Nuno Miguel Mendonça Correia das Neves

B.Sc. in Computer Science and Informatics

Database Replication applied to Network Management

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: José Legatheaux Martins, Professor,
NOVA University of Lisbon

Co-adviser: Nuno Manuel Ribeiro Preguiça, Professor,
NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2018

Database Replication applied to Network Management

Copyright © Nuno Miguel Mendonça Correia das Neves, Faculty of Sciences and Technology, NOVA University Lisbon.
The Faculty of Sciences and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

I start by thanking my adviser, Professor José Legatheaux Martins for accepting me in this project and taking me under his wing. By any means, I was in my best shape during the development of our work, and still, I was able to get more than the necessary guidance throughout the whole duration of this project. I thoroughly appreciate the patience, effort, and help provided from start to finish.

I also would like to thank Professor Nuno Preguiça, which was still able to provide the necessary support when it was needed, while at the same time being overloaded with work or out of academic functions.

My third acknowledgment goes to all the involved colleagues that directly helped in the development of components for this project, especially the Ph.D. students Pedro Lopes e Gonçalo Tomás, that had to stop working on their respective dissertations to provide help.

Another acknowledgment goes to FCT-NOVA, Department of Computer Science, remaining professors, and employees that helped not only during this project but that shaped and assisted my whole academic journey from the start to where it stands.

A special thanks to all my family members, most importantly, my mother and father. Both during their lives had to make enormous sacrifices in order for me to be where I am today. I appreciate the confidence placed on me from an early age, letting me choose my path and take my decisions and risks, even though most of the times they did not agree them. I also appreciate their availability and willingness to help in whatever they could.

A final thanks to all the friends that I have met along the way, particularly the ones that spent their time and attention listening and supporting me during this thesis. A shoulder to lean on is always welcomed, and I was fortunate enough to have a few I could lean on.

Abstract

Software Defined Networking (SDN) is a recent approach used to manage networks. Most of the time it is paired with OpenFlow, a low-level communication protocol used by controlling and switching devices to communicate. Since it is low-level, it does not grant the possibility to explore all the switching functionalities, especially as they get extended with more and more features.

It is therefore required to find alternative ways of coordinating controlling and switching devices without resorting to low-level protocols to be able to access those functionalities.

One of the possible approaches, which was recently implemented in a data center, uses databases and its respective replication to store and exchange information between the devices. Applying the same approach to manage wide area networks would provide a more flexible way to control them.

The goal of this work consists of improving an existing prototype that simulates a small network. It was built originally using a SQL database and an asynchronous external replication software. We replace them with a NoSQL database that natively supports replication, which enables us to remove unnecessary software from the prototype while taking advantage of the database features.

Some of the features, the more notable being non-uniform replication with the help of CRDTs, are used to improve network monitoring, which is a recent addition to the prototype. Network monitoring is a highly important component of network management that facilitates decision making processes.

We evaluate the new version of the prototype by comparing with directly with the old version. We collect the convergence time of the network after an event on a device triggers a modification in its state to help with the comparison. By splinting the convergence time into a sum of smaller actions, we take conclusions regarding different moments of the convergence process.

Keywords: Software Defined Networking, Databases, Replication Mechanism, Network Monitoring

Resumo

Software Defined Networking (SDN) é uma abordagem recente utilizada na gestão de redes. A maioria das vezes é implementada com ajuda do *OpenFlow*, um protocolo de comunicação de baixo nível utilizado pelos dispositivos de controlo e *switching* para comunicarem entre si. Sendo que é um protocolo de baixo nível, não permite a utilização total de todas as funcionalidades dos *switches*, especialmente com o aumento da sua complexidade.

Logo, é necessário encontrar caminhos alternativos para coordenar dispositivos de controlo e *switching* sem recorrer a protocolos de baixo nível para aceder a essas funções.

Uma das possíveis abordagens, que foi recentemente implementada num centro de dados, utiliza bases de dados e os seus mecanismos de replicação para guardar e trocar informação entre dispositivos. Aplicar as mesmas directrizes desta solução para gerir redes de longo alcance proporciona um modo mais flexível de as controlar.

O objectivo deste trabalho consiste no melhoramento de um protótipo já existente que simula uma pequena rede. Foi construído originalmente usando uma base de dados *SQL* e um mecanismo de replicação externo assíncrono. Estes componentes foram substituídos por uma base de dados *NoSQL* com mecanismo de replicação nativo que permite a remoção de *software* desnecessário do protótipo e, ao mesmo tempo, tirar vantagem das funcionalidades da base de dados.

Algumas funcionalidades, sendo a mais importante replicação não uniforme com ajuda de *CRDTs*, são utilizadas para introduzir e melhorar a monitorização da rede, um dos mecanismos recentemente adicionados ao protótipo. Monitorização da rede é um componente importante da gestão de redes que facilita os processos de decisão.

A nova versão do protótipo é testada através da comparação directa com a versão antiga. O tempo de convergência após acontecer um evento num dispositivo que desencadeia uma modificação no estado da rede é colectado para ajudar à comparação. Ao decompor o tempo de convergência na soma de acções mais pequenas, é-nos permitido tirar conclusões referentes a diferentes momentos do processo de convergência.

Palavras-chave: *Software Defined Networking*, Bases de Dados, Mecanismos de Replicação, Monitorização de Redes

Contents

List of Figures xv

List of Tables xvii

Listings xix

1 Introduction 1

1.1 Context 1

1.2 Motivation 1

1.3 Contributions 2

1.4 Document structure 3

2 Related Work: Software Defined Networking 5

2.1 What is SDN? 5

2.2 History of SDN 7

2.2.1 Active Networking 8

2.2.2 Separation of control and data planes 9

2.2.3 Open Flow and network OS 10

2.3 Case Studies 10

2.3.1 OpenFlow 10

2.3.2 Google's B4 SDN network 12

2.3.3 SDN network using a database approach 13

3 Related Work: Replication and Consistency 17

3.1 Replication models 17

3.1.1 Synchronous vs Asynchronous 17

3.1.2 Single-master vs Multi-master 19

3.1.3 Partial and non-uniform 22

3.2 Consistency models 22

3.3 CRDTs 24

3.4 Case studies 24

3.4.1 PNUTS 24

3.4.2 Dynamo 25

3.4.3 Pileus . . . 26
3.4.4 AntidoteDB . . . 26
3.5 Summary . . . 27

4 Prototype Description **29**
4.1 Overall architecture . . . 29
4.1.1 OpenSwitch OPX and LLDP . . . 30
4.1.2 Data-plane . . . 30
4.1.3 Control-plane . . . 31
4.2 Data model . . . 33
4.2.1 Data model - Switch information . . . 34
4.2.2 Data model - Neighbors information . . . 35
4.2.3 Data model - Routing information . . . 36
4.3 Replication mechanism . . . 38

5 Network Replication with AntidoteDB **41**
5.1 Introduction of AntidoteDB in the prototype . . . 41
5.2 AQL . . . 42

6 Prototype Testing **45**
6.1 New prototype (AntidoteDB and AQL) vs old prototype (MySQL and SymmetricDS) . . . 45
6.2 Tests performed . . . 47
6.2.1 First Test: Interface state change . . . 47
6.2.2 Second Test: Neighbor deletion and insertion . . . 48
6.3 Results and Discussion . . . 49
6.4 Deployment enhancements . . . 53
6.4.1 Installation script . . . 53
6.4.2 Bootstrap script . . . 53

7 Network Monitoring with AntidoteDB **55**
7.1 Network monitoring overview . . . 55
7.2 Data model extension . . . 56
7.3 Conventional implementation . . . 58
7.3.1 Switch . . . 58
7.3.2 Controller . . . 59
7.4 Implementation with CRDT use . . . 60
7.5 Testing process and discussion . . . 60
7.5.1 First test: Number of updated replicated . . . 60
7.5.2 Second test: Impact on convergence . . . 61

8 Conclusions **63**

8.1 Future Work 64

Bibliography **67**

A Script Files **71**

List of Figures

2.1 Example of a basic SDN network with one controlling element and three switching elements. 6
2.2 Example of a traditional network with four switching elements. 6
2.3 Timeline that encompasses multiple projects relevant for SDN evolution. . . 8
2.4 OpenFlow switch overall features. 11
2.5 Components of a switch that uses a database approach to exchange information. 14

3.1 Example of two updates being sent to two different replicas following an asynchronous approach. 18
3.2 Example of two updates being sent to two different replicas following a synchronous approach. 19
3.3 Example of two updates being sent to two different types of nodes following a single-master approach. 20
3.4 Example of two updates being sent to two master nodes following a multi-master approach. 21
3.5 Scale containing consistency models order by guarantees they offer. 23

4.1 Prototype switch components. 32
4.2 Prototype controller components. 33
4.3 Entity-relationship model that incorporates all the data regarding switch characteristics. 35
4.4 Entity-relationship model that incorporates all the data regarding switch characteristics, as well as neighborhood data. 37
4.5 Entity-relationship model that incorporates all the data regarding switch characteristics, neighborhood data, and routing information. 38
4.6 Old prototype architecture that includes the old database (MySQL) and replication mechanism (SymmetricDS). 39

5.1 New prototype architecture with AntidoteDB and AQL. 43

6.1 Testing topology used for the various tests. 46

7.1 Entity-relationship model that includes the newly added statistics information (abstracted for simplicity purposes). 57

List of Tables

3.1 The possible combinations of the presented replications models and some of their guarantees. 21
3.2 Analysis of the mechanisms of the presented case studies. 27

6.1 Available resources for each one of the virtual machines. 46
6.2 Versions of the software installed in the virtualized systems. 46
6.3 Hardware specifications of the server that was utilized as host machine for the new prototype. 47
6.4 Average ($\overline{x}$) of the values obtained for the first test (interface state change) that directly compare the old prototype and the new prototype. 49
6.5 Standard deviation (σ) of the values obtained for the first test performed in the new prototype. 49
6.6 Average ($\overline{x}$) of the values obtained for the second test (neighbor discovery) performed in the new prototype along with the available old prototype values. 52

7.1 Number of updates received by the controller with and without resorting to CRDTs. 61
7.2 Average ($\overline{x}$) of the values obtained for the test that compares the prototype with and without monitoring. 62

Listings

5.1 Example of AQL query used to create a new table in the database. 43
5.2 Example of AQL query used to insert values in a table. 44
5.3 Example of AQL used to retrieve values from a table. 44
A.1 Bash script created to install all the necessary software components for both controlling and switching devices. 71
A.2 Bash script responsible for initializing every necessary software component for the controlling device to work. 73
A.3 Bash script responsible for initializing every necessary software component for the switching devices to work. 73

Chapter 1

Introduction

1.1 Context

The majority of the information technology fields registered big advances in recent years. Network related fields are no exception, thanks especially to large improvements in hardware performance and quality, as well as in software, due to the increasing interest in areas like security and cloud computing.

Also, with the number of devices connected to networks constantly increasing, that went from simple desktop computers to cellphones, tablets, and more recently, watches and cars, networks evolved in many aspects to cope with new requirements.

Even though they evolved, when compared to other areas of computer science, its progress has been more limited.

These limitations steam mostly from traditional inflexible network control solutions that are closed and proprietary, low level and hard to extend. There is a long history of unwillingness by network device manufacturers to provide unrestricted access to their systems and respective features, or even provide information about them.

So, researchers, developers, and administrators had their jobs hindered for many years, and their solutions had to be created without knowing the full details of the devices they were working with. Couple with this, successfully developed solutions could not be immediately spread and had to go through not only vendors but specific network institutions to be standardized and published.

1.2 Motivation

In the last decade, an approach used to counter these limitations has emerged. It is called Software Defined Networking and aims to provide an alternative way to manage and

control networks.

This approach redirects the control of the network to external components, instead of the standard approach where all devices make decisions based on the information they have stored locally. These external components, usually called controllers, possess a wider view of the network than switching devices, which enables them to better regulate the network state.

It also allows individuals to manage networks without vendors having the necessity to fully expose their hardware and software. Controllers can be developed without restrictions as long as they support an efficient way to communicate with the devices. It enables the creation of specific management solutions that are more oriented to each project requirement. Multiple renowned companies already implemented SDN based networks, usually in data centers linked to cloud computing and virtualization.

There are multiple protocols and rule sets used to establish communication in SDN networks. The most used one is OpenFlow[18]. Even though SDN materialization is fairly recent, individuals have already begun to search for alternative options and possible improvements for the established protocols. One of them where the authors take advantage of databases and their replication mechanism to implement the same dialog is described in [10].

This alternative served as inspiration for a prototype[20] of a wide area network control solution utilizing a SQL database and an orthogonal asynchronous replication mechanism. There was room for its improvement, especially due to its dependency on an external software to execute the exchange of information between devices. This dissertation aims to upgrade this prototype by introducing a database with embedded replication mechanisms called AntidoteDB[1]. Another goal is to leverage the multiple features of this database to improve existing functions and introduce new mechanisms.

One of the many features AntidoteDB supports is a replication mode were the devices do not replicate their data to every node in the network, called partial replication. It is especially useful since switching devices do not need to replicate information between each other.

It also supports non-uniform replication, which allows the switching devices to replicate data with different levels of priority. Data that does not immediately impact the network routing state is not required to be replicated immediately after it is updated.

1.3 Contributions

The main contributions of this dissertation are:

- **Improvement of the overall prototype functions** - The prototype is extensive, complex and composed by a lot of functions and classes. Like the SDN principles of

[1] https://www.antidotedb.eu

simplicity, we aim to make the structure of the prototype, as well as its functions more clear and user-friendly.

- **Introduction of a new database with embedded replication** - The old prototype has a database that is separated from the replication mechanism. It mainly focuses on cross-platform and flexible configurations which are not a concern in this implementation. Databases with and embedded replication mechanisms generally outperform external replication mechanisms, so we replaced the old database with a new one with embedded replication.

- **Exploit the new replication options** - The new database brought new options regarding replication. While maintaining the basis defined by the old prototype, we took advantage of mechanisms like non-uniform replication.

- **Introduce network monitoring** - We introduced network monitoring in the prototype. By doing this, we not only changed the database, the replication software, and took advantage of the database features but also implemented a new mechanism. Network monitoring is an essential part of networks that supply details and information so that multiple decisions and changes regarding its configuration and state are executed with more precision.

To understand the impact that the changes have in the prototype we compared the performance of the new prototype with the old one. We also evaluated the impact that new features have on the overall performance. When implemented network monitoring, we did not desire to compromise the prototype efficiency, so tests were performed to understand its impact.

1.4 Document structure

The remainder of the document is structured as follows:

Chapter 2

Presents the evolution of the SDN approach since its origin. It details some case studies that help to understand SDN concepts and their implementation. The last case study served as a starting point for this project.

Chapter 3

Represents the other part of the related work, associated with replication and consistency models. Describes one specific important data type and multiple case studies. It also presents AntidoteDB in more detail.

Chapter 4

Describes the overall design of the prototype. Connects design decisions with the SDN approach. Dissects the data model and ends with a presentation of the old external replication mechanism.

Chapter 5

Picks up on the end of the previous one to detail the important features that AntidoteDB provides to the prototype and how they are used. Also details the interface used in order not to change the previous data model.

Chapter 6

Describes the testing environment and process that compares both prototypes and ends with a discussing regarding the collected results.

Chapter 7

Introduces and explains the network monitoring functionality added to the prototype. Compares the version using the non-uniform replication mechanism offered by AntidoteDB with the one not using them and its impact on the prototype.

Chapter 8

Ends this dissertation with the conclusions drawn from the results obtained in the previous chapters and a description of the future work that could be executed.

Chapter 2

Related Work: Software Defined Networking

In this chapter, the SDN approach will be expanded on. In the first section (2.1), a presentation of the overall architecture will be given, comparing it with more traditional architectures. Section 2.2 follows up with an overview of its history to better contextualize its evolutive process.

Relevant cases studies are presented in section 2.3. The most commonly used communication protocol in SDN environments is described in this section, along with an example of a real-life SDN implementation that utilizes that protocol. The chapter ends with the case study that served as inspiration for the prototype presented in 4.

2.1 What is SDN?

Software Defined Networking specifies an approach of a network controlling architecture where the control algorithms do not run inside the standard physical devices (such as routers and switches) but in external components, generally called controllers. There is a clear separation between devices that are responsible for managing the network state (also known as control-plane), from devices responsible for forwarding the traffic (also know as data-plane).

The differentiation of planes and their flexibility allows for more straightforward insertions, deletions, and updates in multiple functions of network related services like management, security, and recovery. This does not apply to traditional network settings where any new functionality requires a new protocol or the modification of an existing one. Also, due to the need for standardization, the process of releasing a protocol becomes long and expensive, with almost no flexibility.

The controller operates in a logically centralized way (figure 2.1), managing the information that is necessary to operate the network. It possesses a global view, which differs from how networks are traditionally operated (figure 2.2), where each device has its own limited view.

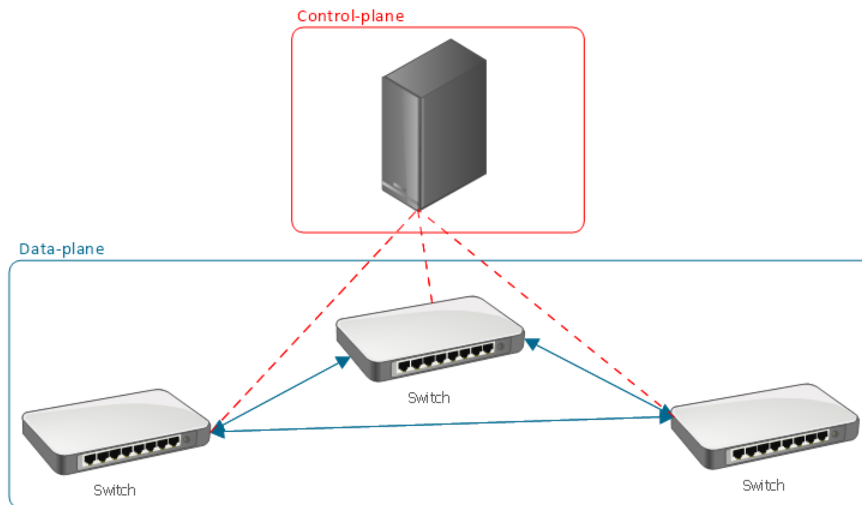

Figure 2.1: Example of a basic SDN network with one controlling element and three switching elements.

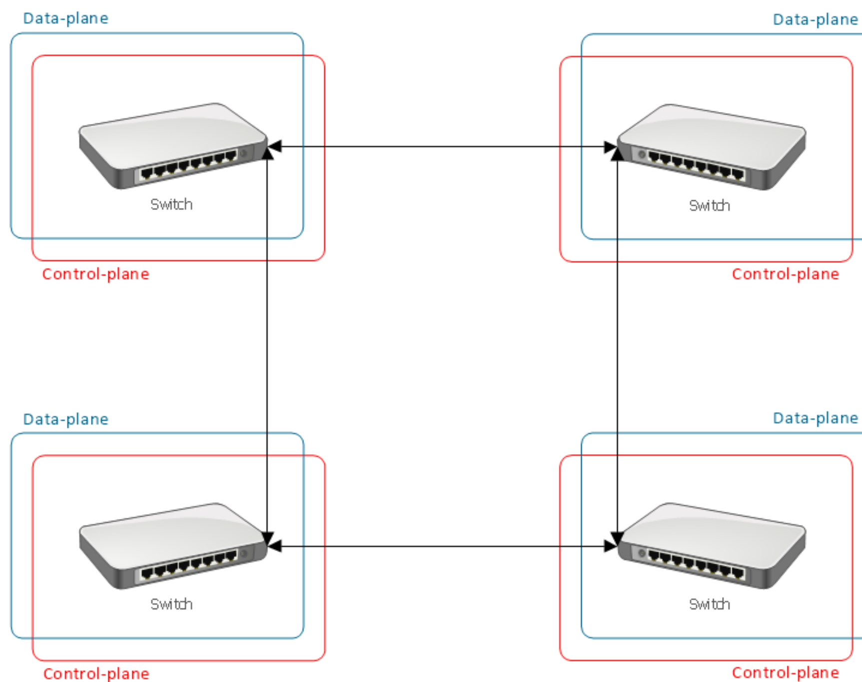

Figure 2.2: Example of a traditional network with four switching elements.

One of the positive aspects of the SDN approach relies on switching devices communicating through standard interfaces with the control-plane. It enables the use of multiple devices in the same network sold by different vendors without having to deal with the

interoperability between them. This diminishes the problem of having to develop and test software to deal with specific manufacturers hardware. The implementation of this approach generally does not increase costs since the control software can generally be installed in off-the-shelf hardware.

One of the biggest uses of the SDN approach is present in network virtualization and service supply in the cloud since it allows network administrators to easily manage traffic and bandwidth, while also creating an isolation level between multiple clients with minimal costs and resources.

It is expected that the adoption of this approach continues to grow in the coming years. A survey[1] conducted by the Network World in 2017 to 294 networking professionals showed that around 50% of them were considering or already in the process of developing a network architecture based in SDN. 18% of the survey respondents were already working on networks managed via SDN implementations.

2.2 History of SDN

The concepts of management, performance, flexibility, and adaptability of computer networks have been in constant evolution, especially in current years, and the SDN approach is one of the most recent products of this progression. However, the evolution was not linear and took some time to mature, with multiple adversities and obstacles encountered along the way.

One of the biggest issues found, which was already mentioned before, was the problem of having to deal with devices that had complex, proprietary and not very flexible control software. It was software that after its first deployment started to incorporate a lot of modifications/extensions developed to overcome obstacles or meet new requirements. This approach goes against the principles of SDN, which are based on the idea of simple and easy development and deployment of networking functions.

To the road that led to Software Defined Networking, which started with the basic idea of creating better network protocols (more efficient, more simple, easier to update) to where it stands today, can be divided into three different eras, like authors of the article[12] proposed:

- Active Networking (2.2.1);
- Separation of control and data planes (2.2.2);
- Open Flow and network OS (2.2.3).

Relevant projects carried out during those eras that impacted SDN appearance and evolution are shown in figure 2.3. These eras will be briefly presented in the next three subsections.

[1] https://cdn2.hubspot.net/hubfs/1624046/State of the Network Executive Summary.pdf

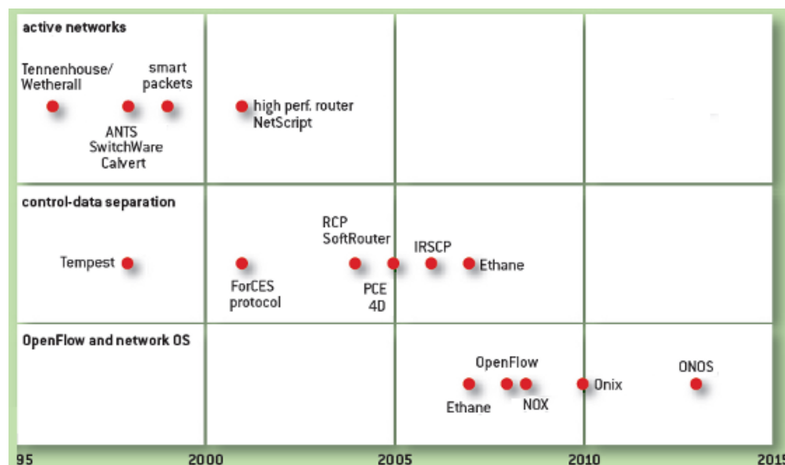

Figure 2.3: Timeline that encompasses multiple projects relevant for SDN evolution.

2.2.1 Active Networking

In the '90s, researchers took advantage of the sudden explosion of the Internet and the emergence of new functionalities to start new projects that developed and tested more complete network protocols.

Some of the projects developed during this era achieved meaningful results in smaller scales. The ones that kept receiving funds, were taken to IETF[2] so they could be standardized, which was an extremely slow process that needed a constant source of income that frustrated many researchers.

Active Networking appeared with the vision of allowing researchers to have an alternative to that process. It was based on enabling access to an interface that provided access to the device resources (memory, queues, processing capabilities) of individual nodes of the network. That access allowed the development of custom functionalities that were applied to specific packets handled by those nodes.

There were two different ways to execute custom made functions: the capsule model and the programmable router/switch model. The first represents a model where the code being executed in the nodes was carried by data packets. In the other model, the code executed by the nodes was established externally. Both of this solutions circumvented the initial problem of needing the approval of entities like IETF to see the outcome of their research deployed on a bigger scale.

Active Networking initiated some of the concepts that later became building blocks of SDN: having the possibility of programming network functions, having an interface that enables access to resources, as well as avoiding the problem of dealing with interoperability, among others.

While a lot of meaningful results were achieved in the laboratories, most of them were not widely deployed. Either they did not solve a problem that was considered to

[2]Internet Engineering Task Force, the organism responsible for standardizing multiple Internet protocols: https://www.ietf.org/.

be critical, or they did it but did not outperform established existing solutions. Active Networking focused too much on searching for solutions focused on the data-plane, while the years that followed were more heavily invested in the control-plane.

2.2.2 Separation of control and data planes

In the early 2000s, with the increasing volume of traffic and emphasis given to concepts like reliability, optimization, and performance, network administrators and researchers constantly sought alternative options to achieve better results in these respective fields.

One of the biggest challenges they found was that devices did not include a clear distinction between control and data planes, even though this separation was already well documented and known at the time. It hindered the possibilities of achieving better results through techniques that took this split into account. So, multiple efforts were made to make more clear the distinction between the two planes, which resulted in projects like:

- ForCES[25] (Forwarding Control Element Separation) which was standardized by IETF and introduced an open protocol used in communications between control and data planes.
- RCP[6] (Routing Control Platform) and PCE[24] (Path Computation Element) that introduced logically centralized control of the network.

Both innovations provided extra tools such as an open and programmable control-plane with wide network visibility and way to communicate with forwarding devices. Their creation, paired with many others, was facilitated by two factors: the appearance of routing code in open software that simplified the creation of prototypes and the advances in server technology that allowed for a single unit to compute and save all routing decisions.

In its preliminary stages, the notion that the separation between control and data plane would not be optimal existed since there was no obvious proof that a network would perform correctly in cases where the control components misbehave, malfunctioned or failed.

An easy solution found for this problem was implemented by inserting packet routing logic in the device's hardware so that even if the control software did not perform correctly, an efficient routing scheme would still be applied. Another technique used to prevent controller failures focused on having replicated instances of the controlling device that would guarantee that if one fails, others could take its place.

However, other problems did not have such clear resolutions. Almost every vendor had little to no incentive to adopt these mechanisms since that could trigger the emergence of direct competition by revealing vendors closed and proprietary features. In conjunction with the necessity of using already existing protocols that were not suitable, it restricted the group of new functionalities that researchers could create and implement.

In the following years, multiple entities created architectures from scratch, avoiding the necessity of dealing with the complexity of existent ones. Some of this projects materialized with success, like Ethane[7], but eventually it was concluded that this approach was not viable in most cases since creating architectures from scratch is a complex and demanding task, that requires a lot of resources.

2.2.3 Open Flow and network OS

In the later 2000s, various concepts that were implemented in smaller scales proved not to be viable in a real-life context. Being aware of this, individuals kept looking for different options.

A group of developers created OpenFlow, which helped in the introduction of a meeting point between the idea of having a fully programmable open network and a functional implementation. OpenFlow is a communication protocol that provides an open interface that allows the control-plane to communicate with the data-plane and vice-versa. It ended up being included on existing and working switch technology, allowing a easier and bigger dissemination.

The protocol started by being tested in networks located on university campuses. It rapidly expanded and opened the doors for other projects and new technologies based on the SDN approach to take place in different environments, such as data centers. In these infrastructures, hiring engineers to write control programs that monitored and managed a large number of devices proved to be more economically viable than consistently buying new switches with closed and proprietary software to have access to new features.

OpenFlow brought numerous contributions for the materialization of SDN, such as the generalization of network devices and their functions. Also helped with the creation of what can be called a network operating system. SDN became the tool that allowed individuals to explore different options on how to manage networks, a platform to experiment with and create new solutions and services.

Since OpenFlow is one of the main protocols used in SDN implementations, it is presented as the first case study.

2.3 Case Studies

2.3.1 OpenFlow

The OpenFlow protocol, standardized by the ONF[3], is available in multiple devices (switches, routers, access points) and enables routing decisions to be taken by identities external to the devices thanks its open communication interface that allows independence from the manufacturer closed features.

[3]Open Networking Foundation, a non-profit organization formed to promote SDN based networking: https://www.opennetworking.org/

A device that supports OpenFlow, contains at least three elements, as figure 2.4 shows: a version of the OpenFlow protocol, a secure channel that connects a device to the controller and enables bidirectional communication between the two and a flow table with entries that have information about the actions to be taken for each flow.

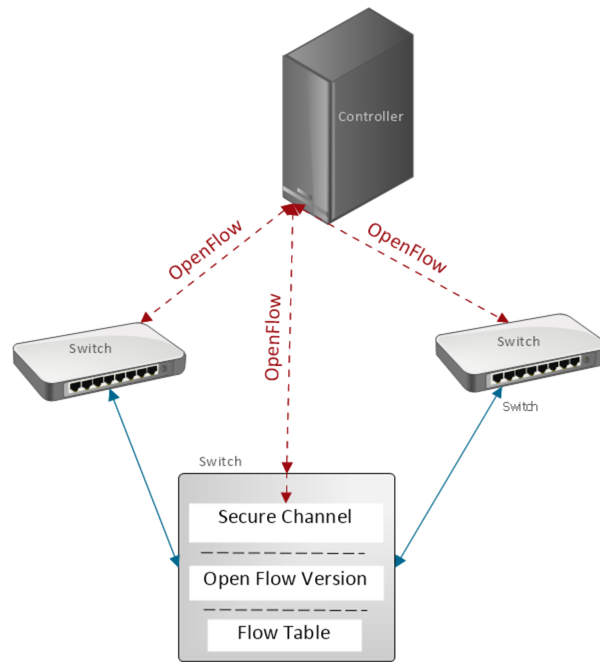

Figure 2.4: OpenFlow switch overall features.

Each entry in the flow table has three important fields: a pattern that matches the header of each packet and identifies to which flow it belongs, a group of statistics which allow to track values like number of packets and bits processed by a flow and an action to be applied for each packet that is part of a flow.

The three basic actions that every device that supports OpenFlow has to execute are:

- Send a packet belonging to a flow to a port;
- Throw away the packet;
- Send the packet to the controller.

The packets that are sent to a controller are the ones were the header does not match any entry in the flow table. In this case, the controller proceeds to build and implement a flow for that packet, so that future ones with similar header do not need to be forwarded to the controller. The packet that was sent to the controller is then sent through that flow.

The controller adds and removes entries from the table of each device. A basic controller can be a single application running on a commodity machine, handling a small number of devices. In real-life contexts, the controller tends to be more complex, typically created in a cluster topology to avoid cases where it fails. It also manages a large

number of devices, dynamically removing and adding entries to multiple tables in an effort to operate efficiently and correctly.

One of the many upsides of architectures using OpenFlow relies on them having the possibility of allowing normal traffic (non-experimental) to be handled independently from the experimental one (thanks to extra actions for flows), isolating one from the other. This allows tests to be performed without having a direct impact on the normal network traffic.

Examples like Hedera[3] or PortLand[19] are some of many that use devices that support OpenFlow and their respective features. The next example presents another SDN setup where OpenFlow is used.

2.3.2 Google's B4 SDN network

Google data centers operate through two distinct WAN's[4]. The first one is mainly dedicated to answering user made requests coming from external domains. The second one provides a stable connection between multiple data centers and is known as B4[14].

The most significant reasons behind the creation of B4, besides the different objective that fulfills when compared to the other Google data center network, were:

- Possibility of having an elastic bandwidth, since most of the traffic between data centers originates from the synchronization of big chunks of data between them. It obviously benefits from having high bandwidth, but can easily tolerate temporary failures and scarcity without much drawback.
- Having control of both applications and end sites allows administrators to easily enforce priorities, configurations and control bursts at the network's edge.
- Little to no concerns when facing scalability challenges since there is only a moderate number of data centers (15) and a very low probability of fast exponential expansion.

Its architecture can be dissected in three different layers. Each data center includes a first layer composed of switches that route traffic. Since this is a SDN based implementation, these devices do not run complex control software. So, there is a second layer that contains network control servers, responsible for managing the previous layer.

The communication between layers is facilitated by the usage of the OpenFlow protocol. The main goal of the second layer is to maintain an operational local network state, managing dynamically entries in the device tables, based on occurring events that have a direct impact on the network like failures and overloads.

The third layer is composed of global application servers (SDN Gateway and TES[5]), that are replicated in each data center and possess a global view of the network. SDN

[4]Wide Area Network, long distance network that covers a big geographical area
[5]Traffic Engineering Server, server responsible for the enforcement of traffic engineering rules.

Gateway abstracts OpenFlow and hardware details so that TES can easily manage traffic and bandwidth between data centers using multiple paths, aided by the applications located in each one and information like statistics.

The last presented case study is a SDN implementation that does not rely on OpenFlow to exchange information between the two planes.

2.3.3 SDN network using a database approach

Not every SDN implementation resorts to OpenFlow. For instance, Bruce Davie et al. (2017) describe a solution[10] for managing client networks in a data center utilizing a different approach.

One of the biggest and most common challenges found in architectures that avoid using established protocols like OpenFlow is the necessity of dealing with interoperability between control systems and data-plane devices. Throughout the years, the most common way to solve this issue was by defining and creating new protocols. With the emergence of new and complex requirements, it became a more impractical solution.

A study was developed by these researchers to find a way to shape the information that network components had to share to correctly operate. Another concern was how that information would be distributed and synchronized between them. Then, the two biggest questions regarding the choices made for the overall system architecture were:

- What abstractions are appropriate for information exchange?

- What is necessary to establish an adequate mechanism for synchronizing information?

Regarding the first question, instead of using low-level protocols and their data models, the abstraction level was elevated. This means that the control-plane devices were not limited to protocol available functions and that they could send configurations to the data-plane devices and expect them to make use of their most optimal features to implement the provided configuration.

This different approach was taken mostly because even though switch ASIC's[6] support a big number of routing operations, with the usage of a protocol like OpenFlow, access to some of those operations is lost, since low-level protocols do not provide access to all of them.

The answer to the second question is centered on the hypothesis of information management being treated as a generic database synchronization problem instead of a specific management protocol exercise.

These factors led the developers of this solution to opt for databases to avoid protocol restrictions. This way, after the information is stored in the devices, they are responsible for, with the given information, executing their functions using their best mechanisms.

[6] Application Specific Integrated Circuit, circuits prepared to achieve better performance values when used for a specific task, instead of general purpose CPU's

Additionally, they used database replication techniques to share and synchronize information between both planes. Figure 2.5 shows a simple SDN network implemented using this approach, which can be compared with figure 2.4 to understand the overall difference between both.

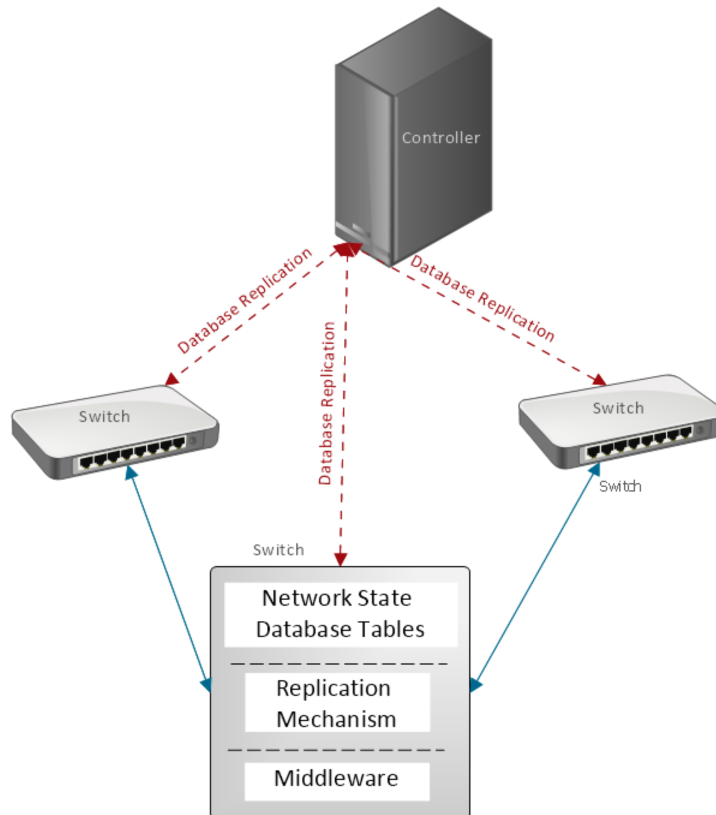

Figure 2.5: Components of a switch that uses a database approach to exchange information.

Using this approach, network operations are separated for state synchronization and from details of protocols used for communication between devices. With the use of databases, there was no necessity of creating or developing a new system, since there are multiple and well-documented database management ones.

This approach also brought some other benefits: there was no need to deal with the complexity of eventual consistency, common in network control protocols since, usually, database synchronization protocols offer more stronger levels of consistency. Additionally, every time modifications needed to be handled in the data model, the process was simple since database schemes are easy to modify or extend.

To manage and implement the database system in this solution authors used OVSDB[7] (Open Virtual Switch Database), an open-source database management protocol for SDN environments. Each switch runs an OVSDB database server and for each existing column

[7]http://docs.openvswitch.org/en/latest/ref/ovsdb.7/

in the database, either the switch or controller are the master, with their counterpart being responsible for having a synchronized and up-to-date replica in memory.

The switch opens a connection with the controller and after the controller acknowledges the connection, the switch writes and sends database queries to extract all the necessary updates. After this, for each column where the switch is the master, the controller installs triggers so that every update is sent back to it. In the opposite case, the controller simply sends updated values to the switch.

The database schema used in this implementation is divided into three layers:

- Physical;
- Logical;
- Logical-Physical Bidding.

The reason for this division relies on the fact that this particular solution had to deal with virtualized and non virtualized components.

The first layer contains a table called *PhysicalSwitch* that includes, among other information, management IP's and tunnel IP's. It also contains the information about all the switch physical ports in a table called *PhysicalPort*.

To provide information about the logical switches, the scheme includes a table which is called *LogicalSwitch* that belongs to the logical layer. The key for every entry in this table matches every packet sent to this switch. The controller is responsible for spreading this table.

To facilitate the mappings between logic and physical switches, two tables were created: *UcastMacsLocal* and *UcastMacsRemote*. They contain data that facilitates routing packets and the creation of links between virtualized and non-virtualized switches.

The idea for this dissertation originated from this solution. It can be applied while circumventing the necessity of delving into complex management protocols and tools while working with relatively accessible concepts like database replication, data modeling, and consistency. The next chapter of the related work gives an overview of these concepts.

Chapter 3

Related Work: Replication and Consistency

In this chapter, the second part of the related work is presented. Section 3.1 and section 3.2 detail respectively, replication and consistency models that are used or serve as the basis for the ones used by network related applications. The next section (3.3) also discusses a mechanism used by many distributed storage systems.

Multiple case studies are presented in section 3.4. These cases studies include the models presented in 3.1 and 3.2 or extensions of them. This section also presents the database used for the newly implemented prototype, along with its details.

3.1 Replication models

With the number of users that own network connected devices growing, as well as the number of devices connected to the network constantly increasing, centralized storage systems cannot fulfill all the requests that are made to applications. Thus, developers resort to the use of distributed storage systems to answer the growing number of requests.

A distributed storage system requires mechanisms to exchange information. They resort to replication to send updates between their nodes. Some replication models will now be presented.

3.1.1 Synchronous vs Asynchronous

One of the first decisions made when building a distributed storage system is to choose whether the replication between multiple storage nodes is synchronous or asynchronous.

In the asynchronous approach, the replication of an update does not have to occur immediately after it is received. In this case, progress can generally be assured without needing the acknowledgment of other replicas and replication can occur, for example, on a scheduled basis.

This tends to be an advantage in terms of performance, but can also be a disadvantage since it can lead to situations where it is necessary to solve conflicts created by concurrent operations made on the same object, generally resorting to reconciliation methods.

Figure 3.1 shows a small example of this model where two updates (1 and 2) on the same object are performed by different clients in different replicas. After a short period of time when they replicate the updates (3 and 4), they have to resort to convergence operations since the updates were not originally performed on the object they have stored in memory.

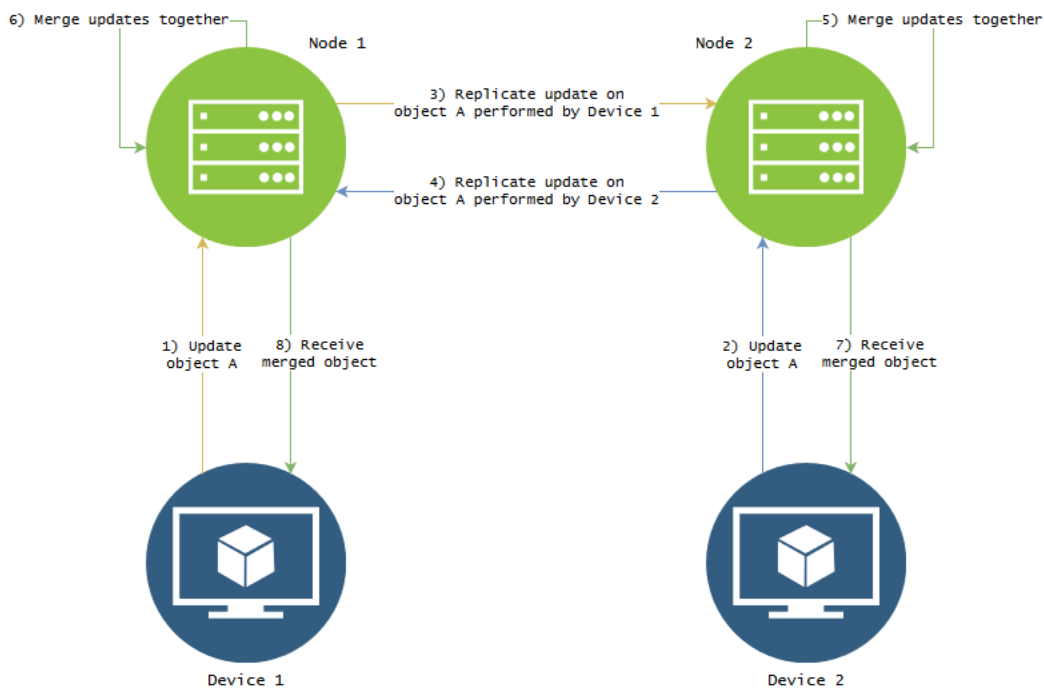

Figure 3.1: Example of two updates being sent to two different replicas following an asynchronous approach.

The alternative approach is synchronous replication. In this model, an update is replicated to every other replica (or a majority of them, depending on the architecture [13]), that has to execute it and acknowledge its results to the replica where the update originated before progress can be made. This usually requires a stable and efficient connection between the parties involved to guarantee progress and efficiency.

In this case, there are no inconsistencies in the data and there is no need to use reconciliation methods. Successful read operations always return the most recent object available. A disadvantage of this model is observed when there are considerable queue times that are a consequence of the coordination process with all replicas that store the same object.

To contrast with the previous model, figure 3.2 shows an example using the synchronous approach. In this case, when the second client tries to update object A (2), it has to wait for the first update (1) to be replicated and acknowledge by every replica. Only

after he can update the object, if necessary.

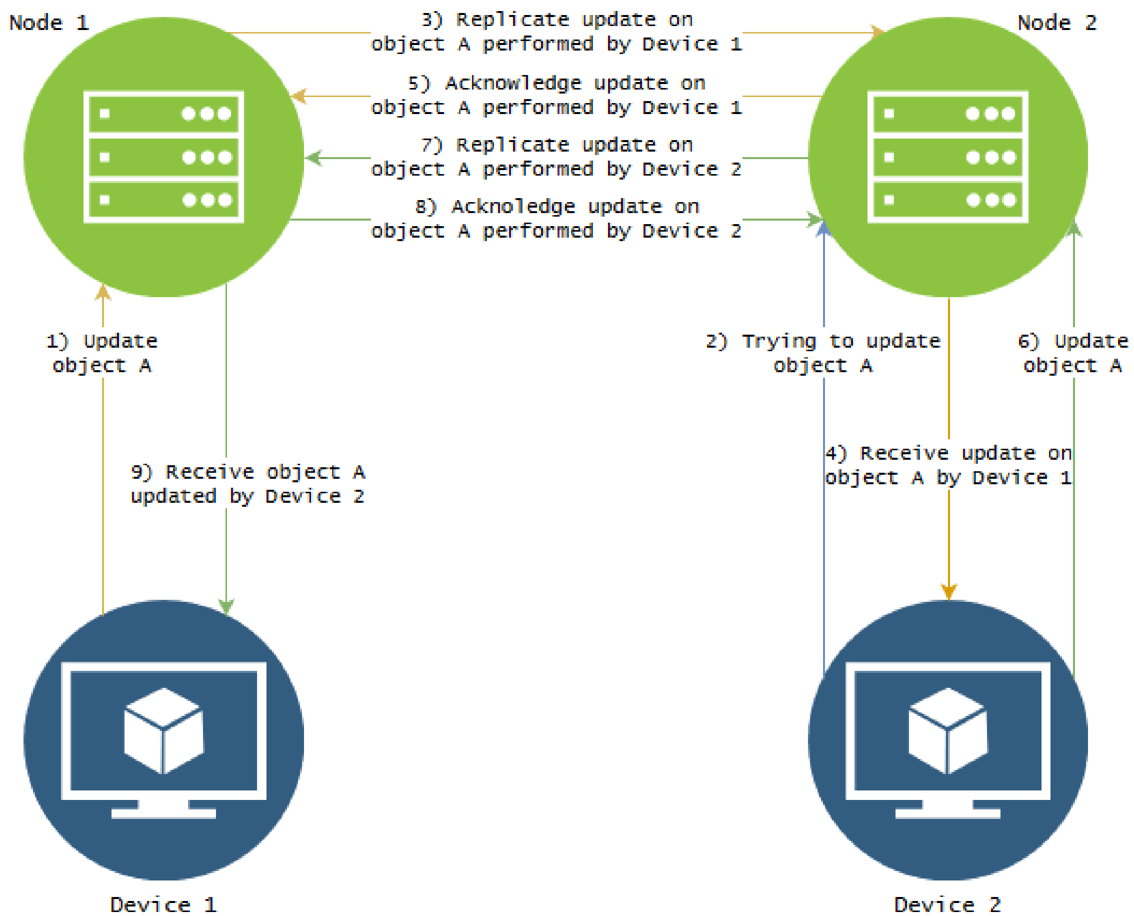

Figure 3.2: Example of two updates being sent to two different replicas following a synchronous approach.

3.1.2 Single-master vs Multi-master

Another available option for distributed storage systems regarding replication is whether objects have one or more masters.

In the case where there is only one master of an object, model known as single-master or primary-secondary, the master is responsible for keeping the object's most recent version. Every update has to be accepted and executed by the master before it is replicated to every other replica.

When an update is received in a node, if it is not the master, it sends a remote procedure to it requesting the modification in the object. If accepted, the master makes the change and propagates it to every other replica. Figure 3.3 shows an example where a direct update on the master (1) does not require any acknowledge by other replicas, but an update made on a non-master replica (4) requires the acceptance of the master of that object.

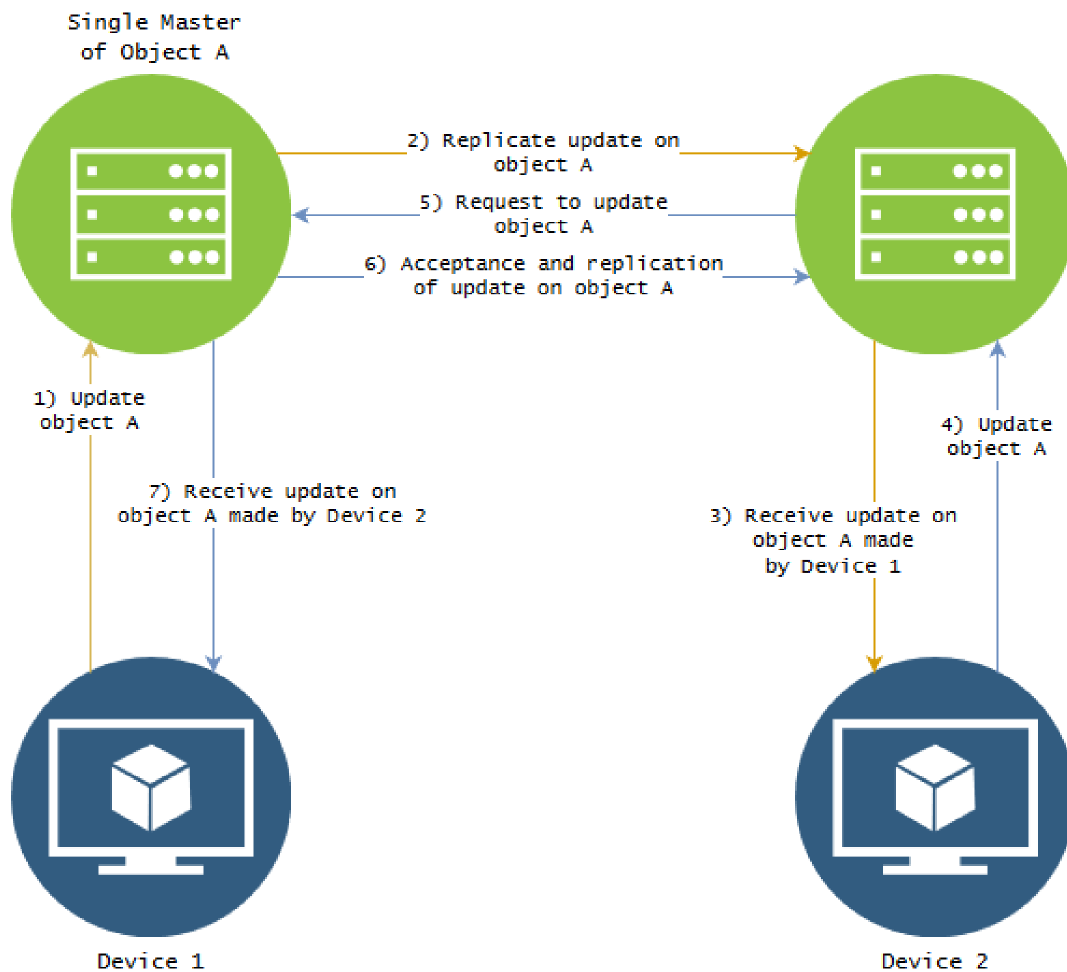

Figure 3.3: Example of two updates being sent to two different types of nodes following a single-master approach.

Delays can take place in the whole system if the master receives multiple requests for the same object, more than he can handle in a reasonable amount of time, or simply takes too much time handling requests on its own due to internal issues.

In the multi-master model, every master replica can update an object without coordinating with other replicas. Figure 3.4 shows an example where two different nodes are master of the same object, and both can receive updates (1 and 4) and simply replicate them after.

A problem with this model occurs when multiple different masters concurrently update the same object and then try to replicate those updates. It can end up in the same situation presented in 3.1, where reconciliation methods are required. If no method is used, the information can end up being stored with different values in different replicas.

Table 3.1 shows a comparison between options available when the previously presented models are combined. The main differences between them are found in performance levels and how updated is the data when a read operation is performed.

Asynchronous replication coupled with multi-master present the best performance

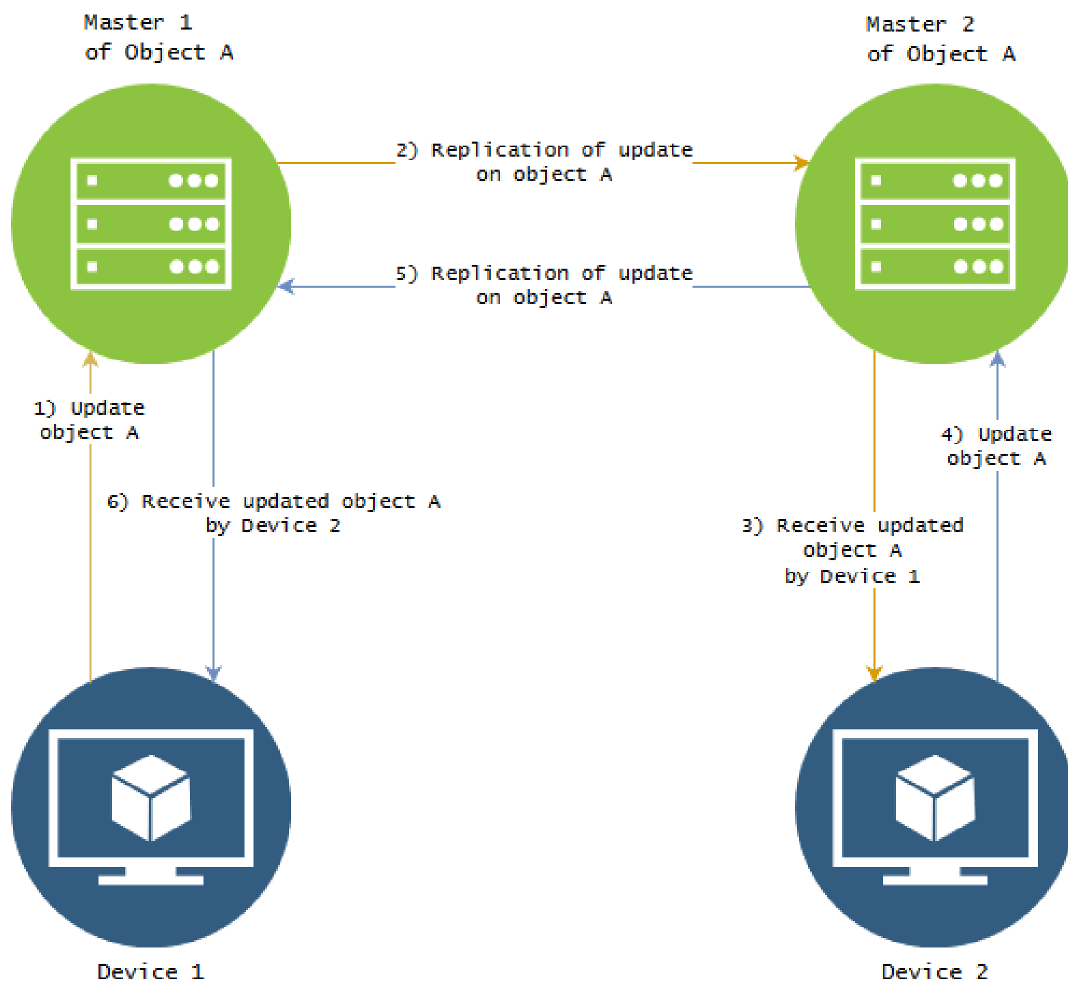

Figure 3.4: Example of two updates being sent to two master nodes following a multi-master approach.

level since both, by default, do not require any acknowledgment process to confirm the reception of updates. In the opposite case, synchronous replication coupled with single-master generally assures that the read values are the most updated ones, but it lacks performance when compared to other approaches.

Table 3.1: The possible combinations of the presented replications models and some of their guarantees.

Ownership / Propagation	Single-master	Multi-master
Synchrounous	++ Recent Data - - Performance	+- Recent Data +- Performance
Asynchrounous	+- Recent Data +- Performance	- - Recent Data ++ Performance

3.1.3 Partial and non-uniform

In recent years, there is been an increase in the size of information required by applications to answer multiple client requests spread around geographically. Coupled with the respective increase in the number of data centers that are built to better serve those clients, the option of not maintaining the application's data in every data center became a reality.

So, in partial replication, each object is only replicated in a subset of nodes. This improves capacity/scalability and increases performance since multiple nodes can store different parts of data. However, nodes may not be able to process every query, since they do not store all the information, adding complexity to the querying process.

Non-uniform replication[5] was proposed as a solution where every replica keeps only part of the data but all replicas can answer most queries. It relies on the concept that for some data objects, read operations do not have to access the entire set of data.

One of the examples presented to clarify this idea is the leaderboard example. In this example, an object can be used to store the leaderboard of an online game. Each user only needs to have his own information saved in the nearest data centers (to achieve lower latency and be fault tolerant). In this case, it is not necessary to replicate the information of all users in all replicas, since only the information about users in the top position needs to be replicated to answer the leaderboard query.

This approach uses eventual consistency and the data type described in section 3.3 to easily meet their goals and to remain competitive with other replication schemes. The refereed consistency model along with more examples will now be presented.

3.2 Consistency models

Replication models are always paired up with consistency models. A consistency model defines a set of rules that describe the guarantees that the system offers when operations are executed on data it stores.

Depending on the chosen model, some provide weaker consistency levels (they allow read operations to return values that are not the most recent), while others provide higher ones (all read operations return the most recent value). To elaborate more on consistency models, a few key ones will be presented in ascending order, taking into account the consistency level that they offer (figure 3.5).

- **Eventual consistency** represents the most relaxed model that is commonly utilized. In this model, it is expected that after a certain time where no updates are performed, the state of all replicas converges to the same. This allows read operations to return older values.

  It is generally paired with a reconciliation method to avoid having different final values for the same object in different nodes. The reconciliation method can be as

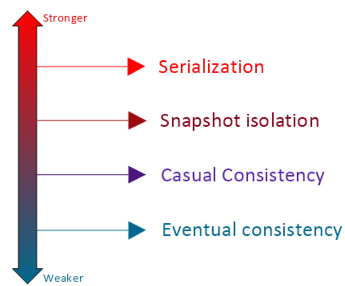

Figure 3.5: Scale containing consistency models order by guarantees they offer.

simple as last write wins, which simply takes into account the most recent update of the concurrent ones, although different solutions are also available.

- **Causal consistency** model assures that reads and writes that have a causal relation are seen by every replica in a common causal order. For example, when there is an operation like the creation of a post in a social network and then a comment on that post, causal consistency guarantees that the first operation happens before the second one, in all replicas.

  This is especially helpful when two non commutative operations take place, since the order by which they execute matters. However, like eventual consistency, does not guarantee that concurrent writes that are not causally related are seen by the same order in every replica.

- **Snapshot isolation** assures that every read operation in a transaction is carried out over a consistent view of the database. The view usually corresponds to the state of the database at the beginning of the transaction, which means that if its values are updated during that period when operations of the transaction are taking place, the view does not represent an updated version of the database.

  Transactions only fail if values are updated after the view was created and during their execution since they usually do not support any convergence mechanism.

- **Serializability** is the model used when the goal is to get strong consistency and isolation. It guarantees that the execution of a set of concurrent transactions applied over one or more objects is equivalent to a sequential execution of those transactions. It generally does not require any convergence or reconciliation method. In contrast with other weaker models, it can limit availability since usually a transaction over an object has to finish completely before another transaction on the same object takes place. It makes this model less attractive for distributed storage systems when compared to the other ones presented since availability is an important factor in these systems.

The presented models are four of the many available. It is also common for developers to merge concepts from different models to better suit their solutions. The following

section presents the emerging data type that is usually paired up in some of the weaker models.

3.3 CRDTs

A wide variety of replication models, as it was referred before, have the problem of dealing with concurrent write operations that can lead to inconsistencies in different nodes. To solve this issue, they have to resort to some type of reconciliation mechanism, that can be costly in either availability and/or performance.

Conflict-free Replicated Data Types[21] or CRDTs are data types (for example, counters, sets, maps, and sequences) that can be concurrently updated by different nodes without needing any coordination between them since they always end up with the same state. Since there is no need to use any coordination mechanism, they offer, by themselves, guarantees regarding the convergence of values that the standard data types cannot offer. CRDTs are used in multiple online applications, for example, online betting application Bet365[1] and online game League of Legends[2]).

The are two main types of CRDT replication. One focuses on the object state and the second one on the operations that are executed on an object.

In the first case, after an update changes the state of an object in a node, the resulting state is replicated to the other replicas. Since CRDTs have the special property of mathematically being always possible to merge, replicas can receive concurrent states from different nodes.

The second one is similar to the first one, but instead of replicating states, it replicates the performed operations to be applied locally by the other replicas. CRDTs require the operations to be commutative in order for the final objects to be equal in all nodes.

3.4 Case studies

The following case studies provide details about multiple systems and protocols designed by renowned entities such as Amazon or Microsoft, describing real-life examples of distributed storage systems that implement the models presented in prior sections.

3.4.1 PNUTS

Big web applications have the necessity to be constantly available, be highly scalable and have consistent and fast response times, for a high number of clients dispersed geographically. Depending on the applications, they can tolerate weak consistency levels without compromising their standard behavior.

[1] https://www.bet365.com/

[2] https://play.euw.leagueoflegends.com

It is with this conditions in mind that Yahoo! engineers developed PNUTS[9], a globally distributed database system used by their applications.

Some of the baseline characteristics of this system are:

- Simple and clear relational model exposed to users with support for basic queries.
- High availability for read and write operations even in the presence of failures thanks to high redundancy.
- Data replicated in multiple locations to offer lower latency and allow usage by multiple applications.

PNUTS provides the *Per-record timeline* consistency model, also presented in [9], which guarantees that all replicas that store an object apply all updates in the same order.

PNUTS uses the *Yahoo! Message Broker* (YMB), which is a topic based publisher/subscriber system. In more detail, YMB is based on the single-master replication model. Updates are sent to the YMB, which at some point is responsible for asynchronously propagating them to the replicas.

YMB guarantees that updates are not lost by storing them on multiple disks in different servers and not removing the update from the YMB log until it is verified that it was acknowledged by every replica. Since it guarantees that in case of data loss, every accepted update is recoverable from a remote replica, it eliminates the necessity of recovering the storage unit itself. Updates originated from non-master replicas must be forwarded and accepted by the master before being committed. It does not require immediate answers from the replicas to make progress. Since this system is globally spread, there are multiple scattered YMB clusters to improve performance and latency.

3.4.2 Dynamo

Like PNUTS, Dynamo[11] was built by Amazon engineers with the objective of being a distributed storage system that suited their application needs. It was designed to be able to cope with failures, without having a major impact on availability and performance.

It differs from other systems because every service has its own instance of Dynamo. Some of its main characteristics are:

- Implements a consistency model based on eventual consistency, with some improvements like object version control using vector clocks, where multiple versions of an object are kept in the system.
- Supports an update acceptance scheme based on quorums.
- Supports a simple key/value interface.
- Data is partitioned and replicated using consistent hashing.

In consistent hashing, each node in the system has a random value that is in range of a hashing function. To check to which node a data object belongs to, it is only needed to execute a hash function with the key of that data object, and then compare the output with node values and check which one is the closest. So, in case of node failures, only its neighbors are affected.

Every data object is replicated an arbitrary number of times N, which is always big enough to ensure that the system copes with failures. Each node is responsible to replicate the data to their next N-1 successor in a clockwise direction.

3.4.3 Pileus

When designing a distributed storage system, the developers have to choose a consistency model. However, it happens frequently that developers do not have all the necessary information to make this decision. Changes in key factors like server and client location, network bandwidth and configuration details, among others, can drastically change the necessities are requirements.

After acknowledging this premise, Microsoft researchers created Pileus[22], a distributed storage system that allows applications to declare their consistency and latency priorities through Service Level Agreements (SLAs). The system tries to match the application request by choosing dynamically which servers provide the best service.

A SLA contains multiple pairs of consistency and latency values. The first pair on the SLA is representative of the preferred values requested by the application. The list of pairs is sorted by preference level, in a descending order. It should be noted that in the case where no pair of values can be meet by any server, just an error message is showed. In Pileus, unavailability is defined as the impossibility of executing operations while meeting the requested values.

The system is divided into two main components: storage nodes, which are mainly responsible for storing data and providing interfaces for access to that data and replication agents, which are responsible for propagating updates between storage nodes.

Pileus replication model is based on the single-master model. A master node is tasked with the function of receiving updates for the object and soon after replication agents proceed to propagate that object after the update is concluded.

3.4.4 AntidoteDB

When compared with NoSQL databases, traditional SQL databases tend to be less suited for replication across multiple locations due to some limiting factors like lower availability and lower recoverability speed when network failures occur.

They scale vertically, which means they scale by adding more processing power (RAM and CPU, for example), which is more costly than scaling horizontally as NoSQL databases do. Scaling horizontally is performed by increasing the number of machines and not the by improving their components.

However, NoSQL databases can show data inconsistencies when using asynchronous replication, which is an issue. AntidoteDB was developed to deal with this issue while maintaining the benefits of a NoSQL database. It is a highly available, transactional database with low latency, with a simple API written in native Erlang.

AntidoteDB takes advantage of many features already presented in this chapter. It supports CRDTs and non-uniform replication.

The system executes the Cure[2] protocol that provides a consistency model called *Transactional causal consistency* (TCC). TCC is an extension of the causal model, that ensures causal order of updates, allows reads to be done over snapshots of the system that includes the effects of all transactions that causally precede it. Also assures that a transaction updating multiple objects respects atomicity and takes advantages of CRDTs to guarantee that the state of the replicas converges, even in the presence of concurrent operations.

There are some design options that were adopted to allow for TCC model to be used without compromising availability and scalability.

First, it utilizes event timestamps to causally order dependencies which allow replicas to make decisions locally. This excludes the necessity of verification processes to check dependencies that lower performance. Second, it stores versions of an object so it can assist in the creation of unique snapshots with distinct time frames. It separates the problem of propagating updates from the one of making them visible allowing replicas to propagate updates without having to coordinate with each other.

3.5 Summary

Table 3.2 presents all the case studies that were detailed in section 3.4 with their respective consistency and replication models.

The goal of all of them is to provide the best distributed storage system, guided by their requirements and developer guidelines. This is why they differ on the different models they use since there is no best option that fulfills everyone's requirements.

Table 3.2: Analysis of the mechanisms of the presented case studies.

Mechanism / Case Study	Consistency Model	Replication Model	CRDTs
PNUTS	Per-record timeline	Single-master / Asynchronous	No
Dynamo	Eventual consistency (w/ improvements)	Consistent hashing	No
Pileus	Selectable	Single-master / Asynchronous	No
AntidoteDB	TCC	Single-master / Asynchronous	Yes

Chapter 4

Prototype Description

The prototype architecture is detailed in this chapter, along with specifications regarding its design. Section 4.1 connects the overall architecture of the prototype with the SDN approach, clarifying the separation between control and data planes and their respective components. It also presents particular design options that facilitate the prototype implementation.

The data model used for the prototype is presented in section 4.2, along with detailed descriptions of its attributes, ending up with the presentation of the overall entity-relationship model that defines it. Ends with a presentation of the old database replication mechanism in section 4.3.

4.1 Overall architecture

The inspiration for the defined architecture and its prototype came from the database approach presented in subsection 2.3.3. To recap, developers created a unique SDN design utilizing databases in order to not rely on existing protocols or create new ones to encompass devices features.

However, this prototype was not made with the purpose of being a simulation of a production-ready solution. It is supposed to serve as a proof of concept that a wide area SDN based network can be operated using the presented approach.

The prototype is completely virtualized, meaning there are no physical elements besides the machine that hosts the virtualized components. Since it is fully virtualized, the switches hardware is simulated with the usage of OpenSwitch OPX[1].

[1] https://github.com/open-switch

4.1.1 OpenSwitch OPX and LLDP

OpenSwitch OPX supplies the prototype with the possibility of having a fully virtualized testing environment, running in a single host machine. Developed by Dell, it virtualizes the hardware of network switch devices in a Debian operating system. It is also available in this vendor physical switches, which enables physical deployment.

Being a kernel based Linux system allows the usage of C and Python programming languages, which are supported by the OPX API. This facilitates the developing process of the middleware necessary for each switch, while at the same time, avoiding the necessity of acquiring non-virtualized switch hardware. In our case, all the middleware present in switches was fully written in Python.

OPX services enable the use of Linux IP stack network features for packet forwarding that are utilized to transmit packets between switches. The data model used in OPX is called YANG[4], created by a group of individuals in the IETF with the objective of being a model that provides information about characteristics of the nodes in the network and the interaction between them.

OPX also provides a mechanism for this prototype called CPS[2], also know as Control-Plane Services.

CPS is particularly important because it provides, first, an event subscriber mechanism able to detect changes in switches interface state, and second, a mechanism to retrieve switch characteristics. They are stored in the switch database and helps to place the switch in the network topology. CPS also provides an API to control the state of the device interfaces and its routing table.

There is another important mechanism that is used, which is not only available through OPX but is fully standardized and available in most operative systems called LLDP[15], also knows as Link Layer Discovery Protocol.

LLDP is a protocol used by network devices to discover their neighbors and announce their identity and resources. It is standardized and independent from the hardware vendor. While using this protocol, the switch periodically sends frames through each port, in 30-second windows by default, that are collected by its neighbors and allow them to acknowledge the switch existence and its characteristics. The usage of this protocol facilitates the creation and discovery of the network topology, simplifying the controller process of computing routes.

OPX is utilized by every component of the data-plane. As every SDN implementation, there is a clear separation between data and control planes. Descriptions for both these planes will now be given.

4.1.2 Data-plane

The data-plane is composed of virtualized switches. These switches were stripped of most of their decision-making capabilities. They are able to set up and store internal switch

[2] https://github.com/open-switch/opx-cps

information without establishing any connection among them, but they all have to be connected to the controller to be a part of the network.

Like the controller, there is a twist in its design. In this prototype, the switches are able and required to detect their direct neighbors. This avoids the creation of a complex process in the controller to establish neighborhood connections between switches. Switches only possess a reduced view of the whole network, which only includes its direct neighbors. This information is constantly replicated to the controller.

The specific data that a switch replicates to the controller is acquired via a middleware that makes the bridge between the database and the logical hardware layer. To retrieve part of that information, the software makes multiple downcalls to the logical hardware layer, then converts the results and stores them in the database. This information is especially valuable for the controller's view of the network.

In addition to making downcalls, this software is responsible for receiving upcalls from the logical hardware level and transform those results into data that can be stored in the database. These upcalls are events that usually have a direct impact on the network state, so they need to be replicated to the controller so it can compute, update, and transmit the new network state.

The decomposition of the switch is done in four main parts, as figure 4.1 shows:

- The database;
- The database replication mechanism;
- The middleware responsible for detecting and dealing with database changes made by the controller, making downcalls to the logical hardware layer, and receive upcalls from that layer.
- OPX and its features.

4.1.3 Control-plane

The control plane is composed by a single controller. In this prototype, all the components in the data-plane directly communicate with it. In real life implementations, SDN based designs like this have the controller replicated multiple times, to quickly recover from cases where it suddenly fails. This way, one of the replicated controllers can assume its role without compromising the state of the network.

There are two reasons why controller replication was not added to the prototype. First, the goal of this prototype is not to prove that a SDN control-plane can be fault-tolerant (which is already proven), even though it would be an interesting overall addition to its implementation. Second, the main architecture, data model definitions, and features of the prototype are still on developing stages. The objective of this thesis is to focus on the coordination between switches and the controller since time constraints prevent the deployment of too many features which can be added later.

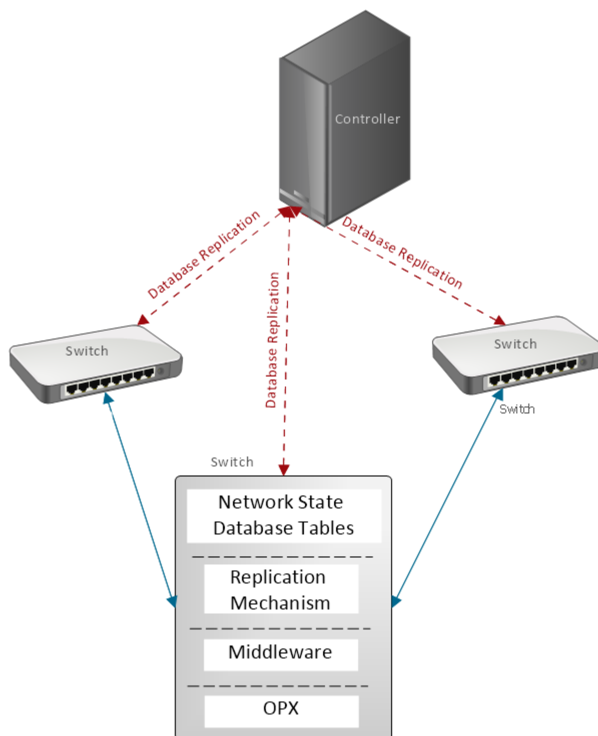

Figure 4.1: Prototype switch components.

The controller stores all the information regarding data-plane components state to create the global view of the network. With this view, it is able to create specific routes for each device that are immediately installed in switches to allow them to forward traffic. These routes are created with the help of a routing algorithm that was built to calculate shortest paths.

The controller is also prepared to constantly receive updates of changes in the network status, and if needed, recompute its view, so that switches are constantly provided with the most updated and correct routing rules.

Like it was mentioned before, there is a twist to the controller when compared to controllers in other designs, such as OpenFlow based ones. It does not wait to receive packets with no defined route (or flow). The controller is proactive in the sense that it calculates the most optimal paths for every possible IP prefix in the network, even though that route may not be necessary at the time it is computed. This simplifies the process of creating routes, and at the same time avoids the necessity of having a mechanism to receive, decapsulate, encapsulate and forward packets in the controller.

The controller can then be broken down into four main parts, as figure 4.2 shows:

- The database;
- The database replication mechanism;
- The routing algorithm;

- The middleware responsible for detecting and dealing with database updates.

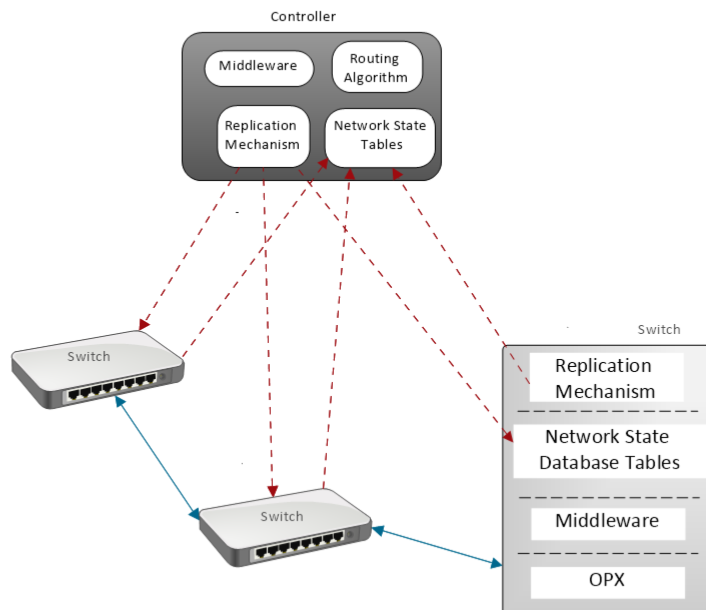

Figure 4.2: Prototype controller components.

4.2 Data model

Since both planes are now introduced, the data model will be presented. This data model stores enough information to control a full IP network, containing details regarding switches, interfaces, neighbors, and routing. To facilitate its presentation, the data model is divided into the three following parts:

- Switch characteristics;
- Neighborhood information;
- Routing data.

Recalling the definitions of control and data plane in this prototype, it is easy to perceive why there is no reference to controller related information. The controller performs actions purely by detecting changes in switches data. In this prototype, there is no information that the controller must store of himself or surrounding environment to manage the network. The controller utilizes the same data model as the switches, with a database that encompasses a replica of every switch data.

In the following subsections, each one of the three parts of the model will be presented along with their respective database tables and attributes.

4.2.1 Data model - Switch information

Switches and interfaces overall details are located in this part of the data model. The majority of this data does not change during the whole time the switch is operating. These are mainly device characteristics extracted via downcalls to the logical hardware layer.

However, some information regarding switches interfaces can be altered by internal events (*Oper-status*, *Enabled*, *Management-ip*. Those are values related to their state, and their update is triggered by upcalls providing from the logical hardware layer.

Both sets of values provide enough information for the controller to identify a device and its resources so that, paired up with neighborhood data, it can establish its routes. The tables that compose this part of the data model are named Switch and Interface respectively. A brief description and example of each attribute will be given, as well as the entity relationship model (figure 4.3):

- Switch:
  - **Switch-identifier (Primary Key)**: Value stored by the switch that uniquely identifies it in the data model.

    Example: 00f57cf4-fed9-4cba-917c-33aedf9dfffb.
  - **Name**: A textual description of the switch identifier. Mainly used for debugging.

    Example: switch1.
  - **Chassis-phys-address**: Know as MAC address, is a six-pair set of hexadecimal values that identifies the switch on the network.

    Example: 00:0c:29:77:6a:c4.
  - **Management-ip**: Ip address used for management purposes, separated from the IPs belonging to interfaces. Used to connect switches with the controller.

    Example: 192.168.34.162.
- Interface:
  - **Interface-identifier (Primary Key)**: Single identifier for an interface in the data model.

    Example: 2c8346b0-a532-46e2-a9f4-e3546d48cd0c.
  - **Name**: A textual description for the interface identifier. Mainly used for debugging.

    Example: e101-001-0.
  - **Oper-status**: Integer that represents the operational status of an interface.

    Example: 1.

- **Enabled**: A single digit that represents the state of the interface, where 1 represents UP and 0 represents DOWN.
  Example: 1.
- **Phys-address**: Know as MAC address, it is a six-pair set of hexadecimal values that identifies the interface on the network.
  Example: 00:0c:29:77:6b:29.
- **Speed**: Maximum speed achievable in communication by components using this interface, measured in bytes per second.
  Example: 40,000,000.
- **MTU**: Maximum transmission unit, defines the largest size of a packet that an interface can exchange without needing to split it, measured in bytes.
  Example: 1532.
- **Ip**: Allocated IP address of the interface. This IP is used to connect a switch with their neighbors.
  Example: 11.1.1.2.
- **Prefix-length**: Set of bits present in the IP address of the interface.
  Example: 24.
- **Switch-identifier-fk (Foreign Key)**: Identifier of the switch to which this interface belongs to.
  Example: 00f57cf4-fed9-4cba-917c-33aedf9dfffb.

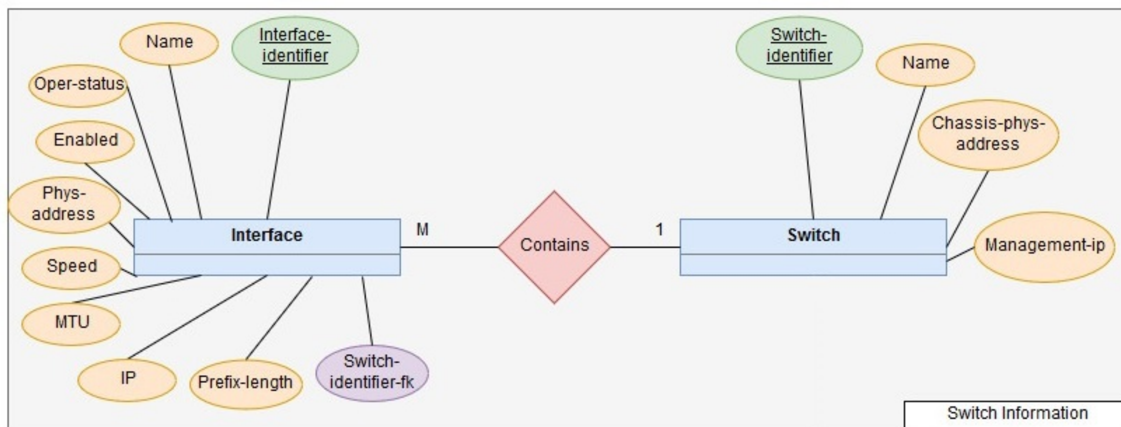

Figure 4.3: Entity-relationship model that incorporates all the data regarding switch characteristics.

4.2.2 Data model - Neighbors information

Neighbors data is a collection of information that is gathered by each switch after the neighborhood discovering procedure takes place. This data is updated by the LLDP process that constantly checks for neighbor presence.

Not all the available information switches acquire about each neighbor is stored in the switch local memory. This avoids replicating information to the controller that would be duplicated in its memory. The switch simply stores the identifiers of its neighbors so that the controller acknowledges the connection between both. A simple query sent to the neighbor using the identifiers can retrieve all the necessary information that the controller needs. These are the following tables, attributes and entity-relationship model (figure 4.4) regarding neighbors details:

- Neighbor:
  - **Chassys-phys-address (Primary Key)**: MAC address of the neighbor. Corresponds to Chassys-phys-address attribute in the table Switch.
    Example: 00:0c:29:77:6a:f7.
  - **Neighbor-identifier (Foreign Key)**: Unique identifier of the neighbor, which corresponds to Switch-identifier value in the table Switch.
    Example: 00f57cf4-fed9-4cba-917c-33aedf9dfffb.
- InterfaceNeighbor:
  - **Neighbor-interface-identifier (Primary Key)**: Single identifier of an interface of a neighbor. Corresponds to Interface-identifier attribute in the table Interface.
    Example: 54c88159-0289-4ceb-a13b-a93bbdf3bc89.
  - **Remote-interface-name**: A textual description for the neighbor interface identifier. Mainly used for debugging.
    Example: e101-002-0.
  - **Phys-Address (Foreign Key)**: MAC address of the neighbor, which corresponds to the Chassys-phys-address value in the table Interface.
    Example: 00:0c:29:ba:b2:74.

4.2.3 Data model - Routing information

The last part of the data model contains the routing rule set. This is similar to a standard routing table, also know as RIB (Routing Information Base), that stores all the information regarding available about routes for a switch. The switch, by itself, cannot define these rules. The controller is responsible for creating the routes and send them to the switches. These routes are constantly updated by the controller following network state changes, and the switch is responsible for receiving them and make downcalls to set up the logical hardware layer correctly.

The last table and of this model is called Ipv4-Rib, and is now presented along with its entity-relationship model (figure 4.5):

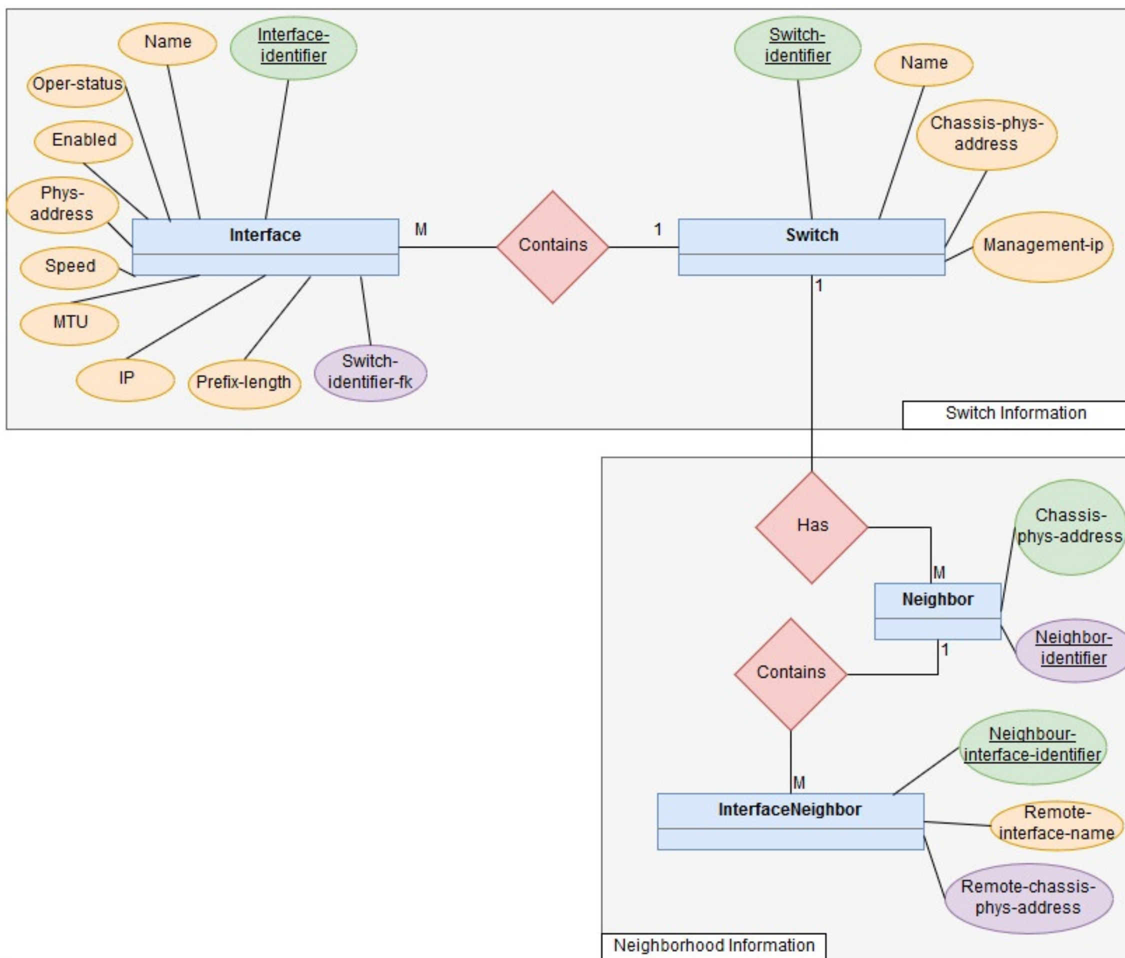

Figure 4.4: Entity-relationship model that incorporates all the data regarding switch characteristics, as well as neighborhood data.

- Ipv4-Rib
  - **Route-identifier (Primary Key)**: Single identifier of a route that uniquely identifies it in the data model.
    Example: d7c4a259-4c2c-48fe-abcc-301026861f3c.
  - **Route-prefix**: IP that represents the final destination of the route.
    Example: 11.1.1.0.
  - **Prefix-length**: Set of bits of the IP prefix.
    Example: 24.
  - **Next-hop**: IP address entry that identifies the next closest switch in the routing path for that route.
    Example: 10.1.1.2.
  - **Interface-identifier-fk (Foreign Key)**: Single identifier of the interface that makes use of this route. Corresponds to Interface-identifier value in the table Interface.

Example: 2c8346b0-a532-46e2-a9f4-e3546d48cd0c.

- **Switch-identifier-fk (Foreign Key)**: Identifier for the switch to which this route belongs to. Corresponds to Switch-identifier value in the table Switch. Example: 00f57cf4-fed9-4cba-917c-33aedf9dfffb.

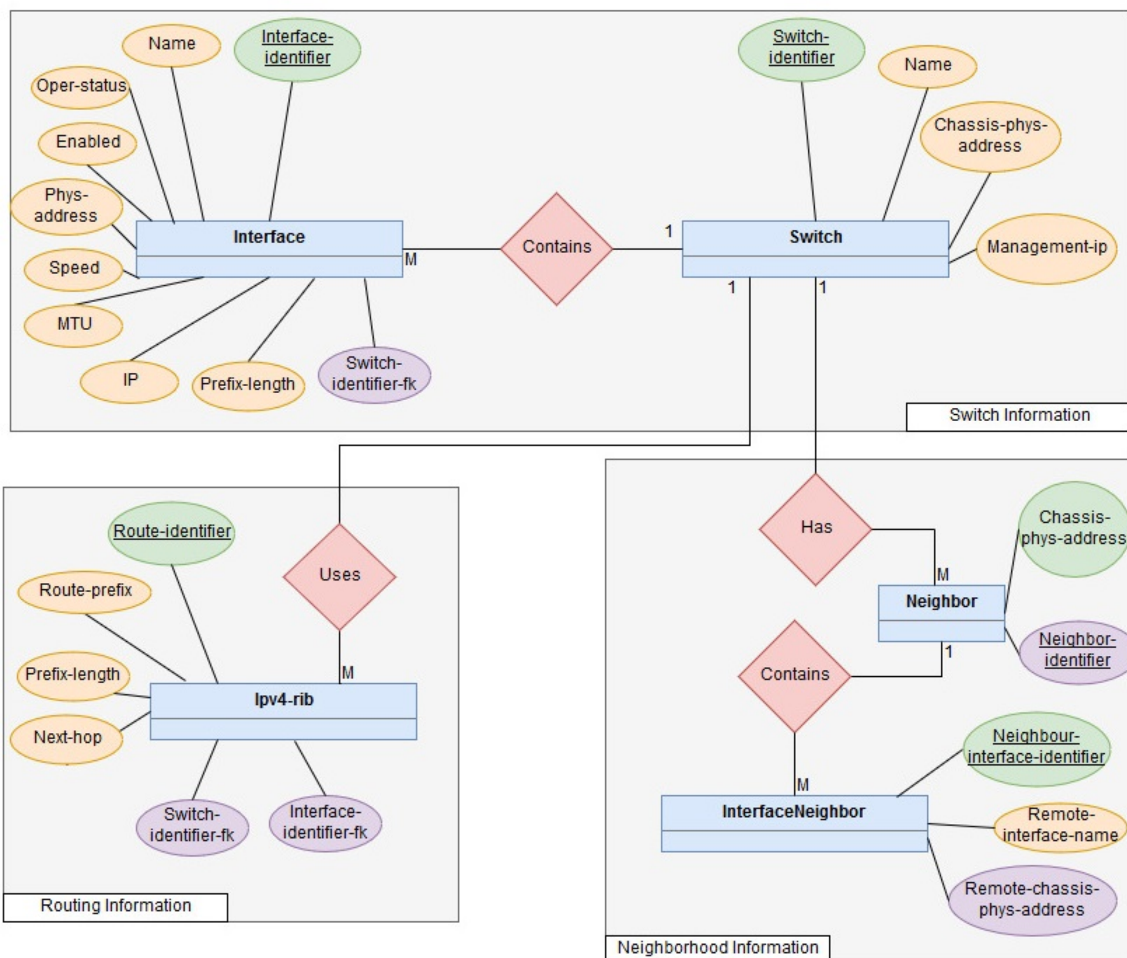

Figure 4.5: Entity-relationship model that incorporates all the data regarding switch characteristics, neighborhood data, and routing information.

4.3 Replication mechanism

The data model presented above was structured to be implemented in a SQL database relational schema. The chosen software for the implementation was MySQL[3], an open source relational database management system acquired by Oracle in 2010.

The database engine complied with every requirement, but even though it supported an embedded replication mechanism, that mechanism was not used. It enabled a clear separation between the replication mechanism and the database.

[3]https://dev.mysql.com

The software that had the responsibility of executing the replication process outside the realm of MySQL was SymmetricDS[4]. SymmetricDS is a data replication software, also open source, optimized for scalability and interaction between multiple database types with flexible configuration. It replicates data asynchronously across multiple nodes and it is dislodged from the database software system itself. Figure 4.6 show the architecture with this replication mechanism included.

In the old prototype, every node (switches and controller) had its own running SymmetricDS instance connected to the database. SymmetricDS creates additional tables outside the ones present in the data model that are necessary for its internal configuration. It requires configuration files, which store properties necessary to establish connections between the database and SymmetricDS of a switch and also the connection between switches and controller (connections between SymmetricDS instances).

The controller is the master of the routing tables. Switches are masters of every other table. SymmetricDS is responsible for installing triggers that detect database changes. After a local update occurs, SymmetricDS is alerted by the triggers and the data is replicated to the slave tables. The triggers and the replication methods are all defined internally in SymmetricDS in property files coupled with the external SymmetricDS ID's that identify each instance.

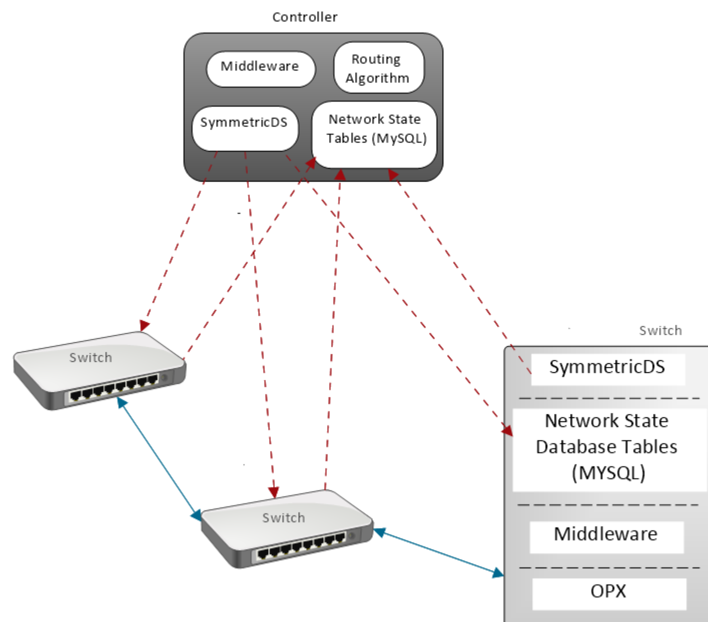

Figure 4.6: Old prototype architecture that includes the old database (MySQL) and replication mechanism (SymmetricDS).

[4]https://www.symmetricds.org/

Chapter 5

Network Replication with AntidoteDB

This chapter presents how and why the new solutions for database and database replication were introduced.

The first section (5.1) expands on section 3.4.4 by presenting features that the new database and its replication mechanism provide and how valuable they are for this prototype.

It is followed up by the introduction of the interface that was used in order not to change the data model in 5.2.

5.1 Introduction of AntidoteDB in the prototype

When looking into what the database could offer to this prototype, the most relevant features were:

- High performance. In every system, high performance is highly valuable. The overall performance of the prototype relies on the components being able to detect changes, store values and exchange them if necessary. It means that there is a lot of processing time that is spent executing database related operations (from reads/writes to replication of data). So, the prototype highly benefits from a well tuned database.
- Easily changeable data model and schema. Requirements in networks are constantly changing, as well as their devices specifications. Coupled with the constant emergence of new services, the prototype requires a database schema that is easy to alter, so it can be easily adapted to new requirements.
- Embedded replication. The prototype requires devices to exchange information between each other. If the database already incorporates a replication mechanism,

there is no necessity to rely on an external one to perform this job. This avoids extra complexity if there is no benefit in using it. Also, embedded replication mechanisms are usually more optimized since they are developed specifically with/for the database.

- Partial replication. Switches are only required to send information to the controller and no other device. Also, they only receive information that is specific to them since the controller is in charge of making global computations. By supporting a replication mechanism that only replicates data to certain nodes, under certain conditions, we make sure that there is the fewest number of updates possible that ensure that the network is properly functioning.

- Non-uniform replication. Replicating data that does not alter the network state at specific points in time instead of replicating it immediately improves the overall performance of the system while reducing the overhead created by the unnecessary replication of those values. It enables the reduction of the number of updates and improves the overall performance.

The database used for the initial prototype was a SQL database (MySQL) and its replication was handled by an external Java written software (SymmetricDS).

The inclusion of SymmetricDS in the prototype represents additional software that requires configuration files and code to set it up. It is expendable if it does not provide any benefit to this specific prototype over, for example, an embedded replication mechanism. Also, one of the main benefits of this software is cross-platform compatibility, which is not a concern of this project since all the devices work with the same database.

AntidoteDB was chosen as the new database and it supports or all the features mentioned above. It includes an embedded replication mechanism, facilitating the exclusion of SymmetricDS from the prototype and all the additional content it requires to work, removing the necessity of relying on an outside software component to deal with replication.

Since AntidoteDB is a NoSQL database, its usage would require the alteration of the data model from a table and row SQL style schema to a NoSQL one. However, we took advantage of an interface available for AntidoteDB called AQL[16] in order not to change the already established model presented in section 4.2. Figure 5.1 represents the architecture of the prototype after the replacements were made. AQL is detailed in the following section.

5.2 AQL

As it was mentioned before, AntidoteDB is a NoSQL database, which left us with two options regarding the data model used for the new prototype.

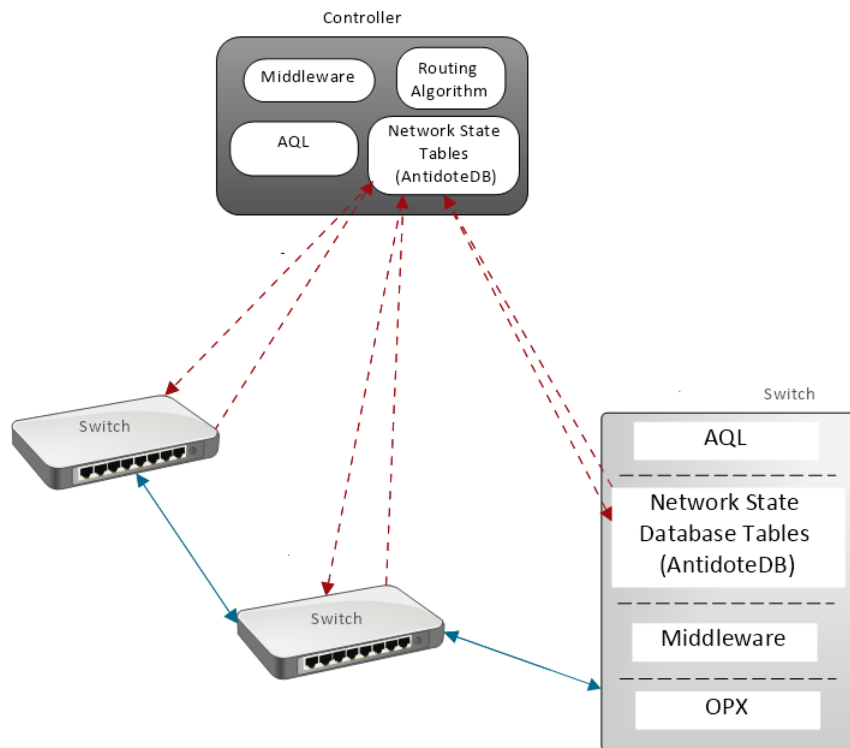

Figure 5.1: New prototype architecture with AntidoteDB and AQL.

We could change the data model to match AntidoteDB standards, which is far from similar to the SQL based one defined for the old prototype. This was undesirable since, first, it would require a lot of work and analysis on how to switch to a different model, while keeping the necessary integrity and structure of the data model.

Second, it would disable the option of having an exportable and generic model, since we would switch from the well-established SQL standard to a database (AntidoteDB) specific one. Coupled with this, we would have to change all the performed queries.

The other option was to use AQL, a SQL-based interface for the AntidoteDB data storage system. The inclusion of AQL enables the usage of SQL-like queries while taking advantage of the features provided by AntidoteDB. It also enabled the option of keeping the same data model. The queries did not follow the exact generalized SQL semantics but were close to the point where a few changes had to be done.

Some of the operations that were not possible without the usage of AQL, coupled with a few examples of queries are now presented:

- Operations regarding table management that allow creations and deletions. Listing 5.1 shows an example of one of this operations, used to create the table presented in **??**.

Listing 5.1: Example of AQL query used to create a new table in the database.

```
1 CREATE AW TABLE switch (identifier VARCHAR PRIMARY KEY,
2  name VARCHAR, physaddress VARCHAR,managementip VARCHAR).
```

- Record handling operations like insertions, deletions and updates. Listing 5.2 shows an example of one of this operations.

Listing 5.2: Example of AQL query used to insert values in a table.

```
1 INSERT INTO switch(identifier,name,physaddress,managementip) VALUES
2 ('00f57cf4-fed9-4cba-917c-33aedf9dfffb','switch1'
3 ,'00:0c:29:77:6a:c4','192.168.34.162')"
```

- Read operations with support for basic clauses like *WHERE*. Listing 5.3 shows an example of one of this operations.

Listing 5.3: Example of AQL used to retrieve values from a table.

```
1 SELECT * FROM switch WHERE physaddress = '00:0c:29:77:6a:c4'.
```

- Database assertions like primary keys, foreign keys, and numeric invariants.

Queries are parsed by AQL and translated to the AntidoteDB semantics before being sent to the respective local instance. The interface is launched in conjunction with AntidoteDB and does not require any additional configuration files or functions besides the installation process.

Chapter 6

Prototype Testing

This chapter presents the testing process of the prototype. Section 6.1 presents the testing environment which includes host machine specification, virtual machine configuration, and software versions. Also includes the architecture of the network topology used for the tests.

Details regarding the specific tests that were performed, including values that were measured, are presented in section 6.2. The results and the comparison between them and the ones obtained by the old prototype are discussed in 6.3 together with the observed differences.

The description of the scripts that help the deployment of the software in the devices, as well as its testing detailed in section 6.4.

6.1 New prototype (AntidoteDB and AQL) vs old prototype (MySQL and SymmetricDS)

The goal of the following tests was to compare the performance between the old and the new prototypes. It can be interpreted as a comparison on how AntidoteDB, its embedded replication, and AQL performed versus MySQL and SymmetricDS.

To carry out these tests, it was used as much as possible a similar setup to the one used to test the previous prototype. Figure 6.1 presents the topology used for testing both prototypes, that is composed of five switches and one controller. There are three switches with three links and two switches with four links, making a total of eleven links. The controller is omitted in the figure 6.1 to simplify its presentation, but it is connected to every switch.

Virtual machine configuration was also the same (table 6.1), where the number of network adapters depends on the type of the virtualized device.

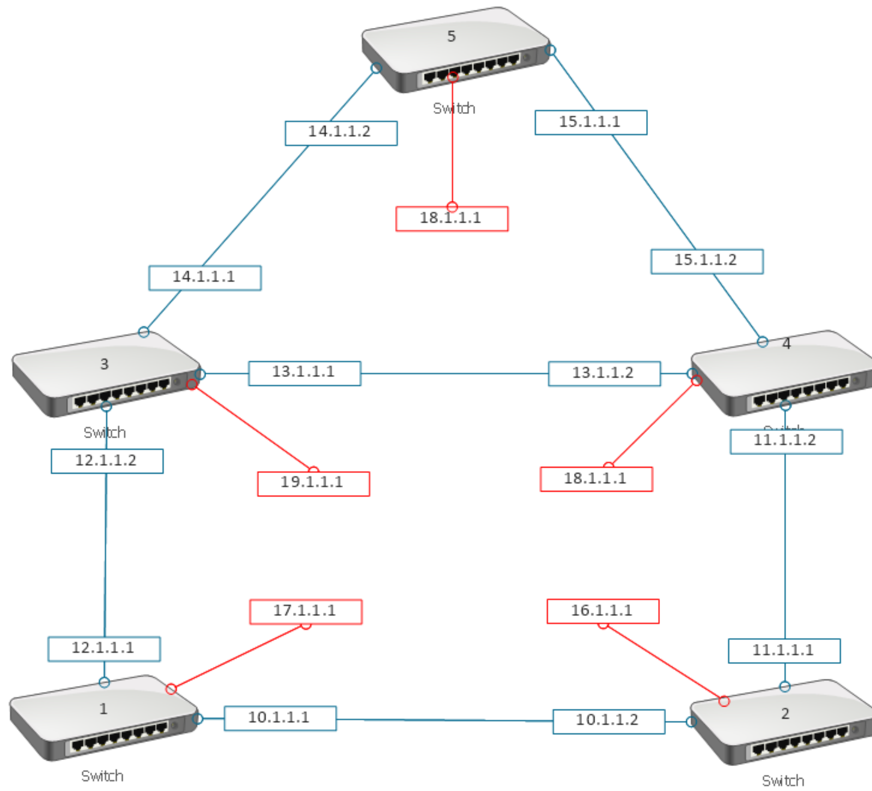

Figure 6.1: Testing topology used for the various tests.

Table 6.1: Available resources for each one of the virtual machines.

RAM: 1.5 GB
Total Processor Cores: 8
Hard Disk: 16 GB
NetworkAdapters: 1 (Controller),4 or 5 (Switch)

The switches had the same version for both OPX (where Debian is the base operating system) and Python. No versions of MySQL, Java, and SymmetricDS were required in the new implementation. Instead, Erlang, AntidoteDB, and AQL were installed. The controller contains the same software as the switches except for OPX since there was no necessity for its inclusion. The development of the controller was performed in the same Debian version that OPX was built on. The software versions are presented in the table 6.2.

Table 6.2: Versions of the software installed in the virtualized systems.

Debian 8.0
OPX 2.1.0
Python 2.7
Erlang 19.0
AntidoteDB (unstable)
AQL (unstable)

However, due to practical/availability reasons, tests could not be executed in a host machine with the same hardware components that the old prototype was tested on. Nevertheless, they were executed in an environment that simulated the same components and allocated resources. The hardware of the server used as the host for this prototype is presented in table 6.3.

Table 6.3: Hardware specifications of the server that was utilized as host machine for the new prototype.

Server: HP DL380 G9
RAM: 128 GB
CPU: 2 Intel Xeon E5-2670 v3
External Disk Array: HP MSA 2040 with 4 600GB disks

6.2 Tests performed

The first test relied on comparing how both prototypes reacted to changes in an interface state of a single switch, and measure the convergence time of the network due to the necessity to update routing tables.

The second consists in the deletion of the information of a neighbor from the database to force its detection, and the same global measurements done in the test presented above were made, since it also forces an update on the routing tables.

6.2.1 First Test: Interface state change

To execute the first test, a Python script was created to disable and then enable, in five-second cycles, an interface of a switch. These actions were only performed on interfaces responsible for one of the links present in the figure 6.1.

This script ran until all the interfaces of a switch were disabled and enabled once, providing results 3 or 4 times, depending on the number of enabled interfaces of that switch. It was run in every switch five times. Every interface change directly affected the network configuration, forcing the controller to recalculate it.

In this occurrence, the overall network convergence time (NCT1) can be broken down to the sum of the time that it took for each one of the following actions to finish:

$$NTC1 = KCPS + EVDB + SDBC + NRC + CDBS + DBOPX + OPXRIB$$

- **Kernel to CPS (KCPS)**: When an interface is disabled, the CPS monitor on the switch is responsible for detecting the event that is triggered at the kernel level. Measured from: the moment that the script disables the interface until the detection by the CPS monitor.

- **CPS to local database (EVDB)**: After the event is noticed by the switch, it is registered in the local database. Measured from: the moment where the detection of the event happens until the last insertion/modification in the database is made.
- **Switch database to controller database (SDBC)**: When a modification is made in the local database of a switch, it is replicated to the controller. Measured from: the last insertion made in the switch database until all the data is replicated to the controller's database.
- **New routing computation (NRC)**: This represents the time that it takes for the algorithm responsible for calculating the new network state to execute after all new information is present in the local database. It also accounts for the creation of the new network configuration and its storage in the controller's database. Measured from: the point where all the new data is present in the controller's database to where the new configuration is created and fully stored in the same database.
- **Controller Database to Switch Database (CDBS)**: It is the reverse process that *SDBC* measures. In this case, it is the replication of data from the controller database to every respective switch database. Measured from: the point where the data is fully stored on the controller to when it is fully stored on all switches.
- **Switch Database to OPX Services(DBOPX)**: After the switch receives all the new data, it performs downcalls, with the help of the CPS services, to implement the routing changes. These changes consist in the creation, update and/or deletion of routes. Measured from: the point where data is fully stored on the switch database to when the last downcall is processed.
- **Opx Services to RIB (OPXRIB)**: The last action to take place starts when the downcalls are executed to establish the changes on the routing tables at the kernel level. This value is not measured since there was no efficient and practical way to do it and is considered to be equal to the one obtained in the first variable *KERCPS*.

6.2.2 Second Test: Neighbor deletion and insertion

Like in the previous test, a Python script was created to execute it. In this case, the script disables the interface that connects the switch to a neighbor and deletes that neighbor's reference from the database of the switch. When the interface is enabled, it forces the LLDP monitor to discover the neighbor. It was performed for every neighbor of a switch, providing results 3 or 4 times, depending on the number of neighbors.

This test differs from the first, since just disabling the interface would not force the process of discovering a neighbor. Like the previous one, this test was performed in every switch five times and always forces the controller to calculate a new network state. Nevertheless, the NCT2 function is similar to the first one except for the first variable *KLLDP*, which is presented next.

$$NTC2 = KLLDP + EVDB + SDBC + NRC + CDBS + DBOPX + OPXRIB$$

- **Kernel to LLDP (KLLDP)**: After the interface of a neighbor is enabled, the LLDP monitor is responsible for detecting that event. Measured from: the moment that the script enables the neighbor's interface until the detection by the LLDP monitor.

6.3 Results and Discussion

The results obtained for each one of the tests will now be presented and discussed. Table 6.4 contains the average values obtained for the first test for each variable (action), in comparison with the ones obtained by the old prototype.

To perform this test, results were collected 85 times, which corresponds to disabling and enabling every interface of a switch 5 times for each switch. Results were measured in milliseconds and always rounded up. Table 6.5 contains the standard deviation for each variable.

Table 6.4: Average ($\overline{x}$) of the values obtained for the first test (interface state change) that directly compare the old prototype and the new prototype.

Variable	Old prototype average ($\overline{x}$)	New prototype average ($\overline{x}$)
KCPS (Kernel to CPS)	120	29
EVDB (CPS to Database)	21	12
SDBC (Switch DB to Controller DB)	70	34
NRC (New Routing Computation)	72	240
CDBS (Controller DB to Switch DB)	75	51
DBOPX (Switch DB to OPX Services)	67	36
OPXRIB (OPX Services to RIB)	120	22

Table 6.5: Standard deviation (σ) of the values obtained for the first test performed in the new prototype.

Variable	New prototype standard deviation (σ)
KCPS (Kernel to CPS)	5.44
EVDB (CPS to Database)	2.44
SDBC (Switch DB to Controller DB)	7.95
NRC (New Routing Computation)	13.23
CDBS (Controller DB to Switch DB)	8.75
DBOPX (Switch DB to OPX Services)	4.75
OPXRIB (OPX Services to RIB)	5.44

Conclusions regarding each individual action will now be given.

KCPS

The first obtained value, *KCPS* is significantly lower (around 75%) in comparison with the one calculated for the old prototype. The difference between both values is hard to

justify.

First, there was no alteration to the process that detects changes in interfaces state. The same mechanism of subscription to an event using the CPS API was used and the OPX version was also the same. The Python code written to deal with this event was similar since it did not involve any contact with the database itself up until this point. No extra functions were added or simplified.

Second, the testing topology did not suffer any change. The testing process was also similar since these tests were based on the ones that were performed in the previous prototype.

Results present in table 6.5 show that there is a low probability of the *KCPS* value increasing to a point where it matches the old prototype. A factor that could increase this value in the old prototype was the exhaustion of resources derived from the usage of multiple virtual machines that require a decent percentage of resources like RAM and CPU. However, the difference between values is so high when compared to the difference between others that is hard to justify just based on the exhaustion of resources.

EVDB

EVDB values obtained for both prototypes are small when compared to other obtained values, lowering the impact that this variable has in the whole convergence time. They are also almost similar to each other and since they are both small, discussing the difference between them does not translate into useful conclusions.

By looking at the prototype code structure it is possible to deduce why these values would always be small: the whole process is only composed by a single operation made on the database, which represents the update of the changed interface state.

SDBC

The replication time of the data providing from the switch was measured stored in this variable. The value obtained for the new prototype is more or less half of the value obtained for the old prototype. Both of them depend on the replication mechanisms used by each one (SymmetricDS and AntidoteDB embedded one) and their configurations and do not take into consideration code or other services.

SymmetricDS was configurated to operate periodically, which made the replication not immediate. AntidoteDB is set up to replicate data almost immediately which can be the difference maker between both values. Also, since it just corresponds to the replication of one value (interface state), it makes sense that replication does not consume a lot of time.

NRC

New routing computation (*NRC*) value in the new prototype is extremely high (240ms) when compared to the one in the old prototype (72ms). The explanation for the discrepancy between values relies on the usage of the AQL interface and the type of queries made to the database.

Most of the queries performed in this step, which are mainly read operations over database values (similar to the one in listing 5.3), do not use to the primary key to filter them, but other attributes (conditions inside the *WHERE* clause in *SELECT* statements that do not use the primary key for that specific table).

Performance decreases thanks to the way that this type of operations are executed by AQL, especially when compared to operations that only depend on the primary key. The loss in performance happens because AQL has to make a full scan of the table, which means reading the primary index of the table, and then every entrance of the table before checking it with the *WHERE* clause present in the statement.

In this step there are multiple iterations over several tables, making this process extremely time-consuming.

CDBS

The time it takes for the calculated route changes to be replicated by the controller is measured by *CDBS*. The conclusions presented for this value are equal to the ones presented for *SDBC*. Since the only factor that impacts this value is the replication time between devices, the new prototype value being lower derives from the replication time difference between SymmetricDS and AntidoteDB.

DBOPX

DBOPX is another value that is around half when compared to its counterpart. In the process measured by this variable, only the access to database values (a small number of *SELECT* operations) changed, derived from the databases changing from one prototype to the other. The rest of the code used to deal with the event of the appearance of a new routing configuration remained similar.

Also, taking into account the testing topology (6.1), after a singular interface state changes, a small number of routes are introduced, deleted or updated in each switch. So it is hard to attribute the difference in values to the performance of AntidoteDB over MySQL when executing operations since their number is small.

This makes the gap between both values hard to justify, which again could derive from the resource exhaustion when testing the old prototype.

OPXRIB

As was stated, the last value *OPXRIB*, was not measured, and its value was directly duplicated from the one obtained in *KCPS*, like in the old prototype. Even though this value could be obtained through certain methods (like constantly polling the routing table at the kernel level), it was decided that there was no precise and accessible mechanism to perform it.

Table 6.6 contains the results of the second test performed. Unfortunately, there were no individual values for this test in the old prototype, except for *NRC* and *KLLPD*, but only the sums of the first three and the last three variables. However, for the new prototype, all the individual values were obtained.

Table 6.6: Average ($\overline{x}$) of the values obtained for the second test (neighbor discovery) performed in the new prototype along with the available old prototype values.

Variable	Old prototype average ($\overline{x}$)	New prototype average ($\overline{x}$)
KLLDP + EVDB + SDBC	1145(932+213)	981(919+13+38)
NRC	68	251
CDBS + DBOPX + OPXRIB	361	108(52+36+20)

KLLDP

Since the obtained values for both prototypes are so similar, few conclusions can be drawn besides that they are both high when compared to any other results. Their high value is a direct consequence of the LLDP protocol and its configuration.

Since the average time for the detection of a new link in the old prototype was 932 milliseconds, the difference in the value that is composed by the sum of the three first variables derives from the ones that are database and replication related (*EVDB* and *SDBC*).

NRC value conclusions are similar to the ones presented for the first test since the number of operations performed by the algorithm is almost equal.

The sum of *CDBS*, *DBOPX*, and *OPXRIB* is around a third of the sum obtained in the old prototype. Taking into account the breakdown of the values obtained in the first test, it is easy to understand the difference. Since each individual obtained value for the new prototype was smaller than its counterpart in the old prototype, the sum of its values it's obviously smaller.

Overall, the biggest drawback of the new prototype is found in the algorithm that calculates the new routes, which highly increases the overall convergence time. However, this increment is traced back to the usage of AQL, and not to the database itself or the middleware. A possible future work could be adjusting the data model to better suit AQL queries in a way that lowers their execution time.

Processes that involve database operations (outside the one mentioned above) kept similar values or achieved slightly better ones in the new prototype. These results derive

from faster processing times that were expected since both databases are highly tuned to perform operations in a quick manner. Also, the number of operations per time span is usually not high enough to create queue times or flood the database itself.

The new replication mechanism proved to be an overall benefit to the prototype, lowering replication time. This was expected since, overall, embedded replication mechanisms tend to outperform external replication mechanisms, and especially one that was designed to operate with multiple databases (cross-platform) and not a specific one.

6.4 Deployment enhancements

To help with the deployment and usage of the prototype, three shell scripts were created. They were designed with two objectives in mind: first, to facilitate individuals that have to pick up the project by automatically setting up everything. Second, to simplify the testing process and speed up the re-deployment of the prototype. Since they have distinct functions, they will be presented separately.

6.4.1 Installation script

This script installs every necessary component for the devices to operate. The only argument it requires is a string (*Switch* or *Controller*), which enables the user to install the specific software for the respective devices. It is presented in the appendix A.1.

It first updates the system and installs Erlang, the language that developers used to write AntidoteDB, which is installed right after. Then the AQL interface is installed. The last part, besides cleaning up and removing the unnecessary content, places the files containing database configuration inside the respective directories so that the databases are loaded up with them included.

6.4.2 Bootstrap script

Although in a standard SDN approach switches must receive their configuration from the controller, in both prototypes, to speed up development, each switch initial configuration (interfaces and their local IP addresses) are set up manually.

This script hides the complexity of booting the devices and its components in addition to setting them up to connect with each other. Devices require a specific order for their components to be initialized and this script orders them correctly. There are two different variations of this script since the switch and the controller have different requirements.

The switch receives IP's as arguments, as many as links the user wants to set up. Each IP is assigned to an interface and to an interface only. The controller receives each switch IP that is part of the network. This IP is not one of those that were set up manually, but the one pre-assigned to the first ethernet or wireless interface.

The next sequence is similar for both the controller and the switch. The script starts an instance of AntidoteDB. Compiling and creating a AntidoteDB release is not always

necessary, but is mandatory when is initialized for the first time or when new functions are installed. After a short period of time starts AQL to make sure that AntidoteDB is up and running. Then, the switch sets up the interfaces with the received IP's, and the controller starts the script that connects all the devices. Both actions are followed up with table creation, which is also similar for both.

These scripts are presented respectively in appendix A.2 for the controller and appendix A.3 for the switch.

Chapter 7

Network Monitoring with AntidoteDB

This chapter describes how the replication features of AntidoteDB were used to improve the newly implemented network monitoring feature in the prototype.

The first section (7.1) of the chapter introduces the concept of network monitoring and why it is important. The necessary extensions performed on the data model to accommodate it are presented in section 7.2.

They are followed by section 7.3, that describes the newly created software functions added to both the controller and the switching devices. Section 7.4 includes an explanation of how AntidoteDB improves network monitoring efficiency in this prototype.

It ends with the description of the tests performed to verify its efficiency and a discussion about their results in section 7.5.

7.1 Network monitoring overview

Networks evolved to a point where they are composed by a large number of devices. The increase in numbers requires networks to fulfill new requirements while also dealing with constantly changing paths and loads. Tuning networks to achieve better results in multiple fields (performance, bandwidth, link utilization) has become a more important task.

Before changing the way a network operates, it is first needed to extract the necessary information to get to that conclusion. This requires monitoring devices and respective traffic.

Network monitoring and traffic analysis enable network management through the supply of valuable information. Either done by hand or via software, network management ensures that fewer network resources are spent, and that network performance is maximized.

There are multiple well documented network monitoring and traffic analyses techniques[8]. Some are available through software embedded in the devices, while others are external to them, which are generally more limited than the internal ones.

Monitoring in SDN networks is also a familiar concept. Different options have been described[23] and implemented, like OpenNetMon[1], which is an open source software implementation for OpenFlow networks. Concepts from one solution in particular called Short-Sighted Routing [17] were utilized as the basis for the created network monitoring mechanism in this prototype.

The introduction of this mechanism opens the possibility of the inclusion of a more efficient control software, that executes operations like route calculation and load distribution more efficiently, thanks to the provided information. The following section presents the extension to the data model performed to accommodate this feature.

7.2 Data model extension

To store information regarding network monitoring in the prototype, a new table was created, called *NetworkStatistics*.

Table attributes represent a small sample of the possible ones that are available[1]. They are retrievable by making downcalls similar to the ones that are already executed to obtain the presented values.

It was decided just to include a small number of attributes that could satisfy the proof of concept to simplify its presentation and discussion. In order to implement all the features described in [17], more statistics and functions would be needed.

The extension of the data model is presented in the entity-relationship model in figure 7.1. The newly created table is composed of the following attributes:

- NetworkStatistics:
  - **StatisticsId (Primary Key)**: Value that uniquely identifies a specific set of statistics.

    Example: 12.
  - **Octets-in-10**: Number of octets received in the last ten seconds, measured in bytes.

    Example: 678.
  - **Octets-out-10**: Number of octets sent in the last ten seconds, measured in bytes.

    Example: 340.
  - **Counter**: Integer that represents how many times statistics were successfully collected.

[1] https://github.com/open-switch/opx-base-model/blob/master/yang-models/dell-interface.yang

Example: 10.

- **Timestamp**: Timestamp corresponding to the last moment when statistics were collected.

  Example: 1335906993.

- **Ema-in**: Exponential moving average of received bytes.

  Example: 689.

- **Ema-out**: Exponential moving average of sent bytes.

  Example: 332.

- **Switch-identifier-fk (Foreign Key)**: Identifier of the switch where a specific set of statistics belong to.

  Example: 00f57cf4-fed9-4cba-917c-33aedf9dfffb.

- **Interface-identifier-fk (Foreign Key)**: Identifier of the interface of a switch where a specific set of statistics belong to.

  Example: 2c8346b0-a532-46e2-a9f4-e3546d48cd0c.

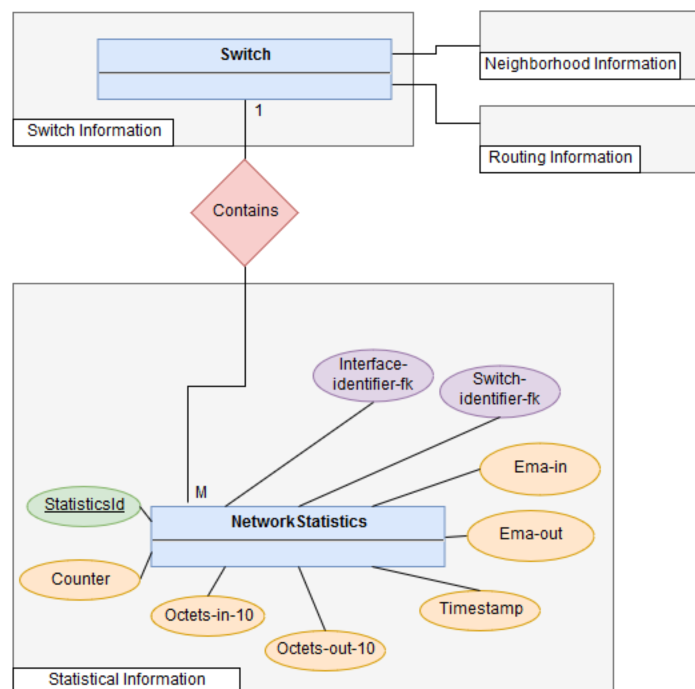

Figure 7.1: Entity-relationship model that includes the newly added statistics information (abstracted for simplicity purposes).

7.3 Conventional implementation

7.3.1 Switch

To implement network monitoring, a new monitor was added in every switching device called *NetworkStatisticsMonitor* (NSM). A monitor in this prototype is either a process that retrieves information via downcalls or a process that waits for upcalls and performs actions based on them.

In this case, NSM is responsible for making downcalls every 10 seconds which are executed with the help of the CPS API to retrieve performance values. The obtained values are stored in the table presented above. After their storage in the local database of the switch, they are replicated to the controller.

Timestamps can be retrieved directly by making a downcall and the counter is simply incremented every time the monitor ends all its operations successfully. The total of octets sent and received since an interface was enabled was also available via downcall like the timestamps.

However, some inconsistencies in the retrieved values were found, since in some occasions after certain periods of time (around 100 seconds) values were reset (set up to 0) even if the interface was still enabled. This disabled the option of storing in the database the values of total of octets sent and received to then later perform the calculations of other values like *Octets-in-10*, *Octets-out-10*, *Ema-in*, and *Ema-out*.

To circumvent this problem, after statistic values were collected, internal values were manually reset, so it would never reach a point where they would reset involuntarily. This also facilitate collecting *Octets-in-10* and *Octets-out-10*.

It was opted not to store every value of *Octets-in-10* and *Octets-out-10*, but only the ones regarding the last downcall, so that the database would not reach a point where it stored an enormous volume of data. Alternatively, we could have stored a limited number of the last read values, but ultimately decided not to since it was not necessary.

Ema-in and *Ema-out* are both exponential moving averages (EMA). An exponential moving average is an example of a moving average were the most recent collected data has a greater impact on the average. This type of average reacts more significantly to abrupt changes than a regular average or a simple moving average since it applies more weight to recently obtained values.

The start of the calculation of an EMA can be done in two different ways. One consists of reading and computing an average of a predefined number of collected results and use that value to seed the EMA calculation. The second uses the first obtained value as the seed and then calculates the EMA from that point onward.

We opted for the second option since the value of the EMA could be calculated right way and not wait for a certain number of reads, allowing the controller to make decisions immediately if needed.

Before calculating an exponential moving average, an intermediate calculation needs

to be performed:

$$K = \frac{2}{N+1}$$

N represents the time period for the moving average. The bigger the N, the slower the EMA reacts to changes. The shorter the N, the more weight is applied to the most recent value, and so the EMA value changes faster. We choose an N equal to 10 which is one of the standard values used when faster changes in EMA values are desired, making detection of network spikes faster. K is also known as the smoothing constant and represents the weight given to the most recent value collected, which in our case was 0.(18) or 18.18%.

The EMA formula is the following:

$$MR \times K + PV \times (1 - K)$$

Where, MR is the most recently obtained value and PV is the previous value of the EMA. The reason behind the calculation of EMA and its usage are presented in the next section.

7.3.2 Controller

In the controller, a monitor was added to process statistical values provided by the switching devices. It constantly checks for new statistic values in the local database that were replicated by those devices.

After new values are found, it verifies if the values for *Ema-in* or *Ema-out* exceed a threshold percentage of the maximum capacity of the link, which we considered to be around 75%. We choose to use *Ema-in* and *Ema-out* over *Octets-in-10* and *Octets-out-10* since *Octets-in-10* and *Octets-out-10* can be misleading. If there is a singular spike on one of the variables that surpasses the threshold for a short period of time, and soon after they return to acceptable numbers, would trigger a response from the controller regardless.

The trigger causes the controller to present a warning message and if algorithms were implemented, for example, to load balance in a multi-path environment, finding a new routing scheme that would minimize the maximum link utilization.

Small spikes over the threshold can be easily tolerated. Depending on the network configuration and state, recomputing it can be costly in performance and time, especially if it is performed every moment the threshold is exceeded.

By instead using *Ema-in* and *Ema-out*, especially with a medium to high value of K, we make sure that the values of *Octets-in-10* and *Octets-out-10* have been over the chosen threshold for an threatening period of time.

We designed the controller so that it verifies if either *Ema-in* or *Ema-out* exceeded the threshold of the maximum link utilization since in traffic engineering each direction in a communication using a point-to-point channel is a differentiated value in the traffic matrix. If either one of the variables surpasses the limit it triggers a response from the controller.

7.4 Implementation with CRDT use

Besides the one presented above, an implementation that also takes advantage of AntidoteDB features was designed. Since the controller is only responsible for receiving the updates and make decisions based on them, there were no changes to it besides not necessitating to check the limits of *Ema-in* and *Ema-out*.

A few modifications were then performed in the switching devices. AntidoteDB instances in the switches, thanks to CRDTs, became responsible for checking if *Ema-in* and *Ema-out* surpass the limit, removing it from the controlling devices. Even though controlling devices, following SDN guidelines, should be in charge of most of the calculations while switches should be stripped of them, we considered it beneficial for the prototype.

It enabled the option of not replicating updates to the controller that not surpass the thresholds, reducing the number of updates that are replicated. The trade-off of having lesser updates and make a simple comparison between two values is beneficial since replicating data that does not impact the network state is more costly than executing a single operation.

Operating this way enables the detachment of the replication decisions from the switches software to the database. This helps to create a clean implementation and separation of components while opening the door for any replication decisions an extension necessitates to be handled by the database, removing the necessity of creating extra unnecessary software or/and functions.

Also with the help of a specific CRDT, he made sure that after an arbitrary number of updates where none is replicated, one was sent to the controller even if its values did not exceed the limit. By doing this, the controller makes sure that the switch is still calculating local statistics and that is able to provide them. It serves as a checkpoint to conclude that monitoring is working properly.

Also, it can still check the validity of performed actions on the network state. For example, if the controller performs a reconfiguration of the routing scheme, it is able from time to time, to check if the values are the ones expected when reconfiguration was performed.

7.5 Testing process and discussion

To understand how the implementation with CRDTs impacted the prototype and its overall performance, as well as how it compares against the implementation without its use, two tests were performed.

7.5.1 First test: Number of updated replicated

The first test performed consisted in measuring the number of updates replicated to the controller by the conventional implementation and the one that takes advantage of CRDTs.

To execute this test (and the second one), the same topology (figure 6.1) described in section 6.1 was used. Switching devices were turned on for 600 seconds. In this test setting, each switch updates its *NetworkStatistics* table 3 or 4 times (1 for each enabled interface) every 10 seconds, making a total of 1020 updates in switches database.

For this test, values *Octets-in-10* and *Octets-out-10* were not obtained via downcalls, but from a random number generator, where results range from 0 to the maximum link utilization possible. The reason behind this resides on the fact that to create traffic capable of achieving 75% of the link utilization, we would have to spend an enormous quantity of resources.

We know that we can obtain *Octets-in-10* and *Octets-out-10* values via the downcall approach, and created this shortcut to simplify the testing process. We also made sure that values of *Ema-in* and *Ema-out* surpassed the limit around 25% of the time, while testing both implementations.

Table 7.1 presents the number of received updates by the controller for each implementation.

Table 7.1: Number of updates received by the controller with and without resorting to CRDTs.

Prototype	Number of Updates
Without CRDT usage	1020
With CRDT usage	415

Like it was expected, the number of updates decreases. Comparing both results, we can see that with CRDT usage we get lower than half of the number of updates. Lowering the number of updates, especially if they are not necessary, avoids consuming network and controller resources, like memory and processing power, freeing them up so they can be utilized in other functions.

7.5.2 Second test: Impact on convergence

This test measured the network convergence time of the prototype with statistics gathering (with CRDT use) and replication happening at the same time. The testing process was copied directly from the test presented in subsection 6.2.1.

It was chosen only to copy the first test and not both since the only overall variable that changes is the starting action, and from then on both tests achieve similar values for the same actions.

The factor that separates this test from the original one is that now the prototype supports network monitoring, and the convergence process was timed to occur near the same time a statistical update was being performed. It was created to conclude if the new feature has any type of impact on the convergence time of the network if both happen almost concurrently.

Table 7.2 presents the results obtained in 6.3 for the first test without network monitoring, and the obtained results for the same test with network monitoring.

Table 7.2: Average ($\overline{x}$) of the values obtained for the test that compares the prototype with and without monitoring.

Variable	Prototype w/ monitoring ($\overline{x}$)	Prototype w/o monitoring ($\overline{x}$)
KCPS (Kernel to CPS)	33	29
EVDB (CPS to Database)	10	12
SDBC (Switch DB to Controller DB)	29	34
NRC (New Routing Computation)	279	240
CDBS (Controller DB to Switch DB)	55	51
DBOPX (Switch DB to OPX Services)	31	36
OPXRIB (OPX Services to RIB)	21	22

By looking at the results presented in the table, it is possible to reiterate that the introduction of network monitoring has little to no impact on the overall process of convergence. Almost every new result stays inside the bonds of dispersion that can be calculated using table 6.5.

The main reasons for its irrelevancy when impacting the convergence time are mainly due to first, only making a small number of updates each time and second, updates being performed in different tables, which does not create concurrent access to tables. The second reason would not also be a problem since we are dealing with a database that is prepared to receive concurrent operations and even supports CRDTs to help to deal with them.

Chapter 8

Conclusions

The main goal of this dissertation was to replace and implement new components in an already working prototype of a SDN wide area network with a database approach. The components should not only improve the system but also introduce new features that were not available with the usage of the old database and respective replication software.

Throughout the execution of these changes, multiple obstacles were found. They handicapped what could have been a faster process that would have enabled an even better and quicker improvement of the prototype. Also, some unexpected adversities appeared while dealing with the chosen technologies. The most relevant cases were:

- Many efforts were spent trying to change from virtual machines to Docker containers[1]. The prototype had every component functioning in containers besides OpenSwitch OPX which was available as a container, not supporting all the necessary interfaces and functions. After several weeks, we had to drop this approach.

- AntidoteDB was still being developed at the same time this dissertation was being prepared. That had a significant impact on our progress since we were not involved directly in these developments. Additionally, the old prototype and AQL were also being developed at the same time by colleagues finishing their thesis.

- OpenSwitch OPX did not provide a clear interface to obtain the available values through their data models, neither proper documentation about its functions. This created a long process of trial and error to retrieve them since different values required different methods.

[1] https://www.docker.com/resources/what-container

This facts coupled with the difficulties in using container-based OPX means that the decision to use a prototype relying on OPX to interface the hardware should be revisited. There are other existing options which should be explored.

In what concerns the conclusions related to the main goals of this dissertation, the first one is that the substitution of the database represented an overall improvement of the prototype. Not only outperformed the original plus the replication mechanism but also helped with the introduction of network monitoring.

The increased time of network convergence did not derive from the database itself, but from the SQL interface that was used. Progress in this direction could be achieved in two different ways: the improvement of the AQL interface or by switching to other data model, which was already mentioned as undesirable.

The second one is that non-uniform replication is a powerful mechanism that improves performance, as it was seen in with network monitoring. Network monitoring is also an interesting addition that successfully opens doors for future network management techniques to be introduced.

The overall result of this work indicates that its possible to implement SDN based networks with higher levels of abstraction, without necessitating OpenFlow or similar low-level protocols. Databases and their models are well known concepts making this approach more accessible to people without deep knowledge in the SDN protocols field. Also, just a few number of functions (IPv4 network with shortest path routing computation) were implemented to create a testing environment where tests could be performed. The data models required to create and expose other functionalities, like security, still need to be studied and developed.

Taking this into account, it is hard to compare it directly with another optimized and fully implemented real SDN network, since they undergo extensive processes of planning and deployment. Also, some factors like latency or network overhead are not taken into account. However, like it was mentioned before, there is room for a lot of new features to be introduced and possible different testing environments to be explored.

8.1 Future Work

Although the prototype is in a working stage, extensions and different options could be introduced and explored:

- Introduce a more complete network management mechanism to take advantage of network monitoring.
- Introduce elements like latency that make the testing base more realistic.
- Exploit different routing algorithms.
- Change from a virtualized environment to one that uses physical devices.

- Implement other network services like security and recovery.

Bibliography

[1] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers. "OpenNetMon: Network monitoring in OpenFlow Software-Defined Networks." In: *NOMS*. IEEE, 2014, pp. 1–8. ISBN: 978-1-4799-0913-1. URL: http://dblp.uni-trier.de/db/conf/noms/noms2014.html#AdrichemDK14.

[2] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. "Cure: Strong semantics meets high availability and low latency." In: *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*. IEEE. 2016, pp. 405–414.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. "Hedera: Dynamic Flow Scheduling for Data Center Networks." In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*. NSDI'10. San Jose, California: USENIX Association, 2010, pp. 19–19. URL: http://dl.acm.org/citation.cfm?id=1855711.1855730.

[4] M. Bjorklund. *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*. RFC 6020. http://www.rfc-editor.org/rfc/rfc6020.txt. RFC Editor, 2010. URL: http://www.rfc-editor.org/rfc/rfc6020.txt.

[5] G. Cabrita and N. Preguiça. "Non-uniform Replication." In: *arXiv preprint arXiv:1711.07733* (2017).

[6] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. "Design and Implementation of a Routing Control Platform." In: *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 15–28. URL: http://dl.acm.org/citation.cfm?id=1251203.1251205.

[7] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. "Ethane: Taking Control of the Enterprise." In: *SIGCOMM Comput. Commun. Rev.* 37.4 (Aug. 2007), pp. 1–12. ISSN: 0146-4833. DOI: 10.1145/1282427.1282382. URL: http://doi.acm.org/10.1145/1282427.1282382.

[8] A. Cecil. "A Summary of Network Traffic Monitoring and Analysis Techniques." In: (Sept. 2018).

[9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s Hosted Data Serving Platform." In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1277–1288. ISSN: 2150-8097. DOI: 10.14778/1454159.1454167. URL: http://dx.doi.org/10.14778/1454159.1454167.

[10] B. Davie, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. Gude, A. Padmanabhan, T. Petty, K. Duda, and A. Chanda. "A Database Approach to SDN Control Plane Design." In: *SIGCOMM Comput. Commun. Rev.* 47.1 (Jan. 2017), pp. 15–26. ISSN: 0146-4833. DOI: 10.1145/3041027.3041030. URL: http://doi.acm.org/10.1145/3041027.3041030.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store." In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281. URL: http://doi.acm.org/10.1145/1323293.1294281.

[12] N. Feamster, J. Rexford, and E. Zegura. "The Road to SDN." In: *Queue* 11.12 (Dec. 2013), 20:20–20:40. ISSN: 1542-7730. DOI: 10.1145/2559899.2560327. URL: http://doi.acm.org/10.1145/2559899.2560327.

[13] S. Goel and R. Buyya. "Data replication strategies in wide-area distributed systems." In: *Enterprise service computing: from concept to deployment*. IGI Global, 2007, pp. 211–241.

[14] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. "B4: Experience with a Globally-deployed Software Defined Wan." In: *SIGCOMM Comput. Commun. Rev.* 43.4 (Aug. 2013), pp. 3–14. ISSN: 0146-4833. DOI: 10.1145/2534169.2486019. URL: http://doi.acm.org/10.1145/2534169.2486019.

[15] S. Krishnan, N. Montavont, E. Njedjou, S. Veerepalli, and A. Yegin. *Link-Layer Event Notifications for Detecting Network Attachments*. RFC 4957. RFC Editor, 2007.

[16] P. Lopes. "Antidote SQL: SQL for Weakly Consistent Databases." Master's thesis. Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa, 2018.

[17] J. L. Martins and N. Campos. "Short-sighted routing, or when less is more." In: *IEEE Communications Magazine* 54.10 (2016), pp. 82–88.

[18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. "OpenFlow: Enabling Innovation in Campus Networks." In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: http://doi.acm.org/10.1145/1355734.1355746.

[19] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. "PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric." In: *SIGCOMM Comput. Commun. Rev.* 39.4 (Aug. 2009), pp. 39–50. ISSN: 0146-4833. DOI: 10.1145/1594977.1592575. URL: http://doi.acm.org/10.1145/1594977.1592575.

[20] N. Pinto. "Database Based IP Network Routing." Master's thesis. Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa, 2018.

[21] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-free Replicated Data Types." In: *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Ed. by X. Défago, F. Petit, and V. Villain. Vol. 6976. Lecture Notes on Computer Science. Grenoble, France: Springer, Oct. 2011, pp. 386–400.

[22] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. "Consistency-based Service Level Agreements for Cloud Storage." In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 309–324. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522731. URL: http://doi.acm.org/10.1145/2517349.2522731.

[23] P. Tsai, C. Tsai, C. Hsu, and C. Yang. "Network Monitoring in Software-Defined Networking: A Review." In: *IEEE Systems Journal* (2018), pp. 1–12. ISSN: 1932-8184. DOI: 10.1109/JSYST.2018.2798060.

[24] J. Vasseur, A. Farrel, and G. Ash. *A Path Computation Element (PCE)-Based Architecture*. RFC 4655. Aug. 2006. DOI: 10.17487/RFC4655. URL: https://rfc-editor.org/rfc/rfc4655.txt.

[25] L. Yang, T. A. Anderson, R. Gopal, and R. Dantu. *Forwarding and Control Element Separation (ForCES) Framework*. RFC 3746. Apr. 2004. DOI: 10.17487/RFC3746. URL: https://rfc-editor.org/rfc/rfc3746.txt.

Appendix A — Script Files

Listing A.1: Bash script created to install all the necessary software components for both controlling and switching devices.

```
if [ $# -ne 1 ]
then
 echo "Invalid number of arguments. The number of arguments must be exactly 1"
 exit
fi
if [ $1 != "Switch" ] && [ $1  != "Controller" ]
then
 echo "Invalid name."
 exit
fi
sudo apt-get -y install build-essential
sudo apt-get -y update
sudo apt-get -y upgrade
wget erlang.org/download/otp_src_19.3.tar.gz
tar zfx otp_src_19.3.tar.gz
cd otp_src_19.3
sudo apt-get -y install libncurses-dev
sudo apt-get -y install g++
./configure
sudo make
sudo make install
cd ..
sudo apt-get -y update
sudo apt-get install -y screen
sudo apt-get install -y curl
git clone https://github.com/pedromslopes/antidote.git
cd antidote
if [ $1 = "Switch" ]; then
```

```
29  git checkout 4bdbcb9a92e696b618abdd4ff036b165e6831659
30 else
31  git checkout 25aaa7a5c306ad5ba534c5a6281794922fe6c6a4
32 fi
33 git config --global url.https://github.com/.insteadOf git://github.com/
34 cd ..
35 git clone https://github.com/pedromslopes/AQL.git
36 cd AQL
37 if [ $1 = "Switch" ]; then
38  git checkout dd16fe21d51dd979b228c7cfa3312e7f543a3fda
39 else
40  git checkout eb68cfa20c89187e2b26a0d47b99e46be0b978e9
41 fi
42 cd ..
43 sudo rm -rf otp_src_19.3.tar.gz
44 sudo rm -rf otp_src_19.3
45 sudo apt-get -y install python-pip
46 sudo pip install networkx
47 mv changetoinstall/antidote.erl antidote/src/
48 if [ $1 = "Switch" ]; then
49  mv changetoinstall/triggers.erl antidote/src/
50 else
51  mv changetoinstall2/triggers.erl antidote/src/
52 fi
53 mv changetoinstall/aqlparser.erl AQL/src/
54 mv changetoinstall/aql_http_handler.erl AQL/src/
55 rm -rf changetoinstall
56 rm -rf changetoinstall2
57 rm -rf OPXAQL
58 rm README.md
59 rm 80-dn-virt-intf.rules
60 if [ $1 = "Switch" ]; then
61  sudo rm -rf ControllerDBCP
62  rm createtablescontroller.py
63  sudo rm -rf StartControllerScript
64  mv StartSwitchScript/start.sh .
65  rm -rf StartSwitchScript
66  rm join_dcs_script.erl
67 else
68  sudo rm -rf SwitchDBCP
69  rm createtablesswitch.py
70  sudo rm -rf StartSwitchScript
71  mv StartControllerScript/start.sh .
72  rm -rf StartControllerScript
73  rm switchInterfaceStateChange.py
74  chmod +x join_dcs_script.erl
75 fi
76 chmod +x start.sh
77 chmod +x stop.sh
78 sudo apt-get install ntp -y
```

```
rm install.sh
```

Listing A.2: Bash script responsible for initializing every necessary software component for the controlling device to work.

```
export PB_IP=$(/sbin/ifconfig | grep -A 1 'eth0' |
tail -1 | cut -d ':' -f 2 | cut -d ' ' -f 1)
export IP=$PB_IP
export ANTIDOTE_NODENAME2=antidote@$IP
NODES=$ANTIDOTE_NODENAME2
echo "Starting Antidote"
cd antidote
rm -rf data.antidote@*
rm -rf data.nonode@*
rm -rf log
rm -rf log.nonode@*
rm -rf log.antidote@*
rm -rf _build/default/rel/
make compile > /dev/null
make rel > /dev/null
screen -S antidote -d -m bash -c
'_build/default/rel/antidote/bin/env console;exec sh'
echo "Starting AQL"
cd ..
cd AQL
screen -S AQL -d -m bash -c 'make shell;exec sh'
sleep 1
echo "Connecting Controller to Switch with IP:"
echo $PB_IP
for ip in "$@"
do
echo $ip
NODES+=" antidote@"$ip
done
cd ..
./join_dcs_script.erl $NODES > /dev/null
echo "Starting Table Creation"
sudo python createtablescontroller.py
echo "Starting Controller Initialization"
sudo python -m ControllerDBCP.EventHandler
```

Listing A.3: Bash script responsible for initializing every necessary software component for the switching devices to work.

```
export PB_IP=$(/sbin/ifconfig | grep -A 1 'eth0' |
tail -1 | cut -d ':' -f 2 | cut -d ' ' -f 1)
export IP=$PB_IP
export ANTIDOTE_NODENAME2=antidote@$IP
echo "Starting Antidote"
cd antidote
```

```
7  rm -rf data.antidote@*
8  rm -rf data.nonode@*
9  rm -rf log
10 rm -rf log.nonode@*
11 rm -rf log.antidote@*
12 rm -rf _build/default/rel/
13 make compile > /dev/null
14 make rel > /dev/null
15 screen -S antidote -d -m bash -c
16 '_build/default/rel/antidote/bin/env console;exec sh'
17 echo "Starting␣AQL"
18 cd ..
19 cd AQL
20 screen -S AQL -d -m bash -c 'make shell;exec sh'
21 #make shell
22 sleep 3
23 declare -i i=1
24 for value in "$@"
25 do
26 echo "Setting␣Interface␣e101-00${i}-0␣Up␣with␣IP␣=␣$value"
27 sudo ip addr add $value/24 dev e101-00${i}-0
28 sudo ip link set dev e101-00${i}-0 up
29 i+=1
30 done
31 sleep 2
32 echo "Starting␣Table␣Creation"
33 cd ..
34 sudo python createtablesswitch.py
35 echo Starting Switch Initialization
36 sudo python -m SwitchDBCP.EventHandler
```