Nelson Miguel Costa Pinto

Licenciado em Engenharia Informática

Database-Based IP Network Routing

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

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

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Junho, 2018

Database-Based IP Network Routing

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

I would like to express my gratitude to my advisor, José Legatheaux Martins, for his guidance, advice, support and dedication, and for all the knowledge that he transmitted to me throughout the preparation of this dissertation and in the course of my academic journey.

I would also like to thank my family and friends who have accompanied and supported me always.

Abstract

The Software-Defined Networking (SDN) approach has the goal of simplifying network management. SDN uses a logically centralized approach to enable simpler network programmability and simplify the network architecture. SDN is in general associated with the OpenFlow protocol, which standardizes communication between a controller and network devices. Alternatively, a database approach could be used to tackle data exchange between controller and network devices. This solution requires the installation of a database server inside each switch, and replicas of those local switches databases, in the controller. The database approach offers several potential advantages over OpenFlow such as higher level of abstraction, flexibility and the use of mature implementations of standardised database protocols to propagate information events and commands.

The purpose of this work is to apply a Database-Based Control Plane (DBCP) for SDN networks on a wide area environment. The objective is to implement a replacement of the control plane of a wide area network, currently achieved using a link-state protocol such as OSPF or IS-IS, by an SDN approach based on similar techniques as the ones used in [4].

We conducted an experiment, which we called IP-DBCP, that consisted of the definition of data models and the construction of an SDN network with database replication as the means of communication between one controller and multiple switches. To this end, a switch was developed, using OpenSwitch software as a logical hardware layer, that is capable of executing a MySQL database management system, load it with its characteristics and collect data related to its network neighbourhood. A controller was also developed that executes a MySQL database management system with the replicated databases of all switches. The controller uses those replicated databases to construct routing rules, using a shortest-path algorithm. Ultimately we tested the correct functioning of the solution and evaluated the convergence time by performing network state changes and compared the results with the ones found in traditional link state protocols.

Keywords: Software-Defined Networking, Databases, Wide area networks

Resumo

Uma abordagem *Software-Defined Networking* (SDN) tem como objetivo simplificar a gestão da rede. SDN utiliza uma abordagem logicamente centralizada de forma a permitir a programação da rede de forma mais simples e simplificar a sua arquitetura.

SDN é geralmente associado ao protocolo *OpenFlow*, que padroniza a comunicação entre o controlador e os dispositivos da rede. Em alternativa, uma abordagem de base de dados pode ser usada para lidar com a troca de dados entre um controlador e os dispositivos da rede. Esta solução necessita da instalação de um servidor de base de dados dentro de cada *switch* e réplicas dessas bases de dados locais, no controlador. A abordagem com bases de dados oferece diversas potenciais vantagens como um maior nível de abstração, flexibilidade e o uso de implementações maduras e padronizadas de protocolos de bases de dados para propagar eventos e comandos.

O propósito deste trabalho é de aplicar *Database-Based Control Plane* (DBCP) para redes SDN de grande âmbito. O objetivo é implementar um substituto de um *control plane* numa rede de computadores de grande âmbito, atualmente conseguido utilizando protocolos como o *OSPF* ou *IS-IS*, por uma abordagem SDN baseada em técnicas semelhantes às usadas em [4]. Realizámos uma experiência, que chamámos de IP-DBCP, que consistiu na definição dos modelos de dados e na construção de uma rede SDN com replicação de base de dados como o meio de comunicação entre um controlador e vários *switches*. Um *switch* foi desenvolvido, usando o *software* OpenSwitch como uma camada de hardware lógico, que é capaz de executar um sistema de gestão de base de dados MySQL, carregá-lo com suas características e recolher dados relacionados com sua vizinhança de rede. Um controlador também foi desenvolvido sendo que executa um sistema de gestão de base de dados MySQL com as bases de dados replicados de todos os *switches*. O controlador usa essas bases de dados replicadas para construir regras de roteamento, usando um algoritmo de caminho mais curto. Testámos o funcionamento correto da solução e avaliamos o tempo de convergência, realizando alterações no estado da rede e comparando os resultados com os encontrados em protocolos tradicionais de *link state*.

Palavras-chave: Software-Defined Networking, Bases de dados, Redes de computadores de grande âmbito

Contents

List of Figures

Chapter 1

Introduction

Management of traditional networks can be complex, time consuming and error prone. This is due to the fact that networks can consist of a large number of vendor-specific devices which are usually closed systems with limited interfaces. Often, policy enforcement and system management must be done directly on the infrastructure, device by device. Additionally, with traditional network protocols, the control plane is distributed and scattered by all devices, therefore, managers have difficulty achieving a coherent and logical view of the network.

The Software-Defined Networking (SDN) approach has the goal of simplifying network management [5]. SDN uses a logically centralized approach to enable simpler network management and simplify its control plane.

In an SDN controlled network, the control plane functions are concentrated in one or more control servers, known as SDN controllers. Switches have only flow tables (the data plane) and no other intelligence besides a control module, that fills the tables entries as ordered by the controller. The usage of SDN approaches are being widely tested and the success of its application is clear in a data centre, where hundreds of switches (with no control plane functionality) are controlled by logically centralized controllers.

Outside this natural setting, an SDN approach faces additional challenges related to the extra latency between controller and switches, heterogeneity of the system components and greater difficulty in setting up a logically centralized view at scale.

SDN is in general associated with the OpenFlow protocol, which standardizes communication between a controller and network devices in an SDN environment. An SDN architecture approach doesn't necessarily need to use OpenFlow, other solutions are possible to setup the dialogue between switches and controllers.

Alternatively, a database approach can be used to tackle data exchange between controllers and network devices as described in [4]. This solution requires the installation of a database server inside each switch, and replicas of those local switches' databases, in controllers. The controller drives the switches by making updates to its replica databases, and the switch transmits information or its state the other way around. Control algorithms executions are triggered by the switches' database updates. This database approach has several potential advantages over OpenFlow: higher level of abstraction; higher flexibility since device enhancements are translated into database models' modifications; low level custom communication protocols are replaced by standardised database protocols (query, updates and replication); well defined semantics in the dialogue between the different systems components (Eg. transactions can be used); the protocols, model and mechanisms are well understood and mature implementations are available. This kind of system architecture has yet to be tested in a wide area network because of its intrinsic challenges.

The purpose of this work is to test and assess a database approach to implement a replacement of the control plane of a wide area network, currently achieved using a link-state protocol such as OSPF or IS-IS, by an SDN approach based on similar techniques as the ones used in [4].

In a data centre, dialogue and state synchronization among switches and controllers is easier since latency is very low. In the wide area we will face an extra challenge related to higher and heterogeneous latencies, harder to implement controller availability requirements and higher scalability.

The main contributions of this work are:

- The definition of the data model required to implement the approach
- Study the implementation challenges
- Make a preliminary assessment of the performance challenges brought by this approach.
- Highlight its advantages and drawbacks.

This document is structured this way:

- Chapter 2 will introduce Software-Defined networking, by giving the motivations for its creation and explaining its main principles. It also introduces the main components and views of an SDN and after, design approaches are discussed such as controller policies, scalability concerns and SDN applications.
- Chapter 3 presents Database-Based Control Plane approach (DBCP). It describes the DBCP model as well the data models it requires.
- Chapter 4 describes the IP Network Database-based Control Plane (IP-DBCP) which is an experiment based on the DBCP approach. This chapter also describes the implementation of this prototype.

- Chapter 5 presents the tests and results measured with the IP-DBCP prototype.
- Finally, Chapter 6 presents the conclusions of this work and discusses future work.

Chapter 2

Related work

2.1 Introduction

Traditional computer networks can be characterized as a layered architecture composed of three planes: the data, control and management planes. The management plane may include software services which allow for functionality control, configuration and network policy definition. The control plane enforces those polices. It represents the protocols used to generate the network topology and data plane forwarding tables. The data plane represents the network devices and is responsible for implementing the policy by forwarding data accordingly.

In traditional networks, the data and control planes are coupled and embedded in the same devices, allowing them to take decisions on their own. This highly decentralized architecture contributed to the success of the Internet as it is today, assuring a fundamental requirement for operability: network resilience [11].

However, this comes at a cost of increased complexity: the tight coupling of data and control planes (vertically integrated) means that decisions about data flows are made onboard the device. The deployment of new functionality and policy enforcement must be done directly on the infrastructure which, together with the fact that there is a lack of a common interface to all devices and the presence of dynamic environments, requires a huge amount of time and can be error prone.

To cope with the lack of functionality, and the increased control requirements, specialized devices such as firewalls, middle boxes and intrusion detection systems may be inserted into the networks which may further increase network design and operation complexity [11][16].

Networks can be composed of a large number of different, vendor-specific, routers, switches and other devices, usually being closed systems and with limited interfaces.

The need for interoperability between different devices leads to the creation of specific protocols which may take years to develop and to evolve to new functionality [11][4].

2.2 Software-Defined Networking

Software-Defined Networking (SDN) is a proposal for a "programmable network", breaking the vertical integration of network devices by decoupling the control from the data planes, with the goal of simplifying network management and enabling innovation and evolution [11][16].

The Open Networking Foundation (ONF) defines SDN as follows: "Software-Defined Networking (SDN) is an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today's applications. This architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services." [5]. The SDN architecture is based on three principles [7]:

The decoupling of traffic from control: As described before, an SDN characteristic is the decoupling of the data and control planes which is a precondition for logically centralized network control. Network devices become simple forwarding elements. This decoupling also allows for separated evolution and software life cycles.

Logically centralized control: An SDN major characteristic is the logical centralization of control. As followed by the decoupling of the data and control planes, a single entity (which could be distributed) called an SDN controller or a Network Operating System (NOS), orchestrates resources and provides abstractions to facilitate the programming of forwarding devices.

Programmability of network services: This principal goal is to provide agility throughout information exchange between a client and a SDN controller allowing configurations before and during the lifetime of a service.

These principles are subject to interpretation and may lead to different implementations. As a traditional network, an SDN network can still be viewed as a layered architecture with three planes, the data, control and management planes, each layer with its specific functions as shown in Figure 4.2. A general description of each layer will be presented.

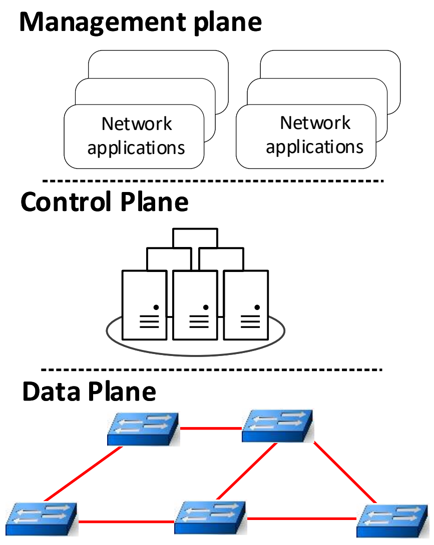

Figure 2.1: Plane oriented view of the SDN architecture

2.2.1 Data plane

The data plane, also considered a forwarding abstraction, is the first layer of an SDN network and is directly associated with the network infrastructure. It is represented by several interconnected forwarding devices, each containing a set of traffic forwarding and processing instructions. Its main role is to implement the forwarding decisions made by the control plane. A characteristic that differentiates it from traditional networks is the removal of intelligence from the devices. They become simple forwarding devices and must receive instructions from the controller plane through an open interface called the southbound interface. The SDN data plane can be described by two layers: Network infrastructures and Southbound interface [11][6][7].

Network infrastructure

Network infrastructure is the composition of interconnected hardware and software forwarding devices such as routers, switches and virtual switches. The centralization of intelligence on the controller transforms these network devices in simple packet forwarding entities that must receive instruction from the controller on how to operate. Network devices may still contain a minimal set of management and control functions allowing the necessary configurations to establish communication between devices and the controller [11][6].

Southbound interface

The southbound interfaces are a crucial element of a clear separation from controller and data planes as it bridges communication between both. By means of an API, it standardizes communications promoting interoperability between devices and the controllers.

There are several approaches such as OpenFlow, ForCes and database. [11][16][7][4].

We will only refer the first one because it is the most known and the last because it plays a crucial role in this work.

Openflow

OpenFlow, managed by the Open Networking Foundation (ONF), is a communication protocol which standardizes information exchange between controllers and data plane devices. OpenFlow requires a compatible switch or router which may exist as two variations: a pure or a hybrid device. A pure device has no intelligence outside the boundaries of OpenFlow, a hybrid has both support for OpenFlow protocol but maintains other traditional protocols allowing it to function outside an SDN environment.

The OpenFlow implementation has two abstractions: the controller communication and the flow tables. The flow tables consist of flow entries defining how packets from a flow should be forwarded in a switch and are composed of three parts: 1) Matching rules; 2) Actions for matching packets; 3) Statistical counters. Using the arrival of a packet to a switch as an example, the header fields are extracted and compared to the matching rules. If the result is a match, the appropriate actions for that match are taken for the packet. If the lookup process returns a miss, which could mean the switch is facing a new flow, it should use a default rule which is usually set to send the packet to the controller to be processed. The communication between the controller and data plane are made using the OpenFlow protocol which sets a number of messages, understood by both parties. [16][6][7].

Database approach

Alternatively, OpenFlow can be replaced by a replicated and distributed database approach as described in [4]. The principal characteristic of this alternative is to install a database server inside each switch and maintain replicas of those databases in the controller. Database techniques and protocols are then used to update both switches' and controllers' databases. A controller may force new rules on to a switch by making updates in its corresponding database replica. A switch may also transmit information or its state the other way around. Control algorithms executions are triggered by the switches' database updates.

This database approach has several potential advantages over OpenFlow [4]:

- The use of standard database synchronization protocols, abstracts the details of data transfer between entities and conceals the low-level protocol details from the developers, who use an higher level of abstraction;

- This approach increases flexibility because there is no need to evolve the communication protocol to support new features which can be made directly on the database schema;

- Well defined semantics in the dialogue between the different systems components (e.g. transactions can be used...);

- Low level custom communication protocols are replaced by standardised database protocols (query, updates and replication);

- The protocols, model and mechanisms are well understood and mature implementations are available.

- Low level details of packet processing are left to the switch control software, which increases separation of concerns.

This approach has only been tried in a data centre environment, which means there is an incomplete understanding of how this may behave in a different environment such as a wide area network. It needs to be taken into account that every switch on the network must maintain a database in order to be able to communicate and maintain configurations.

2.2.2 Control Plane

The control plane, considered the brain of the network, implements what is also known as the Network Operating System (NOS), solves the networking problems, provides services and creates abstractions to lower-level interconnecting devices. It facilitates network management by means of a logically centralized control and programmatic interfaces to the network.

The controller must understand the network topology under its domain and take decisions accordingly, based on policies defined by the management plane. Example of those are: through which switches and links should a packet be forward or is the packet authorized to go to the requested destination.

Controllers don't require special hardware equipment as they can be deployed on traditional servers and platforms using traditional software. They have three communication layers: the southbound interface, already described before, which standardizes communication between the controller and data planes and the Northbound and Eastbound/Westbound interfaces [11][16][6].

2.2.2.1 Northbound

Northbound communications allow for communication between the controller plane and the management plane, exposing interfaces through which the controller may receive instructions. It creates an abstraction from concrete controller implementations promoting applications portability and interoperability [11][16].

2.2.2.2 Eastbound/Westbound

Some SDN implementations may require the deployment of multiple distributed controllers across the network. The Eastbound/westbound is a horizontal layer of communication providing the means for inter-controller communication. Such layer may be used for controller state synchronization and may vary depending on the implementation. There are no standards for Northbound, Eastbound and Westbound communications [11].

2.2.3 Management plane

The management plane uses the abstractions provided by the controller plane and is composed by services and applications. An application can be perceived as instructions that controls a set of resources provided by the controller [11][7].

2.3 Design approaches and challenges

As part of the description of the SDN architecture, there are some architectural design aspects that should be considered.

2.3.1 Reactive vs Proactive Controller Policies

When a network device receives a packet and gets a miss from the lookup process on the flow tables, a decision has to be made by the controller about that packet. The point is to forward that packet through the network until it reaches its destination or it leaves the network on the controller domain. In this scenario, the controller may adopt one of two policies: Reactive or Proactive. With a reactive policy, the controller installs a new flow entry on the network devices allowing the packets on that flow to be forward. With each new flow, setup time has to be considered and can be lengthened by geographically dispersed remote controllers. Also, with this policy, the controller becomes a bottleneck because the network work rate depends on the controller response performance. The proactive policy tries to anticipate the arrival of new flows and pre-installs multiple generic rules on all devices. Although this may cause a loss of precision, this approach may significantly reduce the number of new flow requests to the controller [16].

2.3.2 Scalability concerns

An important aspect of an SDN is how design choices cope with scalability. The most obvious is that the control plane is now the centre of the control. With the increasing load on, and complexity of the network, a single controller could rapidly become overloaded by network device requests and stall the network. One solution may reside in scaling-in the controller, increasing the hardware power. Another is to reduce the number of requests to the controller by implementing proactive policies with the cost of loss of precision and flexibility.

A distributed set of controllers can be seen as an alternative to mitigate this problem. Although it might seem beneficial, those controllers now require synchronization to allow the same view of the network. Consistency algorithms imposes design trade-offs. As the network grows, the use of strong consistency may lead to the reduction of throughput and increased response time. Alternatively, one could use weak consistency with the cost of possible loss of precision during synchronization periods. On flow based architectures, the initial flow setup delay may impose some scalability concerns. The constant appearance of new short lived flows, plus the use of a reactive policy by the controller, could mean a constant network device-to-controller communication overhead. Also, the memory available inside the switches is limited which raises scalability concerns with the increase of flow entries [16][25].

2.3.3 SDN applications

Data centres are an agglomerate of a large number of servers and network devices, usually inside a large building, where latency is small. Most data centres are shared among several different customers and constitute a multi-tenant environment.

Data centres have to cope with constantly changing environments, requirements and the need to enforce multiple network policies. Data centres also face challenges such as the need for optimization of resources, definition of QoS, power saving, provisioning, security and flexibility.

The low latency environment in a data centre provides a realistic environment for the deployment of a Software-defined Network. One of the pillars of an SDN is logically centralized control. In a data centre, a controller or cluster of controllers, could be installed in the network connected to all network devices. This allows the controller to obtain a global view of the network topology. Such an approach enables the programmability of policies and configurations at a single point, avoiding the application of those policies directly to the infrastructure, which could be expensive and error prone due to the large number of different devices. With an SDN approach, network devices would become simple forwarding entities orchestrated by the controller. The controller may push new flow entries on to devices to deal with incoming traffic through proactive or reactive policies. The global view of the controller enables it to optimize resources when needed, such as power saving.

SDN might also be implemented on other environments such as enterprise networks. Those usually have multiple devices connected to the network, some not controlled by the company, such as mobile devices. An SDN implementation might help to simplify the network design by removing devices such as middle-boxes and firewalls, centralizing resources on the controller, allowing a centralized management and policy enforcement [16].

2.3.3.1 SDN in the wide area environment

Wide area networks interconnect multiple local area networks and are characterized by its geographical span. Therefore, a characteristic of this kind of network is the increased latency of communication between network devices. Also, due to its role, it has a requirement of a high availability since down time could be costly. Wide area networks must also be able to scale to withstand increased demands from its edges.

An SDN approach could be used in the wide area, just like other environments such as data centres, to enable simpler network management and simplifying its control plane. In a data centre, dialogue and state synchronization among switches and controllers is easer since latency is very low. In the wide area we will face extra challenges related to the higher and heterogeneous latencies. The switch to controller communication has increased latency, consequence of it geographical position, which has to be considered. Also it is harder to implement controller availability and higher scalability requirements.

Moreover, in order to deal with the challenges and avoid a central point of failure, the logicically centralized control of the network requires a hierarchical and recursive view of the control plane [13].

A production SDN for the wide area environment, was designed by Google and described in [10]. This work shows an SDN deployment for a wide area network (WAN) that connects Google's data centres across the planet. Consisted on an hybrid approach with support for existing routing protocols and OpenFlow.

2.4 Database approach versus OpenFlow

A database approach for controller and switch communication was considered in the work [4]. This work discussed the construction of an SDN environment for data centres context, that achieved interoperability of different vendor network hardware devices. The work focussed on the controller and switch communication.

For the design of the architecture, they considered that the OpenFlow protocol wasn't sufficient enough to answer their requisitions and the need for interoperability. They stated that because OpenFlow is a low level protocol, only devices that support it, can be inserted on the system. Also, different vendor switches require special OpenFlow protocol adaptation in order to integrally make available all the features supported by switches' ASICs to the controller. As this process of adaptation becomes complicated, the need for a higher level of abstraction arises. OpenFlow was also considered not flexible enough to freely evolve to new features.

The alternate solution proposed was to achieve interoperability on southbound communication by treating the problem as a generic database synchronization problem using databases replication protocols. According to the authors, this solution avoids the creation of a specific protocol, by using already implemented database protocols, and also separates the replication mechanisms from the network services. This approach abstracted the implementation details of data synchronizing from the network operations and presented a clear state consistency model that could be used by the whole system.

With this database approach, the challenge resided on constructing a data model that could be generic enough to be supported by all vendors and still allow the implementation of their specific hardware optimizations.

The flow definition processes was left to the switches themselves. The controller computes the control plane details and shares them with the switches using the Open vSwitch Database (OVSDB)[21] protocol. By doing so, they considered an higher level of abstraction.

A reference is made to a constraint of the database approach. Although new features could be expressed as a change to the data model, the controller and switch still require new code to consolidate the new capabilities. Also there is the case where different switches could be running different data model versions, which requires further consideration.

2.5 Open Shortest Path First

Open Shortest Path First (OSPF) is a routing protocol for IP networks, based on link-state technologies, that distributes network state information between routers belonging to the same Autonomous System (AS) [14].

The OSPF protocol produces, in a decentralized way, the routing rules that routers use to forward IP packets. Each router running OSPF contains a global view of the network and produce its own routing rules using a shortest path algorithm.

The global view of the network is built by the combined effort of every router. Routers use the Hello Protocol to periodically send announcement packets to it's neighbours through all router's interfaces. The Hello protocol establishes and maintains neighbour relationships between routers and probes bidirectional communication between neighbours. This way neighbours may form an adjacency. It also allows the detection of router/link failures.

Each router contains a database, referred to as the Link-State Database, that describes the autonomous system topology. This database is composed by the router's local hardware state and it's neighbour relationships. The database is complemented with link-state advertisements (LSAs), produced by every router, consisting of each router's local hardware state and it's neighbour relationships and thus fully describing the network.

The link-state database of every router is kept updated by flooding reliable updates. When a change occurs in a router's sate, it begins a flooding process to propagate it to all routers in the same AS. Every router's link-state database, after the synchronizing period, is identical. It is important that the link-state database is kept updated to ensure a common vision on the network topology.

Based on the link-state database each router creates a shortest path tree with it self as root. This tree gives a path to every other reachable router and their locally associated IP prefixes and defines the best next-hop. When a topology change occurs, the changes are flooded and every router computes new shortest path trees based on the newly updated link-state database.

An Autonomous System can be dived in special groups called areas. These areas were conceived to break down the scale of the AS into multiple groups, reducing the convergence time and scale of the traditional routing protocols and creating isolation between different regions. Areas may be composed by multiple networks interconnected by routers. At the edge of these areas, special routers called "Area Border Routers" work on forwarding traffic outside the area when needed. To maintain multiple areas in a single AS, a backbone area exists composed by multiple interconnected routers that communicate with the areas border routers. This backbone is called the area 0. Each area runs an independent version of OSPF assuring the convergence of routers inside the area. The backbone also runs an independent version of the algorithm.

There is another link-state protocol, on which OSPF got inspiration, the IS-IS protocol, which is popular among ISPs.

Although the notion of areas makes part of both protocol definition, most large scale ISPs, only use one area since the introduction of areas introduces constraints and extra complexity on the network management processes.

2.6 Conclusion

In this chapter we introduced the main concepts behind Software-Defined Network as well as the most complex issues brought by this approach. We also analysed and compared two southbound protocols, namely OpenFlow and database-based approaches.

We also showed that some IGP protocols, namely OSPF and IS-IS, also rely on centralized visions, replicated in each router, to create its own forwarding rules. In order to extend the information flooded, or to change the way forwarding rules are computed, it would require a new version of the protocol and its deployment in every router of each vendor.

In the next chapter we will address how this goal can be reached using a southbound approach based on a databases.

CHAPTER 3

Database-Based Control Plane

In this chapter we start by introducing the Database-Based Control Plane (DBCP) approach and compare it with other southbound protocols such as OpenFlow. We present the reasons for the construction of DBCP and discuss its' components and models. We also present a particular experiment based on the implementation of this model, the IP Network Database-Based Control Plane (IP-DBCP). Finally, the chapter defines the conceptual data model and logical data model that we introduced in the IP-DBCP.

3.1 Database-Based Control Plane

SDN was introduced with the promise of simplifying network management. To achieve this goal, it centralizes the control and network management on a central controller. This is in contrast with the traditional approaches where network control is decentralized and scattered across all switches. The communication between controller and switch in SDN is an essential part of the network operation. It's through this communication layer, also called southbound communication, that controllers and switches exchange data required for the network operation. The importance of this communication layer raised the need for definition of new protocols in response to this requirement.

One popular southbound protocol is OpenFlow. For SDN networks, it strictly defines the information exchange between controller and switch. OpenFlow changes the way switches work by stripping it off any intelligence and requires the controller to define all the operations a switch can execute. OpenFlow switches work on per flow level by implementing low level flow processing tables. Switch and controller dialogue is bidirectional and achieved using OpenFlow semantics. Those semantics define low level messages, supported by both switches and controllers, and consist of network packets, flow details, configurations and statistics. Dialogue is essentially composed of downcalls from the controller to the switch and upcalls from the switch to the controller.

The controller implements a distributed flow management service that defines the appropriate flow entries for each switch it controls. These flows entries are constructed based on the controllers global view of the network topology. The network topology discovery and setup is fully enforced by the controller. The controller may also collect statistics about the functioning of the network. Figure 3.1 shows an example of an OpenFlow network.

OpenFlow is widely used but still presents some limitations. As the work in [4] states, the use of low level semantics for the definitions of data exchange makes OpenFlow inflexible, over-specified in low level details and hard to evolve, not allowing for a clear separation of concerns and abstractions.

As a consequence, an OpenFlow switch is completely dependent on a controller to leverage any hardware optimization. Moreover, OpenFlow lacks a synchronization model when involving multiple switches. There is a lack of specification that guarantees the ordering and consistency of the data exchange operations across multiples switches.

In this work, we attempt to test a different approach to controller and switch communication by dealing with some of the OpenFlow challenges, using databases concepts inspired in [4]. In this work we also consider this database approach on a wide area (WAN) environment as opposed to the work presented in [4] that is applied inside a data centre.

To assess both challenges, we started with the definition of Database-Based Control Plane (DBCP), to replace a traditional shortest-path IP routing algorithm, with a centralized approach using SDN and database techniques.

The DBCP approach aims to elevate the level of abstraction by using databases schemas, models and replication protocols as the way of sharing data among controllers and switches. The use of standard database techniques, abstracts the details of data transfer between entities and conceals the low-level protocol details from the developers, who use an higher level of abstraction. This approach increases flexibility because there is no need to evolve the communication protocol to support new features, which can be now made directly available on the database schema. Also, database replication protocols models and mechanisms are well understood and mature implementations are available. Those protocols offer a well defined distributed semantics.

DBCP is an approach to change the way switches work and how data is shared among controllers and switches on an SDN environment. The ultimate goal of this experiment is to replace the control plane of switches in a wide area network, currently achieved using link state protocols such as OSPF or IS-IS, by an SDN approach that uses SQL data models and databases and is based on techniques similar to the ones used in [4]. In more detail, this approach aims to insert a database management system on each switch and controller, which will be used to orchestrate them on a wide area network. The objective is to use those databases and related protocols to synchronize state and configuration updates among network devices over the network. Figure 3.2 shows a DBCP network.

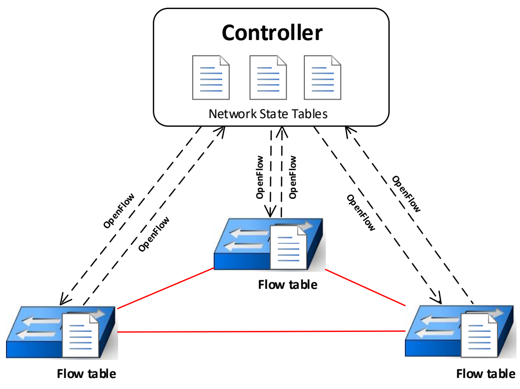

Figure 3.1: OpenFlow network.

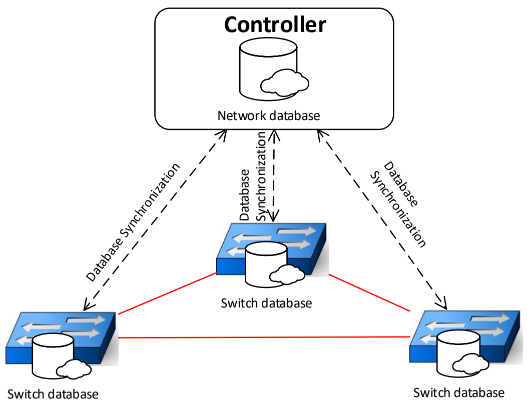

Figure 3.2: DBCP network.

In order to demonstrate that the DBCP approach is realistic in the wide area network, it is necessary to define the adequate data models to build the abstractions required for the necessary network control goals. However, it is also necessary to demonstrate that the database protocols used to replicate data among the different components are able to converge in a timely manner at least as good as, if not better, than the ones implemented by traditional purpose-built network protocols. On top of the paradigm change, the DBCP architecture must still be designed to support essential WAN requirements such as packet routing and network management.

To explain in more detail and to give a more general perspective about how DBCP is characterized, DBCP is composed of switches and controllers. Switches communicate between themselves using links connected by their interfaces, and also communicate with the controllers through its local database. State and configurations changes are converted to database operations locally, which are then synchronized with the controllers. Switches maintain a local view of their directly connected neighbours and forward packets based on decisions received from the controller. Controllers are independent entities that use information from their local database to produce routing decisions and enforce configurations.

An SDN characteristic is the removal of intelligence from the switches. OpenFlow completely removes the intelligence from switches. As opposed to OpenFlow, in DBCP this removal is considered only partial. Switches lose the capability of reacting to configuration changes and knowledge of a global view of the network. On the other hand, switches still maintain a local vision of the neighbourhood, so the topology gathering process in DBCP is considered distributed on the switches as opposed to being centralized in OpenFlow, since this approach is still essentially a local one.

The switches outsource the capabilities of producing routing rules, to the controller. The controller produces routing data based on the network topology and propagates it to the switches.

This work focuses on a first assessment of the DBCP approach designated by "IP

Network Database-Based Control Plane"(IP-DBCP). The objective is to define and develop a first approach of an SDN setting, having controllers and switches communicate through their individual databases. We focus on the essential aspects of the wide area routing that enable its functioning, leaving more specialised components, such as QoS and traffic engineering, for future developments.

3.2 Components of the DBCP architecture

DBCP has two main components: switch and controller. Controller and switches communicate through database replication techniques over the network. This communication layer is called southbound communication. The rest of the chapter describes DBCP in more detail as well as the components and what was taken into consideration for the implementation.

Switch

In DBCP a switch is characterized by the partial removal of the control plane. As opposed to the OpenFlow definition, in DBCP the switches still conserve a part of their control plane. As will be clearer below, all distributed control plane functions are, as with OpenFlow, under the responsibility of the controller, however most centralized control plane functions are still with switches in DBCP. Some of the control plane responsibilities are delegated to the controller using a database management engine. Figure 3.3 presents DBCP view of switches and controllers. A switch can be modularized by three components: a database, the switch control software and the logical and physical hardware.

The database is the divergent element on switches, as compared to other SDN approaches. As stated before, this database is used as the base element of communication with the controller. For that reason, it is essential that the switch has the knowledge to be able to populate that database with the required data. Additionally, it must be able to drive its hardware, on the basis of information received from the controller. This database component is part of a database management system and requires mechanisms for database replication.

The control software is an essential component of switches. This software has the responsibility of bridging hardware data and database operations by sitting as the intermediary between database and hardware changes. The control software in the switch is based on downcalls and upcalls to the switch logical hardware layer, and queries and updates to its database. The switch control software is able to interpret hardware configurations, state and capabilities and transform those into database updates. Also, it transform database updates into low level hardware changes. A switch control software requirement is to react quickly to both hardware configuration changes and database updates. This requirement is important to increase the speed at which those updates reach the controller in order to reduce convergence time. The same requirement applies to the database changes and consequent downcalls to the hardware.

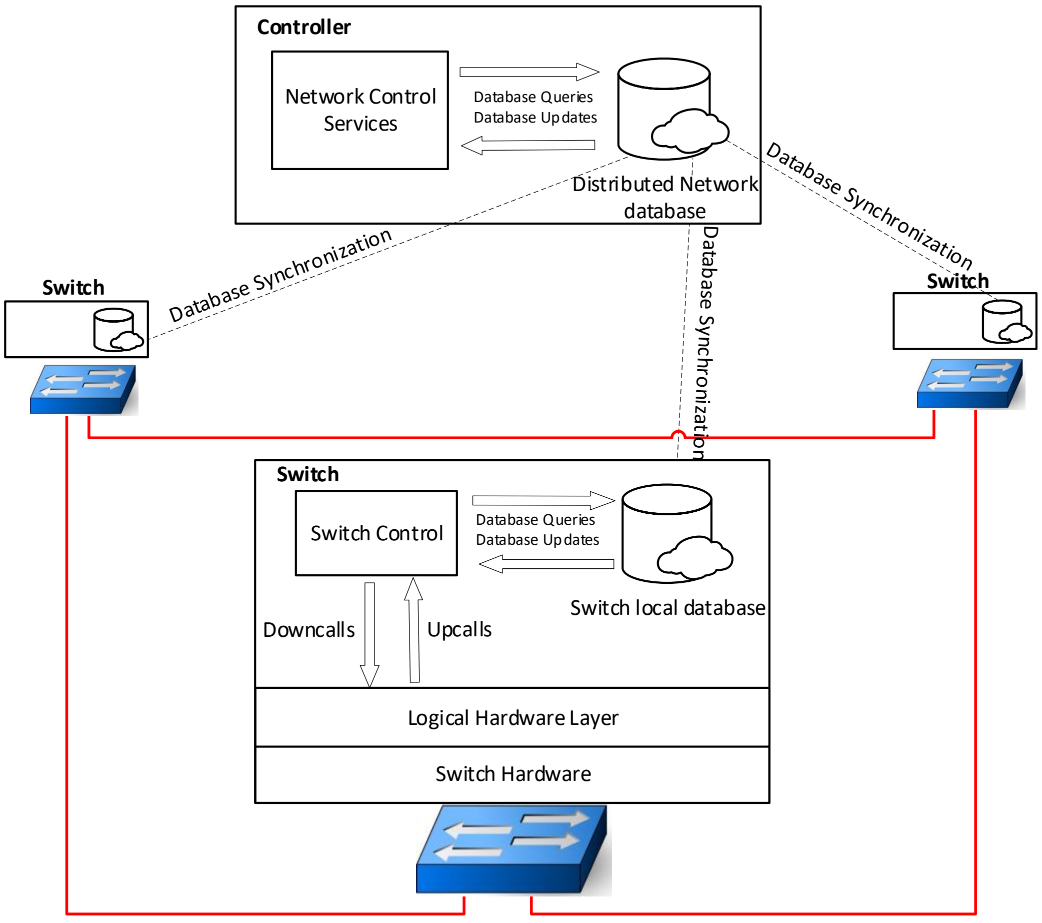

Figure 3.3: DBCP view of switches and controller

The hardware components are the low level mechanisms that enable packet forwarding, flow creation and capability configurations such as interfaces states. It can be further distributed in a logical hardware layer and the real hardware.

A switch may be connected to multiple other switches and hosts through its existing interfaces and links. The switch's main role is to receive and forward incoming packets to the requested destination or next-hop. To achieve this, while not relaying on a specific control plane, it operates on the base of the controller's commands which arrive to the switch as database updates. For the purpose of this experiment, the data models used to implement routing include the routing information base (RIBs) and should also encompass interfaces, their state and neighbourhood.

To implement IP routing it is assumed that there is a level of abstraction on the switch that knows how to interpret the routing table, in order to generate a corresponding flow table. Also, packet forwarding mechanisms components and interface management are also assumed to be present.

An essential requirement for link state protocols is the knowledge of the network topology. This information is essential for the creation of routing rules that enable traffic on networks to be forward to the destination. The creation of new routing rules is dependent on a complete view of the network, in order to make them precise and efficient. To create a global view, protocols like OSPF detect state changes through purpose-build protocols that periodically monitor all interfaces on switches and flood the changes to all other switches.

We could adopt a solution similar to the one available in most OpenFlow controller/switches. In general, OpenFlow upcalls signal to the controller the available interfaces, their status and state change. However, discovering which neighbours are available via each interface is implemented by the controller, in general using a process identical to the implementation of Link Layer Discovery Protocol(LLDP) [9].

We decided that switches in DBCP should be in charge of discovering their neighbours by reason of separation of concerns. It is possible to implement at the switch level, this discovery process almost in a centralized process. Therefore we decided to consider this a low level requirement the controller may ignore. This neighbour information collected by switches is considered for the database model of controller and switches.

A switch view of the world can be summarized to its database and directly connected neighbours. The switch view of the controller is abstracted by the database. Instead of communicating directly with a controller, a switch sees the database as repository of knowledge that enables its operation.

Controller

In DBCP, a controller, just like in other SDN approaches, is the central pillar of the architecture. The controller has the purpose of orchestrating the switches' activity and maintaining a global view of the network based on information originated from switches. Figure 3.3 also shows the role played by the controller on the network. A controller is characterised as containing the replicated state of all switches' databases and a Network Control Services module. The controller's goal is to use those databases to build a global view of the network and apply configuration rules to all switches that enable the appropriate routing of packets. The Network Control Services are able to query the replicated databases to build a global view of the topology, configure switches and execute shortest path algorithms using the database's information. Those services must also be able to react to database changes and apply corresponding solutions. This reaction is critical for the convergence time.

For routing, we opted for the implementation of a pro-active approach. The controller creates beforehand the necessary routes based on the current knowledge of the network topology and inserts those new routes on the database. Compared to a specific approach this does not require that, when a new packet arrives, that packet is sent to the controller to be analysed. We opted for this approach to follow what's already done with link-state protocol and to essentially simplify the routing procedure.

The controller's view of the world is restricted to its database. It is able to build a global view of the network based on the individual vision of the switches.

Southbound communication

As stated before, the action of communicating between controllers and switches and vice-versa is reduced to local databases updates. The communication is abstracted by making all operations seem local. The databases then use replication protocols to synchronize any update that occurs.

3.3 Network model and other assumptions

The implementation presented in this dissertation focuses on wide area network environments. Since these networks are complex, some assumptions about the implementation were made in order to scale down this experiment.

The solution at this phase will only consider IPv4 routing and neighbour discovery, leaving IPv6 for future implementations. Also, unlike OSPF or IS-IS which supports the creation of multiple areas [14], it was decided that in the context of this work, an area free environment would only be considered.

To facilitate the construction of the experiment, we opted for the implementation of two separate networks: the management and production networks. The production network corresponds to all the links interconnecting all switches and focuses on transporting production packets. The management network is a private network, connecting switches and controllers, with the purpose of only being used for management data.

For switch-to-switch connectivity of the production network, we assumed only point-to-point links. Point-to-point links are characterized by connecting only two nodes.

Link and switch failures may occur on a network, which must be dealt with in order to maintain the network resilience. For this experiment, we assume only link and switch failures.

SDN implementations usually deploy more then one controller. On this experiment only one controller is considered.

3.4 Data model

This section presents the DBCP data model. A data model is a design model that describes data, data relationships, data semantics, and consistency constraints. It provides the means to describe the design of a database.

The data model plays an important role since it serves as the base for controller and switch southbound communication. The main objective of the data model is to define the rules for a database that can support the insertion of data, by both a switch and a controller, required for the functioning of the network. A same base data model will be used by both the controller and the switch.

3.4.1 Conceptual Data model

A conceptual data model is an abstract view of the data model. It describes the semantics and main concepts within certain subjects.

For the definition of the data model it is important to first consider the requirements. These requirements are critical to build the data model since they define the rules and conditions necessary to achieve the defined objectives. At this phase, only the base data model is discussed. The base data model is composed by essential information that is common to the controller and switch.

The essential goal of the data model in DBCP is to be the base that supports the communication from switch to controller and vice-versa. For this reason, the base data model must support the distributed insertion and retrieval of data by both a controller and switches.

This base data model must be designed in such a way that the same model can be deployed on both a controller and a switch. It must also be generic enough to support basic IP routing in the network functioning and still allow different configurations to be enforced by the controller. Also, the base data model must take into consideration the network model and the assumption previously presented.

So, for the definition of the base data model, we must first analyse what data is required to be exchange between controllers and switches. This data will drive the base data model design.

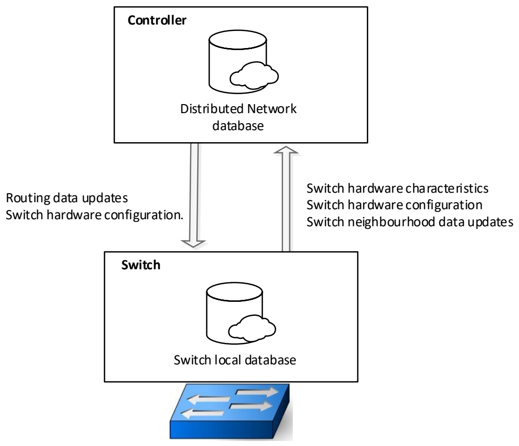
Controller
Distributed Network database
Routing data updates
Switch hardware configuration.
Switch hardware characteristics
Switch hardware configuration
Switch neighbourhood data updates
Switch
Switch local database

Figure 3.4: Data flow view between controller and switches.

To discuss the required data exchange, we will look at what a controller and a switch require from one another in order to achieve their individual goals. For the definition of data exchange, we will look at the controller's and switch's perspective. The question is, what data does a switch require from the controller and the controller from the switch, in order for both to operate as established.

Switch

As discussed before, the vision of a switch became restricted to their lower level functioning and direct neighbourhood, which consequently leads to the need for intervention by the controller in the definition of the guidelines for the switch functioning. Namely, in what concerns all data deployment of the network (as a whole). In this sense, for the functioning of a switch we can group data into two groups:

- Switch hardware configuration.
- IP Routing data

These two data groups essentially answer the lack of intelligence caused by the SDN design. A switch expects a controller to decide about its configuration. The hardware configuration group composes all the data produced by the controller that changes the hardware status of the switch.

The routing data group, considers all the data related to packet forwarding. A switch expects to receive, from a controller, all routing related data since it has no global network view.

Controller

From the controller's perspective, the network is constituted by a set of distributed switches. To fulfil this vision of the network, the controller needs to understand the characteristics and capabilities of each switch as well as their neighbourhood status. This way, we can group the data required by the controller as follows:

- Switch hardware characteristics
- Switch hardware configuration.
- Switch neighbourhood data

One of the key and fundamental roles of the controller is to produce routing data. To this end, it requires a global view of the network topology. In DBCP the network view acquisition is distributed across every switch. A controller constructs a global view by collecting the individual neighbourhood view of each switch.

The controller also needs a view of the characteristics of every switch. This necessity is based on the vision that individual capabilities of each switch must be taken into consideration for the creation of routing rules. In this context, these characteristics are followed by related configurations and statistics that enable knowledge of the current state of the switch.

A modular view

Based on the data requirements by both controller and switches, the base data model can be grouped as follows: the *switch characteristics/configuration*, the *neighbourhood* and the *routing* base, often called the Routing Information Base or RIB. Figure 3.5 shows a modular view of the base data model.

Switch characteristics/configuration
Routing
Neighbourhood

Figure 3.5: Modular view of the data model

The switch characteristics/configurations: this module makes reference to all switches' available characteristics and possible configurations. The *Characteristics* of a switch can be grouped by all the information that describes a switch, or as its features and capabilities. An example of this kind is the physical address of a switch or the number of interfaces available, followed by all their individual characteristics. The *Configuration* part corresponds to all the values that map to possible configurations that may impact switch operations. Such status changes can be a change of an interface's state or name. This information is important to the controller to enable better knowledge of all features supported by network devices and to allow possible state configurations changes to the switch. This module also includes statistics.

Routing: this module corresponds to the routing data. It would contain all information relevant to routing, produced by the controller and used as the guiding map for the switches forwarding mechanisms.

Neighbourhood: this module's main function is to gather all neighbour related information that was collected by the switches. Such information is important for the controller to create the global view of the network.

3.4.2 Logical data model

A logical data model is an implementation of a conceptual data model, presenting a more detailed view of a data model. It describes entities, attributes and relationships but abstracts implementations details.

For DBCP we start by defining an Entity-Relationship Model (ER). The Entity-Relationship Model is composed of entities, relationships and attributes that represent a description of interrelated things of interest.

For the definition of the ER model we consider the assumptions and requirements defined in the conceptual model. The base data model is divided into three groups: *The switch characteristics/configuration*, *Routing*, *Neighbourhood* . We started by defining the entities and relationships that play a role inside each of those groups. An entity is an object that is distinguishable from other objects.

Switch characteristics/configurations encompasses entities that support the switch characteristics and configurations. First of all, we defined an entity called *switch*. Other pillar of switch management are the interfaces configurations. Although an interface is a characteristic of a switch, we consider it as an independent entity. In this sense we add a new entity called *interfaces*. This entity serves as the repository for all data related to interfaces in a switch. This entity needs to maintain a relationship with the *switch* entity. This relationship is expressed as one-to-many. It is considered that an interface has to necessarily be associated with one switch, but a switch has more than one interface.

The routing module hosts data related to routing and the forwarding of packets in DBCP. It is assumed there is only one entity called *routing*. This entity hosts data related to several routes. Because the network model defines the network connections as being point-to-point, it can be assumed that there exists only one other switch behind an interface. So it is possible to define routing rules based only on the exiting interfaces. This originates a relationship between the *interfaces* and the *routing* entity. This relationship is mandatory for the *routing* entity and can be expressed as one-to-many. One interface can be associated with many routing entries but one route can only be associated with one interface. Also, every route must be associated with a switch, representing the switch it belongs to. So, a one-to-many relation is the solution, where one switch can have multiple routes and one route belongs to one switch.

The next module is the *Neighbourhood* one. The goal of this container is to store all data related to the topology view. For the logical data model we consider only one entity: The *neighbour switch*. The entity represents a switch that is a neighbour of another switch. In a relational way, the *neighbour switch* is related to the *switch* with a one-to-many relation. A switch can have multiple neighbour switches but a neighbour switch is only associated with a switch. This first relation aims to create a first relational link view between a local switch and a remote switch directly connected. The *Neighbour switch* also maintains a relation with the *interfaces*. This association is important to further specify how a neighbour switch relates to another switch, in this case, through which interface.

This is a one-to-many relation where a neighbour switch can be associated with many interfaces, assuming that two switches can be connected through two different links and interfaces, and an interface can only be associated with a neighbour switch, as for the limitation of point-to-point interfaces.

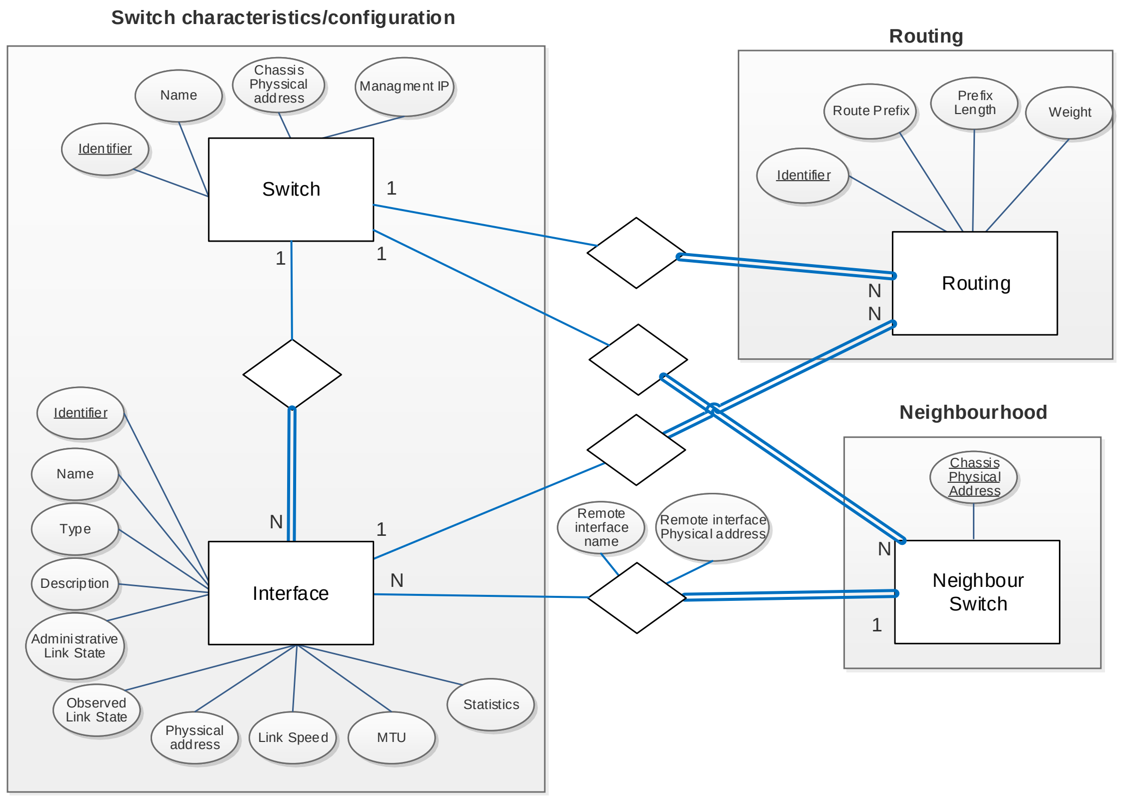

Figure 3.6: An entity–relationship diagram for the base data model.

Entities can have attributes. An attribute is a property of an entity. In the following list we will describe those attributes, reason to its introduction and their importance to the data model.

Figure 3.6 shows an ER diagram with entities, relationships and attributes.

Switch

Identifier - Each switch has an identifier. This identifier allows the unique identification of a switch on the network and the data model.

Name - The name of the switch. The objective of this attribute is to assign a human readable name to identify the switch.

Chassis Physical address - This attribute relates to the chassis's MAC address of a switch.

Management IP - The management IP is the IP address assigned to the switch on the management network.

Interface

Identifier - Each interface has an identifier. It allows the unique identification of an interface on the network.

Name - A name of the interface. The objective of this attribute is to assign a human readable name to identify the interface.

Type - The type of interface.

Description - A textual description of the interface.

Administrative Link State - The administrative link state relates to the configured desired state of the interface. Its purpose is to enable the configuration of an interface's state. This is the state that controller imposes.

Observed Link state - The observed link state is the actual state of the interface reported by the switch.

Physical address - The MAC address of an interface.

Link Speed - The link speed of with an interface.

MTU - The Maximum Transmission Unit (MTU) of the interface.

statistics - Encompasses data produced by the switch that characterizes the functioning of the interface.

Routing

Identifier - Each routing entry has an identifier. This identifier will be unique for each route entry and allows unique identification of an route on the network.

Route Prefix - The network destination IP address.

Prefix Length - The IP prefix length of the network destination.

Weight - A weight associated with this route entry.

Neighbourhood

Chassis Physical address - Represents the MAC address of the neighbour switch.

Remote interface name - Represents the name of the remote neighbour interface.

Remote interface physical address - Represents the physical address of the remote neighbour interface.

The above presented model has been established in an iterative way, being influenced by the implementation options presented in the next chapter. It is worth noting that this model has been established after the analysis of available RFC [2] [3].

3.5 Conclusions

This closes the presentation of the global model of DBCP as well as the base data model used to support DBCP in a network. We left for future work the separation of the data model in two parts: one specific to the DBCP approach in general, and the other containing the parts specific to the IP-DBCP experiment. Its centralization as well as a description of the implemented prototype are the object of next chapter.

Chapter 4

IP Network Database-based Control Plane implementation

4.1 Introduction

This implementation is an experiment, based on the DBCP model discussed in the previous chapter, and consists of the development of a system that is capable of testing database protocols as the means for communication between a controller and switches and, at the same time, support the implementation of real IP packet forward and network management mechanisms on a wide area network.

The implementation of this experiment consisted of the development of a controller that contains a local database and is able to produce routing rules, and a switch that also contains a local database. Both implement all the necessary tools that enables the operation of the network by using their installed databases as the means of communications. Figure 4.1 shows an overview of the completed system architecture.

Switches were implemented as devices running OpenSwitch OPX operating system as a logical hardware abstraction on a virtual machine environment. OpenSwitch OPX offers programmability interfaces that allow the control of a network device and the extraction of configuration, lower level hardware details and network details. The implemented switches are able to forward IP packets, and collect their neighbourhood state using the LLDP protocol.

Switches execute a database management system using a pre-defined data model that stores data related to its underneath capabilities, statistics and also data inserted by the controller. Switches run a control software called *SwitchDBCP* that intermediates the flow of data between OpenSwitch OPX services and the database.

The controller runs a database containing the replicated state of all switches' databases

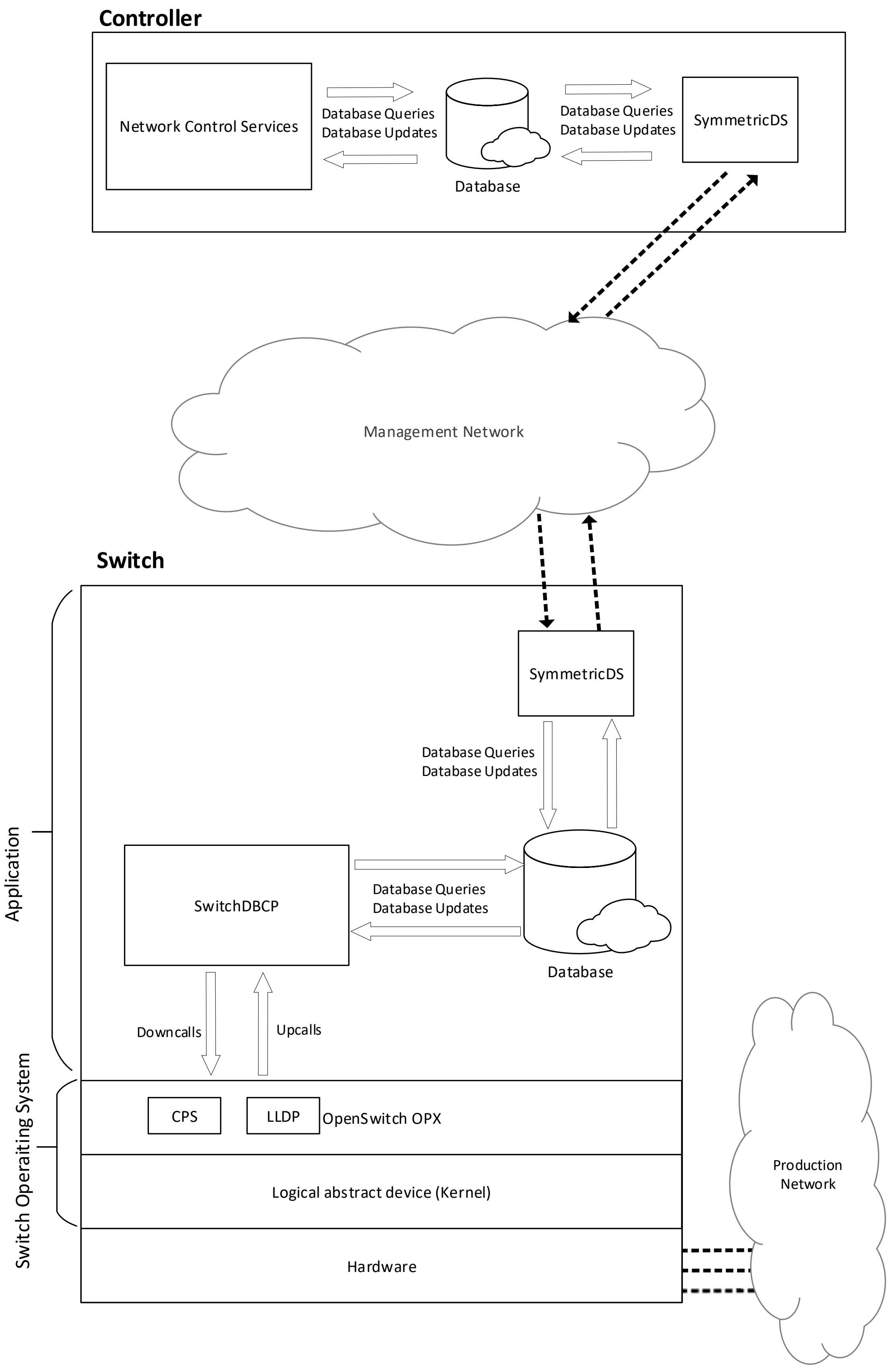

Figure 4.1: Complete view of the implemented system components

and the *Network Control Services* software that manages data from the database and produces routing rules based on the network topology. These routing rules are inserted in the controller's database and propagated to the switches by the replication protocols.

Switches and the controller communicate with each other using the management network emulated in a virtual machine environment. The communication is achieved through the insertion of data into their local database and the use of SymmetricDS software, deployed on both controller and switch, that enables the replication of the database's state. SymmetricDS is orthogonal to the database to separate, as much as possible, the implementation of the switch and controller from the replication mechanisms since we think that an alternative replication substrate will be necessary in the future and is being designed right now.

Switches use the production network to forward packets to their destination.

To achieve the implementation of the purposed solutions the following tasks were pursued:

- Preparing an OpenSwitch OPX image based on one from [19].
- Integration of OPX virtual machines with VMware and the creation of custom virtual networks.
- Implementation of *SwitchDBCP* control software.
- Implementation of the data model schema based on OpenSwitch data model and launching of MySQL databases on switches and controller.
- Integration of SymmetricDS software, on switches and controller, with MySQL databases and a replication model.
- Implementation of the *Network Control Services* software on the controller.

Each of these tasks, as well as the related implemented solutions, will be described on the following sections.

4.2 Switch

4.2.1 OpenSwitch

To represent a switch we opted to use OpenSwitch OPX[18]. OpenSwitch OPX is a Linux-based Network Operating System (NOS) which aims to allow the direct control and abstraction of a network device based on custom made network hardware(e.g. a chassis with ASICs and NICs) or a general purpose server with conventional NICs. It uses an unmodified Linux Kernel and is based on Linux Debian, providing the features of standard Linux IP stack, tools and open source software available for the platform [20].

OpenSwitch OPX provides to this experiment an abstraction of hardware details, allowing us to focus on the implementation and testing of the mechanisms supporting communication between controller and switches.

OpenSwitch, being a conventional Linux operation system, enables the execution of custom-made applications and the use of a wide range of software that is suitable for this implementation. These features are essential to deploy a database management system and related protocols. Furthermore, the use of OPX services helped us use already implemented and tested Linux IP stack network features for packet forwarding.

OpenSwitch OPX has several layers, starting with the hardware. It is currently available in Linux platforms as well as in a range of Dell switches[20]. Thus it is easy to build virtual network devices in a virtual machine or container-based environment. Although OPX OpenSwitch allows testing by virtualization, a similar result is possible on a concrete hardware switch as well as using software-based routers.

On top of the hardware, a standard Linux distribution runs alongside OPX services.

The Control Plane Services (CPS) [17] is an object-centric API, provided by OPX, that enables an interaction between client applications and the OPX platform control and network abstractions. This API was crucial for the implementation, since it provided a way of interacting with OPX services by enabling the retrieval of switch network and hardware details and also the enforcement of configurations. Alternately, networking features could be accessed using the Linux standard API. However, with the former approach, the IP-DBCP software stack can run directly on any hardware platform based on OPX.

The CPS API uses YANG. YANG is a modular data modelling language created to standardize the definition and configuration of the network device characteristics. The YANG language is defined in [1]. Each attribute is represented by a tree path over the YANG model. The YANG model used by the CPS API influenced the design considerations and the definition of the physical data model.

The neighbourhood search and acquisition process was achieved using the Link Layer Discovery Protocol (LLDP) [9]. The LLDP protocol was used to fulfil the requirement for searching and constant updating of the neighbourhood state for the purpose of building the network topology.

LLDP is a vendor-neutral layer 2 discovery protocol used by network devices to advertise their capabilities and identify neighbours. It periodically sends LLDP frames to neighbour devices. Those frames contain information, such as the switch's chassis MAC address or management IP address. Receiving devices collect and store those frames on a management information database (MIB) and complete the received data with local information such as the interface from which the frame arrived.

LLDP is a protocol that is implemented by almost all network devices and servers. Openflow-based SDN environments also use LLDP but the protocol, in those cases, is in general implemented by the controllers.

In our opinion, the right place to implement LLDP in our environment is at the switch level, since it deals with details that only concerns the switch itself and doesn't require any distributed coordination. OpenSwitch OPX includes an implementation of the LLDP protocol.

4.2.2 Switch deployment

For the deployment of switches we opted for the use of VMware virtual environment[23]. This option allowed a faster development of the software modules and offers flexibility in testing and evaluating the developed components on a larger scale. We decided to use independent virtual machines to represent each device. Each virtual machine runs OpenSwitch OPX operating system.

OpenSwitch OPX has support for hardware virtualization. The OpenSwitch community provides an OPX version that simulates basic hardware functionality. For the development of this experiment an image was used, that is available on [19], as a starting point. This image is composed of a base Linux(Debian) operating system with pre-installed OPX services and software modules and their requirements, such as Python for example. Several updates were made to the base image in order to support the dependencies of the experiment. The following software was installed:

Python 2.7.9

Python is used by the CPS API and was also used for the implementation of *SwitchDBCP* and the *Network Control Services*.

MySQL Server 5.7.2

This version(or higher) is required in order to support the creation of multiple triggers with the same action on the same table. This is a requirement to support both *SwitchDBCP* and SymmetricDS triggers that monitor the database.

Python MySQL connector 2.1.7

The MySQL Python connector that supports MySQL and Python versions.

SymmetricDS 3.9.2

The latest version available at the time of implementation was installed.

Java 8

Java 8 is a requirement of SymmetricDS version 3.9.2 .

LLDP 0.95-1

LLDP comes already installed with the OPX base image. Even so, LLDP was updated to version 0.95-1 to enable the output of the results in JSON format.

Switches' interfaces are created and managed by VMware virtualization. Each virtual interface is mapped by the OPX operating system. Each switch is connected to other switches and hosts through links abstracted by VMware networks.

To implement the production and management networks described in the previous chapter, private virtual networks were created on VMWare. These private networks are composed of a virtual switch that connects every interface to that network.

The production network was implemented by setting the interfaces destined for production traffic as part of a specific private network. A link between two switches is characterized by the involved interfaces being on the same private network. Every link is represented by a new private network, exclusive to that link. The network model described before, states that in IP-DBCP we only assume point-to-point connections. To access the existence of a real point-to-point connection all interfaces had to be assigned IP addresses. We decided that for this experiment, the numbering of interfaces would be a manual process.

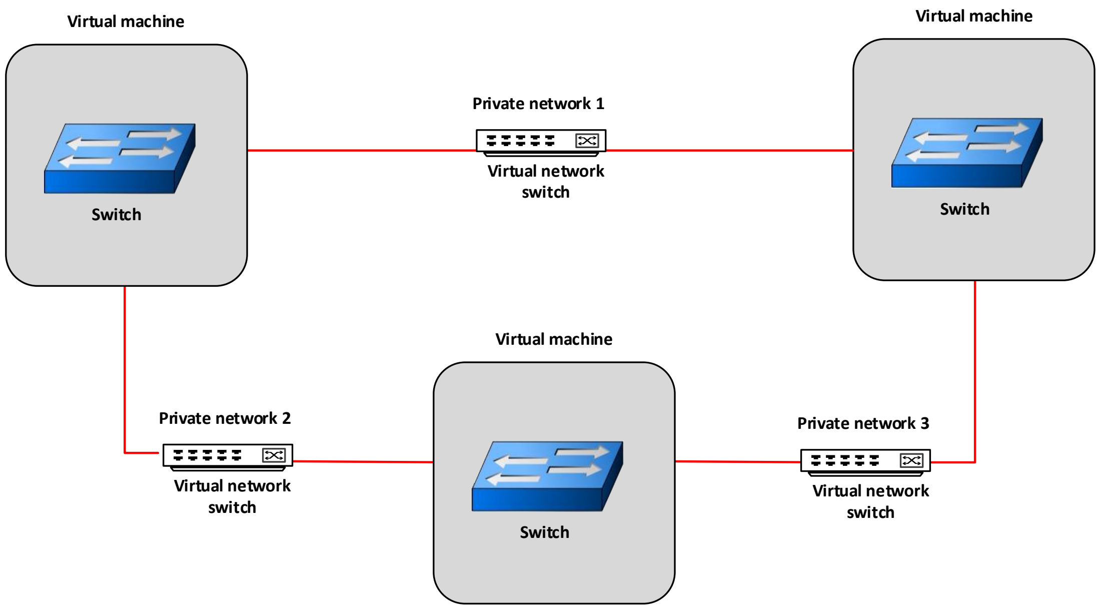

Figure 4.2: Production network: Virtual networks created to connect switches running on virtual environments

4.2.3 SwitchDBCP implementation

SwitchDBCP is the control software that sits between the switch logical hardware abstraction (OPX) and the database management system (MySQL). Its function is to guarantee that both the database and OPX are in an equivalent state. The state of the database must reflect the underneath hardware abstraction and vice-versa. *SwitchDBCP* reacts to both database and OPX state changes and applies updates in accordance with these changes. Figure 4.3 shows the architecture of *SwitchDBCP*. It shows the developed modules, their

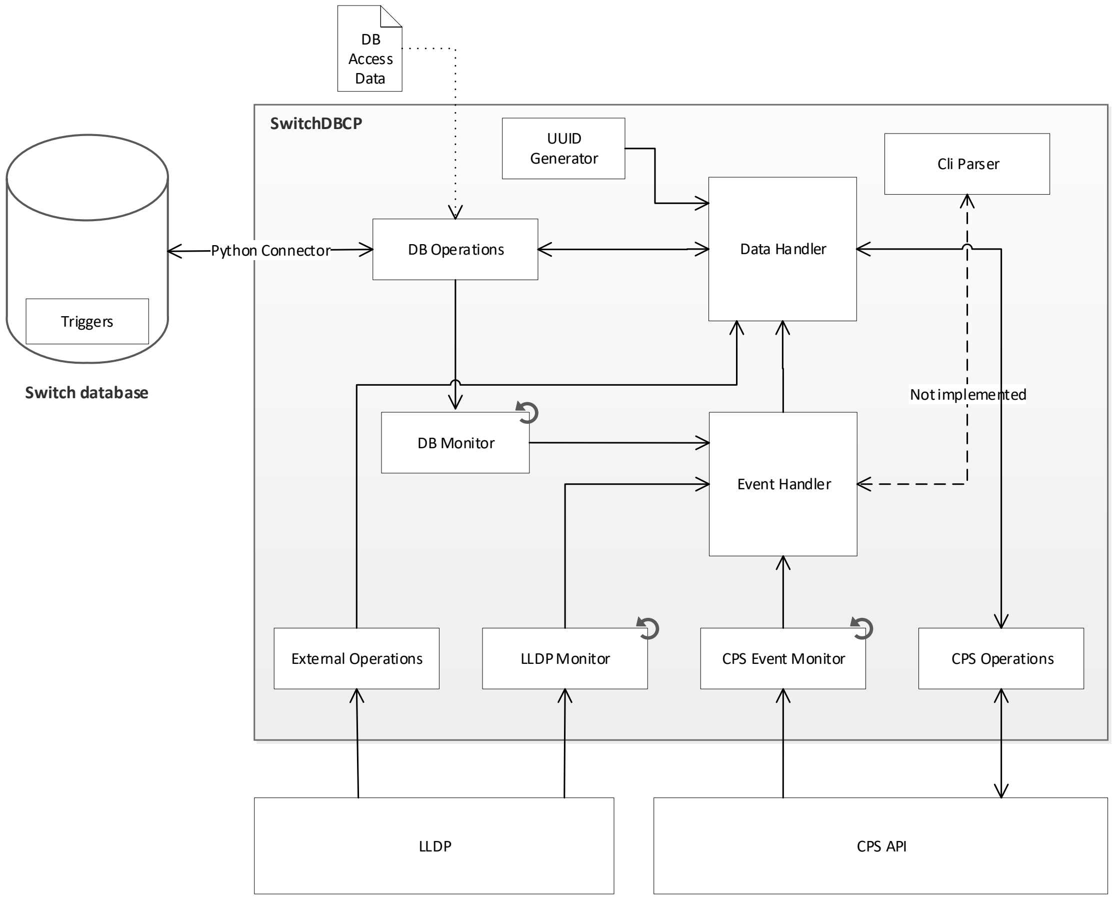

Figure 4.3: A modular view of SwitchDBCP

interactions between them and with the external components. *SwitchDBCP* was completely implemented in Python to satisfy the CPS API dependency. This control software is composed of listeners that continuously monitor: the MySQL database, OPX services and LLDP events. *SwitchDBCP* uses the CPS API exposed by the OPX services to update configurations. *SwitchDBCP* is composed by the following classes:

DB operations

DB operations class is used to communicate with the MySQL database using the MySQL Python connector. This class imports the database access data from a file named *"database_config.txt"* that exists on the same package. This file is structured has follows:

1. The user name used to authenticate with the MySQL server.
2. The password to authenticate the user with the MySQL server.
3. The IP address or the host name of the MySQL server (e.g localhost).
4. The name of the database to use when connecting to MySQL server.

DB Monitor

DB Monitor is a listener that constantly monitors the database for changes. Those changes occur only when the database is updated by the remote synchronization process. Database changes are collected by triggers and placed inside a queue to maintain the order of those updates. *DB Monitor* periodically retrieves those updates from the database.

On this implementation we assumed that a controller could only update the *interfaces* and *ipv4-rib* tables. *DB monitor* constantly queries the *ipv4-rib-changes-log* table and *interfaces-changes-log* table.

The complete process of acquiring the new updates works as follows: triggers were created and inserted on the database that monitor the *interfaces* and *ipv4-rib* tables. Any updates made on the *interfaces* table are registered on the *interfaces-changes-log* using the *identifier* attribute of the updated *interfaces* entry and a reference to the attribute that was updated. Three more triggers were created to detect the creation, update and removal of new *ipv4-rib* entries. They store on *ipv4-rib-changes-log* the *identifier* of *ipv4-rib* and the type of operation.

When *DB monitor* detects new updates, it collects those and individually sends them to the *Data Handler*. Their entries are then deleted from the log tables. It is assumed that the switch update process, after deleting the logs, never fails. *DB Monitor* is executed in a separate thread. The rate at which the *DB Monitor* consults the database is controlled using a variable. This variable is configured by default to 50 milliseconds. In future implementations this process should be optimized.

LLDP Monitor

The *LLDP monitor* is a listener that monitors the LLDP daemon. It detects any neighbourhood changes and transforms those into database updates. *LLDP Monitor* works as follows: the listener is executed in an independent thread that executes the Linux Command Line (CLI) command "*lldpcli watch -f json*". This command monitors any neighbourhood changes and returns a report immediately after the change. When a neighbour change occurs, *LLDP Monitor* collects the result in JSON format and parses it in order to transform it into a database update operation. *LLDP Monitor* sends those results to the *Data Handler*. The rate of neighbour detection is exclusively controlled by *lldpcli* configuration. The interval between LLDP packets sent to other switches and the time to live (TTL) of already discovered neighbours values can be modified using this configuration.

CPS Monitor

The CPS monitor is a listener that monitors data changes from OPX. Its purpose is to transform lower level changes into database updates. Currently the implementation only reflects interfaces changes, more precisely, updates to the attribute *oper-status*. The listener works as follows: it starts by registering it self to the CPS API event libraries using the YANG key "dell-base-if-cmn/if/interfaces-state/interface". Whenever there is an update, CPS API reports back to the CPS Monitor with an *CPSObject*. CPS Monitor collects the interface name and current state of the *oper-status* field and sends it to the *Data Handler*.

Event Handler

The *Event Handler* class has the purpose of handling the execution of the *CPS Monitor*, *LLDP Monitor* and *DB monitor* listeners. It controls the launch of their threads and forwarding of their updates. Every update produced by the listeners is placed in a queue, that is constantly being emptied.

Data Handler

Data Handler is a class that centralizes all data processing logic and defines how data is forwarded. It adapts data collected by the listeners and decides which way it is forwarded, either placed on the database or deployed on OPX.

Listing 4.1: An example of a downcall using CPS API (changing the administrative link state)

```

@staticmethod
def setInterfaceAdminLinkState(if_name, state):

cps_obj = cps_object.CPSObject('dell-base-if-cmn/if/interfaces/interface')
cps_obj.add_attr('if/interfaces/interface/name', if_name)
cps_obj.add_attr('if/interfaces/interface/enabled', state)

cps_update = {'change': cps_obj.get(), 'operation': 'set'}
transaction = cps.transaction([cps_update])

if not transaction:
raise RuntimeError("Error␣change␣the␣interface␣state")
```

CPS Operations

CPS operations is the class used to implement all methods that communicate with the CPS API. This is the class used to query the YANG model and to propagate changes from the database to OPX. The listing 4.1 shows a sample method of a downcall using the CPS API. The methods implemented are the following:

def setInterfaceIpAddress(if_name, ip_addr, pfix_len):

This method allows the assignment of an IP address to an interface. It uses the name of the interface which uniquely identifies an interface on the YANG model. The interface assignment is completed with the IP address to assign and the prefix length associated. This update is usually triggered by a database update to the IP address of an interface.

def addIpv4RouteEntry(route_prefix, prefix_len, if_name, weight,next_hop):

This method adds a new routing entry to OPX routing table. This method requires the destination IP address and the prefix length. Also, we set new routes as having both a next-hop IP address and outgoing interface. In a scenario where a controller creates a new route and places it on the database, this method would be called to process that update on the concerned switch.

def deleteIpv4RouteEntry(route_prefix, prefix_len):

This method removes a routing entry from OPX routing table. OPX routes are identified by its IP prefix and length, so this method only requires those in order to delete the entry. This method is called in a scenario where a controller deletes a route from the database.

def getAllInterfacesData():

This method method is used to obtain all data from OPX that relates to the device interfaces. This includes its attributes and statistics. This method is used during the bootstrapping process of the switch.

def getChassisMac():

This method obtains the physical address of the switch chassis. This is the address that identifies the switch and is collected during the bootstrapping process.

External Operations

The External Operations class is used for methods that interact with the outside world of *SwitchDBCP*. On this implementation this class is used to obtain the initial neighbourhood state from LLDP and the IP address of the management network interface of the switch, assigned by an external DHCP server, during the bootstrapping phase.

We now turn to the description of the controller implementation.

4.3 Controller

A simple controller was implemented based on the DBCP model defined on the previous chapter. The implemented controller consists of a set of software modules designated altogether by *Network Control Services*. The implemented controller is capable of constructing a network topology view based on the local database state and use that data to compute the routing rules for all switches of the network. The implemented controller also allows the configuration of some of the switch's characteristics.

The controller executes a MySQL database management system with the same base data model as switches. This database, after the bootstrapping process, contains the replicated state of every other switch database that it controls. The controller also reacts to database updates and applies the corresponding modification to the switches' state.

The implemented system only supports the execution of a single controller, leaving the inherent problems such as controller failure, out of this implementation. At this moment, the implemented controller only reacts to changes to interface's state or the formation of new neighbour relations.

The controller, just like switches, runs on a virtual machine environment. Switches communicate with the controller through a management network using an outbound approach. Outbound communication is characterized by all switches being directly connected to the controller over dedicated links. Management interfaces exist on every switch and are used to support communication between switches and the controller. Although not realistic in a real wide area network, for this experiment, we took this alternative to simplify the implementation and bootstrapping processes.

The management network was built using a single private network that connects every switch to the controller. Figure 4.4 shows a management network.

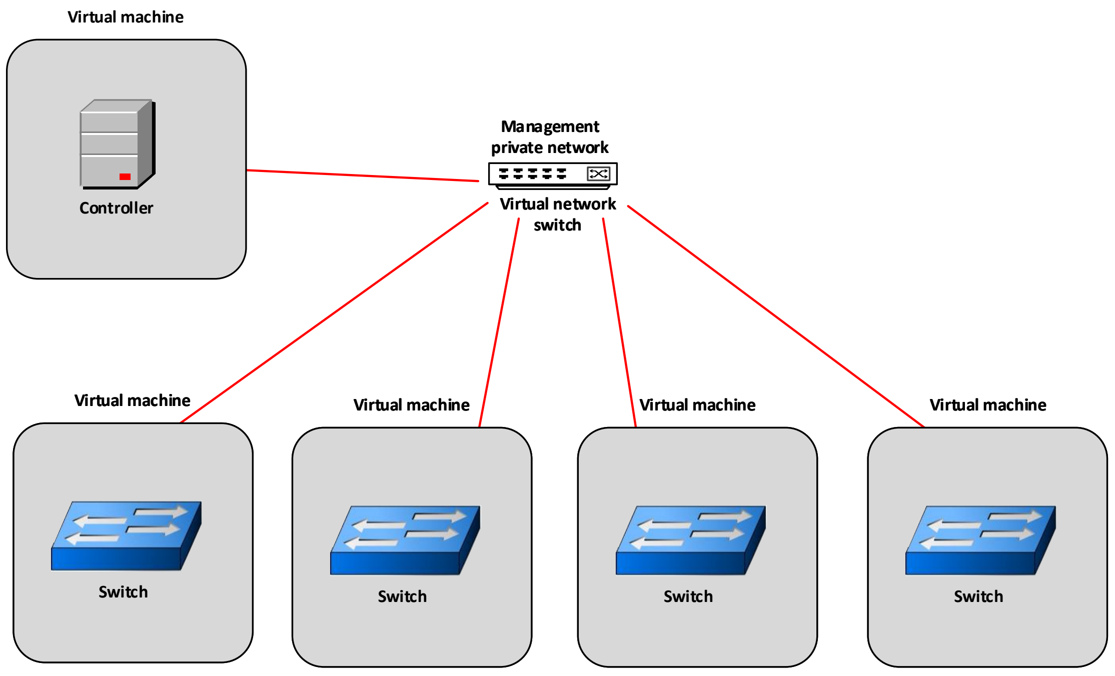

Figure 4.4: Management network: A Virtual network connecting switches and the controller

4.3.1 Network Control Services implementation

Network Control Services is the software that composes the brain of the controller. Its function is to produce new routing rules, based on information shared by the switches, and enable the configuration of the switches it controls. This software reacts to the controller's database changes and produces updates in accordance. It was implemented in Python and has no special hardware or operating system requirement. The architecture of the implemented modules is similar to *SwitchDBCP*. Figure 4.5 shows the architecture of *Network Control Services*.

The *DB operations* class is used to communicate with the controller's MySQL database and has the same characteristics as the implemented class on switches. The *Data Handler* class is used for data processing logic. The *Event Handler* class shares the same implementation characteristics as *SwitchDBCP*, but in this case it only deploys a single listener: *DB Monitor*. *DB Monitor* is a listener that shares the same characteristics of the already mentioned *SwitchDBCP*'s *DB Monitor* listener. For the controller's implementation, this listener monitors the *interface_neighbour-changes-log* and *interfaces-changes-log* tables. It checks for changes to the operational state of interfaces (*oper-status*) and for the formation of new neighbour relations. The rate at which the *DB Monitor* consults the database is also controlled using a local variable configured by default to 50 milliseconds.

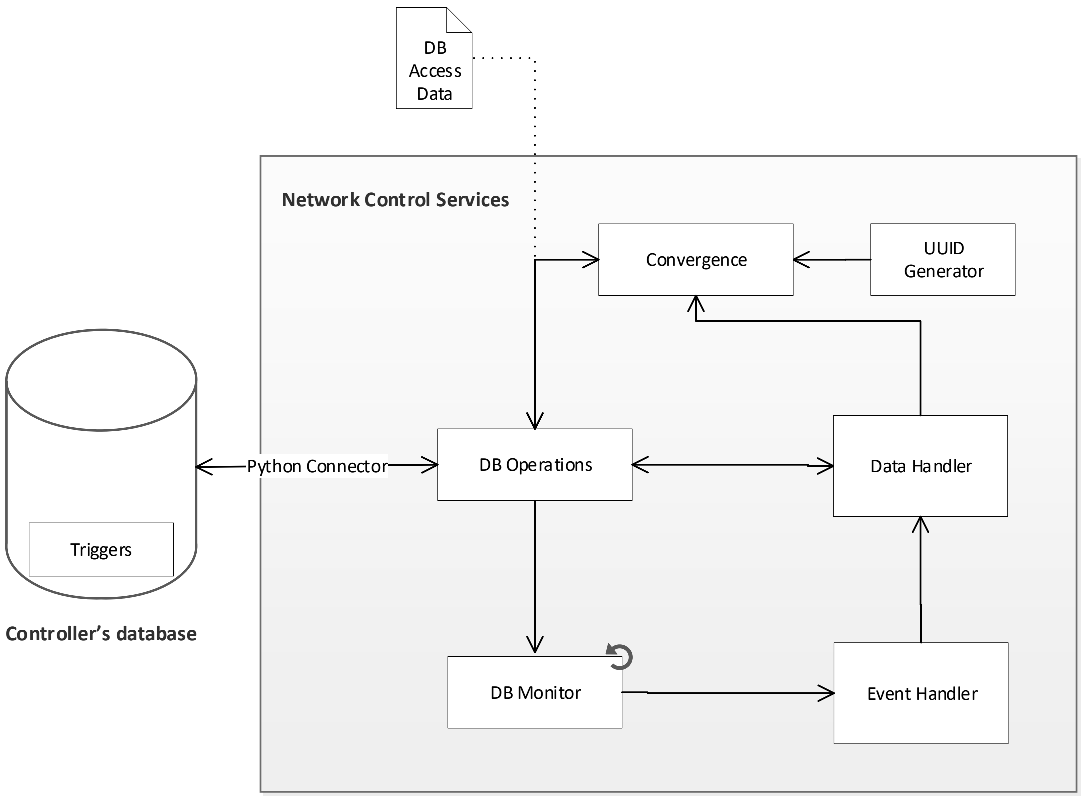

Figure 4.5: A modular view of *SwitchDBCP*

Network Control

The network routing information is computed by the *Convergence* class. This class simultaneously constructs a network topology view and the routing rules for all switches.

The controller builds a network topology graph from the various pieces of data produced and shared by all switches through database replication. To build the network graph, the controller first defines the nodes of the graph. Nodes consist of every switch known by the controller. The controller extracts switch information from the *switch* table. The edges of the graph are the point-to-point links that connect the switches. This information was previously constructed by the switches using the LLDP protocol. When switches discover new neighbours, they create a neighbour relationship expressed through a database insertion on both the *neighbours* and *interface_neighbour* tables. This database insertion is made on both neighbours and on each neighbour it expresses a unidirectional path to the other. To complete bidirectional connectivity, the information of both neighbour switches is used. A link is only considered functional when at least two operational interfaces are connected to it.

A query is made to the *interface_neighbour* table to extract all unidirectional neighbour relations. Those relations are translated into weighed oriented edges on the graph. The weight of each is edge is calculated as shown in Equation 4.1. The network topology build is showed on algorithm 1.

$$Weight = \frac{Reference\ Bandwidth}{Interface\ Bandwidth} \tag{4.1}$$

where:

$ReferenceBandwidth$	= A reference value that was set to 1Gbps;
$InterfaceBandwidth$	= The interface bandwidth is extracted from the database's table *interfaces*, attribute *speed*.

Algorithm 1 Topology graph creation

```
1: procedure BuildNetworkTopologyGraph
2:     $SwitchList$ = $Database$.getAllSwitches()
3:     for all $Switch$ in $SwitchList$ do
4:         $NetworkGraph$.addNode($Switch$)
5:     end for
6:     $NeighbourList$ = $Database$.getInterfaceNeighbourRelationships()
7:     for all $NeighbourRelation$ in $NeighbourList$ do
8:         $weight$ = $NeighbourRelation$.calculateWeight()
9:         $NetworkGraph$.addOrientedEdge($NeighbourRelation.originSwitch$, $NeighbourRelation.destinationSwitch$,$weight$)
10:    end for
        return $NetworkGraph$
11: end procedure
```

The route creation computation can be broken into five steps: acquiring reachable available IP prefixes and associated switches; the creation of individual switch view of reachable IP prefixes and associated switches; for each switch, the execution of shortest path algorithm; the election of next-hops switches; and the creation and filtering of new routing table entries.

The algorithm starts by querying all interfaces with configured IP addresses and a *UP* operational state(*oper-status*=1). The algorithm then extracts the IP prefixes reached by each interface and for each of these IP prefixes, registers the switches to whom they are connected. After the network is build, the next step is to list all the IP prefixes connected to the different switches, see algorithm 2.

Algorithm 2 IP prefixes acquisition

```
 1: procedure IP PREFIXES ACQUISITION
 2:     $InterfaceList$ = $Database$.getAllInterfacesWith($ip$ != $Null$ and $oper\text{-}status$ = 1)
 3:     for all $Interface$ in $InterfaceList$ do
 4:         $ipPrefixes$ = Extract IPv4 network from ($Interface$)
 5:         if $ipPrefixes$ not in $IpPrefixesList$ then
 6:             $IpPrefixesList$.addIpPrefix($ipPrefixes$)
 7:         end if
 8:         $switchId = Interface.ownerSwitchId$
 9:         if $switchId$ not in $IpPrefixesList[ipPrefixes]$ then
10:             $IpPrefixesList[ipPrefixes]$.addSwitch($switchId$)
11:         end if
12:     end for
         return $IpPrefixesList$
13: end procedure
```

The next steps is to associate all IP prefixes known with each switch, in order to compute how each switch gets there. Moreover, all local switch's prefixes are locally available. The algorithm 3 shows this process.

Algorithm 3 Create each switch view of the reachable networks

```
 1: procedure BUILDNETWORKSWITCHVIEW
 2:     $SwitchList$ = $Database$.getAllSwitches();
 3:     $IpPrefixesList$ = IpPrefixesAcquisition()                    ▷ Algorithm 2
 4:     for all $Switch$ in $SwitchList$ do
 5:         $SwitchIpPrefixesViewList$.addSwitch($Switch$)
 6:         $tmpIpPrefixesList = IpPrefixesList$
 7:         for all $IpPrefixes$ in $tmpIpPrefixesList$ do
 8:             if $IpPrefixes$ is reached by $Switch$ then
 9:                 $tmpIpPrefixesList[IpPrefixes] = Switch$
10:             end if
11:         end for
12:         $SwitchIpPrefixesViewList[Switch]$.addNetworkList($tmpIpPrefixesList$)
13:     end for
         return $SwitchIpPrefixesViewList$
14: end procedure
```

The next step consists of executing a shortest-path algorithm for each switch of the network topology graph. This process results on the creation of a path list for every switch containing the paths to every other switch. It makes use of shortest path algorithm from package [15]. Algorithm 4 shows this process.

For each switch it is now possible to determine the next-hop for each destination switch.

Algorithm 4 Path creation from every switch to other switches

```
1: procedure BuildNetworkPath
2:     $SwitchList$ = $Database$.getAllSwitches();
3:     $NetworkGraph$ = BuildNetworkTopologyGraph()                ▷ Algorithm 1
4:     for all $originSwitch$ in $SwitchList$ do
5:         for all $destinationSwitch$ in $SwitchList$ do
6:             $path$ = shortestPath($NetworkGraph$,$originSwitch$,$destinationSwitch$)
7:             $pathList[originSwitch,destinationSwitch]$ = $path$
8:         end for
9:     end for
       return $pathList$
10: end procedure
```

The next step is the election of next-hop switches. Routing rules must be created for every switch on the network, therefore for each IP prefixes, next-hops must be calculated. So, using the IP prefixes list and associated switches from algorithm 3, for each switch and for each individual IP prefixes list, the associated switches list is extracted. They serve as the destination point for a IP prefixes. For each of these switches, the paths generated in algorithm 4 are used to elect the switches (more than one if they have the same cost) that has the least cost to reach that IP prefixes. Finally, for each of the selected switches, the next-hop is extracted from the path that terminates on the elected switch. Algorithm 5 shows this process.

The last step is to construct the routing rules that are later deployed to the switches. This process starts by creating the new routing rules. The output of algorithm 5 is used. For every switch and for every associated network and associated next-hops, a new routing rule is created. Finally, the new routing rules are filtered. If a new route is already present on the database, it is ignored. If there is a routing rule present on the database but not on the new routing list, that route is deleted from the database. The remaining routes are then insert on the controller's database. Algorithm 6 shows this process.

Algorithm 5 Next-hop acquisition

```
procedure BuildRoutingNextHop
    $pathList$ = BuildNetworkPath()                                   ▷ Algorithm 4
    $SwitchIpPrefixesViewList$ = BuildIpPrefixesSwitchView()          ▷ Algorithm 3
    for all $SwitchView$ in $SwitchIpPrefixesViewList$ do
        for all $IpPrefixes$ in $SwitchView$ do
            $candidateSwitchList$ = $IpPrefixes$.associatedSwitches()
            $finalCandidateSwitchList$ = []
            for all $candidateSwitch$ in $candidateSwitchList$ do
                for all $path$ in $pathList[SwitchView.switch]$ do
                    if $path$ reaches $candidateSwitch$ then
                        $cost = path.cost$
                        if $cost$ is the first one the be calculated then
                            $finalCandidateSwitchList$.add($candidateSwitch$)
                        end if
                        if $cost$ is the smallest then
                            $finalCandidateSwitchList$ = []
                            $finalCandidateSwitchList$.add($candidateSwitch$)
                        end if
                        if $cost$ is equal to the smallest then
                            $finalCandidateSwitchList$.add($candidateSwitch$)
                        end if
                    end if
                end for
            end for
            $nextHopList$ = []
            for all $path$ in $pathList[SwitchView.switch]$ do
                for all $candidateSwitch$ in $finalCandidateSwitchList$ do
                    if $path$ reaches $candidateSwitch$ then
                        $nextHopList$.addNextHop($path$.getNextHop())
                    end if
                end for
            end for
            $SwitchNetworkViewList[SwitchView][Network] = nextHopList$
        end for
    end for
     return $SwitchNetworkViewList$
end procedure
```

Algorithm 6 Routing build and filtering

```
procedure BuildFinalRoutingRules
    $SwitchIpPrefixesViewList$ = BuildIpPrefixesSwitchView()        ▷ Algorithm 5
    $NewRouteList$ = []
    for all $SwitchView$ in $SwitchIpPrefixesViewList$ do
        for all $IpPrefixes$ in $SwitchView$ do
            for all $NextHop$ in $IpPrefixes$ do
                $NewRouteList$.newRoute($SwitchView.switchid$,$IpPrefixes$,$NextHop$)
            end for
        end for
    end for
    $DatabaseRoutelist$ = $Database$.getAllRoutes()
    for all $DatabaseRoute$ in $DatabaseRoutelist$ do
        for all $NewRoute$ in $NewRouteList$ do
            if $NewRoute$ exists in $DatabaseRoutelist$ then
                $NewRouteList$.delete($NewRoute$)
            end if
        end for
        if $DatabaseRoute$ not in $NewRouteList$ then
            Delete from the database's table ipv4-rib: $DatabaseRoute$
        end if
    end for
    Insert all routes on the database's table ipv4-rib: $NewRouteList$
end procedure
```

4.4 Physical data model

The physical model deals with the concrete implementation of the database tables and attributes. The implemented data model is based on DBCP and its logical data model. It was also been influenced by the YANG model supported by OpenSwitch OPX. The same base data model was implemented and deployed on switches and controller. In addition to the base data model, some tables have been added to address some of the implementation requirements which were different on controller and switches.

Five tables were implemented that compose the base data model: *switch*, *interfaces*, *interface _neighbour*, *neighbours* and *ipv4-rib*. The direct mapping between some attributes of DBCP base data model and those of the OPX YANG model lead us to use as much as possible the names and types in both modules. RFC [2] was used to define the *interfaces* table, and RFC [3] to define the *ipv4-rib* table.

These tables are used by switches to store their local information and to receive updates from the controller. On the controller there is a replica of all the switches' tables.

Each switch is uniquely identified on the network with a Universally Unique Identifier (UUID). This decision allows each switch to generate its own identifier, without the need for distributed coordination, thus avoiding collisions. Likewise, all switchs' interfaces and routes created are also identified by UUIDs generated by each switch and the controller.

Some modifications in terms of attributes were made from the logical model, in order to overcome some of the limitations of the implementation environment used.

The virtual network environment used to deploy the production network could not support a real point-to-point connection since it deploys a virtual switch to connect every interface. This resulted in a change to the data model since a route could no longer use the outgoing interface as a next-hop. The solution was to add a new attribute to the *ipv4-rib* table called *Next-hop* that contains the address of the destination interface.

For switches, the data model contains three complementary tables: *controller*, *interfaces-changes-log* and *ipv4-rib-changes-log*. The *controller* table is used to store data related to the controllers associated with a switch . This table stores the IP address of the controller on the management network. The controller has the *interfaces-changes-log* table and *interface_neighbour-changes-log* table.

4.5 Database replication

Database replication among controller and switches is the key part of this experiment. The focus on delivering data from switches to controller and back using their local databases, without the intervention of the control modules, lead us to use SymmetricDS[22] software. At the time of development, it was considered flexible and dynamic enough to be rapidly deployed on switches and controllers and satisfied the requirements for this experiment. Also, the reason behind the choice of using an orthogonal solution to the database is due to the need to separate as far as possible the database replication mechanisms from the software being developed. On the other hand, SymmetricDS is a temporary alternative which may not fulfil all the timing requirements for data synchronizing on DBCP.

SymmetricDS is an open source Java software that supports the replication of relational database tables between multiple nodes over the network. It has support for one-way and multi-master replication and can replicate data asynchronously at scheduled moments or periodically. Also, it is agnostic of the database management system.

SymmetricDS installs triggers on a target database and constantly monitors its tables for local changes. Any changes detected are sent over the network, using HTTP, to pre-defined destinations. Also, it applies remote data changes that may have occurred on a remote database, to the local tables. Configuration of each running instance of SymmetricDS is made using a properties file.

SymmetricDS allows the definition of a replication model enabling the definition of the tables and columns of the database that should be synchronized (also defined as vertical filtering) and also to which replica should specific rows go (also called horizontal filtering). Synchronization can also be defined as going on one direction or both directions.

On SymmetricDS a node is considered an instance of SymmetricDS. Nodes exchange data through pushes and pulls. A central node, or master node orchestrates other instances and allows their registration. A push sends changes to nodes and pulls are periodical checks for available changes.

For this experiment we used standalone installations of SymmetricDS version 3.9.2[22], that were deployed on the controller and switches. The role of SymmetricDS in this experiment is to replicate the state of every switch database to the controller and vice-versa. In this implementation every switch executes an instance of SymmetricDS with connection to its local MySQL database. This instance is started during the bootstrapping process and uses a properties file that is built during the bootstrapping process. The controller runs a single SymmetricDS instance connected to its MySQL database.

When switches start, SymmetricDS installs on the MySQL database, the tables and relations required for its execution. Then it attempts to register their instances with the master instance on the controller, using the IP address and the name of the SymmetricDS instance defined on the properties file. When a switch is registered, it receives from the controller instance the synchronizing model defined for that database. The model is translated into triggers and insertions to SymmetricDS tables.

The controller's properties file is static and pre-created.

Listing 4.2: A sample of the controller's SymmetricDS properties file

```
engine.name=controller-000
db.driver=org.h2.Driver
db.url=jdbc:mysql://localhost/dbcp-controller
db.user=root
db.password=XXXXXX
registration.url=
sync.url=http://192.168.1.15:31415/sync/controller-000
group.id=controller
external.id=000
auto.reload.reverse=true
job.purge.period.time.ms=72000
job.routing.period.time.ms=500
job.push.period.time.ms=1000
job.pull.period.time.ms=1000
initial.load.create.first=true
```

Listing 4.3: A sample of a switch SymmetricDS properties file

```
engine.name=123e4567-e89b-12d3-a456-426655440000
db.driver=org.h2.Driver
db.url=jdbc:mysql://localhost/switch_control_database
db.user=root
db.password=XXXXXX
registration.url=http://192.168.1.15:31415/sync/controller-000
group.id=switch
external.id=123e4567-e89b-12d3-a456-426655440000
job.routing.period.time.ms=500
job.push.period.time.ms=1000
job.pull.period.time.ms=1000
```

The SymmetricDS properties files are composed of instructions for the local database connections, remote connection to other SymmetricDS, synchronization time definitions and identifiers that uniquely identify the instance. Listings 4.2 and 4.3 show a configuration example used for the controller and the switches respectively. In both examples, synchronizing periods were chosen just for illustration purposes. For the controllers' properties file, the name and external ID are arbitrary. Database connection is set to its local database. The controller instance is set as a master and to accept the initial database load from other registering switches.

The switch properties file is similar to the controllers' but differs in its external ID and group. The database connection is similar to controller. The switch instance uses the tag *registration.url* to define the IP address of SymmetricDS master instance. The switch uses this address to first register itself on the controller and then to check for pulls. The *external.ID* and *engine.name* for switches are unique. They are equal to the switch *identifier* attribute defined on the *switch* table. This is an important aspect that enables the horizontal filtering of database updates on the controller. This properties file is created during the bootstrapping process.

The replication model is defined on the controller and then replicated to all switches when they register. This models is inserted through database updates on the local controller's database. For this experiment the following model was defined: SymmetricDS triggers were defined that target the tables *switch*, *interfaces*, *neighbour*, *interfaces_neighbours* and *ipv4-rib*. Those triggers signal SymmetricDS to monitor those tables for changes, that will be later replicated. The replication model then defines which kind of data from the tables is replicated and to which node. Data can be sent from a switch to the controller, from the controller to all switches and from the controller to a single switch. Data replication from a controller to a single switch is based on the switch's identifier and the defined *external_ID* on the SymmetricDS properties file.

4.6 The Bootstrapping sequence

The complete process of system bootstrapping starts with the controller. An instance of SymmetricDS, connected to a previously created MySQL database with the aforementioned data model, is started to prepare for future switch connections. Before starting it is assumed that the properties file for the SymmetricDS instance is already present and configured. Also, for the lunch of SymmetricDS, we assume that the SymmetricDS tables are created and the replication model was defined.

The controller's *Network Control Services* are then initiated. Bootstrapping these processes consist of first initiating the *DB Monitor* listener and then executing the convergence algorithm to the current state of the database.

Switches can then be initiated. The switch bootstrapping process initiates with *SwitchDBCP*. Before starting it is assumed that OpenSwitch OPX services are operational. *SwitchDBCP* first starts a connection with the local MySQL database. The database is assumed to be already created and with the switch data model. All data is cleared from the database. *SwitchDBCP* then fills the *switch* and *interfaces* tables on the database, retrieving its data from the CPS API, and is followed by retrieving the current neighbourhood state of the switch from the LLDP protocol. This initial data acquisition process enables the database to start with an updated view of the underneath hardware layer.

When the database has been filled, *SwitchDBCP* starts its listeners: the *DB Monitor*, *LLDP Monitor* and *CPS Event Monitor*. As described before, those listeners will assure that the previously inserted view of the switch is kept updated. Next, *SwitchDBCP* creates a properties file for SymmetricDS. This file is created based on a template. Currently *SwitchDBCP* creates this properties file by configuring it with the current IP address and name of the controller but also by setting the switch *external_ID* equal to its identifier on the database. The IP address of the controller was previously transmitted to *SwitchDBCP* as an initial argument when starting. *SwitchDBCP* then starts an instance of SymmetricDS. SymmetricDS immediately synchronizes the current state of the database. Figure 4.6 represents a switch bootstrapping process.

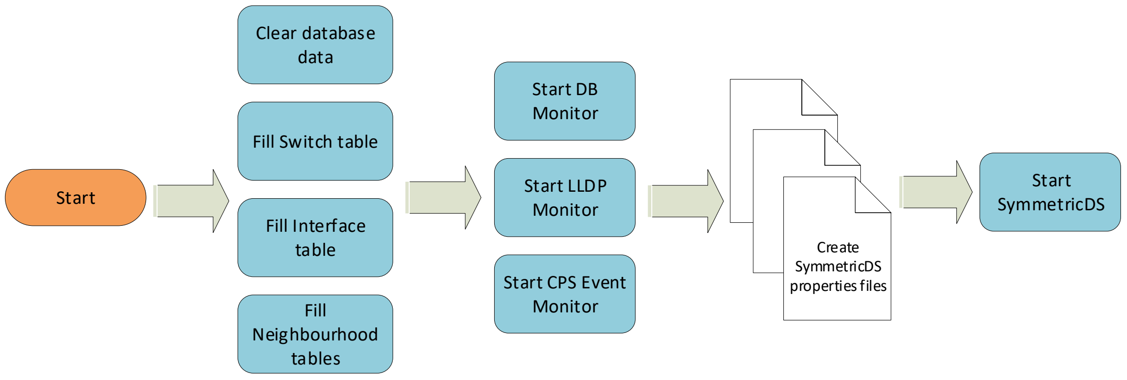

Figure 4.6: The bootstrapping sequence of a switch

4.7 Conclusions

This ends the description of the prototype IP Database-based Control Plane. As is easily recognizable, the prototype is organized in modules and functions that fulfil the DBCP model.

However, it lacks several optimizations that will be addressed later, namely what concerns the LLDP and the neighbourhood discovery process.

The optimizations that concern the database replication process will be subject of later research. This justifies why we only adopted an orthogonal solution that must be considered provisional.

Chapter 5

Prototype evaluation

5.1 Introduction

In this chapter, we present the results of the experimental evaluation which was carried out to assess the implementation of the system presented in the previous chapter. The test environment and the characteristics of the tests are first detailed. Then follows the presentation of the tests that were performed. Finally, the conclusions of the tests are discussed.

5.2 Evaluation methodologies and environment

In order to carry out the evaluation of the developed components, the environment in which they were executed is presented. As described in the previous chapter, the switch and controller were implemented as virtual machines. As such, to perform the tests, a virtual environment provided by VMware Workstation 14 [24] was used.

Each switch corresponds to an independent virtual machine. The hardware characteristics of each virtual machine corresponding to a switch are described in Table 5.1. Each switch is composed of the OpenSwitch OPX operating system, *SwitchDBCP* software, presented in the previous chapter, SymmetricDS software with version 3.9.2 and MySQL version 5.7.2. Also Java 8 and Python 2.7 were used to support the execution of the different software modules.

Table 5.1: Virtual machines configuration

Virtual CPUs	8
RAM	1.5GB
Storage	16GB

For the controller, a Debian operating system was used to support the execution of the *Network Control Services*, also described in the previous chapter. Alongside those services, the controller also contains a MySQL Server version 5.7.2, SymmetricDS software with version 3.9.2, Java 8 and Python 2.7. The controller uses the same hardware configuration as the switches, as described in Table 5.1.

The tests were performed using a single host machine which supported the execution of multiple virtual machines and virtual networks. The hardware and operating system of the host machine is described in table 5.2. The hardware of the host machine limited the size of the emulated network, mainly due to the limited *RAM* available to execute a larger number of virtual machines.

Table 5.2: Host hardware characteristics and operating system

Operating System	Windows 10 64-bit
Motherboard	Asus P8Z68-V PRO
CPU	Intel core i7 2600k 3.90GHz
RAM	Kingston 12.0GB
Storage	HDD WDC 5400RPM

The same network topology was used for all the tests. This network is composed of five switches, one controller and seventeen IP prefixes. Figure 5.1 shows a representation of the network topology. The number of switches used was limited to 5 due to the limitation of the host's available RAM. The configuration of the switches' interfaces was a manual process accomplished using the Linux API for interface management. This process could be enforced by the controller but this option was not considered due to lack of time. Since the network was built on a single host machine, links latency are negligible, therefore for this evaluation the impact of latency was not considered. Also, there was no packet-loss introduced in any of the links.

The evaluation of the system had a substantial focus on observing the execution times of the various elements composing the network. The times, extracted from logs and created by software, use the local clock of each operating system. A need to make several timing comparisons between switches and the controller, lead to use the Network Time Protocol[12] for all virtual machines.

Several Python scripts were developed to support the testing process. These scripts provide automation for the execution of the tests and support for the process of collecting and processing of the obtained samples.

5.3 Evaluation

5.3.1 Evaluating the system

The evaluation of the system consisted of the measurement of the execution times of its various components. The main focus was on the network convergence time and how the

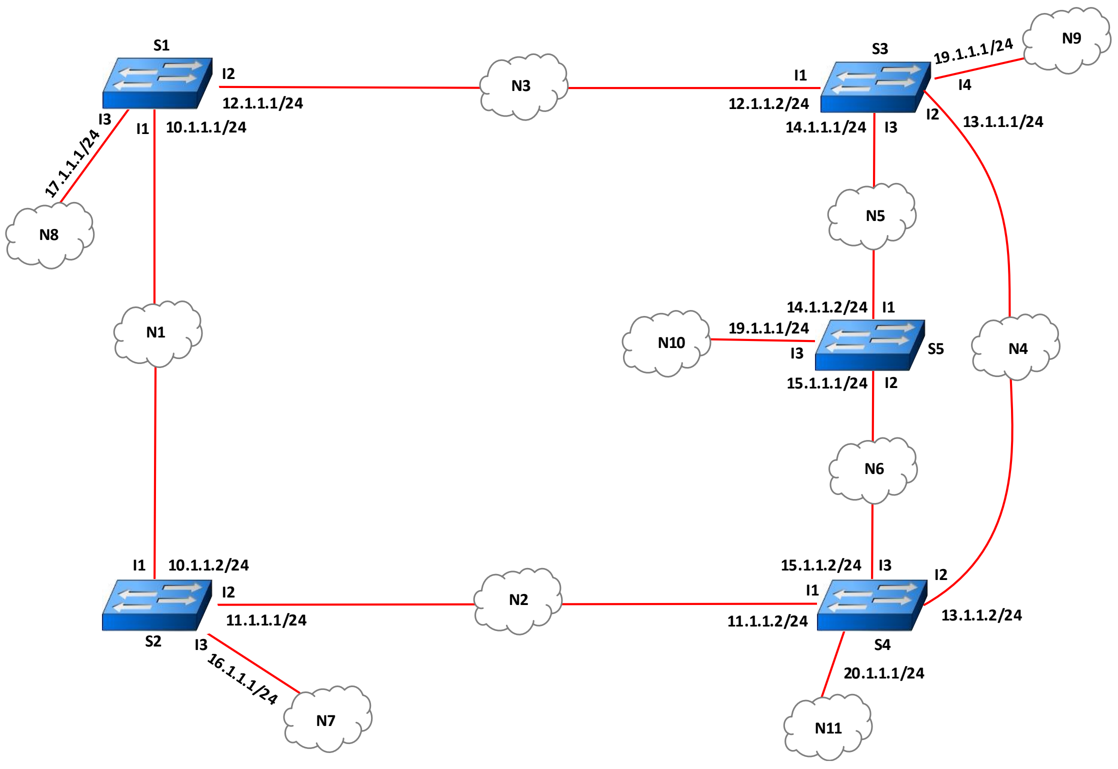

Figure 5.1: The production network topology

developed software and data replication utility impacts the convergence time of switches in the presence of a link or interface state change. This analysis allowed us to determine which components have a greater impact on the convergence time and thus understand the advantages and limitations of this IP-DBCP approach.

A state change of an interface or link causes the reaction of switches which in turn propagate this change to the controller. The controller is in charge of making decisions (building new routes) and consequently updating all the switches in response to those changes. In these tests, we considered two possible changes to the network's state: the change of interfaces' and link's state.

The event of an interface state change leads to the following sequence of operations: this event is first detected by the kernel of the switch where it occurred. OPX services, running on the switch, detect the change and trigger an CPS event that is detected by *SwitchDBCP*. *SwitchDBCP* collects and processes the event information. It then updates its local database, namely changes the *oper-status* field on the *interfaces* table. SymmetricDS, running on the switch, senses the change in the local database and replicates those changes to the local controller's database.

In the controller, the *DB Listener* detects the modification in the database. As the modification corresponds to the state change of an interface, this causes the execution of the convergence algorithm, causing the computation of new routing rules. At the end of the algorithm execution, the *ipv4-rib* table of the local controller database is updated to reflect the new defined routes. SymmetricDS, running on the controller, detects these updates and replicates those to the switches' databases. On each switch, *SwitchDBCP* detects the change in the local database through the *DB Listener*, processes the new updates and executes a downcall to the CPS OPX services. The OPX services then update the RIB table.

Based on the previously described sequence of operations, the convergence time of the switches, after a state change of an interface in the production network, can be characterized as follows:

$$NetworkConverganceTime = F + FE + SR + C + (CR + SD + RIB)_{lastSwitch} \tag{5.1}$$

where:

F = Elapsed time since the occurrence of the state change up to the creation of an event by the OPX CPS API by the switch where the state change occurred;

FE = Elapsed time since the CPS API event until the last insertion in the switch's local database;

SR = The database replication time from switch to the controller's local database;

C = The total time the controller took to react to the database changes and deploy new configuration back to the database;

CR = The time the database replication process took from the controller to a local switch's database;

SD = Interval since the end of the replication process and the downcall to OPX CPS API;

RIB = OPX time to update the RIB.

In a different scenario, where the creation of a new link exists, the process is similar to the previous one discussed. However, initially this process is triggered by *SwitchDBCP* due to the execution of the LLDP protocol. When the LLDP protocol detects the existence of a new neighbour, *SwitchDBCP* collects this change and updates the switch's local database.

We can then characterize the convergence time when a new neighbour relation forms, as such:

$$NetworkConverganceTime = N + NE + SR + C + (CR + SD + RIB)_{lastSwitch} \tag{5.2}$$

where:

N = Elapsed time since the creation of the link up to the detection of the new neighbour by the LLDP protocol and creation of the event by *SwitchDBCP*;

NE = Elapsed time since the creation of the event up to the last insertion in the switch's local database.

5.3.2 Results

First, a test was performed to understand the impact of the data replication software on the time it takes to replicate data between the switch's and controller's databases. In addition, this test made it possible to compare 3 distinct SymmetricDS configurations (the batch creation and push intervals).

The test consisted of executing a script that, at 5 second intervals, performed an insertion on the switch's local database. The time when that operation took place was noted. On the controller side, another script collected the time at the end of the replication operation. The same data flow was applied in reverse order from the local controller's database, to a target database of a switch. 100 samples were collected for both cases and the mean value in milliseconds is shown in Table 5.3. Figure 5.2, shows the minimum, average and maximum values obtained in this test.

Table 5.3: Database replication time using different SymmetricDS software configurations

	switch to controller (SR)	controller to switch (CR)
batch creation: 200ms batch push: 500ms	390ms	390ms
batch creation: 100ms batch push: 250ms	190ms	187ms
batch creation: 50ms batch push: 100ms	70ms	75ms

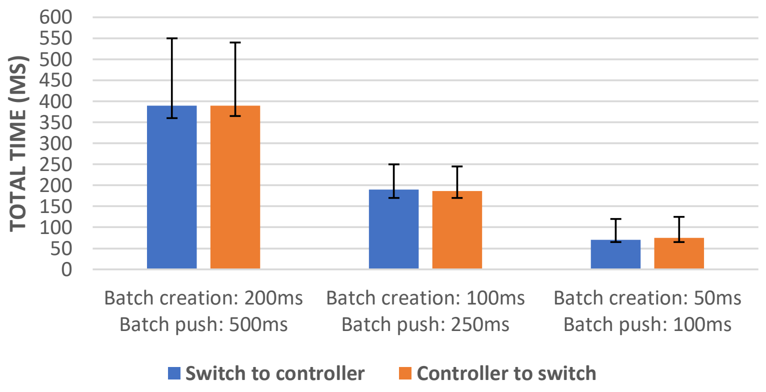

Figure 5.2: Database replication time using different SymmetricDS software configurations

A second test was carried out in order to understand the impact of the remaining components on the convergence time. This test was based on an interface's state change and consisted of automatically setting an interface to a *Down* state and measuring the elapsed time in each component.

In this test, a script was used to automatically, at intervals of 5 seconds, change the state of interface *I2* from switch *S2* between *Up* and *Down*, see Figure 5.1.

At the same time several other scripts were placed on switches and on the controller in order to collect the moments of execution of the various components. For this test, 100 samples were collected. SymmetricDS was used with the best configuration obtained in the previous test (batch creation: 50ms; batch push: 100ms). The *DB listeners* in the controller and in the switches were both configured with the value of 50 milliseconds.

Data was collected along the entire flow that makes up the convergence process in the presence of an interface state change. Using Equation 5.1 as reference, the values of *F*, *FE*, *SD*, *C* and *SD* were repeatedly collected. The value of *F* was calculated from the difference between the time of the interface state change, extracted from the script that caused it, and the time *SwitchDBCP* received the CPS Event. The value of *FE* is the difference between the time of the last insertion on the database, provided by MySQL, and the CPS event. The value of C_1, from the end of the replication process to the start of the convergence algorithm. C_2, from the total time of execution of the algorithm. C_3, from the end of the algorithm's execution to the last inserted update on the database.

Figure 5.3 shows the average results obtained in milliseconds. The image also shows the values of *SR* and CR obtained on the previous test. The value of the variable *RIB* on this test was not measured and was assumed to be similar to the value of *F*.

Another test was conducted to measure the impact of the LLDP protocol used to detect and acquire neighbourhood data from switches. The objective of this test was to understand the impact of the protocol when detecting neighbourhood relation changes, which is a fundamental process in the creation of a network graph. In this test we also considered different configurations' parameters supported by the protocol.

For this test, the failure of a link and the creation of a new link were considered. The test consisted of executing a script that, at intervals of 5 seconds, changed the state of interface *I2* of switch *S2* between *Down* and *Up* states. In switch *S4*, another script was also placed to collects the logs related to the generation of new events by *SwitchDBCP*, related to *LLDP Monitor*. In this test, the times in *S2* and *S4* were collected and compared in order to extract the elapsed time. Three different types of configurations of the LLDP protocol were used, namely for the attribute *transmit delay*, which controls the interval between sending of LLDP packets, and *transmit hold* (TTL) the number of failures made to trigger an event. These were configured with values to the second so the smallest configurable value was 1 second. In this test, 100 samples were collected. Table 5.4 shows the results obtained.

Table 5.4: Test results from the LLDP protocol.

	Link *down*	Link *Up*
Transmit delay: 4 Transmit hold: 30	97.185s	3.782s
Transmit delay: 2 Transmit hold: 10	13.645s	1.567s
Transmit delay: 1 Transmit hold: 2	2.118s	0.932s

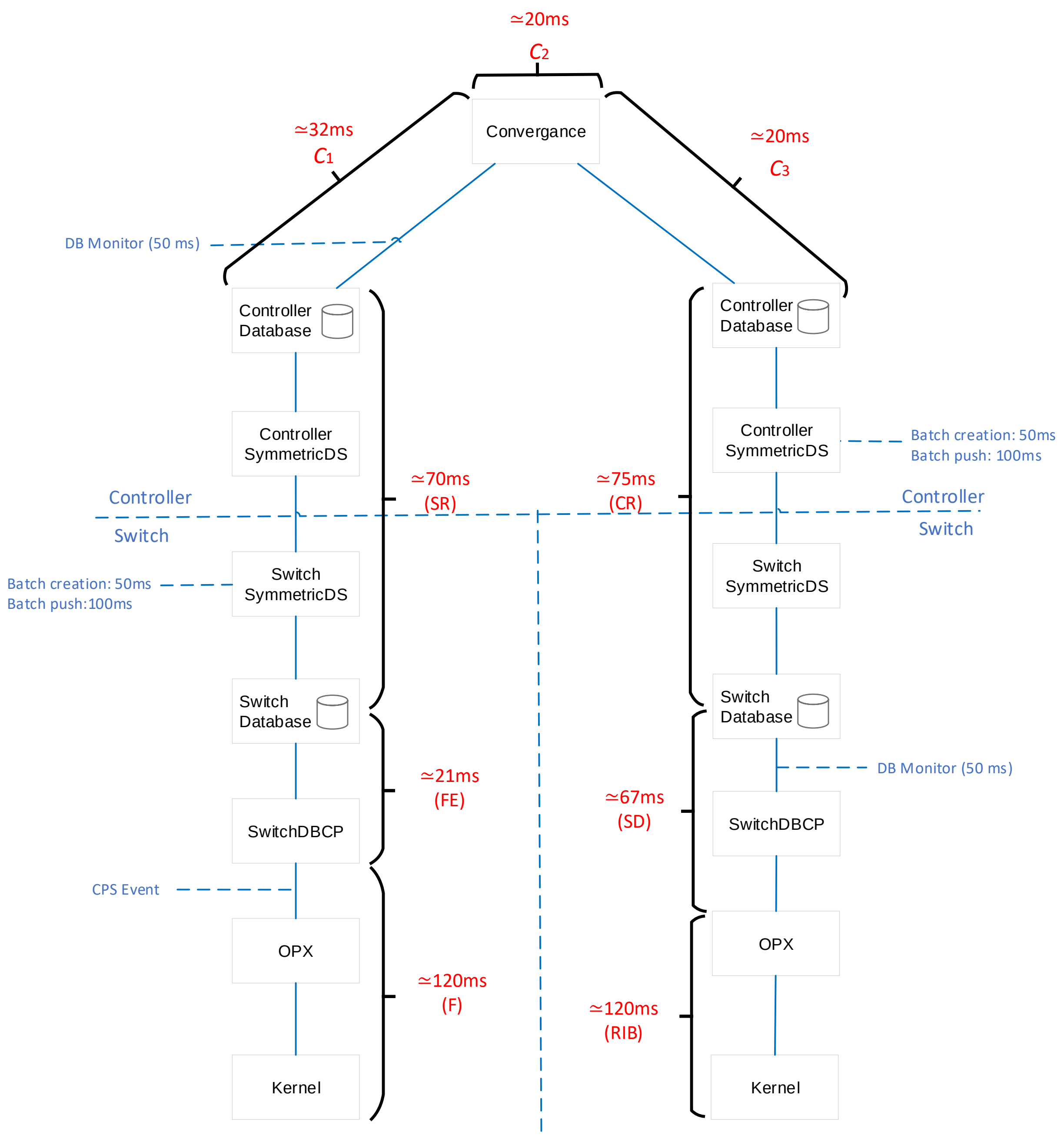

Figure 5.3: Network convergence events flow after an interface state change

Another more general test was carried out in order to better understand the real convergence time of the network. This test consisted of the analysis of the convergence time of the network in the presence of changes at any point in the network. The objective of this test was to obtain the mean total convergence time by analysing several possible cases of network state changes.

A script was developed that in a pseudo-random way selects a switch and one of its interfaces, and changes its state. This test started with a complete and fully operational network as described in Figure 5.1. The selected interface's state is changed to a *Down* state, and after 5 seconds, its changed back to an *Up* state.

To get the actual convergence time, only the time of the last switch to apply the new routes created by the controller is considered.

In this test the following configurations were used: *SymmetricDS* was used with the best configuration obtained (batch creation: 50ms; batch push: 100ms); *DB listener* on controller and switches were configured with a value of 50ms; and finally, the LLDP protocol was configured with the values of *Transmit delay*: 1 *Transmit hold*: 2.

500 samples were obtained and the results are presented in the Figure 5.4. The first column of the table shows the average time elapsed since the change in the network until the update is replicated to the controller database (*N* + *NE* + *SR* or *F* + *FE* + *SR*). The second column represents the mean time since the end of replication to the end of the processing of the routes by the controller, and their insertion in the database *(C)*. The third column shows the average last switch update time (*CR* + *SD* + *RIB*). Finally, the last column shows the total observed average time, that is, the exact elapsed time since the network state change took place and the last switch was updated.

	State change reaching the controller database	**Controller reaction (C)**	**Last switch convergence**	**Total convergence time (Observed)**
New link	1.145s (N+NE+SR)	0.068s (C)	0.361s (CR+SD+RIB)	1.575s
Interface Down	0.297s (F+FE+SR)	0.089s (C)	0.398s (CR+SD+RIB)	0.785s
Total average	0.720s	0.077s	0.368s	1.160s

Figure 5.4: Test results from the total convergence time of the network

5.3.3 Evaluation Conclusions

The observation of the values obtained from these tests allows the drawing of several conclusions regarding the performance of each component as well as the whole system. The various implementation options had a significant impact on performance.

First, it is possible to conclude that SymmetricDS, based on the best tested configuration, introduces an average delay of about 70 milliseconds from the switch to the controller and 75 milliseconds from the controller to the switch. Given the lack of latency in the tested environment, this value can be justified by the way SymmetricDS operates. Since data replication is not immediate, SymmetricDS operates based on periodic database queries. There is, therefore, a time interval between each batch creation and push that has to be taken into account. It was also possible to observe some fluctuations of the obtained results in relation to the mean value. As Figure 5.2 shows, cases were observed where the replication time reached 120 milliseconds (batch creation: 50ms, batch push: 100). These fluctuations were expected given that in the worst case it is expected to reach 150 milliseconds.

Given that latency is negligible, this figure composes poorly with the times reported in [8] for the same propagation. However there is ample space for optimisations in these implementation.

In the second test, the various times that compose the network convergence process were registered. First, this test showed that there is a delay of 120 milliseconds from changing the interface's state until the creation of the CPS event by the OPX services (*F*). Compared with the remaining values obtained in this test, this value is significantly higher and has a greater impact on the total time. The reason for this obtained value is dependent on the kernel itself and the OPX services which is the reason why, in the context of this work it was not possible to justify this result. After the creation of the CPS event, up to the end of the database insertion (*FE*) an average value of 21 milliseconds was obtained. This value essentially consists of the time taken by the database update. It has been observed that the average execution time of updates to the database of a switch, during the operation of the network, is in average of 18 milliseconds. This value can also be justified by the fact that a persistent database was used which could be improved since persistence is not required. This figure also compares poorly with the 10 ms reported in [8] but can also be improved.

In the controller, the value of (*C*) was divided into three separate times. The average time of C_1 is 32 milliseconds. This is essentially composed by the delay introduced by the *DB Listener*. It works on the basis of periodic database queries to detect and obtain modifications. In this test the *DB Listener* was configured with a 50 millisecond value. This result is also influenced by the database query time. The mean time of C_2 obtained was 20 milliseconds and corresponds exclusively to the execution time of the convergence algorithm. The average value of C_3 is 20 milliseconds and is related to the cost of inserting into the database the results obtained from the execution of the convergence algorithm. These values seem reasonable and hard to compare with those reported in [8].

The average elapsed time from the switch's database update, up to the downcall of an update to the OPX (*SD*), was 67 milliseconds. Again, this time is affected by the *DB Listener* configured with the 50 millisecond interval. To the listener time, the collection of the database updates and their processing time is added.

As stated before, the value of (*RIB*) on this test was not measured and was assumed to be similar to the value of (*F*).

The last test performed, allows us to conclude that the total time of the network convergence is in average 0.795 seconds when the status of an interface goes down, and is 1.575 seconds when a new link is introduced.

This test shows that there is a noticeable difference between the network convergence time, when there is a change in the interface's state, and when a new link is created. As observed in the Figure 5.4, the convergence time in the case of a new link is on average 0.790 seconds longer than the state change of an interface.

This difference can be justified by the way the network topology acquisition model was implemented. While the interface state change is detected locally on the switch and sent directly to the controller, the detection of a new link is dependent on the LLDP protocol and its probe execution process. The timings are similar to the ones found with the Hello protocol of OSPF, and are better than the ones that would have been observed if the LLDP functionality had been implemented by the controller. As shown in the Table 5.4, link formation takes an average of 0.932 seconds to be detected with the configuration used in this test. This delay is shown in the first column of the Figure 5.4. After the creation of a new link, updates take an average of 1.145 seconds to arrive at the controller database, whereas the change of state of an interface takes 0.297 seconds.

Still in the context of this test, it is important to refer the results obtained from the test of the LLDP protocol, namely the values shown in Table 5.4, referring to a link failure event. These values are not reflected in the results of Figure 5.4. This is because the implementation of the switch does not use the LLDP protocol to detect link failures. Instead, the interface's state change is used.

Still in Figure 5.4, the values of the second column referring to the total reaction time of the controller were on average 77 milliseconds. This is a value that we can consider good, though, the number of network prefixes and switches is small. However, as noted earlier, this value could be improved by opting for another method of detecting local database changes.

The IP-DBCP model here tested and implemented, can only be partially compared to the traditional model. Firstly, in the traditional model, the network state change detection is done locally by the switches using hardware mechanisms or protocols such as the Hello protocol [8]. This can in fact be compared to IP-DBCP which also implements a local switch network state detection model. However, in the traditional model the network state change is flooded, which differs from the IP-DBCP model where this change is only sent to the controller. Therefore, it's difficult to compare both. The processing of network convergence also differs in both cases. In the traditional model the processing is sequential and repeated on all switches. In IP-DBCP this processing is done only by the controller. After the updates processing, the IP-DBCP model performs an additional step, not performed by the traditional model, which consists of sending the updates from controller to each switch. In the traditional model, as the computation was done locally, the updates can immediately be applied. Finally, we considered that the process of local application of the new updates can be made similar in the two models. We still do not know if the times obtained with IP-DBCP are unbearable, given potential future improvements.

However, we believe that the most relevant differences between the two approaches is in fact concentrated in the state update and propagation process now in charge of the database. In the next chapter we will return to this analysis.

Chapter 6

Conclusions

6.1 Concluding Remarks

In this dissertation we described a work performed with the goal of analysing if a database approach would be suitable to replace OpenFlow in an SDN wide area environment, with the sub-goals of increasing the semantic level of the coordination model between controllers and switches, in terms of data description and also in terms of the coordination semantics among the different parts of this distributed system.

The first chapter introduced the motivations for this work as well as the description of the document structure and its contents.

The second chapter introduced the concept of Software-Defined Network, its architecture, applications and challenges. The OpenFlow protocol was also discussed as well as how it relates to this work. Next, the OSPF protocol was discussed and finally, other works were described which served as the basis for this work.

In the third chapter, the DBCP model was introduced. This chapter described the architecture of DBCP, namely its components, operation and the associated data model, but also the problems that the DBCP propose to solve. A comparison was also made between the DBCP model and others using OpenFlow.

In the fourth chapter, the prototype implementation, called IP-DBCP and based on DBCP, was described. This chapter described the implementation made, the technologies used and the replication model chosen.

Finally, in chapter five, the tests performed and the corresponding environment, were described. Finally, the test conclusions were presented based on the results obtained.

The first conclusion of this work is that it shows it is possible to make an SDN controlled network where the semantics of the data shared between switches and the controller are of an higher level then OpenFlow and can be easily modified and extended, since it is based on a relational data model, a popular framework in all information systems settings.

This approach has been tested with the problem of routing by the shortest path in an IPv4 network, but can be easily extended to more sophisticated forms of routing and network management (e.g. traffic engineering). It is also suitable to acquire the status of the network since it is derived from already defined YANG data models for network management. However, it doesn't rely on the introduction of any new description language.

The second conclusion is related with the fact that we do not need to introduce any new protocol for data, events and commands sent among the different components of the system. Again, these features were implemented using database replication protocols. Although the ones used are only suitable for a proof of concept, the implementation is agnostic to the introduction of more realistic and sophisticated ones since those protocols are under the responsibility of the database engine replication mechanism. They can be changed without the need of any change in the definition of the network protocols details. Also, the semantics of that coordination can be clearly improved in the future.

In what concerns the performance of this proof of concept implementation, it is hard to compare a preliminary experience with the current fully engineered and optimized solutions used in bullet-proof implementations of OSPF or IS-IS. However, the same optimizations that are used in those implementations can also be introduced in this one (fault detection, event propagation, status changes, incremental algorithms, ...) but the following: route computation and communication protocols among the switches and the controller and vice-versa.

All route computations are centralized in the controller instead of being done in parallel in the different switches. However, this is exactly the price of the SDN approach. Current high-end servers configurations seem suitable to deal with this difference.

In what concerns the higher overhead of the coordination protocols, which will never be comparable to the tailor-made fault-tolerant floods used by most network protocols, it is yet too soon to evaluate the trade-offs of both approaches since the DBCP approach is competing, in this regard, with a tailor made reliable flooding protocol. The full evaluation of the trade-offs requires the analysis of case studies where an SDN approach can excel and where more sophisticated data and coordination models are required.

6.2 Future Work

In what concerns the future work, there are many avenues to explore:

- A more sophisticated and optimized implementation of this proof of concept case study.

- The introduction of different database replication protocols

- More demanding case studies where this approach would pay since traditional available flooding protocols wouldn't be enough.
- Further development of the prototype to support more network functionalities.
- Extending the data models to support more network functionalities, devices configuration and statistics.

Bibliography

[1] M. Bjorklund. *The YANG 1.1 Data Modeling Language*. RFC 7950. Aug. 2016. DOI: 10.17487/RFC7950. URL: https://rfc-editor.org/rfc/rfc7950.txt.

[2] M. Bjorklund. *A YANG Data Model for Interface Management*. RFC 8343. Mar. 2018. DOI: 10.17487/RFC8343. URL: https://rfc-editor.org/rfc/rfc8343.txt.

[3] M. Bjorklund. *A YANG Data Model for IP Management*. RFC 8344. Mar. 2018. DOI: 10.17487/RFC8344. URL: https://rfc-editor.org/rfc/rfc8344.txt.

[4] 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*. ACM SIGCOMM Computer Communication Review, 2017.

[5] O. Networking Foundation. *Software-Defined Networking (SDN) Definition*. https://www.opennetworking.org/sdn-resources/sdn-definition. Accessed: 2017-05-02.

[6] O. Networking Foundation. *SDN architecture, Issue 1, ONF TR-502*. Version 1. Open Network Foundation, 2014.

[7] O. Networking Foundation. *SDN architecture, Issue 1.1, ONF TR-521*. Open Network Foundation, 2016.

[8] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure. "Achieving Sub-second IGP Convergence in Large IP Networks." In: *SIGCOMM Comput. Commun. Rev.* 35.3 (July 2005), pp. 35–44. ISSN: 0146-4833. DOI: 10.1145/1070873.1070877. URL: http://doi.acm.org/10.1145/1070873.1070877.

[9] "IEEE Standard for Local and Metropolitan Area Networks– Station and Media Access Control Connectivity Discovery." In: *IEEE Std 802.1AB-2009 (Revision of IEEE Std 802.1AB-2005)* (2009), pp. 1–204. DOI: 10.1109/IEEESTD.2009.5251812.

[10] 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.

[11] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. "Software-Defined Networking: A Comprehensive Survey." In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76. ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2371999.

[12] J. Martin, J. Burbank, W. Kasch, and P. D. L. Mills. *Network Time Protocol Version 4: Protocol and Algorithms Specification*. RFC 5905. June 2010. DOI: 10.17487/RFC5905. URL: https://rfc-editor.org/rfc/rfc5905.txt.

[13] J. McCauley, Z. Liu, A. Panda, T. Koponen, B. Raghavan, J. Rexford, and S. Shenker. "Recursive SDN for Carrier Networks." In: *SIGCOMM Comput. Commun. Rev.* 46.4 (Dec. 2016), pp. 1–7. ISSN: 0146-4833. DOI: 10.1145/3027947.3027948. URL: http://doi.acm.org/10.1145/3027947.3027948.

[14] J. Moy. *OSPF Version 2*. RFC 2328. Apr. 1998. DOI: 10.17487/RFC2328. URL: https://rfc-editor.org/rfc/rfc2328.txt.

[15] NetworkX. *NetworkX Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks*. https://networkx.github.io/. Accessed: 2018-02-01.

[16] B. A. A. Nunes, M. Mendonca, X. N. Nguyen, K. Obraczka, and T. Turletti. "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks." In: *IEEE Communications Surveys Tutorials* 16.3 (2014), pp. 1617–1634. ISSN: 1553-877X. DOI: 10.1109/SURV.2014.012214.00180.

[17] OpenSwitch. *OpenSwitch CPS API*. https://github.com/open-switch/opx-cps/. Accessed: 2018-01-10.

[18] OpenSwitch. *OpenSwitch OPX*. https://www.openswitch.net/. Accessed: 2018-01-03.

[19] OpenSwitch. *OpenSwitch OPX Base image source*. https://github.com/open-switch/opx-docs/wiki/Run-virtual-machine. Accessed: 2018-01-05.

[20] OpenSwitch. *OpenSwitch OPX Wiki*. https://github.com/open-switch/opx-docs/wiki/. Accessed: 2018-01-10.

[21] B. Pfaff and B. Davie. *The Open vSwitch Database Management Protocol*. RFC 7047. Dec. 2013. DOI: 10.17487/RFC7047. URL: https://rfc-editor.org/rfc/rfc7047.txt.

[22] SymmetricDS. *SymmetricDS open source database replication software*. https://www.symmetricds.org/. Accessed: 2018-02-01.

[23] VMware. *VMware Infrastructure Architecture Overview*. https://www.vmware.com/pdf/vi_architecture_wp.pdf/. Accessed: 2018-01-15.

[24] VMware. *VMware Workstation 14*. https://www.vmware.com/support/workstation.html/. Accessed: 2018-02-01.

[25] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali. “On scalability of software-defined networking.” In: *IEEE Communications Magazine* 51.2 (2013), pp. 136–141. ISSN: 0163-6804. DOI: 10.1109/MCOM.2013.6461198.