A Scalable Architecture for OpenFlow SDN Controllers

Filipe Fernandes Azevedo

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Fernando Henrique Córte-Real Mira da Silva
              Prof. Luís Jorge Brás Monteiro Guerra e Silva

Examination Committee

Chairperson: Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. Fernando Henrique Córte-Real Mira da Silva
Member of the Committee: Prof. Luís Manuel Antunes Veiga

November 2015
Acknowledgments

I would like to thank every one, from friends and colleagues to family, who in one way or another helped achieve this life goal.

There are however a couple of you who have been there all along, helping me raise the bar day by day, namely my parents, Sancho and Susana, to whom I hold deep gratitude for everything they have done throughout my whole life, my girlfriend, Mariana, who was there all along offering unconditional support, my aunts and uncles, Madalena, Júlio, Maurício and Catarina who offered me more than I could ask for and without whom I could not get all the way to where I am, and my dear friend João, who through countless lunch hours put up with my problems and supported me along the way.

Last but definitely not least, I would like to express my sincere gratitude to my advisors, Professor Fernando Mira da Silva and Professor Luis Guerra e Silva, for excellent mentorship and all the support given that at times went beyond the role of advisor.
Abstract

The architectural principles of Software-Defined Network (SDN) and its most prominent supporting protocol - OpenFlow - keep gaining momentum. SDN relies essentially on the decoupling of the control plane from the data plane, placing the former in a logically centralized component to be executed on commodity hardware - the SDN controller. OpenFlow's reactive programming enables the programming of the network based on real-time decisions taken as new traffic hits the data plane, but it requires the first packet of every new flow traversing any SDN controlled device to be sent to the controller and evaluated, which when considered in large network environments becomes too large to be handled by a single SDN controller instance. In this work we propose a new architecture for an elastic SDN controller cluster as a solution to overcome the aforementioned limitations by allowing for the existence of multiple SDN controller instances acting as a single controller but handling each a subset of the OpenFlow switches that comprise the network. A proof of concept of the said architecture has been implemented by extending the Floodlight controller and integrating it with the Linux Virtual Server project.

Keywords: Software-Defined Networking, OpenFlow, Network Management, Distributed SDN Controller
Resumo

O padrão arquitetural Software-Defined Network (SDN) e o seu protocolo mais proeminente - OpenFlow - continuam a ganhar impeto. A arquitetura SDN tem por base o desacoplamento do control plane do data plane, colocando o primeiro num novo componente logicamente centralizado a ser executado em hardware de comodidade - o Controlador SDN. O modo de programação reativa do OpenFlow permite a programação da rede em tempo real, tomando decisões de encaminhamento conforme o tráfego dá entrada no data plane, sendo necessário para tal que a primeira trama de cada fluxo que atravesse um qualquer dispositivo de rede gerido por um controlador SDN seja reencaminhado para o controlador para que seja inspecionado. Embora o modo reativo proporcione um método mais conveniente e flexível para programar a rede quando comparado com o modo proactivo, o custo computacional associado à execução das tarefas necessárias torna-se incomportável para ser executado por uma única instância do controlador SDN quando aplicado a redes de grande dimensão.

Propõe-se neste documento uma nova arquitetura que torna o controlador SDN num cluster elástico visando a solução para o problema de escalabilidade apresentado através da existência de várias instâncias de controlador SDN que atuam como um único controlador, ficando no entanto cada instância responsável pela gestão de um subconjunto dos switches OpenFlow que compõem a rede. Houve ainda lugar a uma implementação de prova de conceito através da extensão do controlador Floodlight e da integração com o projeto Linux Virtual Server.

Keywords: Software-Defined Networking, OpenFlow, Gestão de redes, Controlador SDN Distribuído
# Contents

| List of Tables | ix |
| List of Figures | xi |
| Acronyms | xiii |

## 1 Introduction
1.1 Objectives .................................................. 3
1.2 Document structure ........................................ 3

## 2 Background
2.1 Software-Defined Networking ...................................... 5
  2.1.1 Southbound interfaces ...................................... 8
2.2 OpenFlow ..................................................... 9
  2.2.1 OpenFlow flow programming .................................. 11

## 3 Architecture
3.1 Elastic SDN controller cluster .................................. 13
3.2 Request Router .................................................. 16

## 4 Implementation
4.1 Elastic SDN controller cluster .................................. 21
  4.1.1 Proof of concept implementation ................................ 22
  4.1.2 Distributed network policy module .......................... 24
4.2 Request Router .................................................. 25
  4.2.1 Linux Virtual Server (LVS) .................................. 25
  4.2.2 Elastic Cluster Integration .................................. 27
  4.2.3 Anycast routing .............................................. 28

## 5 Evaluation
5.1 Test environment .............................................. 29
  5.1.1 Network testbed ............................................. 29
5.2 Elastic SDN controller cluster .................................. 30
  5.2.1 Scenarios .................................................. 30
  5.2.2 Results .................................................. 31
5.3 Distributed network policy module tests .......................... 31
  5.3.1 Scenario .................................................. 31
  5.3.2 Results .................................................. 32
5.4 Analysis .................................................... 32
List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Cluster messages</td>
<td>16</td>
</tr>
<tr>
<td>5.1</td>
<td>Flow programming results with a two-instance controller cluster</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>Flow programming results with a three-instance controller cluster</td>
<td>32</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Two node network according to the traditional network node architecture.</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Two node network according to the SDN architecture.</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>OpenFlow switch architecture.</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>OpenFlow reactive flow programming</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>OpenFlow SDN controller clustering</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>SDN elastic cluster membership registration</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Static SDN Controller load balancing</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Request Router integration with SDN controller instances</td>
<td>17</td>
</tr>
<tr>
<td>3.5</td>
<td>Logical overview of the proposed architecture</td>
<td>18</td>
</tr>
<tr>
<td>3.6</td>
<td>OpenFlow switch management connections load balanced by Request Routers</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Proof of concept implementation high level block diagram</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Request Router integration component high level block diagram</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>Typical datacenter topology</td>
<td>30</td>
</tr>
<tr>
<td>5.2</td>
<td>Performance indicators for the distributed network policy module</td>
<td>33</td>
</tr>
</tbody>
</table>
Acronyms

API  Application Programming Interface. 2 3 5 13 21 24
ARP  Address Resolution Protocol. 25
BDDP  Broadcast Domain Discovery Protocol. 21
BFD  Bidirectional Forwarding Detection. 28 29 35
BGP  Border Gateway Protocol. 25 28 29 35
CPU  Central Processing Unit. 8
ECMP  Equal-cost multi-path. 28
ETSI  European Telecommunications Standards Institute. 2
ForCES  Forwarding and Control Element Separation. 8
IaaS  Infrastructure as a Service. 1 8 29
IDS  Intrusion Detection System. 1 2
IETF  Internet Engineering Task Force. 8
IPv4  Internet Protocol version 4. 23 25
IPVS  IP Virtual Server. 25 28 35
IS-IS  Intermediate System to Intermediate System. 28
IT  Information Technology. 1 2 8
JSON  JavaScript Object Notation. 23
LLDP  Link Layer Discovery Protocol. 21 24
LVS  Linux Virtual Server. 25 31
MIB  Management Information Base. 5 13 14 16 22 23 30 31 33
MPLS  Multiprotocol Label Switching. 1 28
MTU  Maximum Transmission Unit. 23 24
NAT  Network Address Translation. 1 25
NFV  Network Function Virtualization. 2

NOS  Network Operating System. 8

OF-CONFIG  OpenFlow Management and Configuration Protocol. 9

ONF  Open Networking Foundation. 8, 9

OSI  Open Systems Interconnection. 23, 25, 26, 31

OSPF  Open Shortest Path First. 28

OVSDDB  Open vSwitch Database. 9

POF  Protocol Oblivious Forwarding. 8

QoS  Quality of Service. 11

RIP  Routing Information Protocol. 28

RPF  Reverse Path Forwarding. 26

SDN  Software-Defined Network. III, V, XI, 2, 3, 5, 6, 8-14, 16-19, 21-23, 25, 27, 29-31, 33, 35

TCP  Transmission Control Protocol. 10, 18, 23-25, 31-33

TLS  Transport Layer Security. 10, 18

UDP  User Datagram Protocol. 25

VLAN  Virtual Local Area Network. 1, 22

VM  Virtual Machine. 1, 29

VRF  Virtual Routing and Forwarding. 1

WAN  Wide Area Network. 1
Chapter 1

Introduction

Today’s computer networks are essentially sets of interconnected network nodes. These nodes are mainly proprietary black boxes that allow for data input/output, exposing a configuration interface through which network operators can configure its behavior by means of a vendor specific configuration workflow and interface. These black boxes are typically switches and routers that implement, in software and/or hardware, a set of standard or proprietary protocols that allow them to communicate with each other and provide service to the endpoints logically or physically connected to them. Because the process of creating and implementing new protocols is slow, and due to the complex dependency between multiple protocols running on the same node, new network features have typically been implemented resorting to middle boxes designed to perform specific functions such as Firewalls, Intrusion Detection Systems (IDSs) and Wide Area Network (WAN) optimizers, thus effectively increasing the overall complexity and in turn further aggravating the network manageability issue at hand.

This has held true for the last couple of decades and is now an increasingly focus of attentions due to the recent trend of Information Technology (IT) virtualization, that has changed the typical data center environment drastically, with the number of virtual network ports surpassing that of physical network ports and the amount of virtual servers largely exceeding the number of physical servers [1]. However, IT virtualization has a much deeper impact on the network infrastructure of the data center than the mere increase of network ports and hosts. With modern IT virtualization technology, creation, removal and relocation of Virtual Machines (VMs) between physical hosts became trivial operations. Therefore, the concept of Infrastructure as a Service (IaaS) has emerged, which enables the creation of elastic computing services [1] such as Amazon Elastic Compute Cloud (Amazon EC2) [2]. The IT virtualization operations supporting IaaS can be performed in a matter of minutes. However, the network infrastructure also needs to be reconfigured to support the changes being made to the environment. This operation is still done manually and often node by node, taking hours or even days to be completed.

Modern networking approaches already introduced virtualization mechanisms, such as Multiprotocol Label Switching (MPLS) for path virtualization, Virtual Local Area Network (VLAN) for layer 2 virtualization and Network Address Translation (NAT) and Virtual Routing and Forwarding (VRF) for layer 3 virtualization [1], however, these solutions are still device-centric and therefore still require configuration to be performed node by node through the same vendor specific workflows and interfaces, meaning that there is still no global network view nor global network configuration [1] and in some cases these mechanisms are no longer able to cope with the high density of compute instances introduced by IT virtualization [3] thus leaving the task of network reconfiguration to still require a large amount of time, be error prone and in some cases ineffective. As a result, networks are now the technological bottleneck that is dragging the operational workload and hindering innovation in IT with heavy focus on data center environments. To that extent a new network management model is now required, one that can cope with these require-
ments while also being future proof.

In order to solve these issues new approaches were introduced in the past couple of years, such as SDNs and Network Function Virtualization (NFV). These two approaches share some core concerns, such as the use of commodity hardware to implement network features, but they both aim at different goals, being complementary but not dependent on each other [4][5].

NFV was proposed by the European Telecommunications Standards Institute (ETSI) and stems from IT virtualization itself, proposing the virtualization of entire network node classes, such as Firewalls, Routers and IDSs, using commodity hardware as the underlying physical infrastructure. This approach results in the dismissal of function-specific hardware middle boxes by means of virtualization, thus easing the physical constrains of adding new features to the network, as new instances of virtualized nodes can be deployed using existing commodity hardware already in place in different regions of the network [4]. This means that at core, NFV attempts to improve the scalability of current networks by removing some of the hardware complexity and cost, but leaves the global network manageability issue yet to be solved.

The current notion of SDN was developed in Stanford University while researching for a network configuration protocol, from which resulted OpenFlow, and is an approach that keeps gaining momentum both on the academic and business environments. SDN refers to the architectural principles while OpenFlow is an implementation of a supporting protocol for the said architecture. SDN relies essentially on the decoupling of the control plane from the data plane, placing the former in a logically centralized component to be executed on commodity hardware - the SDN Controller.

OpenFlow, being a protocol that implements SDN, allows for network programmability in a flow-oriented fashion through a well-defined Application Programming Interface (API) providing two different approaches to do so: proactive and reactive flow programming. While both approaches can be used simultaneously, there is a lot to be gained from the latter, which provides a mechanism to program the network to forward data based on real-time decisions taken as traffic hits the data plane, while the former provides a mean for static network programming before traffic reaches the data plane. The reactive approach, however, has a much higher computational cost when compared to the proactive approach, since for every new traffic flow traversing the network controlled by a given SDN controller the first packet of such flow must be sent from the OpenFlow switch receiving it to the SDN Controller, have the SDN controller evaluate the packet, determine appropriate action, program the OpenFlow switch accordingly, and then forward the packet back into the data plane.

When applied to large scale networks, the inherent computational cost of performing the aforementioned tasks becomes far too high to be handled by a single SDN controller instance. One way to overcome this limitation is to have several SDN controller instances running separately, each handling a subset of the OpenFlow switch set. However, this approach cannot be easily implemented since network policies would have to be explicitly configured on each and every SDN controller, which is susceptible to SDN controller failures, requires a complex and cumbersome configuration of the OpenFlow switches or alternatively a mechanism of coordination between the instances of the SDN controller in order to ensure equal load sharing across SDN controllers. Finally, for any given flow traversing multiple OpenFlow switches controlled by different SDN controllers instances the aforementioned process would have to be executed on each SDN controller instance involved.
1.1 Objectives

This work proposes a new architecture for OpenFlow-based SDN controllers that offers both load balance between different controller instances and a resilient infrastructure, which enables for a scalable use of reactive flow programming, regardlessly of the network size. This new architecture does so by relying on two key concepts: the clustering of controllers and the introduction of a load balancing layer. The clustering mechanism takes the notion of the controller as a logically centralized component and introduces the concept of elastic controller clustering, implementing the mechanisms necessary to scale out the controller, as needed and in real time, by providing means to on-demand add and remove controller instances from the cluster no service disruption. The load balancing layer, which is also executed on commodity hardware, is logically situated between the OpenFlow switches and the SDN Controllers, acting as a level of indirection that assigns a controller to each OpenFlow switch in a consistent fashion that ensures an equal distribution of OpenFlow switches across all the available SDN controller instances.

1.2 Document structure

This document is composed of 6 chapters structured as follows: Chapter 2 covers the state of the art regarding SDN and OpenFlow; Chapter 3 describes the proposed architecture; Chapter 4 cover the implementation details of the architecture and strategies used; Chapter 5 presents the results obtained by testing the resulting implementation; Chapter 6 summarizes the work developed, the problems identified and future work.
Chapter 2

Background

2.1 Software-Defined Networking

SDN relies on two fundamental changes to the traditional network design principles in order to solve the management issues of traditional networks mentioned in Chapter 1: the decoupling of network control and forwarding planes and network programmability [6]. Traditional network nodes are built of the paradigm of a three strongly-coupled planes of functionality. These are the management plane, which allows for service monitoring and policy definition, the control plane, which enforces the policies in network devices and the data plane which efficiently forwards data. Figure 2.1 depicts a two-node network following this networking architecture approach.

Having these three planes tightly-coupled and implemented in the network nodes is the reason behind both the box-centric configuration style and vendor-specific configuration workflow and interface. SDN decouples these planes from one another, defining the abstraction of Specification, corresponding to the management plane functionality, Distribution, corresponding to the control plane functionality, and Forwarding, corresponding to the data plane functionality. By doing so, this new architectural concept makes it possible to logically centralize the Specification and Distribution while keeping the Forwarding implemented in the network nodes [1][6][7], thus creating the premises for a network infrastructure that harnesses the benefits of both distributed data forwarding and centralized management, dismissing today’s decentralized network management style.

The second core feature of SDN is the ability to programmatically configure the network, which is accomplished by having the Specification expose a set of APIs to be consumed by the network applications which are referred to as Northbound APIs. These APIs are generally capable of exposing the network topology through global and abstracted views, as well as providing methods of global policy definition and common network functionality such as routing, access control and bandwidth management [6]. The policies, which are kept in the form of Management Information Base (MIB) databases, are then applied to the appropriate network nodes by the Distribution through the use of a communication protocol implemented both on this abstraction and on the network elements.

These two core concepts lead to the definition of the architecture of SDN in three layers: the Application layer, the Control layer and the Infrastructure layer [7][1] as depicted in Figure 2.2. The Infrastructure layer is where all the network nodes reside, performing all the Forwarding functions. The Control layer encompasses the Specification and the Distribution, therefore dealing with all the network management and control mechanisms. The Application layer is where the network applications reside. These applications define the network policies through the Control’s APIs, taking into account factors such as (but not restricted to) network topology and the network operator input.
This three-layered architecture, depicted in figure 2.2, is commonly extended to a four-layered architecture by breaking the Control layer in two, creating a layer solely for network virtualization that sits between the Control and Infrastructure layers.

The implementation of the Control layer is known as a SDN controller or Network Operating System (NOS) and it is the main building block for Software-Defined Networks. A SDN Controller exposes three distinct APIs: Northbound, Southbound and Eastbound/Westbound. The Northbound API is intended for interaction with network applications, corresponding to the Specification functionality. The Southbound API, in the scope of the Distribution functionality, is intended for interaction with the network elements. The Eastbound/Westbound API is intended for integration with other SDN controllers, and is mostly seen in distributed controllers [1]. The specification of Northbound and Eastbound/Westbound APIs varies according to the controller implementation. The Southbound API however, must implement the same protocol as network elements do in order for network programmability to work, which highlights the importance of defining good standard protocols, and the acceptance of said protocols by vendors, as a key point for SDN's success.

By comparing figures 2.1 and 2.2 it is easy to understand that this new architecture simplifies management by having it done in a single point (Control layer) instead of multiple points (network nodes) and fosters innovation as the implementation of new network policies or custom network behavior can now be promptly implemented by introducing an application to do so without the need of knowing how it will be implemented in the network infrastructure or even of how the network infrastructure is composed. Furthermore, applications can be written to dynamically adjust network policies according to the network status without the need for Human intervention, thus drastically reducing the time required to adjust network policies upon network events such as topology changes and therefore potentially increasing the effectiveness of said policies, should the event pose a security threat or risk of service outage. Another inherent advantage of this solution is the consistency of the policy throughout the network, since that the
Figure 2.2: Two node network according to the SDN architecture.
policy is now defined by the network operator through a network application in a global scope and in a single point, therefore eliminating inconsistencies in configurations between different network nodes. There are currently several implementations for SDN controllers such as OpenDaylight [8], Floodlight [9] and ONOS [10] [11], as well as southbound interfaces such as OpenFlow, POF, ForCES and OpFlex [11]. Although, at first, it may not be obvious, the fact that multiple southbound interfaces exist, presents itself as an issue, given that either network elements or SDN controllers must implement more than one of them in order to achieve compatibility between SDN controllers and network elements. Because there is now a dominant southbound interface - OpenFlow - the vast majority of network elements implementing the SDN architecture are implementing only OpenFlow. The [Open Networking Foundation (ONF)] has made OpenFlow a standard protocol and is working on standardizing the Northbound interfaces [6], which will make network applications controller-agnostic, which in turn will streamline development and make application portability possible.

Today's data center's multitenancy is growing as a consequence of IaaS and therefore infrastructure virtualization becomes increasingly important both for security and service reasons. With more and more companies migrating their infrastructure to private cloud infrastructures, network virtualization starts to gain new perspectives as traffic isolation is not enough, with support for custom tenant network topologies being required to smooth the transition. There are also several network virtualization implementations for SDN such as FlowVisor [11][12] and OpenVirteX [11]. These implementations are analogous to the hypervisors in IT virtualization, sitting between the SDN controller and the network elements, acting as a proxy for the communications between both. Multiple SDN controllers are allowed to interact with the same instance of the virtualization layer, one for each virtual network.

### 2.1.1 Southbound interfaces

The SDN architecture relies on the existence of protocols for programming the forwarding plane, also referred to as southbound interfaces (from a SDN controller perspective). There are currently several protocols, such as Protocol Oblivious Forwarding (POF) [13], Forwarding and Control Element Separation (ForCES) [14], OpFlex [15] and OpenFlow [6]. POF attempts to enhance the forwarding plane programmability by defining generic matching rules that make use of offset definitions instead of packet header definitions. In practice, matching in POF is performed against a sequence of bits with specified length in a packet starting at a given position, effectively allowing the definition of matches against any packet header regardless of the protocol [13]. ForCES despite also aiming at decoupling control from forwarding, does not require the existence of a network controller and instead the control is allowed to remain in the network nodes [11]. OpFlex mixes concepts from both OpenFlow and ForCES having the management being centralized while the control remains distributed in the network elements [16].

OpenFlow, the most prominent of them, is a standard protocol for forwarding plane programming, defined by the ONF and designed for Ethernet networks [6]. It is analogous to the Instruction Set of a Central Processing Unit (CPU) in the sense that it defines primitives that can be used by external applications to program the network node much like it is done with CPUs [6]. Forwarding decisions are based in the notion of flows, which according to the Internet Engineering Task Force (IETF) are "a sequence of packets from a sending application to a receiving application" [17]. OpenFlow is supported by several vendors such as Cisco [18], Juniper [19], HP [20] and Alcatel-Lucent [21].

While the OpenFlow protocol allows for forwarding plane programming, some network management aspects are not entirely bound to forwarding decisions. Some of these aspects include turning network ports on and off and establishing tunnels between network elements. There are however other southbound protocols that introduce these management capabilities and rely on OpenFlow to implement.
forwarding plane programming, namely the Open vSwitch Database (OVSDB) and the OpenFlow Management and Configuration Protocol (OF-CONFIG) [22]. OVSDB is a component of Open vSwitch and it is designed to manage Open vSwitch implementations. OF-CONFIG is defined by the ONF specifically to be used alongside with OpenFlow and may be implemented in both software or hardware network element implementations.

2.2 OpenFlow

Having had several versions released since 2008, when version 0.8.0 debuted, OpenFlow is currently on version 1.5 [23]. However, because changes made from version 1.3 to version 1.5 [23][24] are not relevant to the scope of this work, and because both hardware and software switch implementations mainly implement version 1.3, we will focus on version 1.3 for the purpose of this work.

As previously mentioned, there are now several available implementations of OpenFlow-enabled network elements, however OpenFlow implementation and traditional forwarding plane implementations are not mutually exclusive, thus the definitions of OpenFlow-only network elements (which implement only the OpenFlow forwarding plane) and OpenFlow-hybrid network elements (which implement both the OpenFlow and legacy forwarding planes). In the latter, OpenFlow specifies means of forwarding traffic from the OpenFlow forwarding plane and into the legacy forwarding plane [24]. This is in fact a major upside to OpenFlow adoption, as it makes the adoption of OpenFlow by hardware vendors as easy as releasing a new software version that implements the OpenFlow forwarding plane on top of existing network hardware, and allows adopters to gradually roll-out OpenFlow in existing networks possibly with existing network hardware.

The OpenFlow specification defines the architecture of an OpenFlow switch implementation in three main building blocks: the OpenFlow Channel, the Processing Pipeline and the Group Table as depicted in figure 2.3. The OpenFlow Channel defines the interface for communications between the OpenFlow switch and the SDN controller, through which controllers can statically or dynamically create, modify and remove flow entries from flow tables. Further detail on the differences between static and dynamic flow entry programming will be addressed later in this document. All communications between an Open-
Flow switch and an SDN controller are encrypted by default using Transport Layer Security (TLS) even though it is possible to use plain Transmission Control Protocol (TCP) [24]. These communications are always initiated by the OpenFlow switch, and each switch may have one or more SDN controller IP Address specified in its base configuration. The OpenFlow switch is required to maintain active connections with all configured SDN controllers simultaneously. Because multiple SDN controllers may be configured in a single switch, each controller is assigned one of three possible roles: Master, Slave or Equal [24].

The Master role grants the SDN controller full management access to the switch, allowing it to program flow entries and receive notifications of events occurring on the switch, such as flow entries expiring and port status updates. There can be at most one SDN controller configured as master per switch. The Slave role grants the SDN controller with read-only access to the switch, allowing it to receive only port status updates. Any SDN controller instance that is already connected to the switch can request to change role from slave to master, at which point the instance that previously held the master role will be updated to the slave role. This mechanism provides a fault tolerance mechanism by allowing for the existence of Active-Standby SDN controller clusters. The Equal role is essentially the same as the Master role, with the exception of allowing one switch to be connected to multiple SDN controllers in Equal role. This specific role aims at introducing a mechanism for fault tolerance but also load balancing, requiring however that the SDN controllers coordinate with each other.

The Processing Pipeline is composed of several flow tables (with a required minimum of one), each containing a set of flow entries. A Flow entry is defined by a set of fields, namely:

- **Priority**, which defines its precedence over other flow entries in the same table, with the highest value taking precedence
- **Match fields** consisting of header patterns that when positively matched against a packet identify a flow
- **A set of instructions** that determine the course of action the switch should take to handle the flow
- **Timeouts** defining for how long the flow entry is to be maintained in the flow table, namely a hard timeout which forces the flow entry to be removed $\Delta$ seconds after it was installed and a soft timeout specifies for how long the flow entry shall be kept after the last matching packet as been matched
- **Counters** that increment each time a packet is successfully matched against the flow entry
- **Cookies**, which are used exclusively by the SDN controller to annotate the flow entry

For each flow table there might be one special flow entry that matches all packets and with a priority of zero, called the table-miss flow entry and it defines the default action set to be taken by the switch for any given packet. Flow matching is performed in a one way direction, starting in flow table 0 (which is the only mandatory flow table) and ending with either a set of actions to be executed (defined by matching entries) or alternatively dropping the packet. This means that once processing has reached table $n$, it can then either resume processing in table $m$ where $m > n$ if the matching flow entry defined such instruction, or execute the actions in the instruction set defined by matching entries in tables 0 through $n$. If no entries that match the packet occurred then either there is a table-miss flow, usually redirecting the packet to the SDN controller or simply resuming the pipeline processing in the next table, or the switch drops the packet completely. Last but not least, the Group Table contains group entries. Group entries allow the definition of action buckets, which are defined by a Group Identifier uniquely identifying the group entry, a Group Type defining the behavior of the Group entry, and a set of Action Buckets which
are themselves sets of actions to be executed by the switch. Group entries are somewhat analogous to action macros, simplifying forwarding functions such as broadcast and multicast [24]. An OpenFlow switch may also implement a meter table allowing for the implementation of simple Quality of Service (QoS) features such as queue management and rate limiting [24]. The aforementioned mechanisms grant OpenFlow the ability to perform highly granular control over how specific flows should traverse the network, enabling the differentiation of traffic according to its profile [6]. Consequently, network protocol implementations on network elements become increasingly deprecated, leading to the dismissal of said implementations in favor of OpenFlow forwarding plane implementations instead, thus having for the forwarding decisions being programmed from the SDN controller according to the global policies set for the network [6].

2.2.1 OpenFlow flow programming

SDN controllers may program flow entries in OpenFlow switches by issuing flow table modification messages. These messages allow the SDN controller to manipulate flow tables by adding new flow entries and modifying or removing existing flow entries. Flow entry additions performed by the SDN controller by issuing a flow table modification message to predefine the forwarding behavior ahead of data transmission is referred to as static flow entries and configures the proactive flow programming paradigm. These flow entries usually have both their hard and soft timeouts set to zero, meaning that they will not expire [24]. Having these flow entries present in the flow tables enable the traffic to be forwarded immediately when reaching the OpenFlow switch. However, it has the disadvantages of having to preprogram flow entries to cover every single flow possible according to the global network policy, which leads to a quick exhaustion of the flow tables available in the switch. Furthermore, the action set being preprogrammed might not be optimal for a particular flow being processed at a particular point in time. A typical usage of static flow entries are the table-miss flow entries, which have already been discussed. Table-miss flow entries, defining an instruction to forward the packet to the SDN controller through a packet-in message, on the other hand enable the SDN controller to examine the packet and define the proper actions to be executed by the switch in a reactive fashion. The SDN controller may respond to this message with either a packet-out message defining the actions to be executed exclusively for that packet or alterna-
tively with a flow table modification message that will create a new flow entry to handle all the packets matching that flow, including the original packet included in the packet-in message [24], as depicted in figure 2.4. The resulting flow entries are referred to as dynamic flow entries, and it is one of the most powerful features of OpenFlow as it allows for forwarding decision-making to be performed by a centralized system that has global view and control over the network - the SDN Controller - thus configuring the reactive flow programming paradigm. This however comes with a considerable computational cost, since every first packet of each new flow being admitted into the network must be sent to and evaluated by the SDN controller which in turn will instruct the switch(es) on how to behave for that flow. When considering large-scale networks, such as that of a big data center, the amount of packets that must be sent to and processed by the controller is far too great to be handled by a single controller instance, therefore rendering this approach unfeasible.
Chapter 3

Architecture

As previously mentioned, the OpenFlow specification provides mechanisms for resilience and load balancing across multiple SDN controller instances. These mechanisms however require complex mechanisms of inter-instance coordination and cumbersome configuration tasks on the OpenFlow switch(es). The basic principle of the proposed architecture is to replace the SDN controller by a cluster of SDN controllers, which keep a consistent MIB between them. Any individual controller in the cluster is able to manage any OpenFlow device in the same network domain. Load balance between controllers is provided by introducing southbound and northbound request routers, which may themselves be replicated for redundancy proposes. The proposed solution enables controllers to be dynamically added to or removed from the cluster without network disruption or OpenFlow switch reconfiguration, therefore providing an elastic structure.

3.1 Elastic SDN controller cluster

OpenFlow allows for an OpenFlow switch to be connected to several SDN controllers simultaneously, however, the latter are expected to have some sort of coordination between them in order to achieve load balancing and instance failure resiliency, which is achieved by exposing eastbound/westbound APIs to be consumed by neighboring controllers within the same management domain. Having the instances coordinated with each other and keep consistent states in a distributed environment requires therefore a SDN controller cluster and configures an architecture such that of figure 3.1. SDN cluster instances must then provide eastbound/westbound APIs that allow for cluster membership operations. The resulting cluster should support key aspects of the SDN controller component, such as keep a global view and management scope. Because the goal is to perform load balancing between all instances, each instance will be actively managing any given number of OpenFlow switches in the same management domain, and therefore the network state is distributed in nature. In order to improve scalability and resiliency, new SDN controller instances can be added and removed on demand from the cluster, without requiring any reconfiguration or service interruption.

To that extent, when a new SDN controller instance is initialized it must advertise itself to all existing SDN controller instances, forming a cluster if only one other instance exists prior to the advertisement or otherwise joining the existing cluster. In order for this to be accomplished either the newly instantiated SDN controller would need to have prior knowledge of all existing instances or, instead, some mechanism of peer instance discovery must be available. Peer instance discovery offers a mechanism much closer to the zero-configuration goal that this architecture targets. The discovery mechanism considered is that of group communication, which is itself a form of indirect communication, in which the sender is
not aware of the receivers and thus instead of sending the message to a specific receiver it sends the message to a group, which will then have the message delivered to all members of such group [25]. Upon joining a cluster, regardless if new or existing, the joining instance(s) must then synchronize its MIB with any other peer instance, from which point on it will proceed to process updates from other peers. The SDN controller instance cluster registration process is defined in Figure 3.2 and the set of messages for cluster membership in Table 3.1.

Because SDN controller instances are prone to failure due to software, hardware or network failures, at any given point in time, which prevents them from being able to notify peers, it is important to provide a mechanism that makes every cluster member aware of their peers state. As such, two mechanism have been defined for instance failure detection: session identifiers and periodic health checks.

SDN controller instances must generate a unique session identifier when initializing themselves, which will then serve as an identifying key for cluster members, and therefore must be included in every message exchanged with peering instances. When a SDN controller instance $\alpha$ receives a message from a peer instance $\beta$ which is identified by session identifier $\upsilon$, $\alpha$ must validate that the membership key $\kappa$ it holds for peer $\beta$ has not changed ($\kappa = \upsilon$) before processing the message. Should the session identifier have changed ($\kappa \neq \upsilon$), $\alpha$ must invalidate all MIB entries associated with peer $\beta$.

In order to ensure fast failure detection and quickly invalidate any MIB entry that might lead to erroneous forwarding decisions, every SDN controller instance participating in a cluster must send periodic (every $\Delta$ seconds) Keepalive notifications to its peers. Should an instance $\alpha$ fail to receive three consecutive Keepalive notifications (3$\Delta$ seconds) from a peer instance $\beta$, then $\alpha$ must assume that $\beta$ is no longer available and therefore render all MIB entries associated with $\beta$ invalid.

In order to maintain the aforementioned behavior, active instances must issue cluster membership and MIB updates to their cluster peers and be prepared to update themselves with the information issued by their peers. The propagation of MIB updates must be triggered for every network event detected by a given instance. The minimum set of cluster membership and MIB update messages and associated events, along with which instances must be notified, are specified in Table 3.1. When a SDN cluster instance propagates updates to other peer instances, it must implement causal consistency, thus ensuring that any dependencies between updates are met. Causal consistency ensures that messages sent by an instance $\alpha$ are seen by every other instance receiving them in the exact same order that they where sent by $\alpha$, thus guaranteeing potential causal ordering of events throughout the cluster [25]. This property is crucial to maintain the integrity of the distributed data model, as it guarantees that scenarios such as adding an inter-switch link before adding the switches themselves do not happen.

This architecture enables the necessary for coordination between SDN controller instances, providing...
Figure 3.2: SDN elastic cluster membership registration
Table 3.1: Cluster messages

<table>
<thead>
<tr>
<th>Scope</th>
<th>Message</th>
<th>Event</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster membership</td>
<td>Join cluster</td>
<td>New SDN controller instance created and</td>
<td>cluster</td>
</tr>
<tr>
<td></td>
<td></td>
<td>requesting to join the existing cluster</td>
<td></td>
</tr>
<tr>
<td>Cluster membership</td>
<td>Leave cluster</td>
<td>SDN controller instance shutting down and</td>
<td>cluster</td>
</tr>
<tr>
<td></td>
<td></td>
<td>requesting to leave the cluster</td>
<td></td>
</tr>
<tr>
<td>Cluster membership</td>
<td>Instance health</td>
<td>SDN controller instance periodic health</td>
<td>cluster</td>
</tr>
<tr>
<td></td>
<td></td>
<td>report</td>
<td></td>
</tr>
<tr>
<td>Cluster membership</td>
<td>Full Synchronization Request</td>
<td>SDN controller instance requesting a full MIB update</td>
<td>specific instance</td>
</tr>
<tr>
<td></td>
<td>Full Synchronization Reply</td>
<td>SDN controller instance replying to a full MIB update</td>
<td>specific instance</td>
</tr>
<tr>
<td>MIB update</td>
<td>Add Switch</td>
<td>New OpenFlow switch connected to an instance</td>
<td>cluster</td>
</tr>
<tr>
<td>MIB update</td>
<td>Remove Switch</td>
<td>OpenFlow switch disconnected from an instance</td>
<td>cluster</td>
</tr>
<tr>
<td>MIB update</td>
<td>Add Host</td>
<td>New host connected to the network</td>
<td>cluster</td>
</tr>
<tr>
<td>MIB update</td>
<td>Remove Host</td>
<td>Host disconnected from the network</td>
<td>cluster</td>
</tr>
<tr>
<td>MIB update</td>
<td>Add Inter-switch link</td>
<td>New neighbor adjacency detected</td>
<td>cluster</td>
</tr>
<tr>
<td>MIB update</td>
<td>Remove Inter-switch link</td>
<td>Neighbor adjacency lost</td>
<td>cluster</td>
</tr>
</tbody>
</table>

each instance with a global view of the network regardlessly of the network node(s) it manages. This approach, however, still requires that every OpenFlow switch within the management domain be reconfigured every time an SDN controller instance is added, to or removed from, the SDN controller cluster, which takes us back to a manual, decentralized and vendor-specific network management model.

3.2 Request Router

The mechanism provided by OpenFlow to load balance between clustered SDN controller instances and inherently provide resilience requires one of the following:

1. OpenFlow switches are configured with all the IP Addresses of all the controller instances, and then either let Master/Slave role election occur or have the controller implementation extended such that multiple instances in Equal role can coordinate between themselves which instance will process which event;

2. The network is partitioned such that different subsets of network are controlled by different controllers as depicted in Figure 3.3.

The first approach adds a considerable amount of complexity both to the OpenFlow switch configuration and to the controller implementation, providing however the basis for equitable load balancing and controller resiliency. The second has the advantage of removing complexity both from the OpenFlow switches and the controller implementation, but it is still suboptimal as it does not guarantee equal load balancing across controller instances. In fact, a subset $\Psi$ of OpenFlow switches may process more traffic flows than a subset $\Gamma$, resulting in a higher computational load on the instance controlling subset $\Psi$ when compared to the instance controlling subset $\Gamma$.

In order to provide a solution that keeps the advantages and eliminates the disadvantages of both approaches, it is then necessary to implement of a mechanism that transparently assigns the least-loaded
instance to an OpenFlow switch - a load balancing component. This new component, the request router, must be completely transparent to all three components of the SDN architecture, while providing load balance between SDN controller instances without compromising infrastructure resiliency.

The request router encapsulates all the controller instances as a single virtual instance from the OpenFlow switches and network applications point of view, by logically sitting between the SDN controller instances and the OpenFlow switches and having the OpenFlow switches connect to a request router instance instead of directly connecting to controller instances. The request router is then responsible to determine which is the best SDN controller instance to handle the OpenFlow switch and forward the request to that instance.

To be able to forward connections to SDN controller instances, it is necessary for the request router to be aware of all existing SDN controller instances such that the logical topology described in Figure 3.4 can be achieved. This can be attained simply by having the request router instances joining the communication group described in Section 3.1 and listening for the cluster membership messages described in Table 3.1. The same Keepalive mechanism described in Section 3.1 must also be implemented on the request router in order for the request routers to be also able to detect failing SDN controller instances.

In order to maintain infrastructure resiliency, it is imperative that the execution of the load balancing mechanism does not affect the connections between OpenFlow switches and SDN controller instances.
and that a failure of any given request router instance does not have impact on the load balancing service. The request router must then implement a clustering mechanism that allows the existence of several request router instances, in which active OpenFlow switch management connections are processed by any available request router instance, thus load balancing the connections between themselves such that request router instances do not become overloaded, and where any request router instance $\phi$ can take over the tasks performed by instance $\phi$ should $\phi$ fail.

To ensure that OpenFlow switch management connections to SDN controller instances are load balanced between available request routers, without requiring reconfiguration of the OpenFlow switches, request router instances implement the Anycast communication paradigm. By implementing Anycast, OpenFlow switches will connect to whichever request router instance is logically closer to them, thus providing a mechanism to load balance OpenFlow switch management connections between request router instances. Because large network topologies often span over several geographical locations and OpenFlow switches will connect to whichever request router is closer to them, request routers instances in the same request router cluster are allowed to have different forwarding rules so that OpenFlow switch management connections can be forwarded to the closest least-loaded, or to the globally least-loaded SDN cluster instance according to the network administrators preferences, thus resulting the in topology depicted in Figure 3.5.

As mentioned in Chapter 2 Section 2.2, OpenFlow switches are required to maintain active TCP or TLS connections with the configured SDN controller(s). Considering the aforementioned specification of the request router, all OpenFlow switches must be configured to connect exclusively to the request router cluster, which will in turn forward the connection to an SDN controller instance, as per the topology represented in Figure 3.5. The SDN controller instance then replies directly to the OpenFlow switch, as portrayed in Figure 3.6. Having this behavior, as opposed to having the SDN controller instance reply back to the request router instance, improves both performance and scalability as the latency for the response packets will be lower and it removes computational load from the request router instances, thus allowing them to handle more request packets.

Because both TCP and TLS (which uses TCP as its transport layer protocol [26]) are connection oriented, request router instances must always forward packets coming from an OpenFlow switch to the
same SDN controller instance. Therefore, the request router must implement a deterministic destination selection for all packets coming from the same OpenFlow switch, such that any request router instance will forward the packets from switch $\rho$ to the exact same SDN controller instance $\alpha$. Should $\alpha$ become unavailable then request router instances are required to stop forwarding packets to $\alpha$ and assign a new SDN controller instance $\beta$ and forward all packets from $\rho$ to $\beta$. 

Figure 3.6: OpenFlow switch management connections load balanced by Request Routers
Chapter 4

Implementation

To demonstrate the feasibility and validity of the proposed architecture, a proof-of-concept implementation has been developed by means of extending an existing SDN controller and integrating a Linux kernel module for clustering.

The requirements set for the controller selection were controller completeness, popularity and availability of source code, which lead to a final decision between OpenDaylight and Floodlight. While both OpenDaylight and Floodlight are modular open-source SDN controllers implemented in Java, highly popular and supported by major players in the networking industry, at the time of the decision, Floodlight, which was at version 1.1, had a more stable implementation when compared with the OpenDaylight, which was at release Helium. OpenDaylight also offers support for multiple Southbound Interface protocols which, while falling outside the scope of this work, would render the architecture of the platform more complex than that of Floodlight, and would therefore add more complexity to the implementation. For these reasons, Floodlight was chosen as a basis for the implementation of the proof-of-concept for the proposed architecture.

4.1 Elastic SDN controller cluster

Floodlight is a modular implementation of an SDN controller developed by Big Switch Networks using the Java language, and is currently in version 1.1. Floodlight shares its core implementation with Big Switch Networks’s own Big Network Controller. It offers stable support for OpenFlow versions 1.0 and 1.3 and experimental support for versions 1.1, 1.2 and 1.4 through OpenFlowJ Loxi, a library that encapsulates the OpenFlow protocol and exposes functionality through a protocol version agnostic API.

Floodlight’s architecture is highly modular, composed by a base Module Loading System that loads a set of registered modules, allowing for the establishment of inter-module dependencies as well as service exposure and consumption by registered modules. There are a set of Controller Modules which implement core SDN controller functionality which is then either exposed by service APIs or by propagating as events to registered listener modules, thus enabling an event-driven programming model. These modules implement features such as OpenFlow switch management connection handling (FloodlightProvider and OFSwitchManager modules), inter-switch link discovery through Link Layer Discovery Protocol (LLDP) and Broadcast Domain Discovery Protocol (BDDP) (LinkDiscoveryManager module), network host discovery and tracking through packet-in inspection (DeviceManagerImpl module) and network topology and routing service (TopologyService module).
Floodlight defines a unique device as a VLAN/MAC Address pair and considers that there may be at most one attachment point per network domain [28]. In order to maintain compatibility the same principles were taken into consideration for the implementation of the proof-of-concept.

### 4.1.1 Proof of concept implementation

The proof-of-concept implementation followed Floodlight’s architectural design. A floodlight module was therefore implemented, encapsulating the clustering mechanisms detailed in Chapter 3 Section 3.1 and providing both APIs and triggering events to registered listeners. The main building blocks composing the developed module, depicted in Figure 4.1, are as follows:

- **Module and Service registration:** Handles all the tasks necessary to integrate with the Floodlight controller, including component initialization, declaring dependencies and exposing the services provided by both Global State Service and Global Topology Service by implementing the IFloodlightModule interface, which is then used by the Floodlight Module Loading System to register and execute the module.

- **Global State Service:** Handles the global state of the MIB and exposes an API to provide message exchange between Floodlight modules executed in different instances of the cluster, therefore allowing for the extension and enhancement of the distributed environment properties to module specific implementations. The message exchange API provided is based on the concept of named message queues, in which a module registers in a queue and is allowed to send messages to, and listen for, incoming messages on the queue. This module depends on the Cluster Communications Service to exchange messages with peer controller instances and on the Cluster Service in order to determine the address of the destination peer and whether or not a peer is a member of the cluster.
• **Global Topology Service:** Locally, it provides the same services as the TopologyService module by exposing an API to be consumed by other Floodlight modules and triggering events upon topology changes, using however the Global State Service's MIB instead of the local MIB. It also listens for local topology changes, such as new OpenFlow switch connections, new inter-switch links and hosts, updating its MIB and propagating the changes to peer instances accordingly. This block registers to a special queue in Global State Service reserved for the exchange of the messages specified in Table 3.1 under the scope of MIB update. All messages queued for sending through the exposed API are sent encapsulated (and de-encapsulated when received before being registered to the corresponding queue) within a special purpose container message that holds properties such as the queue identifier.

• **Cluster Service:** This block is responsible for all the clustering logic such as that of the registration of peer cluster instances (including the registration of the local instance with existing ones) described in Chapter 3 Section 3.1 and the exchanging and processing of all the messages stated in Table 3.1 under the scope of cluster membership. Messages are sent and received using the API provided by the Cluster Communications Service.

• **Cluster Communications Service:** Encapsulates all of the network communication details, providing a unified API to both Cluster Service and Global State Service, thus enabling them to send and receive messages to and from peer controller instances. The messages are encoded/decoded to a suit transmission format by the Message Codec block, and then sent and received by the network access specific implementations. The network access is done using Internet Protocol version 4 (IPv4) since it is the most widespread network protocol. The network access specific implementations are contained in the IPv4 Multicast and IPv4 Unicast blocks, allowing for the easy replacement of the network protocol used simply by implementing new blocks that expose the same API provided by the Cluster Communications Service.

• **Message Codec:** Encodes and decodes messages to a suitable format to be sent through the network, which is essentially an array of bytes. In this implementation, the method chosen to encode/decode is the serialization of Java objects, however, any other method can be implemented such as JavaScript Object Notation (JSON) or otherwise a special purpose Application Layer protocol in order to make messages compatible with any SDN controller implementation, regardless of the programming language it was developed in.

• **IPv4 Multicast:** All of the IPv4 multicast implementation is contained within this block. The multicast group used for this implementation is 224.0.1.20, which is a special group address reserved for private experiments [29] within the multicast addresses block for portocol traffic that is allowed to be forwarded through the internet [30].

• **IPv4 Unicast:** This block implements all of the IPv4 unicast using only TCP as it provides guaranteed delivery independently of message size.

The group communication paradigm specified in Chapter 3 Section 3.1 is implemented using the IPv4 multicast mechanism provided by the IPv4 Multicast block, which presented an additional challenge as IPv4 multicast does not provide a connection oriented communication, therefore requiring for manual fragmentation, ordering and reassembly of messages with size greater than the Maximum Transmission Unit (MTU). In order to overcome this limitation, without going into complex message delivery implementations and while still keeping the desired properties of group communication, a workaround was implemented in the Cluster Communications Service block in which if the size of a message targeting all

---

1 Considering the Open Systems Interconnection (OSI) model
cluster members is large enough not to fit into a single packet (in which case the IPv4 Multicast block will raise an exception), instead of using the multicast mechanism, unicast TCP sessions are established to all registered clustered members, conveying the message through these sessions. All of the messages specified in Table 3.1 within the scope of cluster membership and targeted at the whole cluster, are guaranteed to fit in a message which size is lesser than the MTU and are therefore not affected by this workaround, thus keeping the group communication properties required for cluster formation and maintenance intact.

In order to reduce the amount of data transmitted and at the same time provide a fail recovery mechanism, the Join cluster and Instance health messages defined in Table 3.1 were merged into the a single message (Join cluster), having each instance the responsibility of differentiating the message purpose according to whether the instance sending the message was already registered (in which case a timestamp indicating the last time the instance reported activity is updated) or not (in which case the registration process is executed).

In order for inter-switch links connecting OpenFlow switches controller by different instances to be properly detected and registered into the global topology, the LinkDiscoveryManager module had be modified in such a way that the incoming LLDP packets sent by other controller instances are processed according to the topology made available by the Global topology Service instead. This represents the only change required to the Floodlight core modules in order for the proof-of-concept implementation to work.

4.1.2 Distributed network policy module

Floodlight comes bundled with a set of network applications that provide network policy functionality out of the box. One of the bundled applications is the Forwarding module, which implements a forwarding policy that permits all flows to go through the network (also known as permit from any to any), installing the necessary flow entries in a reactive fashion [31].

The nature of this module makes it a perfect test subject for the proposed architecture, however, in order to better demonstrate the full potential of the proposed architecture this module was extended to take advantage of the inter-instance communication mechanism provided by the Global State Service so that the forwarding policy is only computed on the SDN controller instance managing the OpenFlow switch to which the device that initiates the flow is connected to. Controller instances managing other OpenFlow switches in the path that the flow takes to traverse the network are simply instructed which flow entries to add to which switches instead of having to compute the policy by themselves. This approach increases scalability as the computational cost of determining the actions to be performed for any given flow is only inflicted in one controller instance, leaving other controller instances free to compute the flows being admitted into the network through OpenFlow switches that they control directly.

This extended version of the Forwarding module, registers to a queue named after itself using the Global State Service API which will be used to exchange messages to coordinate policy programming actions throughout the network. Upon receiving a Packet-In for a packet that hit the table miss flow, it invokes the API of the Global Topology Service in order to determine if the destination device is already known in the topology (at global scope) and if so which is the shortest path between the source and destination devices. Should the destination device be unknown to the topology, the packet is flooded to all ports of the corresponding broadcast domain in the local OpenFlow switch and the process terminates. If however the destination is known in the global topology, it takes the path computed by the Global Topology Service and in turn computes the flow entry to be programmed in each OpenFlow switch pertaining to path and sends out Flow Programming Request messages to the controller instances managing the involved OpenFlow switches that are not locally managed. Flow Programming Requests are then processed by the receiving instances which notify the instance that sent it with a Flow Programming Confirmation.
message upon completion. The instance that computed the policy is only allowed to program the flow entry in the OpenFlow switch that triggered the Packet-In, after receiving Flow Programming Confirmation messages from all remote instances involved, thus making sure that there will be no other Packet-In triggered in the path.

4.2 Request Router

The request router component was implemented by integrating Linux virtual machines running Linux kernel 3.16 compiled with the IP Virtual Server (IPVS) module, with the Elastic SDN controller cluster. The IPVS module, pertaining to the Linux Virtual Server (LVS) Project, provides an efficient kernel-mode OSI layer 4 switching facility, suited to load balance client connections between several clustered servers. The implementation of the Anycast communication paradigm assumes that the management network infrastructure, to which the management interfaces of OpenFlow switches and SDN controller instances are connected to, supports the Border Gateway Protocol (BGP) routing protocol and is able to establish BGP peering with the request router instances. The configuration of the management network infrastructure is not within the scope of this work.

4.2.1 Linux Virtual Server (LVS)

The LVS project, in particular the IPVS module, provides an out of the box load balancing facility for client-server applications connecting through the network and using the IPv4 stack. This module, implemented as a kernel module, introduces minimal overhead in the client-server communication and can be deployed as a cluster, in which mode forwarding states will be synchronized between cluster members, thus allowing for the required infrastructure resiliency seeing that a request router instance failure will not impact normal operation as its peer instance is able to take over the forwarding function seamlessly.

The load balancing is performed at the Transport Layer of the OSI model, supporting both User Datagram Protocol (UDP) and TCP protocols. IPVS receives the connection data from the client for a configured service cluster and forwards it to the one of the servers available for that cluster in one of three possible ways:

- In NAT mode, IPVS terminates the client connection and initiates a new connection with the service cluster instance determined by the load balance algorithm, forwarding all the packets received from the client to the server, masquerading however the source IP Address, effectively forcing data communications to go through the load balancer in both ways. While this mode offers the advantage of allowing the service cluster servers to be located anywhere on the network and requiring no configuration on the clients nor the servers, it has limited scaling capabilities as NAT itself is limited when it comes to scaling.

- In IP Tunneling mode, IPVS does not terminate the client-server communication and instead encapsulates the data being received from the client inside an IP-IP packet and forwards it to the service cluster instance determined by the load balance algorithm. This mode also offers the advantage of allowing the service cluster servers to be located anywhere on the network, requiring however that the servers support the IP-IP tunneling (which introduces minimal overhead) and the configuration of loopback interfaces with the service cluster IP Address in all of the service cluster servers and disallow Address Resolution Protocol (ARP) on that loopback. It is however much more scalable than NAT mode, since this mode does not perform NAT nor does it process packets coming from the server to the client.
In Direct Routing mode, IPVS behaves essentially in the same way as in the IP Tunneling mode, however, instead of performing encapsulation, the load balancer is required to be on the same OSI layer 2 as the service cluster servers, forwarding the packets received by the client directly to the service cluster instance determined by the load balance algorithm. While this mode offers the same advantages as the IP Tunneling mode and without the IP Tunneling drawback, the requirement of being in the same OSI layer 2 as the service cluster server represents a limitation in terms of solution design.

Both IP Tunneling and Direct Routing modes met the requirements set for the request router component. Although for the purpose of this implementation Direct Routing has been chosen due to constraints with the deployment target, IP Tunneling mode should be used instead as it does not limit the deployment topology and IP-IP tunneling is natively supported in modern Linux kernels, requiring only that management network routers have the Reverse Path Forwarding (RPF) feature disabled. It is worth to note however that the forwarding mode is determined by server and not by service cluster, which means that it is possible to use more than one of these modes simultaneously for different servers in the same service cluster.

IPVS also offers a number of load balancing algorithms that can be used to determine to which service cluster instance the client connection should be forwarded to:

- The **Round Robin** algorithm iterates through the configured servers, equally distributing client connections to each server.

- The **Weighted Round Robin** algorithm allows the definition of weights per server, which are then taken into account when assigning client connections, such that servers with higher weight are assigned more connections.

- The **Least-Connections** algorithm assigns new client connections to the server with least active connections.

- The **Weighted Least-Connections** algorithm is the default algorithm and like the Weighted Round Robin it allows for the definition of weights per server. It then assigns new client connections to the server with the lowest Connections to Weight ratio.

- The **Locality-Based Least-Connections** algorithm assigns incoming connections to the same server if it is not overloaded or unavailable, in which case it assigns the connection to another server with fewer connections.

- The **Locality-Based Least-Connection with Replication** algorithm defines a server set that can serve the client’s request and forwards the packets to the server in that set with least active connections. If all servers in the set are overloaded, a new server belonging to the service cluster is added to the set of servers that can serve the client and the packets are forwarded to that server.

- The **Destination Hashing** algorithm consistently assigns server by looking up a hash table using the service cluster IP Address as key.

- The **Source Hashing** algorithm consistently assigns server by looking up a hash table using the client’s IP Address as key.

- The **Shortest Expected Delay** algorithm assigns client connections to the server that is expected to have the shortest delay, which is estimated by \(\frac{(C+1)}{U}\) where \(C\) is the number of connections for the server and \(U\) the Weight assigned to the server.
The Never Queue algorithm assigns client connections to one of the idling servers. Should there be no idling server the Shortest Expected Delay is used to assign a server.

The optimal algorithm, according to the requirements set for the request router, would be the Source Hashing since it guarantees a deterministic load balancing decision. However, for the purposes of testing this implementation using a network emulator, the Least Connections algorithm was instead chosen since the Source Hashing algorithm would always assign the same server because the source IP Address would always be the same for all OpenFlow switches (that of the emulator).

4.2.2 Elastic Cluster Integration

The request router must integrate with the SDN controller elastic cluster in order to determine the set of active controller instances to which it can forward OpenFlow switch connections. To do that, a new component was developed, much like the developed Floodlight module, joining also the same multicast group in order to receive Join messages through which it infers the active set of controller instances. Because the message encondig/decoding is performed using Java’s object serialization, this components implementation necessarily had to be made using the Java language as well. The building blocks of the component are depicted in Figure 4.2 and are as follows:

- **Request Router**: This block implements all of the load balance logic, handling events generated by the Elastic Cluster Integration and triggering the necessary configuration changes to IPVS through the LVS Integration block.
- **LVS Integration**: This block handles the necessary parsing and command issuing with the ipvsadm tool in order to ensure the correct configuration of IPVS.
- **Linux Shell Integration**: An abstraction layer to allow for the execution of Linux shell commands.
- **Elastic Cluster Integration**: This block encapsulates the same cluster membership logic present in the Cluster Service block of the Floodlight elastic clustering module, and is intended to detect cluster member changes and trigger adequate events to the Request Router block.
• **Cluster Communications Service**: Implements essentially the same logic for listening to cluster messages as its homologous block in the Floodlight elastic clustering module.

• **Message Codec**: Implements exactly the same logic as its homologous block in the Floodlight elastic clustering module.

• **IPv4 Multicast**: Implements exactly the same logic as its homologous block in the Floodlight elastic clustering module.

The component interfaces with the IPVS module through ipvsadm, a command line tool designed to control IPVS enabling the creation of service clusters and the addition and removal of servers from service clusters. To do so, the developed component launches new processes for each execution of the ipvsadm tool necessary and parses its output in order to extract status information and command execution success.

### 4.2.3 Anycast routing

The request router anycast communication paradigm is implemented by adding a third component to the solution, the Quagga Routing Suite. The Quagga Routing Suite is a routing software suite that implements several routing protocols such as Routing Information Protocol (RIP), Open Shortest Path First (OSPF), Intermediate System to Intermediate System (IS-IS) and BGP as well as the MPLS Label Distribution Protocol and Bidirectional Forwarding Detection (BFD).

The anycast communication paradigm is achieved by having the request router instances establish a BGP session with the closest management network router and announcing the cluster IP Address to which OpenFlow switches will establish management connections to. In practice, this means that each router in the management network will have routes to the cluster IP Address that have the next-hop defined as whichever request router instance is closer to them, and as expected different routers in different network regions will have different next-hop defined for that route. The Equal-cost multi-path (ECMP) feature of BGP is also used to guarantee that load balancing is also achieved between request router instances, as it allows management network routers to hold multiple active routes for the same destination through different next-hops, thus providing the mechanism to load balance between multiple request router instances provided that they are at the same routing distance.

To improve the robustness of the solution and minimize convergence time in case of a request router instance failure, BFD is also used. BFD is a Hello protocol intended to provide fast failure detection of a network path between two systems regardless of which component might have failed [35]. When BFD is combined with BGP the time it takes to expire a route for a next-hop that is no longer available is reduced from minutes to microseconds [35][36].
Chapter 5

Evaluation

Ultimately the goal to be achieved by this work is an architecture for a network infrastructure that while solving existing manageability issues improves the scalability of OpenFlow-based SDN controllers operating in reactive mode. This chapter presents and discusses the results obtained from various tests performed on the proof-of-concept implementation described in Chapter 4 in order to validate that the proposed architecture described in Chapter 3 met its objectives. The scope of the testing ranges from strictly functional tests, in which the correctness of architectural model and of the implementation are validated, to some performance tests to the distributed network policy module.

5.1 Test environment

A virtual test environment was instantiated for the evaluation of this work resorting to a shared IaaS provider. This environment was composed of six VMs which where allocated for the following purposes:

- 1 VM was allocated to execute the Mininet network emulator
- 2 VMs were allocated to the execute the Request Router component described in Chapter 2 Section 3.2 and Chapter 4 Section 4.2
- 3 VMs were allocated to execute the SDN controller instances described in Chapter 2 Section 3.1 and Chapter 4 Section 4.1.

The Operating System chosen to run in these VMs was the version 8 of the Debian Linux distribution since it provided a lightweight environment built on top of stable versions of the libraries required to execute the components supporting this implementation. Due to the lack of resources to emulate a complete management network infrastructure, it was not possible to test the routing integration described in Chapter 4 Section 4.2.3. However, since this implementation uses only proven components and protocols - Quagga, BGP and BFD - it is not expected that any problem arises from this part of the request router component.

5.1.1 Network testbed

Because the test environment was restricted to a virtualized support, it was also necessary to provide a virtualized network testbed solution. Mininet is a network emulator written in python, that is capable of emulating switches and hosts in custom defined topologies, by employing process-based virtualization and making use of Linux’s network
Because its emulated switches are able to support the OpenFlow protocol, Mininet is the network emulator of choice for SDN testbeds. Version 2.2.1 of Mininet was used to emulate the topology in Figure 5.1, which was used in the tests described in this chapter.

5.2 Elastic SDN controller cluster

The first batch of tests carried out were functional tests, designed to validate the correctness of the implementation and the validity of the architecture. For the validation of the SDN controller elastic cluster described in Chapter 4 Section 4.1, the correctness of such implementation greatly depended on the correctness of the Floodlight controller, and it is therefore considered that any behavior coherent with that of the base version of Floodlight is therefore correct. Furthermore, all architecture-specific behavior must be validated with the requirements and desired behavior described in Chapter 3 Section 3.1.

The request router, while partially implemented as stated in Section 5.1, was also subject to testing to the remaining components that were implemented in the test environment. The validation of the correctness of the request router is validated with the requirements and desired behavior described in Chapter 3 Section 3.2 and implementation specific particularities indicated in Chapter 4 Section 4.2.

5.2.1 Scenarios

For the SDN controller elastic cluster specific functional tests, the scenarios to be tested are as follows:

1. Validate cluster membership behavior, consistency and exchanged messages. To do so the first two controller instances are to be initiated simultaneously and let them form a cluster while monitoring the network for exchanged messages. After convergence has been reached, a third controller instance is to be introduced and let it join the cluster while still monitoring the network for exchanged messages.

2. Validate MIB consistency throughout the cluster and exchanged messages by manipulating the network topology to add, remove and simulate failure of OpenFlow switches and hosts as well as simulate switch management transference between available controller instances by provoking failure of cluster instances. The management network is to be monitored for exchanged messages throughout the tests.
3. Validate the correct programming of OpenFlow switches. This functionality is to be validated by means of executing the module implemented as described in Chapter 4 Section 4.1.2 to allow TCP connections between Host 1 and Host 3 as well as between Host 4 and Host 5.

For the request router specific functional tests, the key points tested are as follows:

1. Validate request router clustering and state replication by having OpenFlow switches establish management connections to the IP Address of the SDN controller cluster and validating that both request router instances have the same connection state information.

2. Validate that the component developed as described in Chapter 4 Section 4.2.2 properly integrates with the SDN cluster as well as with LVS’s ipvsadm tool by provoking SDN controller cluster membership changes.

5.2.2 Results

The results obtained from the execution of the functional tests on the experimental implementation show that the proposed architecture provides the desired properties and moreover that the implementation’s behavior is coherent with that of the Floodlight. Furthermore, the request router also complied with the specifications and expected implementation behavior.

Through the execution of the functional tests for the SDN controller elastic cluster, it was found that all relevant network events such as switch, host, links registration/de-registration as well as inter-switch adjacencies are being correctly and timely detected and properly handled to update the local MIB and trigger appropriate messages to peer cluster members, which also update their MIB accordingly. It was also confirmed that all messages described in Table 3.1 within the scope of cluster membership and targeting the whole cluster are sent using multicast, therefore keeping the desired group communication properties described in Chapter 3 Section 3.1.

The functional tests to which the request router was subjected to were used to validate its correct behavior. However, these tests also led to the discovery of a Master/Master cluster feature instead of the predicted Master/Slave cluster as per LVS’s documentation.

5.3 Distributed network policy module tests

The use-case scenario of a global policy application that was previously functionally tested was also subject to performance benchmarking. To that extent, the same scenario used to validate the correct programming of OpenFlow switches was also used to retrieve latency metrics for the programming process.

5.3.1 Scenario

Once again, this scenario specified the correct programming of the OpenFlow switches in order to allow for the establishment of TCP connections between Host 1 and Host 3 as well as between Host 4 and Host 5. To fully test the flow programming capabilities, Flow Entries were configured with matches for OSI Layer 2, 3 and 4 source and destination fields as well as OSI Layer 4 protocol number and physical port through which the traffic was being admitted into the network, coming to a total of 8 match fields per Flow Entry.

Since the tests were performed using TCP connections, bi-directionality is implied, which means that for
every TCP session opened there will be two Flow Entries programmed in each of the involved OpenFlow switches. For completeness purposes and in order to provide scalability metrics, these tests were performed using a two-instance controller cluster and repeated using a three-instance controller cluster, with OpenFlow switches evenly distributed among them by the request router component.

5.3.2 Results

The results obtained are stated in Tables 5.1 and 5.2 and plotted in Figure 5.2.

<table>
<thead>
<tr>
<th>1 connection</th>
<th>5 Simultaneous connections</th>
<th>10 Simultaneous connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>6207 milliseconds</td>
<td>6618 milliseconds</td>
<td>11714 milliseconds</td>
</tr>
<tr>
<td>6209 milliseconds</td>
<td>6414 milliseconds</td>
<td>14918 milliseconds</td>
</tr>
<tr>
<td>4406 milliseconds</td>
<td>6614 milliseconds</td>
<td>11270 milliseconds</td>
</tr>
<tr>
<td>6406 milliseconds</td>
<td>6410 milliseconds</td>
<td>11709 milliseconds</td>
</tr>
<tr>
<td>6406 milliseconds</td>
<td>6609 milliseconds</td>
<td>14919 milliseconds</td>
</tr>
<tr>
<td>6406 milliseconds</td>
<td>6414 milliseconds</td>
<td>11705 milliseconds</td>
</tr>
<tr>
<td>6406 milliseconds</td>
<td>6413 milliseconds</td>
<td>10725 milliseconds</td>
</tr>
<tr>
<td>6006 milliseconds</td>
<td>6213 milliseconds</td>
<td>10738 milliseconds</td>
</tr>
<tr>
<td>6206 milliseconds</td>
<td>6609 milliseconds</td>
<td>11722 milliseconds</td>
</tr>
</tbody>
</table>

Table 5.1: Flow programming results with a two-instance controller cluster

<table>
<thead>
<tr>
<th>1 connection</th>
<th>5 Simultaneous connections</th>
<th>10 Simultaneous connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>6606 milliseconds</td>
<td>6614 milliseconds</td>
<td>6681 milliseconds</td>
</tr>
<tr>
<td>6406 milliseconds</td>
<td>6614 milliseconds</td>
<td>6678 milliseconds</td>
</tr>
<tr>
<td>6206 milliseconds</td>
<td>6610 milliseconds</td>
<td>10688 milliseconds</td>
</tr>
<tr>
<td>6406 milliseconds</td>
<td>6614 milliseconds</td>
<td>6682 milliseconds</td>
</tr>
<tr>
<td>6206 milliseconds</td>
<td>6414 milliseconds</td>
<td>6685 milliseconds</td>
</tr>
<tr>
<td>6206 milliseconds</td>
<td>6614 milliseconds</td>
<td>10686 milliseconds</td>
</tr>
<tr>
<td>6406 milliseconds</td>
<td>6618 milliseconds</td>
<td>6690 milliseconds</td>
</tr>
<tr>
<td>6406 milliseconds</td>
<td>6609 milliseconds</td>
<td>10698 milliseconds</td>
</tr>
<tr>
<td>6006 milliseconds</td>
<td>6610 milliseconds</td>
<td>7092 milliseconds</td>
</tr>
</tbody>
</table>

Table 5.2: Flow programming results with a three-instance controller cluster

5.4 Analysis

Due to Mininet’s limitations, switches and hosts cannot be added or removed dynamically, however, they can be isolated from the network by shutting down all ports and in the particular case of switches removing configuration for controller connection. It is not evident however that this limitation impacted the tests in anyway.

The results obtained from the tests to the SDN controller elastic cluster showed that both cluster membership and MIB are kept consistent throughout cluster members after adding, removing and provoking deliberate failures on controller instances as well as OpenFlow switches.

The results obtained while benchmarking the distributed network policy application also yielded positive results, showing that it is already possible to have a 33% efficiency increase by adding a third SDN controller instance to the cluster when there are more than 10 concurrent TCP sessions traversing the controlled network, thus achieving the desired scalability property. However, if we take into account the absolute time required for the completion of flow configuration operations, the results obtained are
unsatisfactory, which might be related to factors such as the overall load of the hardware supporting the test environment, virtualization overhead, network emulation environment overhead and finally to a suboptimal implementation of TCP unicast connections between controller instances, which instead of caching connections between SDN Controller instances for further communications are being opened and closed to send a single message.

The global memory footprint of the controller instances appear to have suffered an increase of approximately 10% when executing the prototype Floodlight module.

Figure 5.2: Performance indicators for the distributed network policy module
Chapter 6

Conclusion

SDN presents itself as the solution for traditional network management problems, and OpenFlow while being the most promising protocol implementing the SDN architecture offers two different approaches to program the network, namely reactive and proactive programming. While reactive programming offers a much more flexible and convenient method to program the network, it comes with a computational cost that becomes a limitation when applied to large scale networks.

In this work we proposed a novel architecture for OpenFlow controllers based on the concept of SDN controller elastic clustering and coupled with a load balance infrastructure. The proposed architecture provides a scalable solution for reactive programming in large networks and, at the same time, increased redundancy and resilience in case of failure of a single controller, while keeping the centralized management paradigm typical to SDN and without introducing complex distributed coordination algorithms in the controller implementation.

To validate the proposed architecture, we implemented a prototype of the SDN controller cluster by extending version 1.1 of the Floodlight controller. A prototype of the load balance infrastructure was also implemented, using free open-source components such as Quagga and IPVS and relaying in proven standards such as BGP and BFD.

The global prototype implementation of the proposed architecture showed that it is able to provide a fully functional elastic controller cluster for SDN applications, thus removing the limitations of OpenFlow’s reactive programming when deployed in large networking environments.

Along with the implementation of the prototype a sample SDN application was also developed in order to demonstrate the use of the architecture. Tests conducted to this applications showed that when network traffic increases and new SDN controller instances are added to the cluster, a performance gain of 33% is attained, further proving the desired scalability properties of the proposed architecture.

Although the implementation described in this work served its purpose as a prototype, a more carefully developed solution that goes deeper into the controller core implementation will provide a considerable increase in performance.

The standardization and extension of the message set exchanged between controller instances pertaining to the same cluster and implementation as an Application layer protocol will make way for the existence of controller clusters composed of controller instances implemented in different languages and more fit to handle specific needs.
Bibliography


Glossary

Anycast  A one-to-nearest communication paradigm in which multiple instances of the receiver exist and datagrams sent by a sender are delivered to the nearest receiver. [18, 25]

IP Address  A numeric address globally identifying a device connected to a network running the Internet Protocol. [10, 16, 25, 28, 31]

Keepalive  A periodic message sent from the station at one end of a communication channel to the station at the other end of the channel in order to check that the channel is still active and keep it active. [14, 17]

MAC Address  The Media Access Control Address is a 48-bit address typically imprinted in the network access hardware that uniquely identifies a device within a local area network scope. [22]