A SCALABLE ARCHITECTURE FOR OPENFLOW CONTROLLERS

Filipe Azevedo
Instituto Superior Técnico
Universidade de Lisboa
Email: filipe.azevedo@tecnico.ulisboa.pt

***Abstract*—The architectural principles of Software-Defined Networking (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.**

I. INTRODUCTION

The current notion of *Software-Defined Networking* (SDN) was developed in Stanford University while researching for a network configuration protocol, from which resulted *OpenFlow*, and is a concept 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, it is susceptible to SDN Controller failures, requires a 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.

In this document, we propose a new architecture that offers both load balance between different SDN 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 SDN Controllers and the introduction of a load balancing layer. The clustering mechanism takes the notion of the SDN Controller as a logically centralized component and introduces the concept of *elastic SDN Controller clustering*, implementing the mechanisms necessary to scale out the SDN Controller as needed and in real time by providing means to add and remove SDN Controller instances from the cluster as needed with neglectable or 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 insures an equal distribution of OpenFlow switches across available SDN controller instances.

This document covers the state of the art in Section II, followed by the proposed architecture in Section III, its implementation in Section IV and finally the evaluation in Section V.

II. State of the Art

A. Software-Defined Networking

Software-Defined Networking relies on two fundamental changes to the traditional networking design principles: the decoupling of the network control and forwarding planes and network programmability [1]. Today's network nodes are built on the architectural paradigm of three strongly-coupled planes of functionality: the *management plane*, enables service monitoring and policy definition, the control plane, which enforces the policies in network devices and the data plane which efficiently forwards data.

Software-Defined Networking 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 functionalities while keeping the Forwarding implemented in the network nodes [1]–[3] thus creating the premises for a network infrastructure that harnesses the benefits of both distributed data forwarding and centralized management, overcoming the limitations of 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 Application Programming Interfaces (APIs), to be consumed by the network applications. 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 [1]. The policies 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 software-defined networks in three layers, the *Application* layer, the *Control* layer and the *Infrastructure* [2], [3]. The Infrastructure layer is where the network nodes reside, performing 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, taking into account factors such as (but not restricted to) network topology and the network operator input.

The implementation of the Control layer is known as *Software-Defined Network Controller* (SDN Controller), and it is the main building block of Software-Defined Networks. A SDN Controller exposes a northbound interface, intended for interaction with network applications, corresponding to the Specification functionality and a southbound interface intended for interaction with the network devices in the scope of the Distribution functionality, as depicted in figure 1. In some cases the SDN Controller might also define eastbound/westbound interfaces for integration with other SDN Controllers (mostly seen in distributed controllers) [2]. There now are several implementations for SDN Controllers [4], as well as southbound interfaces [2], however, because there is a dominant southbound interface - OpenFlow - the vast majority of network elements supporting the SDN architecture do so by implementing only OpenFlow.

B. OpenFlow

OpenFlow is a standard protocol for forwarding plane programming, defined by the Open Networking Foundation (ONF) and designed for Ethernet networks [1]. 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 [1]. Forwarding decisions are based in the notion of flow, which, according to the IETF, is "a sequence of packets from a sending application to a receiving application" [5].

Having had several versions released since 2008, when version 0.8.0 debuted, OpenFlow is currently on version 1.4. However, because there has not been significant changes in version 1.4 when compared to version 1.3 and because current hardware and software switch implementations mainly implement version 1.3, we will focus on the latter for the purpose of this work.

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*. For the purposes of this work we will focus on the OpenFlow Channel and Processing Pipeline blocks.

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; *counters* that increment each time a packet is successfully matched against the flow entry, and finally *cookies*, which is a field 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 performed by the switch. 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 in order to forward the packet (defined by matching entries) or alternatively explicitly dropping the packet. If no entries that match the packet occurred then either there is a table-miss

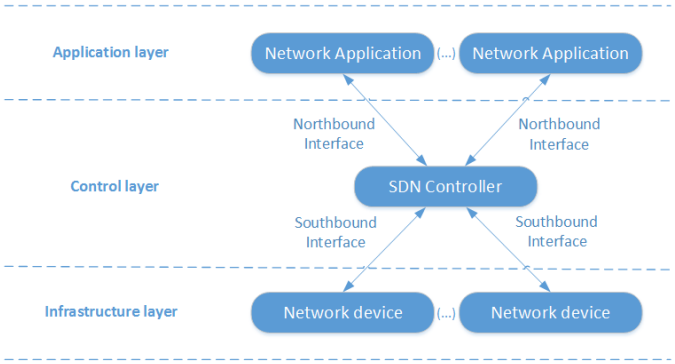

Fig. 1. Software-Defined Networks architecture

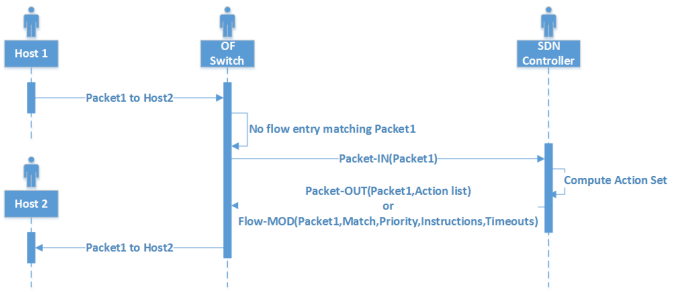

Fig. 2. Reactive flow programming

flow, usually redirecting the packet to the SDN Controller, or the switch drops the packet completely.

The OpenFlow Channel defines the interface for communications between the OpenFlow switch and the SDN Controller, through which controllers can create, modify and remove flow entries from flow tables in a proactive or reactive fashion.

C. OpenFlow switch to SDN Controller connection

All communications between an OpenFlow switch and an SDN controller are, by default, encrypted using TLS, although it is also possible to use plain TCP. These communications are always initiated by the OpenFlow switch, and each switch may have one or more SDN Controller IP addresses specified in its base configuration. The OpenFlow switch is required to maintain active connections witch 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* [6].

The Master role grants the SDN Controller full management access to the switch, allowing it to program flow entries, 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 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.

Although the OpenFlow specification provides mechanisms for resilience and load balancing across multiple SDN Controller instances, it requires the SDN Controllers to load balancers between themselves and also to reconfigure every single switch every time an instance is added or removed from the SDN Controller cluster.

D. OpenFlow flow programming

SDN Controllers program flow entries in OpenFlow capable switches by issuing *flow table modification* messages either in a proactive or in a reactive fashion. These messages allow the SDN Controller to manipulate flow tables by adding new flow entries and modifying or removing existing flow entries.

When proactively programming flow enties, the SDN Controller issues a flow table modification message to predefine a forwarding behavior ahead of data transmission. These resulting flow entries usually have their timeouts set to zero, meaning that they will not expire, and are referred to as static flow entries [6]. 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 instant. A typical usage of static flow entries are the table-miss flow entries.

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 alternatively 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 [6], as depicted in figure 2. 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. 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.

III. Architecture

The basic principle of the proposed architecture is to replace the Controller by a cluster of Controllers, which keep a consistent management information base (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 the introduction of 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, therefore providing an *elastic* structure.

A. SDN Controller elastic clustering

As previously mentioned, although OpenFlow allows for an OpenFlow switch to be connected to several SDN Controllers simultaneously, the latter are expected to have some sort of coordination between them, 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 controller cluster and configures an architecture such that of figure 3.

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 maintain the aforementioned key aspect of SDN Controllers, active instances must propagate their management information base (MIB) on to their cluster peers and be prepared to update their own MIB with updates issued by their peers. The propagation of these updates must be triggered at least when network events are detected and when new adjacencies between OpenFlow switches controlled by different instances are detected. When a instance propagates updates to other peer instances, it must do so guaranteeing causal consistency.

In order to improve scalability and resiliency, new controller instances can be added, integrated with existing instances and removed on demand from the cluster. To that extent, when a new instance joins the cluster it must beforehand generate a unique session identifier, advertise itself along with its session identifier to the remaining cluster peers as a new participating instance and then synchronize its MIB with another participating instance. Upon receiving a join message,

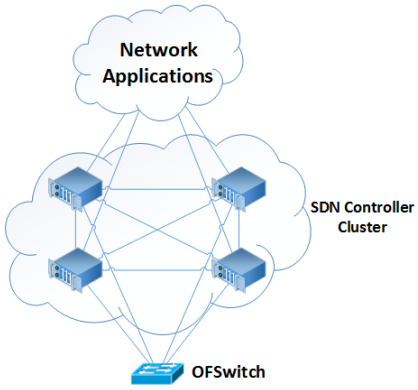

Fig. 3. OpenFlow SDN Controller clustering

existing cluster members must validate that the instance from which the message originated is already a member of the cluster with a different session identifier, in which case the peers must invalidate any information relative to that instance in their MIB and then register the instance and its new session identifier. Cluster membership is kept by propagating a *keepalive* message every Δ seconds. Should an instance α fail to propagate a keeplive message three times in a row (3Δ), peer instances must assume that instance α failed an is no longer part of the cluster, therefore invalidating any information relative to that instance in their MIB.

B. Load balancing SDN Controllers

The mechanism provided by OpenFlow to load balance between clustered 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 4

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 α of OpenFlow switches may process more traffic flows than a subset β, resulting in a higher computational load on the instance controlling subset α when compared to the instance controlling subset β.

In order to provide a solution that keeps the advantages and discards the disadvantages of both approaches, a new load balancing component. This new component, the *request*

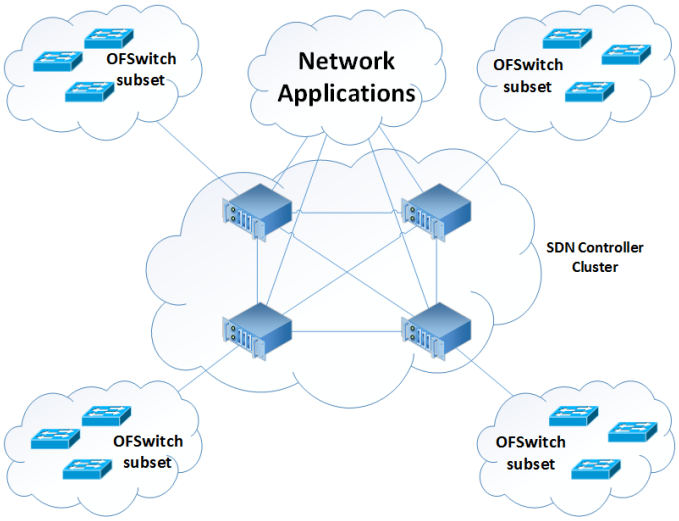

Fig. 4. Static SDN Controller load balancing

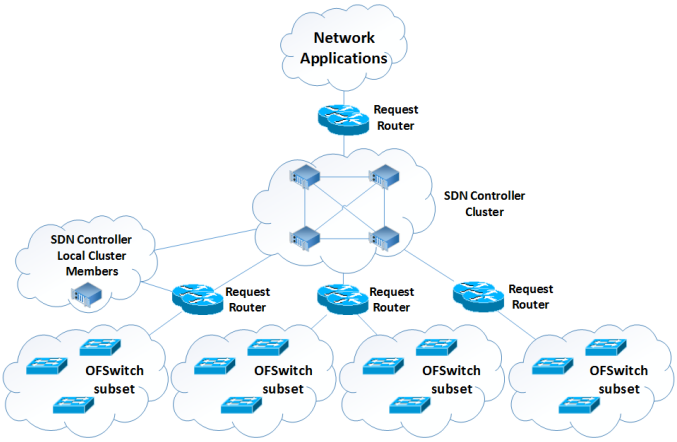

Fig. 5. Proposed architecture overview

router, encapsulates all the controller instances as a single virtual instance from the OpenFlow switches and network applications point of view.

The request router must be completely transparent to all three components of the SDN architecture, and provide load balance and infrastructure resilience.

To do so, the request router must implement a clustering mechanism that allows the existence of several request router instances. These instances must also integrate with the SDN Controller cluster in order to keep an updated list of active controller instances. Because large network topologies often span over several geographical locations, the request router must also support *anycast routing*, guarantying that any given OpenFlow switch will connect to whichever request router is logically closer. However, contrary to the controller cluster members, the request router cluster members are allowed to have different configurations, allowing for a request router instance to prefer a local controller instance over a remote one on the same cluster.

OpenFlow switches within the same management domain must all be configured with the same controller IP address - that of the request router cluster. The connection is initiated by the OpenFlow switches to a member of the request router cluster that will consistently forward the connection to a controller instance.

The architecture portrayed in figure 5 is obtained by combining *SDN Controller elastic cluster* with the request router cluster.

IV. Implementation

An experimental implementation of the proposed architecture depicted in figure 5 has been carried out by extending the *Floodlight* SDN Controller, a modular SDN Controller developed in Java, through the development of a module that implements the clustering functionalities described in section III-A and by implementing the functionalities described in III-B on top of Linux Virtual Server (LVS), more specifically on top of the IP Virtual Server (IPVS) component, along with a managing component that integrates with the developed Floodlight module.

A. SDN Controller elastic clustering

The message propagation between instances in the same cluster has been implemented using IPv4 multicast or IPv4 unicast with Transmissiong Control Protocol (TCP) for specific use-cases such as MIB synchronization, which requires guaranteed delivery. The Link Layer Discovery Protocol (LLDP) implementation distributed with Floodlight has also been extended in order to detect adjacencies between OpenFlow switches controlled by different controller instances.

The developed module holds its own data structures, storing global state information such as a global network graph that includes OpenFlow switches, network links annotated with properties such as link capacity, network hosts and controller affinity for each network switch. MIB information exchange is performed by serializing Java objects and sending them through the network either to the multicast group or to a specific instance, depending on the use-case being considered. This module also exposes a set of services (APIs) to be consumed by other Floodlight modules. These APIs allow other modules to take advantage of SDN controller instance coordination and are designed to provide both access to the global network view and a channel of communication between instances that is specific for a module.

B. Load balancing SDN Controllers

The request router component has been implemented on Linux virtual machines running Linux kernel 3.2 compiled with the IPVS module, which provides an efficient OSI layer 4 switching facility. IPVS is used in IP tunneling mode, which encapsulates the received packet in IP-IP, an IP tunneling technique that encapsulates an IP packet inside another IP

packet, therefore becoming completely transparent for both SDN controllers and OpenFlow switches. This technique requires all controller instances to have a loopback interface configured with the cluster's virtual IPv4 address. The configuration of IPVS is performed automatically by a developed component that also joins the controller cluster multicast group in order to monitor cluster membership events and update IPVS accordingly. The IP anycast functionality is obtained by adding a loopback interface in each request router instance, configured with the request router's cluster virtual IP, and by running Border Gateway Protocol (BGP) that peers with the core network and advertises a path to the request router's cluster virtual IP through the request router instance. The BGP protocol is executed on top of Quagga routing software suite.

C. 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 [7]. 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, 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.

V. Evaluation

This Section presents and discusses the results obtained from various tests performed on the experimental implementation in order to validate that the proposed architecture described.

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.

A. Test environment

A virtual test environment was instantiated for the evaluation of this work resorting to a shared Infrastructure as a Service provider. This environment was composed of six Virtual Machines (VMs), one for emulating the network topology, two

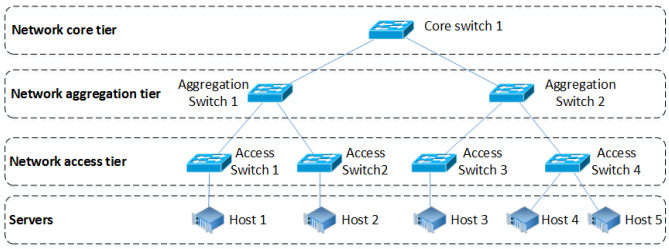

Fig. 6. Typical datacenter topology

for running the request router cluster, and three for running SDN controller instances. 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.

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 namespaces [8]. 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 6, which was used in the tests described in this chapter.

B. 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, 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 previously described in Sections III and IV.

The request router, while partially implemented, was also subject to testing for 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 previously described. 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 distributed network policy module implemented as described in Section IV-C to allow TCP connections between Host 1 and Host 3 as well as between Host 4 and Host 5.

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.

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 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 request router properly integrates with the SDN cluster as well as with IPVS by provoking SDN controller cluster membership changes.

The request router also complied with the specifications and expected implementation behavior.

C. 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. 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. Ten samples were taken using 1, 5 and 10 concurrent connections respectively for each of the SDN controller cluster topologies. The results obtained are stated in Tables I and I and summarized in Figure 7.

1 TCP connection	5 TCP connections	10 TCP connections
6207 milliseconds	6618 milliseconds	11714 milliseconds
6209 milliseconds	6414 milliseconds	14918 milliseconds
4406 milliseconds	6614 milliseconds	11270 milliseconds
4406 milliseconds	6410 milliseconds	11709 milliseconds
6406 milliseconds	6609 milliseconds	14919 milliseconds
6406 milliseconds	6414 milliseconds	11705 milliseconds
6406 milliseconds	6413 milliseconds	10725 milliseconds
6006 milliseconds	6213 milliseconds	10738 milliseconds
6206 milliseconds	6609 milliseconds	11722 milliseconds

TABLE I
RESULTS WITH A TWO-INSTANCE CONTROLLER CLUSTER

1 TCP connection	5 TCP connections	10 TCP connections
6606 milliseconds	6614 milliseconds	6681 milliseconds
6406 milliseconds	6614 milliseconds	6678 milliseconds
6206 milliseconds	6610 milliseconds	10686 milliseconds
6406 milliseconds	6614 milliseconds	6682 milliseconds
6206 milliseconds	6414 milliseconds	6685 milliseconds
6206 milliseconds	6614 milliseconds	10686 milliseconds
6406 milliseconds	6618 milliseconds	6690 milliseconds
6406 milliseconds	6609 milliseconds	10698 milliseconds
6006 milliseconds	6610 milliseconds	7092 milliseconds

TABLE II
RESULTS WITH A TWO-INSTANCE CONTROLLER CLUSTER

VI. ANALYSIS

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 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

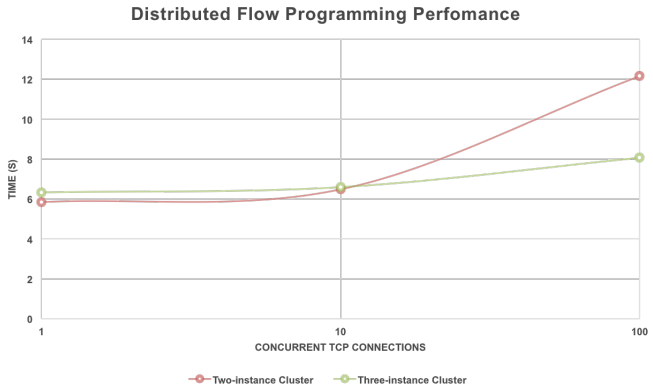

Fig. 7. Performance indicators for the distributed network policy module

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.

VII. Conclusion

We proposed a novel architecture for OpenFlow controllers based on the concept of SDN controller elastic clustering. 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. The prototype implementation of the proposed architecture showed that it is able to provide a full functional controller for SDN applications. Tests conducted to the distributed network policy application 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 of the message set exchanged between controller instances pertaining to the same cluster and implementation as an Apllication 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.

References

[1] Open Networking Foundation (ONF), "Software-Defined Networking: The New Norm for Networks," White paper, Open Networking Foundation (ONF), White paper, April 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[2] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," ser. Proceedings of the IEEE, vol. 103, no. 1. IEEE, 2015.

[3] Open Networking Foundation, "Software-Defined Networking (SDN) Definition." [Online]. Available: https://www.opennetworking.org/sdn-resources/sdn-definition

[4] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou, "Feature-based comparison and selection of software defined networking (sdn) controllers," *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*, January 2014.

[5] Internet Engineering Task Force (IETF), "What is a Flow?" http://www.ietf.org/proceedings/39/slides/int/ip1394-background/tsld004.htm, accessed: 2014-11-27.

[6] Open Networking Foundation (ONF), "OpenFlow Switch Specification v1.3.4," OpenFlow Spec, Open Networking Foundation (ONF), OpenFlow Spec, March 2014. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.4.pdf

[7] Big Switch Networks, Inc., "Floodlight Forwarding Module," https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Forwarding, accessed: 2015-09-12.

[8] B. Brandon Heller, "Mininet," http://mininet.org/overview/, accessed: 2015-04-18.