Software Defined Networking for Community
Network Testbeds

Emmanouil Dimogerontakis

A thesis submitted in partial fulfilment of the requirements for the degree of
European Master in Distributed Computing

Thesis Committee

Supervisor: Leandro Navarro Moldes

July, 2013
Acknowledgements

First of all I would like to thank my amazing supervisor Leandro, whose support and "day-dreaming" were fundamental in this effort. Johan Montelius also contributed vitally to my effort, since if it was not for him EMDC would not exist. Additionally, I would like to give special thanks to an excellent researcher, Ivan Vilata i Balaguer who spend quite a lot of hours discussing with me.

On a more personal level, I would like to thank again Leandro, for all the non-technical support and understanding. A huge thank you also goes to all my EMDC classmates that made the last two years an unforgettable adventure. From them, during my thesis project stand out my workmate and poet Ioanna and holy Maria, with whom I shared many rough thesis nights having beers in Placa del Sol. Moreover, it feels natural to thank my parents since without them this thesis neither me would exist.

Last but certainly not least, I thank my beautiful distraction, that spiced up my thesis life.

Barcelona, July, 2013
Emmanouil Dimogerontakis
"ici un bout d’un souffle
ici un bout d’un rêve
tu ne sais pas quoi faire, quoi dire”

Rupa and the April Fishes
Abstract

Community Networks are a part of the Networking field that has received increasing attention the last years. In an effort to set the cornerstone for an internet without central authorities and monopolies, network engineers throughout the world have started creating community networks. To enhance this effort, Community-lab, a community networks testbed, was created. Using Community-Lab, researchers are able to experiment with new protocols and applications for community networks in a realistic environment. Nevertheless, this testbed does not offer the ability to perform L2 experiments. To address this gap, we decided to develop a system that allows Community-Lab researchers to perform L2 experiments. Moreover, we decided to reach our goal using Software Defined Networking (SDN) techniques, their promise for a complete networking solution and due to the attention they received lately. As a result, we propose an architecture that allows researchers to perform L2 experiments in a community networks testbed. We implemented this architecture for Community-Lab using the OpenFlow SDN protocol, enabling researchers to manage their own L2 experimental topology.
Categories and Keywords

Categories and Subject Descriptors

H.3.4 [Systems and Software]: Distributed Systems

Keywords

Community Networks
Software Defined Networks
Network Virtualization
Network Testbeds
Index

1 Introduction 1
   1.1 Motivation ............................................. 1
   1.2 Research History ................................... 2
   1.3 Structure of the Document ......................... 2

2 Background 5
   2.1 Introduction ........................................... 5
   2.2 Wireless Community Networks ..................... 5
      2.2.1 Wireless Mesh Networks ....................... 5
      2.2.2 Community-Lab .................................. 6
   2.3 Software Defined Networking ........................ 8
      2.3.1 Software Defined Networking ................ 8
      2.3.2 OpenFlow ....................................... 8

3 Challenges & Related Work 11
   3.1 Introduction .......................................... 11
   3.2 Challenges ........................................... 11
      3.2.1 Wireless Mesh Networks ....................... 11
      3.2.2 Community Networks and Community Network Testbeds 12
   3.3 Related Work ......................................... 13
      3.3.1 SDN in Wireless Mesh Networks ............... 14
      3.3.2 SDN in Rural and Heterogeneous Environments 15
      3.3.3 SDN in Mobile Networks ....................... 16

4 Architecture 17
   4.1 Introduction .......................................... 17
   4.2 Architecture Overview ............................. 17
      4.2.1 Decisions ....................................... 17
5 Implementation

5.1 Introduction .................................................. 23
5.2 Poxy - A proxy for POX OF controller ..................... 23
  5.2.1 POX .................................................. 24
  5.2.2 Poxy .................................................. 24
5.3 Pongo - An integration of POX, Django and Community-Lab .... 26
  5.3.1 Django .................................................. 26
  5.3.2 Pongo .................................................. 27
5.4 External Software .............................................. 27
  5.4.1 CONFINE Software ....................................... 28
  5.4.2 Open vSwitch ........................................... 29
  5.4.3 Batmav-adv ............................................. 29
5.5 L2 experiments in Community-Lab .............................. 30
  5.5.1 Controller ............................................... 30
  5.5.2 Nodes .................................................. 30
  5.5.3 Typical System Execution ................................ 31

6 Evaluation & Discussion ........................................ 35

6.1 Introduction .................................................. 35
6.2 Evaluation .................................................... 35
  6.2.1 Functional Evaluation .................................... 35
  6.2.2 Performance Analysis .................................... 37
    6.2.2.1 Communication Overhead ............................. 37
    6.2.2.2 Computation Overhead ............................... 38
6.3 Discussion .................................................... 40
  6.3.1 Tackling the Challenges ................................ 40
  6.3.2 Distributed properties of the system .................... 41
  6.3.3 Generalization for CNs or WMNs ......................... 42

7 Conclusion ........................................................ 43

7.1 Conclusions .................................................. 43
7.2 Future Work .................................................. 43

A Poxy README .................................................... 45
B Pongo README

B.1 Django .................................................. 47
B.2 Poxweb application .................................... 47
B.3 Integration with Community-Lab ..................... 48
List of Figures

2.1 Typical WMN architecture. ....................................................... 6
2.2 Community-Lab architecture. ................................................ 7
2.3 Abstract view of Community-Lab architecture. ........................... 8
2.4 OpenFlow idea. ................................................................. 9
2.5 Idealized OpenFlow Switch. .................................................. 10

4.1 The two possible approaches for achieving L2 connectivity. .......... 20
4.2 Overview of the architecture. .................................................. 22

5.1 Basic idea of Poxy ............................................................. 25
5.2 Overview of the implementation design. .................................... 31
5.3 User view of the topology. ..................................................... 32

6.1 Main page of Pongo. ............................................................. 36
6.2 View of the slivers (above) and the links between the (below) from Pongo. . . . . . . . . . . 36
6.3 Deleting a link from Pongo. ................................................... 36
6.4 The two main points of interest for communication overhead (grey areas). .... 38
6.5 Architecture of the server. ..................................................... 39
6.6 Architecture of the node. ...................................................... 39
Acronyms

CN  Community Network
WCN  Wireless Community Network
WMN  Wireless Mesh Network
L2  OSI Layer 2
L3  OSI Layer 3
SDN  Software Defined Networking
OF  OpenFlow
OFP  OpenFlow Protocol
FOSS  Free and Open Source Software
1.1 Motivation

Community Networks (CNs) are a very good example of decentralized distributed systems. CNs are directly related with the existence of social community networks. In fact they can be described as grassroots computer networks for communities. Their liberal foundation as well as their services are a free good for community members and follow the model of a healthy social community. Built and maintained by their own users, CNs abolish the idea of central authorities and introduce the idea of freedom and equality. Furthermore their environment contains a large amount of hard and interesting challenges to be resolved, fact always welcome by researchers.

In order to create new infrastructures and technologies that would promote and enhance CNs there was the need for an environment where researchers could deploy and test their ideas. Simulation is an established way of performing research but cannot provide realistic enough results. Thus, a testbed for CNs would be an ideal candidate for research. These are the reasons Community-Lab\textsuperscript{1} was created. Community-Lab is a networking testbed which runs on top of existing CNs. That way the CNs researchers can experiment in an realistic environment.

In the process of familiarizing with Community-Lab we encountered the need to allow experimenters to perform L2 experiments. To achieve that goal we would need a way to perform L2 topology virtualization and thus present to the researchers not a low level API but a set of abstract and manageable L2 resources. There is a wide variety of ways to achieve L2 topology virtualization. Nevertheless, we can say that a very prominent approach is the one of Software Defined Networking (SDN), as we explain later. Virtualizing the network resources using SDN does not only provide on the benefits of handling resources. The SDN community is working on a complete solution where except from using virtualizing resources users will be able to configure them and manage them automatically. This is the reason why we chose OpenFlow Protocol (OFP), an SDN protocol, to be one of the basic components of our architecture.

\footnote{Community-Lab \url{http://community-lab.net/}}
CHAPTER 1. INTRODUCTION

As a result of all the above, our effort was focused in creating a component for CN testbeds, like Community-Lab, where experimenters would be able to handle their L2 topology and experiment with it. Furthermore, we want to use the OFP as to virtualize the existing L2 resources, as it can lead to a very complete solution. This approach sets the environment for the creation a generic SDN system used for network experiments in CN testbeds, as also layers above L2 will be easily handled.

1.2 Research History

The initial step of this thesis was to study the characteristics and design of Community-Lab and trace problem areas. During our study we identified the research gap concerning L2 experiments. In order to see how we can deal with this problem we performed brief study of the possible L2 virtualization techniques reaching the conclusion that an SDN approach should be used. Taking that into account, we investigated the SDN technologies and the existing research concerning deploying SDN in CN-like environments. Using the obtained information, we designed the architecture of our system. To proceed, we had to familiarize with the external software used and develop our prototype system. Finally, we performed the evaluation of our system.

This work was performed in the context of the CONFINE\textsuperscript{2} European project. During my work, I benefited from the fruitful collaboration with the remaining members of the DSG team working on CONFINE and more especially Navaneeth Rameshan, Ester López, Davide Vega D’Aurelio and Marc Aymerich from Pangea.

1.3 Structure of the Document

The rest of this document is organized as follows. Chapter 2 provides an introduction to the different technical areas related to this work. Chapter 3 describes the fundamental challenges related to our work and an overview of the related research. In chapter 4 we introduce the architecture of our system. Chapter 5 describes our effort to map the proposed architecture to a prototype implementation. Chapter 6 presents the evaluation of our system and a more abstract

\textsuperscript{2}CONFINE \url{http://confine-project.eu/}
1.3. STRUCTURE OF THE DOCUMENT

discussion about our system and our choices. Finally, Chapter 7 concludes this document by summarizing its main points and future work.
2.1 Introduction

In this chapter we provide some theoretical background about ideas and technologies that are used in our effort to reach our goal. In the following section we introduce Wireless Community Networks and Community-Lab. In section §2.3 we introduce the ideas of Software Defined Networking and OpenFlow.

2.2 Wireless Community Networks

Community Networks (CNs) are community-owned open local IP networks. A CN is open to all members of the community and can provide many services with more important, usually, the Internet service. Nowadays most of the existing CNs have a wireless infrastructure and we are going to assume that is an important part of the CN environment. These CNs are called Wireless CNs (WCNs). For the rest of this document, when we refer to Community Networks, we will imply Wireless Community Networks.

CNs are often designed to be self-organized and self-configured. Their architecture can be very similar to Wireless Mesh Networks (WMNs). More specifically, CNs are similar to WMNs that are deployed in rural environments, but expanded to support a social/community aspect. It is a common case that CNs are a set of interconnected WMNs. Thus, a main technical difference between CNs and WMNs is the diversity of protocols and devices used in the former ones. This is one of the major challenges in managing a CN.

2.2.1 Wireless Mesh Networks

To understand better how CNs function we will describe briefly a typical WMN architecture. Akyildiz and Wang, in [?], describe WMNs as dynamically self-organized and self-configured wireless networks, where nodes establish multihop ad-hoc mesh connectivity. The authors present figure 2.1 as a hybrid WMN, where mesh clients can access a the WMN through
other mesh clients. Two types of nodes participate in WMNs: mesh routers (can be also gateways) and mesh nodes. This WMN architecture is closer to CNs.

The multihop ad-hoc environment cannot be handled efficiently by the standard IEEE 802.11\(^1\) stack or ad-hoc protocols. Thus, many wireless protocols are revised to support WMNs and new WMN-specific protocols are created. Among them BMX6\(^2\), B.A.T.M.A.N advanced\(^3\) and IEEE 802.11s\(^4\) are some of the most promising.

![Figure 2.1: Typical WMN architecture.](image)

### 2.2.2 Community-Lab

Community-Lab[?] is a CNs testbed, set up by the CONFINE\(^5\) European project. Community-Lab is a testbed over existing CNs where researchers are able to conduct their experiments. Researchers are able to test their own protocols and applications deploying them on real community networks. Of course, more outcomes are to be expected from this effort, like the expansion of already existing community networks, the broadening of our knowledge about CNs and the development of open-source software for the testbed.

In order to create a sufficiently large testbed where thousands of experiments can run in parallel there is the need for many CNs to be interconnected. This is achieved by connecting ex-

---

\(^1\)IEEE 802.11 [http://standards.ieee.org/about/get/802/802.11.html](http://standards.ieee.org/about/get/802/802.11.html)


\(^5\)CONFINE [http://confine-project.eu/](http://confine-project.eu/)
2.2. WIRELESS COMMUNITY NETWORKS

Existing CNs, like GUIFI\(^6\) and AWMN\(^7\), with existing networking infrastructures for experiments, like FEDERICA\(^8\). This is depicted in figure 2.2, which shows an overview of the Community-Lab architecture. Furthermore, Community-Lab allows the researchers to perform experiments at different network levels. As a result, it must deal with heterogeneous nodes as well as with varying wireless link technologies and routing protocols.

As can be seen in figure 2.2, three main entities exist in Community-Lab: the testbed server, the testbed nodes and the testbed gateways. The testbed controller maintains information about the users of the testbed, the experiments and the testbed nodes. The testbed gateways allow the inter-CN communication. The testbed nodes are the devices that host the experiments. All these entities are interconnected through the management network. This interconnection is usually established through the underlying CN links. The researcher can create and deploy his experiment through the testbed server and the testbed nodes.

![Community-Lab Architecture](image)

Figure 2.2: Community-Lab architecture.

Figure 2.3 depicts an abstract view of the testbed architecture. Every node has a community device, through which it connects to the CN, and a research device where the experiments are run. The ideas of slivers and slices from Planetlab are deployed, as it represents a successful network testbed and offers interesting ideas for testbed architectures. Each experiment is represented by a slice, which is a collection of slivers. A sliver consists of the isolated resources associated with a slice inside a node.

---

\(^6\)GUIFI [http://guifi.net/en](http://guifi.net/en)

\(^7\)AWMN [http://www.athenswireless.net/](http://www.athenswireless.net/)

\(^8\)FEDERICA [http://www.fp7-federica.eu/](http://www.fp7-federica.eu/)
2.3 Software Defined Networking

2.3.1 Software Defined Networking

Software Defined Networking (SDN) is an approach to networking that aims to create abstractions for Networking that were missing for a long time. The main idea is that a network should be seen as a set of resources. SDN proposes that all network components should be translated into a set of resources reachable by a defined API. That way, layers of abstraction can be introduced, simplifying the life of network researchers and developers. This approach tries to introduce in the Networking field the abstraction techniques used in many Computer Science fields, like Computer Architecture or Operating Systems. The most prominent application of SDN currently, is Network Virtualization, since virtualization is strongly connected with the idea of abstraction. To this date, there exists only one main representative of the SDN approach, the OpenFlow protocol.

2.3.2 OpenFlow

OpenFlow[?] (OF) is an SDN protocol, which creates abstractions of L2, like switches. In order to achieve that, OpenFlow consists of 3 main components: the OpenFlow data structures, the OpenFlow API and the OpenFlow protocol. The OpenFlow API defines the way the users can interact with the resources. The directives given through the OpenFlow API are translated into OpenFlow data structures. Then, the OpenFlow controller is responsible to distribute the information contained in the data structures to the network devices using the OpenFlow protocol.
2.3. SOFTWARE DEFINED NETWORKING

protocol. All these components are described in detail in the OF specification [?].

The key idea behind OpenFlow is the separation of control plane and data plane. As shown in figure 2.4, the control plane is moved from the network resource, here the switch, to a controller. Thus, the switch becomes a dump packet switching device and the controller handles all the logic of the control plane.

![OpenFlow idea](http://bradhedlund.com/2011/04/21/data-center-scale-openflow-sdn/)

**Figure 2.4:** OpenFlow idea.

Figure 2.5, as shown in [?], presents a typical OF setup. The OF controller provides an API to the user through which the forwarding logic can be implemented. Typically, as depicted, the switch uses a different communication channel for the data plane and the OFP. The Flow Table stores the flow rules that are send from the remote controller. The switching happens according to the rules installed in the Flow Table. In a real deployment usually an out-of-band communication channel is used for the OFP communication.

Flow control with OF can either be reactive or proactive. In the reactive approach, the switch informs the controller for every new incoming packet that arrives and waits a reply with the action to be taken. In the proactive approach, the controller populates the flow tables of the switches beforehand and is not concerned about every new flow. In case the proactive approach is used there is no often communication needed between the switches and the controller. Thus, the communication overhead is small and in case that communication between the switch and the controller is lost the system can still function correctly. On the other hand, the reactive approach can offer a more dynamic way of handling the flows, increasing though the latency of the system and the control plane traffic.

OF has undoubtedly evolved since 2008 that it was initially proposed. The OpenFlow Switch Specification 1.3.0[10] is the latest specification providing an increased level of expressiveness to

---


[10] OF Switch Specification 1.3.0 [http://emmdim.pc.ac.upc.edu/of/openflow-spec-v1.3.0.pdf](http://emmdim.pc.ac.upc.edu/of/openflow-spec-v1.3.0.pdf)
the OF user. Nevertheless, the new OFP versions are not yet supported by hardware switches since the porting is not trivial.

As far as the management plane is concerned, there are two competing protocols compatible with the OFP, OF-CONFIG\textsuperscript{11} and OVSDB\textsuperscript{12}. Both of these protocols are designed to act as management protocols, like NETCONF\textsuperscript{13}. They are still quite immature. Nevertheless, the existence of such management and provisioning plane make the SDN solution very complete in handling network resources. Thus, it can be said that the SDN community is seriously trying to provide a complete and correct abstraction environment for Networking. More importantly, SDN promises the ability to programmatically reconfigure the resources, offering a dynamic approach on virtualization. From this point of view, we believe that OF can be used to implement Network Virtualization in a very complete way.

Throughout the rest of this document, when we refer to SDN we will mainly imply OF, since at the current moment is the main SDN protocol.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{Idealized_OpenFlow_Switch.png}
\caption{Idealized OpenFlow Switch.}
\end{figure}

\begin{footnotesize}
\begin{itemize}
\item \textsuperscript{11}OF-CONFIG\url{http://emmdim.pc.ac.upc.edu/of/of-configdot0-final.pdf}
\item \textsuperscript{12}OVSDB \url{http://tools.ietf.org/html/draft-pfaff-ovsdb-proto-02}
\item \textsuperscript{13}NETCONF \url{http://tools.ietf.org/html/rfc6241}
\end{itemize}
\end{footnotesize}
3.1 Introduction

In this chapter we present the main obstacles encountered in our effort to design a system as described in §1.1 and how some of them have been addressed in related research. The following section explains the challenges of using SDN techniques to perform L2 experiments in CN testbed environments. In section §3.3 we present a brief state of the art with approaches on the challenges above in environments that resemble CN testbeds, or CNs in general. It is worth to mention that there is no related research as far as CNs are concerned, while the relevant topic of SDN in WMNs has only recently received attention from researchers.

3.2 Challenges

Deploying the OFP in CNs and WMNs is a challenging topic. Additionally, we have to resolve problems that are related with the nature of testbeds. In this section we try to enumerate the most important of these challenges. It is worth mentioning, that by default, the OFP protocol is designed to manage a homogeneous set of switches that ideally can behave like they are part of a wired environment.

3.2.1 Wireless Mesh Networks

As mentioned earlier, CNs have many similarities with WMNs. Thus, the challenges we present in this section apply also in CNs. The main characteristics of the wireless channels are the origins for many the problems faced in WMNs. Here we refer to the most common WMN problems and we explain how they affect our effort.

Challenge 1: Link Quality Instability

Wireless link quality is highly unstable, especially in the case of WMNs were distances between nodes can get significant. Kim and Shin in [?] explain that links in WMNs, due to their
deployment in large and heterogeneous areas, often experience significant quality fluctuations and performance degradation or weak connectivity. This instability is translated to the WMN with a high node churn rate. As a result, we have to assume that in our network there are going to be often topology changes.

**Challenge 2: Link Capacity**

The wireless link capacity is known to be significantly smaller than the one of the wired links, one to two orders of magnitude. Akyildiz and Xudong in [?], do not mention it clearly, but strongly imply that modelling and improving link capacity is one of the major concerns in WMN research. As this problem does not seem to be solved, WMN routing protocols try to minimize their control plane traffic.

The OFP on the other hand, was designed for wired networks and it does not take into account this kind of restrictions. Using the OFP for managing control plane in WMNs may introduce significant overhead. The often communication between the switches and the controller may both reduce the available bandwidth but also prevent the correct function of the OFP service. Even if the OF is used in proactive mode, when control plane traffic is supposed to be minimal, the communication is often due to the to the frequent messages sent by the switches to check if the controller is alive.

**3.2.2 Community Networks and Community Network Testbeds**

CNs share the WMN problems, since they are actually implemented as WMNs, but they also add extra challenges. One of the greater cause of problems in CNs is the diversity of hardware, software and protocols used.

**Challenge 3: Device and Protocol Diversity**

A characteristic of WMNs is their heterogeneous nature, as mentioned also before. This is a fact also for CNs but not only in terms of physical links, also in term of devices as highlighted in [?], by Neumann et al. . This is a great obstacle in finding a global strategy to handle L2 resources. The homogeneity that the OFP assumes increases further the difficulty of the situation.
3.3. RELATED WORK

Challenge 4: Communication with Non-Testbed Nodes

Inside a CN we can assume that there can exist some kind of homogeneity. For example standard routing protocols that are agreed to be used network-wide or in specific islands of nodes. In a CN testbed created on top of existing CNs, though, new nodes have to be introduced in various locations and various CNs. For example, a testbed can be deployed on top of multiple CNs, thus having to manage multiple kinds of nodes and protocols, as well as their combination applying minimal or no changes to them [?]. Thus, in the case of a CN testbed, it might be the case that all the testbed nodes are similar but this cannot be supported for the adjacent CN nodes.

This implication leads to a problem similar with the one faced in Challenge 3. For a testbed wide solution, we would need to manage a set of diverse L2 resources, in terms of links and nodes. This is a challenge that complicates the virtualization of L2 resources since it is hard to have a view of the complete L2 topology. Furthermore, the deployment of the OFP is harder since, as mentioned above, the OFP assumes an homogeneous environment with OF compatible nodes.

Challenge 5: Out-of-band Channels

When a CN environment is considered, we cannot assume that out-of-band channels are available. As we mentioned earlier in §2.3.1, it is important for the proper functionality of the OFP that it is used in a dedicated channel of communication. Thus, ideally, we would like an out-of-band channel for control plane (OFP) and an in-band channel for the data plane. This is, though, a requirement hard to meet.

3.3 Related Work

Our goal, as stated in §1.1, implies that the first and crucial step is to deploy the OFP in a CN testbed environment. The rest of the problems are going to be handled on top of this infrastructure. As a result, in this section, we present research related with deploying OF in CN-like environments. We have categorized the related work in three main sections: SDN in WMNs, SDN in rural and heterogeneous environments and finally SDN in Mobile Networks.
3.3.1 SDN in Wireless Mesh Networks

Before reviewing the state of the art we would like to mention that no one has addressed the problem of deploying the OFP in an heterogeneous environment like a CN. There exist approaches concerning deploying the OFP in WMNs but considering for example that all the mesh routers involved are OF compatible. Furthermore, most of the efforts are directed in using the OFP protocol to substitute the forwarding plane. In our case, we want OF to experiment in L2. The lack of existing similar approaches makes our effort more challenging but at the same time also more innovative.

The accomplishment most similar to our work, is presented by Dely and Kassler in [1]. This paper describes an architecture for WMNs where OF is used to substitute the forwarding plane, instead of existing routing protocols for WMNs like AODV, OLSR or B.A.T.M.A.N.. The nodes that connect to the mesh split their physical interfaces in two virtual ones, one for the control plane (OF) and one for the data plane. Each virtual interface has a different SSID, achieving that way the separation of control and data plane. The data plane IP connectivity is established through OF rules but the control plane IP connectivity should be established by a normal L3 mesh routing protocol. In their implementation OLSR is used in order to establish IP connectivity for the control plane virtual interfaces. The OF controller lies in another network which is accessible through the mesh gateways. Their architecture includes also a monitoring server, which communicates with monitoring agents on the node side and collects information about link quality and channel utilization. This information, together with data from OLSR are provided by the monitoring server. The OF controller can query the monitoring server and use the data in testing a new routing protocol. Furthermore, their architecture was evaluated as an experiment in a real WMN testbed where they identified with two worth mentioning results. Firstly, slow mesh routers can present problematic behaviour handling big number of rules. Secondly, using OF instead of OLSR in the forwarding plane when there is a considerable number OF rules introduces significant overhead, which increases proportionally to the number of rules. Dely and Kassler’s approach is quite complete taking into account the WMN environment. Their proposed architecture tackles Challenges 1,2 and 5 but in a different context. Their approach intents to use the OFP in L3 and assumes that all the mesh routers are OF compatible, implying a homogeneous environment.

Chung et al. in [2], based mainly on Dely and Kassler’s architecture, present their efforts to deploy OF in WMNs. Their work is focused in presenting the difficulties they faced, concerning
more hardware obstacles. Though their experiments are not extensive enough, they suggest that Dely and Kassler’s approach show acceptable performance comparer to using 802.11s in the forwarding plane. More importantly, they conclude, that due to performance issues OF should not be used to substitute mesh networking protocols, but to provide additional functionality in the WMN.

Nguyen in his Master’s Thesis, [?], proposes three adaptations for OF in WMNs. Firstly, take a proper action when an OF controller realizes that the link in one of the OF switches ports is down. This allows dynamic reconfiguration according to the topology. Second, instruct the switch to work in standalone mode when the connection with the controller is not present and provide backup controllers to deal with the instability of the WMN links. Third, reduce messages that transfer statistics from switches to controller and the messages from the switches that check the health of the controller. The first adaptation can actually be implemented in the custom implemented controller, but nevertheless encounters the topology instability caused by Challenge 1. The second and the third adaptation are interesting technical approaches in order to face Challenge 2.

3.3.2 SDN in Rural and Heterogeneous Environments

Hasan et al. in [?], discuss deploying SDN techniques in rural networks in order to allow these networks to operate as infrastructure providers for ISPs. The paper describes the opportunities and challenges involved in this effort. While rural networks present many similarities with CNs, this work approaches the problem in an abstract a theoretical way without providing any solutions.

A similar but more generic approach was presented in [?], Mendonca et al., in a students’ ACM, conference. The authors provide an abstract description of how should SDN techniques be applied in heterogeneous WMNs. They define the main use cases, a set of requirements that heterogeneous SDN would need to enable the use cases and from those requirements they derive a set of research challenges to be confronted. Unfortunately no architecture or technical advices are proposed, but we consider that their effort is important in encouraging an discussion that would lead to a faster establishment of this research domain.
3.3.3 SDN in Mobile Networks

In [?], Yap with some of the OF prime movers, are presenting OpenRoads, an OF based platform for research in mobile networks. This platform targets primarily mobility experiments. OpenRoads uses OF to control the data plane and SNMP to control the device configuration. To our understanding, OpenRoads is made to be deployed in controlled environments with a provided homogeneous infrastructure. It seems that except maintaining topology, and more specifically only client topology, this project contains no contributions relevant to our work.

As it is shown above, SDN in WMNs or related fields are quite immature. The list of related papers is quite small and the solutions provided, except from [?], are addressing very specific problems with very specific solutions. Also, we can notice, as we claimed before, that there is no actual research in deploying OF in heterogeneous environments. We hope that our effort will be able to contribute in the evolution of this topic.
4.1 Introduction

In the previous chapter we traced the existing gaps in research, as far as our goal is concerned. In this chapter we describe the architecture we propose that allows the deployment of L2 experiments in a CNs testbed using SDN technologies. In the following section we describe and present our basic decisions concerning the architecture and we also present the architecture figure.

4.2 Architecture Overview

To present properly our architecture we have to introduce first the environment where the experiments will take place. We assume a testbed architecture similar to Community-Lab, which is a valid assumption since it deploys approaches from already existing successful testbeds like Planetlab modified to fulfill the needs of WMNs and CNs. Thus, our environment is formed by testbed nodes, community nodes, a testbed server and testbed gateways. The experiments actually take place in the testbed nodes but the traffic can traverse also community nodes. The testbed nodes have at least two network interfaces. Through one interface they connect to the testbed management network and also reach the testbed server. Additionally, each node should have at least one for local wireless connections to communicate with adjacent community nodes. The users connect to the testbed server to create and deploy their experiment. When a user wants to create an experiment he creates a new slice and some slivers that belong on this slice. The slivers are going to be run as VMs inside testbed nodes.

4.2.1 Decisions

Assuming the environment described above, we want to extend the testbed functionality and provide to the experimenters the ability to perform L2 experiments using SDN techniques. The specific category of experiments we chose to address is the ability to manage the L2 topology.
 CHAPTER 4. ARCHITECTURE

of all the slivers that belong in a slice. We want to remind to the reader that this is just an example that will drive our design, but it is certainly not the only kind of experiment that our architecture can support. Having described all that, we will now proceed to present our architecture explaining the most important decisions we made. The presented decisions can be arranged in three categories: the basic, which concern the necessary infrastructure, the functionality, which add the desired functionality to the system, and finally the optimizations, which enhance the performance of the system. We present them in that order, starting from the basic decisions.

Decision 1: OF Switches on the host side of testbed nodes

Since we want to support SDN for experiments, each sliver should have its own dedicated OF switch. These switches could either be located inside the sliver or in the host. It was our choice to place the OF switches in the host so that users would use OF without having to care about technical details. Also, this approach allows OF to be used from testbed administrators to manage network resources of slivers, for example to deploy QoS policies.

Decision 2: OF Controller in Testbed Server

To administer the OF switches we need an OF controller. The OF controller should have connectivity with all the OF switches and hence the related slivers. Furthermore the controller should be accessible from the user as it is the entity through which the switches can be managed. Taking all these into account, together with the fact that our system should be deployed as a testbed service, we decided to place the OF controller in the testbed server. The testbed server is accessible from all the testbed nodes and additionally provides an interface to the user to manage his experiments.

We have now placed the most important entities of our architecture. What remains is to provide the functionality of L2 experiments and attempt to optimize the architecture taking into account the environment.

Decision 3: L2 mesh routing protocol for multihop L2 connectivity

The lowest level accessible from the user is L2. Thus, we need to achieve L2 connectivity between the slivers. In an environment as described above the non-testbed (community) nodes
make this step non-trivial. Assuming we want to let the experimenters define their own L2 virtual topology for their slivers and in the future even adjust the physical L2 links we need to provide a L2 connectivity between the slivers, hence the testbed nodes. As we mentioned before we assume that the node has at least two network interfaces, a management and a local one. Thus, we will now explain which interface and what technique we believe is the most suitable to achieve L2 connectivity between the slivers.

If the management interface is used we would have to create a L2 overlay over the existing management overlay. In case the management overlay functions on L2, it already introduces significant traffic overhead in the CN. In case the management overlay functions on L3, like in Community-Lab, L2 tunnels could be created on top of the L3 overlay. That way, we would create a L2 overlay on top of the L3 management overlay. This approach, depicted on the left stack in the figure 4.1, has several drawbacks. First, we would have to use the management connection both for control plane and data plane. Some isolation mechanism like VLANs would have to be used to separate the different planes. Alternatively an approach to use multiple virtual interfaces on top of the management interface could be used, like proposed in [?]. Creating a L2 overlay with an isolation mechanism on top of the management network would require modifications to the existing testbed infrastructure and would significantly affect the performance not only of our service, but also other testbed services due which would be unacceptable. Second, with this approach there is no direct mapping between the service L2 topology and the actual L2 topology, which would affect the quality and realism of the users experiments. For example, with such an approach it would be difficult to allow the user to handle the physical underlying or to find information about it that would help the simulations. For all these reasons, we decided to reject an approach where L2 connectivity would be achieved through L2 overlays on top of the management network.

The second option involves exploiting the local network interface. This approach is more sensible since we do not have to build a L3 network and then degrade it to L2 using an overlay. On the other hand there exists the problem of establishing the L2 connectivity with nodes that direct connection does not exist. At the same time we have to take into account the diversity of existing nodes. To make the testbed realistic, the traffic should traverse non testbed community nodes. On the other hand, the CN environment allows us to assume that there will be islands where mesh routing protocols are deployed. Thus, we require that the slivers of the experiment are placed in nodes that belong to the same island. Many mesh routing algorithms exist, and
the mesh routing protocols is a domain of active research with main competitors 802.11s, OLSR and BXM6. Nevertheless all these algorithms work on L3 which leads to a scenario similar as using the management network interface. What we would ideally need is a mesh routing protocol that provides connectivity in L2. This at the current moment can be achieved by two protocols, IEEE 802.11s and B.A.T.M.A.N-advanced. This approach is depicted on the right stack in the figure 4.1. A L2 mesh routing protocol creates ideally a fully connected L2 network of the participating nodes. So at the IP layer, every packet sending operation is seen as a one-hop communication. These kind of overlays are also called L2.5 overlays. In the previous approach there was no mapping of the L2 overlay with the physical nodes and links. With this approach there exists a trivial mapping between the upper L2 overlay and the physical nodes, and a more difficult one, but of reasonable complexity between the upper L2 overlay and the physical links. Also, performance-wise, we delegate the responsibility to the L2 mesh routing protocol.

![Figure 4.1: The two possible approaches for achieving L2 connectivity.](image)

From our point of view, the second solution seems to fit our needs in a great level. Nevertheless, there are some drawbacks to consider. The drawbacks are related with the existing L2 mesh routing protocols. Firstly, creating a full L2.5 overlay creates some performance overhead. This is the cost in order to maintain L2 connectivity even when some links fail. Another downside concerns the mapping of L2.5 links to actual physical links. For example, if a user decides to drop a link between two slivers this does not guarantee that the link is not going to be used by the mesh routing protocol. This raises some concerns about the realism of the experiments. It would be ideal to map the changes done by OF to the L2.5 overlay but this is not supported by the current L2 mesh routing protocols.

Now that the required functionality is supported we describe some optimization decisions.
4.2. ARCHITECTURE OVERVIEW

Decision 4: Use OF in proactive mode

As we presented in the related work, §3.3.1, using the OFP in the control plane can create significant overhead to the system, depending on the number of flow rules. In the background, §2.3.2, we mentioned that the OF controller can be programmed to act in proactive or reactive mode. Using OF in proactive mode can significantly reduce control plane traffic. Furthermore, a proactive approach does not eliminate totally the dynamic property of the system, since the controller can still decide to change remove or add rules based on other inputs different than the traffic incoming to the switches.

Decision 5: Local Proxy OF controller in testbed nodes

Using OF in proactive mode we minimize the control plane communication, that concerns exchanging flow rules. Nevertheless, the communication between the OF switch and the OF controller contains also some type of management plane communication. The switch periodically send requests to the controller to ensure that he is alive. Moreover, the controller periodically requests statistics from the switches. To minimize the overhead of this type of communication we introduce a new idea in the SDN community. The idea of the proxy OF controller. The proxy OF controller is collocated with the OF switch and performs all these tasks in a local level. Thus, the switch does not know the health of the central controller and the statistics are gathered by the local proxy controllers. Since we have actually decoupled the switches from the central controller we can design our own asynchronous communication system to do health checks and gather statistics.

Decision 6: Control plane through management interface, data plane through local interface

This decision is a derivative from the previous decisions, we wanted though to state it more clearly. The control plane communication will take place locally between the OF switch and the local proxy OF controller and remotely between the central OF controller in the testbed server and the local proxy controller. The remote communication will take place through the management network interface, since this is the interface the nodes use to communicate with the testbed server. The data plane communication will take place through the local interface, through which the testbed node can access other community nodes in the same island through the L2.5 overlay.
Taking all the previous decisions into account we present our architecture in the figure 4.2.

Figure 4.2: Overview of the architecture.
5.1 Introduction

In this chapter we describe our effort to prototype the proposed architecture. The goal was not only to produce a proof-of-concept system, but also create stable and reusable tools. As a result, while proving the feasibility of our architecture we also contribute to the researchers community.

Our implementation consists of two main components. Poxy, a proxy for the POX OF controller and Pongo, an attempt to integrate POX, Django and Community-Lab. Poxy plays the role of the local proxy as described in § 4.2.1. Pongo, on the other hand, is a centralized service that corresponds to the specifications described in §4.2.1. The two components are totally independent but have the ability to communicate using the OFP.

These software components together with Open vSwitch, a software OF switch, batman-adv, a Linux kernel module that implements the B.A.T.M.A.N. advanced protocol, and the CONFINE software collection are the technologies that enabled us to deploy a prototype system, proving the feasibility of our idea.

In the following sections we describe the software components we developed, provide a brief overview of the rest of the technologies that were used and finally explain how all these were used and deployed proving that our system can function in real environments.

5.2 Poxy - A proxy for POX OF controller

Poxy\textsuperscript{1} implements a proxy for the controller-switch OFP connection, on top of the POX OF controller.

\textsuperscript{1}Poxy https://github.com/emmdim/Poxy
5.2.1 POX

POX\textsuperscript{2} is a, FOSS licensed, programmable OF controller written in Python. As with all the OF controllers, POX provides an API where each user can program his switches building an application that would define their forwarding plane using the OFP. The functionality of the applications ranges from turning the switches in simple hubs to creating custom routing algorithms.

Internally POX is divided in two abstract components: the core functionality and a basic OF controller. The core functionality creates and maintains all the Python environment needed for the controller. The OF controller provides the most basic OF functionalities like accepting and handling switch connections. The most important part of the POX OF controller, as far as the user is concerned, is a set of events that represent OF events, for example the connection of a new switch or the arrival of an OF packet from a switch. These events can be handled from the user to provide the desired functionality.

POX was chosen as base to built the local proxy controller according to the following criteria. Firstly, as explained also in the evaluation, §6.2.2, the performance of the local controller is not critical compared to other parts of the system. The functionality of the local proxy is minimal and consists basically in communicating with a limited amount of local switches. The manner of communication would be the same no matter of the controller choice. This allowed us to choose based on the learning curve, simplicity and expressiveness. After a small research, we reached the conclusion that POX was the more suitable OF controller according to this criteria.

5.2.2 Poxy

Poxy is a, FOSS licensed, proxy for OF controllers based on POX. Poxy gives the ability to place a basic or custom POX OF controller in the middle of every OF connection since it acts as a transparent proxy. Since it is based on the OF protocol, Poxy is not restricted by which OF controller or which OF switch lies on the other side. As show in figure 5.1, it acts as a controller on the switch side and as a switch on the controller side. What is not shown in the figure, is that currently Poxy can handle many OF switch connections that are connected to the same OF controller and, also, many OF switch connections to different OF controllers. The implementation is publicly available and it can run as a standalone application outside the scope of this project.

\textsuperscript{2}POX \url{http://www.noxrepo.org/pox/about-pox/}
5.2. POXY - A PROXY FOR POX OF CONTROLLER

Poxy is consisted of two main components which are presented below.

A local OF Controller

This part of Poxy is used to communicate with switches. Poxy uses as its OF controller the existing basic OF controller of POX overriding some of its functionality. This controller is responsible to establish the connection with the switch and propagate to the switch the messages that arrive forwarded from the proxy. At the same time, it listens for incoming packets from the switches and it propagates them to the proxy. Furthermore, the controller maintains all the information about the connected switches, their features and their state. In a few words, the controller is responsible to establish and maintain connections with the switches, to store switch information and to share all these with the proxy.

A Proxy

The proxy implements the main behavior of Poxy. On the one end, it forwards packets from the OF switch to the remote OF controller. This is achieved by receiving and redirecting the packets that arrive to the local OF controller through the OF connection. One the other end, the proxy has to simulate a switch connection to the remote controller. The POX controller does not use information from protocols of network layers lower than the OFP (though it can when it has to) so a connection with a switch can be totally simulated without being an actual
CHAPTER 5. IMPLEMENTATION

connection to the switch. To achieve that, it replicates and forwards to the remote controller the connection initiation messages that arrived to the local controller. These messages trick the remote controller to believe that on the other side of the connection lies the actual switch. After the initiation, any received messages from the remote controller are forwarded to the local controller, which is responsible to place them in the OFP datapath of the switch.

Poxy is written fully in Python and it can be used as a proxy in accordance with POX and all the POX applications that a user could develop. Some of its functionality is implemented overriding the basic OF controller provided by POX and some as a custom POX application. Nevertheless, the structure of the basic POX controller is not changed, thus the users, are able to program their own controllers as they would before, using Poxy as an additional application. While Poxy is far from a production phase, similar software does not exist up to now, except implementations for internal use like in a project called FlowVisor\(^3\), we believe that it is a very useful contribution to the SDN community, since we can imagine a lot of applications on top of it (for example security-oriented).

5.3 Pongo - An integration of POX, Django and Community-Lab

Pongo\(^4\) is an attempt to integrate POX with Django in order to administer L2 experiments in a collection of nodes.

5.3.1 Django

Django\(^5\) is a Python web framework that can be used in order to create web services. Django was used in order to provide a front-end to the users of our system. Additionally, this choice made the integration of our system with Community-Lab easier, since the testbed controller front-end is written in Django. The compatibility of our central controller with the Community-Lab controller is of great importance, since this testbed can allow us to quickly deploy and verify our ideas.

\(^3\)FlowVisor https://openflow.stanford.edu/display/DOCS/Flowvisor
\(^4\)Pongo https://github.com/emmdim/Pongo
\(^5\)Django https://www.djangoproject.com/
5.3.2 Pongo

Pongo is a, FOSS licensed, Python software component built on top of POX OF controller which acts as an extended OpenFlow controller. The extensions are efforts to integrate POX with Django and Community-Lab. Pongo provides a very simple interactive web interface to Community-Lab researchers that wish to perform experiments using our system. On the contrary with Poxy, Pongo is a normal POX application, without overriding POX main functionality.

The main functionality of Pongo lies in allowing the user to define his own L2 topology adding and deleting links between slivers through the web interface. The choices of the links are translated to OFP rules and through POX they are propagated to the switches. This feature assumes that L2 connectivity between the switches already exists, but the application can be easily reconfigured to work for L3 topologies. Through the web interface, a researcher can see his slivers and the corresponding OF switches. Moreover he can see the existing connections between the OF switches. Whenever a new OF switch is connected to POX, Pongo finds all the testbed related information and stores them to the Django database. Whenever an OF switch gets disconnected, the user can be informed from the web interface, where all the related information and related links are deleted. Overall, through Pongo a user can have information about the state of the L2 connections between his slivers and more importantly handle their L2 topology.

While this application is Community-Lab specific upto a level, it is a good starting point for a researcher that intents to administer a collection of OF nodes through a web interface. The idea of slivers does not necessarily associate our application with a world level testbed. Even a simple collection of nodes can be a collection of slivers. While not as standalone and complete as Poxy, we believe that this application is also an interesting product for the researchers community.

5.4 External Software

Above, we presented the software components we developed to prototype our system. We now briefly present other existing software that our system uses.
5.4.1 CONFINE Software

CONFINE software\(^6\) is a collection of FOSS components that allow everyone to deploy a CN testbed like Community-Lab. Two main subcomponents can be distinguished: the CONFINE node software system and the CONFINE controller software, as described below. As it is mentioned before, Community-Lab, and the CONFINE software collection, was chosen because it is the only testbed that can provide the correct infrastructure to test our system. Furthermore, extending its functionality we are able to contribute to the researchers’ community.

CONFINE Node Software System

CONFINE Node Software\(^7\) consists of a Linux *OpenWrt*\(^8\) image which is configured properly and contains the necessary packages in order to function as a research device(see 2.2.2) inside a CONFINE CN.

The node is a low-performance device that runs a customized OpenWrt image. Each node can host multiple slivers from researchers that conduct experiments. Each sliver is a virtual *LXC*\(^9\) host, created with the help of *libvirt*\(^10\). Isolation between the virtual hosts is achieved using *cgroups*\(^11\). To create the management overlay network, a *Tinc*\(^12\) VPN is used. Furthermore, each node implements a common REST API, which publishes publicly available information about itself, for example the slivers it hosts.

As far as networking is concerned, there are many types of network interfaces that a node can have but we present the ones related to our system. All the nodes have a management network interface, through which Tinc connects them to the management network. The second network interface of our interest is the isolated interface which provides direct wireless access to nearby community nodes.

CONFINE Controller Software

CONFINE Controller Software\(^13\) implements the software stack of a testbed controller. A CONFINE testbed controller offers many services, like slice management, sliver management

---

\(^{6}\)CONFINE Software [http://redmine.confine-project.eu/projects/confine](http://redmine.confine-project.eu/projects/confine)


\(^{8}\)OpenWrt [https://openwrt.org/](https://openwrt.org/)


\(^{10}\)libvirt [http://libvirt.org/](http://libvirt.org/)


\(^{13}\)CONFINE Controller Software [http://wiki.confine-project.eu/soft:server](http://wiki.confine-project.eu/soft:server)
and experiment creation. These services, and many more, are made available to the researcher through a REST API and a web interface built in Django.

5.4.2 Open vSwitch

Open vSwitch\textsuperscript{14} is a FOSS licensed software that implements an advanced edge switch. It is a virtual switch for Linux systems, that lies in the hypervisor layer and enables more fine grained switching for the VMs. Open vSwitch provides a superset of the Linux bridge functionality, while trying to maintain its performance and even improve it in cases. It supports a wide range of virtualization techniques. For our case, the most important feature is its integration with the OFP.

Open vSwitch can function either in Linux user space or in Linux kernel space. As it is obvious, the second option is much more efficient in terms of performance thus it is the solution we adopted. Open vSwitch is not the only existing software switch that supports the OFP. Nevertheless, it was the first to reach the production phase and is a very mature software component. Furthermore, Open vSwitch was created by the key people behind the OFP, fact that increases our trust on it. Moreover, when Open vSwitch was first released it had performance similar to Linux Bridge\textsuperscript{15} according to \cite{?}. Since then, Open vSwitch has been continuously improving, becoming a better candidate even for simple software switching.

5.4.3 Batman-adv

Batman-adv\textsuperscript{16} is a FOSS Linux kernel module implementing the B.A.T.M.A.N. advanced L2 routing protocol. Batman-adv creates a virtual network interface, on top of an already existing Linux interface, which can be considered as a usual ethernet device\cite{?}.

Batman-adv implementation is by far more mature than its competitor from the IEEE 802.11s standard. In 2010, Garropp et al. in \cite{?} claimed that both of them lack in functionality but batman-adv is generally more stable. In 2011, Chissungo et al. in \cite{?} supported that batman-adv network performance is acceptable only between nodes that are a small number of hops aways. In 2012, Quartulli et al. in \cite{?} suggest that batman-adv performance has improved significantly, as a result on a major change of the implementation approach. Considering that

\textsuperscript{14}Open vSwitch \url{http://openvswitch.org/}
\textsuperscript{15}Linux Bridge \url{http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge}
\textsuperscript{16}Batman-adv \url{http://www.open-mesh.org/projects/batman-adv/wiki/wiki}
802.1s is a slow evolving protocol and rather immature, we decided that batman-adv was our best candidate. Nevertheless, we have to bare in mind that despite the performance improvements batman-adv produces significant overhead in dense node areas and that it cannot scale for more than several hundreds of nodes.

5.5 L2 experiments in Community-Lab

All the software components and tools described above are combined in order to build a system for L2 experiments in for a CN testbed. The CN testbed is Community Lab and we are using an architecture which is depicted in figure 5.2. The two main entities are the controller and the node.

5.5.1 Controller

The controller is based on the CONFINE Controller software. The controller runs a different Pongo instance for each of the researchers that wish to conduct experiments using our system. The user can access the web UI through the web interface of the CONFINE Controller software. Pongo communicates with each node’s CONFINE node software system making HTTP requests to it. Also Pongo communicates with the Open vSwitches located on the nodes trough Poxy using the OFP.

5.5.2 Nodes

The nodes are based on the CONFINE Node software system. Each node runs one Poxy instance which can be used by different experiments at the same time. Each user’s sliver has a dedicated Open vSwitch. The main Poxy application does not forward traffic to multiple OF controllers, thus an extension was added that implements the idea of different users. The OFP traffic, the control plane, is routed through the management interface from Pongo. The data plane of the virtual switches is routed through the bat0 interface which is the virtual interface created by batman-adv. This way the separation of control plane and data plane is achieved. The nodes communicate with each other through the batman-adv overlay. For this, we have to assume that all the nodes of an experiment that uses our system run to batman-adv island. Since we assume batman-adv runs on L2 we assume that the OF switches of the researcher are fully connected between them. Thus, when a new switch is connected to Pongo all the links
with rest of the switches that belong to the same experiment are assumed to exist. Any data plane traffic is finally propagated through Open vSwitch inside the slivers.

5.5.3 Typical System Execution

Below we describe the sequence of actions that happen when a new OF experiment is started:

1. The user registers his experiment in the CONFINE controller through the web interface and activates the Pongo application.

2. The sliver instances are booted and the Open vSwitches are configured by the CONFINE
Node software system.

3. The Open vSwitches automatically connect to the local Poxy application.

4. Poxy redirects all the switch connections to the corresponding Pongo.

5. Pongo uses the REST API to extract from the CONFINE Node software system all the related information to each switch.

6. The Pongo interface is populated with the available slivers and the connections between them.

7. Pongo populates the switches flow tables with rules that allow the slivers to see only each other.

8. The user is able to monitor and manage his L2 topology.

Assuming the architecture depicted in 5.2 expanded in three Community-lab nodes, we present in figure 5.3 how the users would see their topology: Bob (on the left) and Alice (on the right).

If the users want to delete a link:

1. The users deletes the link that connects his slivers using the Pongo web interface in the controller.
2. Pongo creates the appropriate OF rules and sends them to the corresponding Poxy instances through the OFP connection.

3. The Poxy uses the local OFP connection to send the rules to the switches which populate their flow tables.

4. The link is now deleted, as far as the user can notice.

In a similar manner users can also create links that they have deleted previously.

This workflow is very simple and it was developed in order to prove that our system functions. It was not in the scope of our effort to implement and exploit all the possible scenarios. Nevertheless, it is extremely easily to add functionalities simply changing the Pongo app. An experienced Python and POX programmer should be able to add a functionality in no more than a couple of hours.
6.1 Introduction

In this section we assess our contributions, as they are described in §4 and in §5. We start in the first section, with the evaluation of our system. Then, in the Discussion section, §6.3, we review our architecture and implementation decisions under a critical prism.

6.2 Evaluation

Taking into account the fact that we have developed a new system there are two types of possible evaluation. First, we need to test that our system functions properly. Second, we need to perform experiments to test the performance of our system. We describe these steps below.

6.2.1 Functional Evaluation

We managed to set up our system in a virtualized environment with three testbed nodes and one testbed server. Then we set up our Pongo in the testbed server, the Open vSwitches, batman-adv and Poxy in each of the nodes, as described in §5.5.3. The view from Pongo web interface is shown in figure 6.1. After the installation, we had a view of our slivers (name: sliver\_id) and their connections (name: sliver1\_id_to_sliver2\_id) from the Django interface of Pongo as shown in figure 6.2. Then we created traffic between the slivers with id 266 and 267 which is routed successfully. Finally, by dropping the link from the Pongo interface (figure 6.3) the traffic is blocked. When the links are recreated through the Pongo interface the traffic is resumed.
CHAPTER 6. EVALUATION & DISCUSSION

Figure 6.1: Main page of Pongo.

Figure 6.2: View of the slivers (above) and the links between the (below) from Pongo.

Figure 6.3: Deleting a link from Pongo.
6.2.2 Performance Analysis

In this section we present a performance analysis of our system, tracing the bottlenecks and the performance critical parts. Based on this analysis we propose a set of experiments that would, in our opinion, support our arguments. We describe separately our analysis concerning the communication overhead and the computation overhead. It is left for future work to perform the evaluation performing the experiments.

6.2.2.1 Communication Overhead

As it is depicted in figure 6.4, there are two areas (in gray boxes) were the communication overhead should be studied.

Area 1 contains the communication that takes place in the management overlay. We want this communication to be very minimal, so that it wont interfere with the proper functionality of the testbed. The REST communication, that performs sliver info retrieval from the node, happens only once for each node, when the OF switch connects to the corresponding PONGO, so it is negligible. Moreover, OF is used in proactive mode so the communication is necessary only upon when a switch is connected, or the user decides to remove/add links. No other communication from our system take place in this area. Experimenting with network overhead in this area is not even needed as our system guarantees almost constant amount of message exchange.

Area 2 contains the communication that takes place in the island of the L2 mesh routing protocol. In this communication part there are no restrictions concerning interfering with the testbed or the community network. On the other hand, this part can easily become the bottleneck of our system. As we suggested in Decision 3 in §4, we delegate the performance concerns of these communication channels to the L2 mesh routing protocol. Up to the current moment, the implementations of L2 mesh routing protocols, even of batman-adv, have improved significantly, but it is our opinion that they are still not ready to support hundreds of nodes and large amount of traffic. Thus, the throughput of batman-adv links may degrade the performance of our system. We assume that the throughput decreases as the distance (in number of hops) increases. Also the control plane used by batman-adv adds overhead to the system but the implementation is designed in order to minimize this overhead. Performing some trivial file transfer experiments for one hop distance with and without batman-adv, in a wired-like environment we found that in the latter case throughput is of the same order of magnitude and only
slightly decreased. In general we expect that batman-adv itself does not introduce the overhead, but it is the wireless environment (for example wireless interference) that creates the problems. Furthermore, we cannot expect the same performance from a L2 multihop environment (virtual L2) and an L2 collision domain. An experiment proving our statement would be out of the range of this thesis, since it is related with the performance of routing protocols in wireless environments and not with our system. Nevertheless, a simple experiment could be performed, where our system is used on top of a switched wireless network (WLAN for example) compared with usage on top of a batman-adv network in a WMN environment. Furthermore, experiments that indicate the relation between number of nodes and throughput could be conducted.

Figure 6.4: The two main points of interest for communication overhead (grey areas).

### 6.2.2.2 Computation Overhead

Computationally, our system is separated in the server and the nodes.

The Pongo applications run on the controller, which is located on the testbed server. As it is depicted in the figure 6.5 and was explained in §5, there can exist multiple Pongo applications, one for each user. The testbed server is designed to run on powerful server hardware, so the computation overhead introduced by the Pongo applications is not important. Additionally, our choice to run OF in proactive mode makes the computation overhead even smaller. No
experiments are necessary for the computation overhead of the server part.

The nodes on the other hand, according to Community-Lab specifications, are low performance and low cost devices. For that reason, we chose to have only one Poxy application running on each node, as shown in figure 6.6. The communication between the Open vSwitches and Poxy happens using traditional Unix sockets which run on the user space. Nevertheless, this overhead is not that important, since it only involves some message exchanges every few seconds. On the other hand, the packet switching in the Open vSwitches and in the real network interface that lies below the batman-adv interface could be computationally intensive. The overhead introduced by the switches depends almost only on the amount of data traffic, which is natural and cannot be changed. As a result the only troubling point, is packet handling for the batman-adv control plane. As we explained previously, this concerns the performance and capabilities of L2 mesh routing protocols. The most interesting experiment concerning the computation overhead in the node would be to measure the overhead caused by the physical batman-adv interface and at the same time compare it with the actual data (payload) that are transmitted or received.
6.3 Discussion

In this section we review our system and argue about the success of our decisions and our general approach. We remind to the reader that our goal was to built a system on top of an existing CN network testbed where users could perform L2 experiments with the help of SDN techniques.

6.3.1 Tackling the Challenges

The main assumption of our architecture was that the environment of the testbed has some similar properties with Community-Lab, which are reasonable. Under this assumption we describe how our architectural decisions, presented in §4.2.1, tackle the main challenges as they are described in §3.2.

Challenge 1: Link Quality Instability

Solution:

The actual problem caused by this challenge was the frequent changes in the topology. In our case we actually care about the testbed nodes. Nevertheless, their L2 connectivity get affected by topology changes. Therefore, we let our L2 mesh protocol handle both the changes and the instabilities. We have to note here, the L2 mesh routing protocols are optimized for wireless network interfaces and are able handle in a smart way these kind of challenges. Thus, Decision 3 resolves this challenge.

Challenge 2: Link Capacity

Solution:

Using OF in proactive mode and introducing the entity of OF proxy controller significantly reduce the link usage by the OFP. Thus, Decisions 4,5 resolve this challenge.

Challenge 3: Device and Protocol Diversity

Solution:

The assumption that the testbed nodes will be located in the same island with a common L2 mesh routing protocol are enough to overcome the existing diversity. Thus, Decision 3 resolves this challenge.
Challenge 4: Communication with Non-Testbed Nodes

Solution:

Addressed in a way similar to Challenge 3. Decision 3 resolves this challenge.

Challenge 5: Out-of-band Channels

Solution:

As we described before, we use totally different interfaces to achieve data plane and control plane communication. Thus, Decision 6 resolves this challenge.

Therefore, we can claim that we succeeded in overcoming the challenges and building an architecture that can provide the required functionality.

6.3.2 Distributed properties of the system

Despite our achievements, it is worth to mention that our system is not fault-tolerant. There are two kinds of fault-tolerance that could exist in our system, component oriented and network oriented. A component failure could be either a failure that concerns the controller or a failure on the node side. On the node side, if the sliver or the node fails the OF switches are able to recover the rules upon the component recovery. Thus, any failure related to the nodes can be surpassed. One the other hand, the controller is a single point of failure and maintains no state. As a result, if the controllers fails the system cannot recover. Network failure mainly refers to the communication channels failure. In general, simple link failures do not affect our system because in a CN environment there are usually multiple routes and the L2 mesh routing protocol can also handle them. Our system, though, will not be able to recover from a network partition. If an update occurs during or after a network partition the update may not reach all the relevant switches. An inconsistency on the switches may lead our system to malfunction. Handling inconsistencies is an interesting topic that we plan to explore in the future. At the current moment, there are no consistency guarantees concerning propagating the rules to the OF switches. This raises the need to invent a method to monitor the distribute state of the switches.
6.3.3 Generalization for CNs or WMNs

One of the interesting points for discussion is the possibility and the benefits of applying our architecture not only in CN testbeds, but also normal CNs or WMNs. Concerning WMNs, our architecture would be very similar with the one proposed in [?], but adapted for L2. Our approach could be used only in case the WMN already uses the L2 mesh routing protocol. Similarly, for CNs our architecture could prove useful in islands that have deployed L2 mesh routing protocols. The separation of control plane and data plane should be implemented like proposed in [?], breaking each physical interface into two virtual ones. Additionally, our contribution of having a proxy OpenFlow controller would be really helpful taking into account the distributed nature of the environment but there is a need for a central controller.

There exists, though, a theoretical contradiction. Routing would be already implemented into L2, why would a forwarding plane be needed? There is no doubt that OF and SDN can provide really interesting applications for CNs and WMNs, like described in [?], but what would we earn in case we had OF working in L2 instead of L3? We believe that especially in the case where OF could interact with the L2 mesh routing protocol there are a lot of interesting applications. Most importantly, if a system of distributed OF controllers was developed, it could provide full control in L2 CN-wide or mesh-wide. Such a claim is very strong but we believe, under the assumption we stated about OF communication with the mesh protocol, it is a valid claim. Naturally, this is what the functionality would be as an outcome, but we have to consider also the performance cost. Therefore, the answer lies in the tradeoff between the desired functionality and the performance overhead.
7.1 Conclusions

Throughout this document we presented how we achieved to create a system that allows L2 experiments in CN testbeds using SDN techniques. We proposed a generic architecture that fulfills our goal and implemented this architecture for an existing CN testbed, Community-Lab. Our implementation consists of two, FOSS licensed, software components. First, we developed Poxy, a proxy for OF traffic. Second, we implemented Pongo, an integration of POX OF controller, Community-Lab testbed server and Django. Using these software components we deployed the proposed architecture in Community-Lab, creating an application that allows managing the L2 topology of a researchers nodes. Thus, we proved the feasibility of our architecture and the proper function of our implementation. Moreover, we performed a performance analysis of our system reaching the conclusion that the wireless multihop environment and the use of L2 mesh routing protocol are the greatest cause of overhead. Finally we discussed how our effort satisfies our goals and under which conditions our proposed architecture could be used in a generic CN or WMN environment.

7.2 Future Work

The next step would be to complete the evaluation by performing the experiments we described in §6.2. We believe that these experiments will confirm our hypotheses. An interesting approach for future work, would be to research the consistency of the system by checking the distributed state of the switches, as proposed in §6.3. The safety and the progress of the system should also be guaranteed in a technical way. Another interesting future approach would be to detach the OF service from the controller server and place it in multiple servers throughout the CN. Then there will be the need to form a protocol for communication between the distributed OF controllers. This approach seems very challenging but very promising for future CNs, as it would lead to scenarios similar to the ones described in §6.3.3. A different, but equally
interesting, approach would be to explore and deploy more the SDN capabilities. For example, new L2 (or upper) use cases could be developed based on the current OF infrastructure and newer versions of OF could be exploited. Finally, in order to exploit the full advantages of using SDN, it is in our future interests to explore the necessity and feasibility of having a management plane, like OFCONFIG or OVSDB. That would create a complete platform for performing experiments in a CN testbed but also could be used by the infrastructure of the testbed itself.
We provide the readme file from the confine branch of the Poxy\textsuperscript{1} application.

This application acts kind of like a proxy. It assumes that there is a local dump POX controller and a remote actual generic OpenFlow controller that does all the smart stuff. The applications job is to register the switch in both controllers but use the local one actually only to simulate that is connected to the controller. The remote one will contain the actual controller logic (for example another POX app).

The main files are:

\texttt{events.py}: Contains the new events created.
\texttt{async.py}: Kind of a socket wrapper for writing to a socket and asynchronous reading
\texttt{proxy.py}: The module that contains most of the logic. Lies in the middle of the asynchronous connection object of async.py and the basic POX functionality.
\texttt{myof\_01.py}: A copy of the switch handler of pox\_01.py which starts also the proxy

In this branch (confine) the user declares the address of his switch (each user has one switch) along with the remote controller. Thus separate switches can be connected to separate controllers. The user should be added before the switch is brought up. Start with debug on and connect to the central controller listening to the

\texttt{IP: RADDRESS (string, default = "127.0.0.1")}
\texttt{PORT: RPORT (int, default = 6634 )}

It is also necessary to declare the MAC address of the first user. For example:
\texttt{switch\_mac="00:00:00:00:00:00"}

Another possible option defines the maximum number of users that are going to use the system. By default it is 5. The option is:
\texttt{max\_users=5}

\textsuperscript{1}Poxy https://github.com/emmdim/Poxy/tree/confine
Example:

```
./pox.py log.level --DEBUG py myof_01 --switch_mac="00:00:00:00:00:00"
[--rport=RPORT] [--raddress=RADDRESS] --max_users=10
```

The py option runs an interactive python CLI. Thus if the we want to add new users later it can be done using the following commands:

```
POX>from myof_01 import *
POX>addUser("00:00:00:00:00:00")
```

Optionally the user can a specific destination other than the default (which is as previously):

```
POX>addUser("00:00:00:00:00:00","127.0.0.2",6633)
```
The Pongo\textsuperscript{1} application is consisted of 3 main parts. The Django part, the poxweb Django application that integrates POX with Django and a small library for integration with Community-Lab

B.1 Django

As a tree structure, Pongo looks like a Django project. The server which leads to the Django administration interface can be started normally by configuring the Pongo/settings.py file properly and running the following commands:

```
python manage.py syncdb
python manage.py runserver
```

B.2 Poxweb application

The Poxweb application is a Django application that also contains the POX functionality and is located in the poxweb folder. We will now describe the most interesting files:

- **models.py**: Contains the Django models that describe the links and the slivers.
- **admin.py**: Includes the Django models in the Django administrations environment.
- **ext/db.py**: A backend library for the POX application, that queries the Django database.
- **ext/dbpoll.py**: A backend daemon for the POX application, that is asynchronously checking if changes concerning the links occured in the database.
- **ext/l2topo.py**: The main POX application that handles the POX OpenFlow events and translates them to Django actions. At the same time it receives information from **dbpoll.py** concerning the database changes and translates them to OpenFlow rules if necessary.

\textsuperscript{1}Pongo https://github.com/emmdim/Pongo
Since it is a part Django application, POX should be started after the Django server. The CLI command to start the POX application is:

`.pox.py l2topo`

B.3 Integration with Community-Lab

`poxweb/ext/slice.py`: This module implements the REST communication between POX and any Community-Lab node. In case the user does not want to use Pongo with Community-Lab but with his own set of nodes he should implement the communication overriding this module.