Contributions for the Standardisation of a SDN Northbound Interface for Load Balancing Applications

Diogo Coutinho
Instituto Superior Técnico
University of Lisbon
diogo.m.coutinho@tecnico.ulisboa.pt

***Abstract*— Software-Defined Networking (SDN) is a new paradigm that is emerging in networking. It has the goal to contribute to deal with the inherent complexity of today's networks. The main concept behind SDN is the separation of the control plane from the data plane, centralizing the control plane of the network devices in a server or SDN controller.**

OpenFlow emerged as the first widely adopted protocol for the communication between the centralized control plane and the data plane, known as the Southbound Interface (SBI). Its success has led SDN to the spotlight. The Northbound Interface (NBI) is the Application Programming Interface (API) for the communication between the control plane and the application layer. This interface provides an abstraction to network application developers, making it possible to implement the desired functionalities without concerns regarding the low-level details of the underlying infrastructure. This has resulted in a more efficient, higher level, network application development process. But, unlike the SBI, there is not yet an accepted open standard for the NBI, which makes SDN applications lose interoperability, leading to a fragmented framework. The implementations of the NBI for specific domains will help define a broader standard, considering a wider range of domains, that will facilitate SDN widely adoption. In this paper, we discuss the standardisation of the NBI and extend the Floodlight controller to provide a relevant NBI for load balancing applications. We developed programs to access this interface and evaluated the system in a typical data center network topology in a real environment.

I. INTRODUCTION

In traditional networks it is not possible to develop network functionalities, like load balancers, without knowing the low-level details of the forwarding devices, such as switches, routers and Access Points (APs). This is due to the conventional architecture of these elements, usually featuring a tightly coupled data plane and control plane. This leads to long development cycles of network applications and extra complexity in their management.

SDN is a relatively new paradigm that decouples the data plane from the control plane, which enables the latter to be centrally manageable. This is done by the so-called SDN controller. The controller is programmable and provides abstraction for the applications running on the underlying infrastructure. It provides two Application Programming Interfaces (APIs). The Southbound Interface (SBI) links the controller to the data plane. The Northbound Interface (NBI) enables external applications to manage the controller. OpenFlow [1], created at Stanford University, was one of the first protocols for the SBI, and today it is considered the industry *de facto* standard. The NBI provides a high-level API to the SDN applications. This makes it possible to implement complex functions in the network, without needing the low-level specification of the network infrastructure.

Separately, there is no standard accepted by the industry for the definition of the NBI. The earlier concern was to have a well defined SBI. Now, SDN has a defined SBI, but is lacking a standard for the NBI. This led to each SDN controller implementing their own NBI, losing interoperability and, therefore, wasting time and resources in porting applications between different SDN controllers. Defining standards for SDN is the focus of the Open Networking Foundation (ONF). An organization with the goal to popularize SDN through the development of open standards. It created a group focusing on defining APIs for the NBI, the Northbound Interface Working Group (NBI-WG) [2]. It is working to develop a domain-agnostic interface, as well as domain specific APIs. This group is functioning to establish standardisation across the wide range of application use cases that the current networks demand.

The goal of this work is to discuss and make contributions to the definition of the NBI. As the NBI has a broad scope of use cases, we will focus mainly on load balancing applications. The implementation will follow the ONF guidelines in order to aid the SDN key players to agree upon a NBI standard for a wider range of use cases.

we contributed with a definition of the network management operations, that can be used by external programs through the NBI, for load balancing applications. We developed a Representational State Transfer (REST) interface and provided useful documentation regarding the specification of this API. Moreover, we contributed with new load balancing features to the Floodlight SDN controller, in order to have a Proof-of-Concept (PoC) implementation and to evaluate the interface in a real environment. Furthermore, we developed two client applications to facilitate the access to the management operations of the implemented NBI.

This document is organized as follows: Chapter II gives an insight to SDN state of the art. Chapter III discusses the overall system architecture. Then, in chapter IV, we address the multiple components implemented. In chapter V, we evaluate the most important aspects of our solution. Finally, in chapter VI, we conclude this work, discuss the system limitations and future work.

II. STATE OF THE ART

A. Northbound Interface

The NBI is the API that enables an application to program the network, via the network services provided by the controller. This controller will translate the application's instructions for the network elements to a language that the switches will understand (which is done through the OpenFlow protocol). The importance of the NBI is that it simplifies the operation of the control plane, by providing a high-level API between the application layer and the controller. This makes it easier for the development of network functionalities.

To continue improving the adoption of SDN, the ONF created a Working Group to develop prototypes for the NBI, but not necessarily standardize it. The Northbound Interfaces Working Group has the goal of building an interface capable of dealing with different levels of abstraction and across a wide range of domains. But before finding the broad solution, it is looking to build use case specific interfaces. Although this group was first created in 2013, to this day, it has not yet publicly released a NBI standardisation draft.

Separately, there are other implementations of the NBI that have been developed by the SDN community. These works will help the industry find the appropriate requirements to be defined, in order to obtain the right interface to be proposed as a standard. The SDN controllers Ryu and Floodlight implement a REST-like API for the NBI. REST architectural design is very popular, because it has the necessary functionalities asked by a NBI. But both Ryu's and Floodlight's NBIs do not fully respect the fundamental designs defined for REST in [3]. This problem is addressed in [4], where a NBI is proposed following the principles for REST APIs. Otherwise, the NBI may not be able to be extensible, scalable and interoperable. Furthermore, two violations of the REST principles are identified in the Floodlight controller: The first violation is exposing the media type in an Uniform Resource Identifier (URI), such as *wm/firewall/rules/json*, which limit the client and the server ability to evolve independently. Alternatively, the servers should instruct clients on how to construct URIs, by defining those instructions within media types and link relations. The second violation is exposing a fixed set of URIs, losing the controller's ability to change URIs and reallocate the resources. The REST API must provide an entry URI, from there, all transition possibilities shall be given to the clients by the server. This concept is part of the hypertext driven approach. Then, the paper recommends some modifications on the API and presents a framework for designing RESTful NBIs following a hypertext driven approach. The NBI is tested using a generalized SDN controller. Results showed that there is a trade-off regarding the performance of the NBI and its scalability and extensibility. Following the same approach RAPTOR [5], addresses the violations of REST in Floodlight and Ryu controllers and creates an interface for translating application requirements into these controllers NBIs. This approach can be useful for networks with multiple SDN controllers, easing the management of the network. But it is fairly limited, as the interaction with the user requires an external application and the interface is dependent of the controllers. This means that as their implementation may change, RAPTOR's has to change accordingly.

Another way to implement the NBI is used in the ONOS [6] SDN controller. It provides the applications a different way to specify their desires through intents. An intent is a policy-based request or connectivity requirement, simplified so that the details of how the service will be performed is abstracted for the application. This is possible due to the mapping used to translate between the simple consumer terms, presented in the intent, and the specific detailed terms, needed for the server to interpret and perform the operation requested in the intent. The ONF has defined the guidelines for implementing a NBI following this approach [7]. Considering these guidelines and ONOS NBI intent framework, [8] proposes a framework designed to leverage the intent-based architecture to provide support to a wide range of applications. The goal is to allow a developer to use the framework to create new services and/or use already existing services that satisfy the requirements for its application.

Independently of the way the NBI is implemented, there are issues related to the security of this interface. In [9], requirements such as confidentiality, integrity, authenticity and accountability are raised in the context of the NBI. A REST-like API is proposed with added security features to ensure that the security requirements are respected in the interface. It is proposed that the SDN controller is associated to a Certificate Authority (CA). Then, the applications would have to trust this CA, as well as have an application certificate signed by a CA. This CA would represent the vendor of the application, enabling the controller to know which applications are to be trusted or not. From there it is possible to give certain permissions to the application, considering the level of trust in the application vendor.

An important requirement of the NBI is that it should be user-friendly, the APIs should be provided to the users with useful documentation and a web Graphical User Interface (GUI).

III. SYSTEM ARCHITECTURE

Before we dive into the specifics of our system architecture, we briefly address the fundamentals of a load balancer [10]. The basics of a load balancer concepts are the virtual server, the server or host, the member and the pool. The virtual server is a proxy for the actual physical or virtual machine. The virtual server has a virtual Internet Protocol (IP), it is to this address that the clients will send their requests. The server or host is the actual physical or virtual machine, where the requests are distributed from the virtual server. The member is an entity that represents a server or host. The member can also represent more than a host, i.e., it can also contain the port associated to a service in that server. This allows for a greater granularity when load balancing the traffic, as it can take into account the

services included in a host. A pool is a cluster of members that can share a property. The pool is usually associated with a virtual server, so that requests sent to this virtual server will be distributed to the members of a specific pool. Our project focuses on data center networks, as it is an environment where Software-Defined Networking (SDN) is gaining popularity, as seen in the previous chapter. In data centers, the most common topology is the tree topology [11]. We will be using it to deploy our solution and evaluate it. The system's high-level architecture is represented in figure 1. Traffic will be generated from clients to the network, represented in green. The network in a tree topology, will receive the incoming traffic at the root Open vSwitch (OVS) A. Then, the switches B and C will forward the traffic to the servers according to load balance rules defined by the SDN controller. The controller gets the instructions necessary to create the load balance activity through the NBI.

The load balancer NBI is an important component of our project, we defined it as an interface based on the principles previously mentioned. It is easily extensible, so in case there is need for more advanced load balancers, the changes can be incorporated into the NBI. This interface is developed according to the fundamental principles for REST web applications, as it provides independency from platform and programming languages to the service. However, Floodlight's NBI has some violations of these principles, as mentioned before. Knowing this, we have created an interface that fixed the first violation of a REST principles in Floodlight. The load balancer REST API is no longer exposing the media type in an URI. However, the second violation corresponds to exposing a fixed set of URIs. This has not been addressed, as we chose to have better performance than scalability. Furthermore, the load balancer API with a fixed set of URIs stays consistent with the other modules' APIs, that also have a invariable set of URIs defining their interface.

The OpenFlow controller we use to build our solution is Floodlight. We have modified the simple load balancing module existing in this controller. This module was incomplete and with some limitations. It addressed Internet Control Message Protocol (ICMP), Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) traffic and had a simple round-robin algorithm to distribute client requests, but no concern for traffic volume. Also, there was no health monitoring system implemented to check on availability of the members. Considering these issues, we have improved and extended the Floodlight load balancer module to better correspond to the load balancing principles. We added features to the Floodlight project, such as the health monitoring system, the statistics and the Weighted Round-Robin (WRR) load balancing algorithms, the capability to handle Transport Layer Security (TLS) traffic, created new statistics collection algorithms and improved and expanded the load balancer NBI. The resources exposed by the load balancer API had only Create, Read, Update and Delete (CRUD) operations on the virtual IPs, pools and members. As seen in figure 1, the forwarding elements used in the network are connected to Floodlight. The controller serves as the control plane for

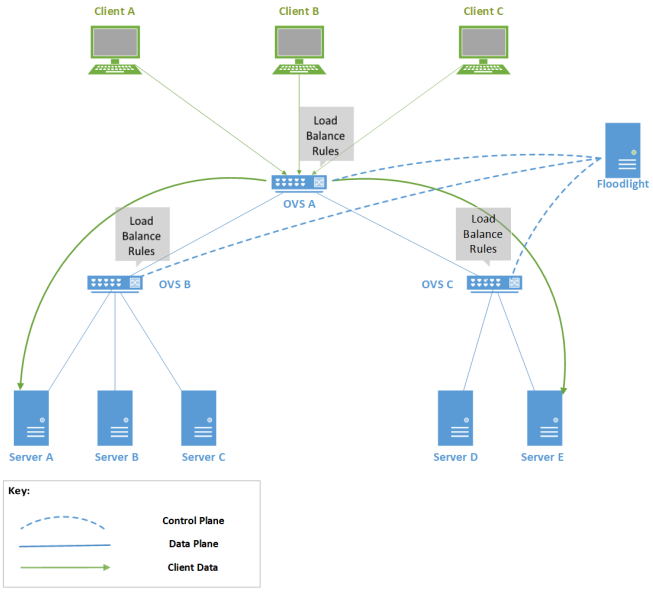


Fig. 1. High-level architecture of the system.

these elements. The switch used for the data plane of our solution is the OVS. This switch is compatible with both Floodlight, the OpenFlow controller, and Mininet, which is used to emulate the system's network.

A. Floodlight Load Balancer

Floodlight is built by multiple application modules. The core services that make use of the OpenFlow protocol to build an OpenFlow controller can interact with these application modules through a Java API. The application modules could be placed outside of the controller and communicate through the NBI. However, due to the high communication with the controller needed by these applications, they are compiled with it. The load balancer, statistics and static flow entry pusher modules are part of those type of applications. The load balancer is the principal component of our project. It is responsible for distributing network traffic through the servers available, thus increasing availability and responsiveness. This module was designed to be compatible with the OpenStack Neutron load balancer as a service[1]. The load balancing principles have a set of concepts that are represented in the data structure of the load balancer module, presented in figure 2. It is an Unified Modeling Language (UML) diagram of this module, corresponding to the most important five classes and their relationships. Below, it is described the components of this UML:

- **LBVip**: This is the class that represents the virtual server and IP of the load balancer. An instance of this class is identified by a unique attribute *id*. The address attribute is the IP that the clients are able to see and can send their request towards it. This class has a list of the pools that are served through the *LBVip* address. The *protocol* attribute is used to set the type of traffic

[1]https://wiki.openstack.org/wiki/Neutron/LBaaS

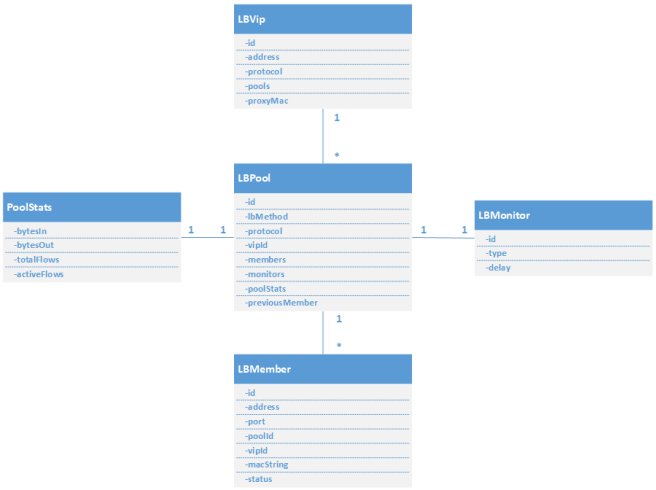


Fig. 2. Simplified data structures of the load balancer module.

which the *LBVip* will be responsible for distributing. The *proxyMac* attribute is the Media Access Control (MAC) address of the load balancer. This attribute is constant for every instance and it is used to respond to Address Resolution Protocol (ARP) requests.

- **LBMember**: This class is the representation of a member, according to the load balancing principles. The attribute *id* is a unique identifier for an instance. A member is composed of an address and a port, which are attributes of this class. Furthermore, a member is associated with a pool and a *LBVip*. The *LBMember* has the MAC address of the physical (or virtual) server that it is representing, saved in the *macString* attribute. The *status* attribute is used to determine the availability of a member.
- **LBMonitor**: One important aspect of load balancing is the health monitor system. This class represents one monitor that is responsible for evaluating the availability of a member. Each entity of this class is identified by a unique attribute *id*. The *type* attribute is the description of the method the monitor will use to query the availability of the member. The *delay* attribute is the period of time between queries to the member. The *LBMonitor* class is responsible for the status change in a member. If a member is deemed inactive because it could not answer to the monitor, then its status will be changed accordingly. One *LBMonitor* can only be associated with one *LBPool*.
- **LBPool**: This class represents a cluster or pool of members. It is identified by the attribute *id*. This attribute must be unique among other instances of this class. The *lbMethod* is the algorithm used by the pool to pick the members associated with it. The available algorithms will be presented and discussed in the next chapter. Every *LBPool* must have a *vipId*, this corresponds to the attribute *id* of a *LBVip*. The members and monitors that are connected to a pool will be saved in two different lists. These lists contain the *ids* of the corresponding *LBMember* and *LBMonitor*. The *LBPool* also has a link to *PoolStats*, which is described below.
- **PoolStats**: To better understand the state of the network, it is provided information about the created pools. The information presented corresponds to the bytes that a certain pool has received, the bytes that it has transmitted, the total flows that have been created by it and the currently active flows. This intelligence can be used to plan modifications to the network or to diagnose possible problems in it.

There are still other important data structures in the load balancer module. All the *LBVip*, *LBPool*, *LBMember* and *LBMonitor* in the network are saved in hash maps. The hash map, as the name sugests, is a data structure that maps keys to values and uses a hash function to provide access to its elements. It has the attribute *id* of the classes as the key, and the instance of the class as the value. This allows to retrieve and access all the components in the network by its unique identifier.

B. Floodlight Northbound Interface

Floodlight's NBI is well documented and it is the recommend way to utilize the features of this controller. The internal modules have to create the resources that they want to see exposed in this interface. For instance, the load balancer module will expose the *LBVip* resources in order to allow external applications to execute CRUD operations in that data structure. Similarly, the statistics module can be enabled or disabled and the statistics collected in the forwarding elements are accessed through the NBI. This way it is possible to have control over the Floodlight internal modules from an external point of view.

Due to the nature of the information exposed in the NBI, it would not be considered secure to have potentially malicious users access the contents of the resources. The Floodlight NBI can be accessed safely using the TLS protocol to avoid this problem. Although, the access to this interface is not interdicted through Hypertext Transfer Protocol (HTTP) without any security protocol. The data itself can protected using TLS, but Floodlight also restrains the access to the NBI if deemed necessary. It is possible to create an access control list or authentication process via the creation of a Java KeyStore (JKS). This is a type of authentication that does not require the user to have a password to access the interface. The way JKS works is through white listing the trusted users that are permitted to access the NBI. The JKS has a management tool - Keytool - that can be utilized to add user's public keys in the JKS, thus giving them the permission to access the NBI on a secure port.

Floodlight provides a web GUI exposed through the NBI, so that it is easier for users to manage the controller. This web interface is important as it can be used a debugging tool and offers all the necessary functions to manage a complex network. We also developed a python Command-Line Interface (CLI) application with the load balancing management functions. Having a GUI as well as a CLI to

manage the load balancer is itself a test to the NBI and overall favorable for Floodlight.

To access the NBI and the resources there defined, a user simply has to address the controller IP and port together with the URI of the desired operation. The API supports requests and responses in the Javascript Object Notation (JSON) data format. Furthermore, this interface returns standard HTTP error messages in case there are failures when processing a request. Moreover, some operations require the input of arguments in order to execute its functions. These parameters have some constraints, if the inputs are not respecting the constraints, the API will return an error. The details concerning the load balancer REST API are addressed in [12].

IV. IMPLEMENTATION

A. Load Balancer

Floodlight modules are created as Java packages. Therefore, the load balancer is a Java package defined, along with others, in the controller. The modules usually define an interface that is common to Floodlight, so that modules can provide their service to others. The load balancer uses this to provide the necessary functionalities. For instance, it uses the statistics module to get information from the switches that is used to power the decision algorithms. Floodlight modules that want to process a packet-in message have to define the corresponding listener. Then, these messages are processed in order, module by module. The load balancer is set to process the packet-in before other modules, such as the forwarding module. This way, if a packet-in does not have a virtual IP as the destination address, it is discarded to continue in the Floodlight process pipeline. Considering a state where the flow tables of the switches are empty, if a request is sent to the virtual address, a packet-in must be sent to Floodlight as the forwarding elements do not know how to process this request. The first thing the load balancer does is get some details of the incoming packet. It gets the IP address and port to which the packet is directed. Then, it deals with the ARP requests. The load balancer responds to ARP requests by pushing a packet-out to the forwarding elements that sent the request with an ARP reply. The reply comes with the MAC address of the load balancer that is addressed in the packet-in. It starts by filtering any TLS protocol messages. It compares the target port to a list of known TLS protocol ports. If the packet is from Hypertext Transfer Protocol Secure (HTTPS), Internet Message Access Protocol (IMAP), Post Office Protocol (POP) or Simple Mail Transfer Protocol (SMTP) TLS protocol ports, it is redirected to a *LBVip* with the attribute *protocol* equivalent to TLS. If the packet is not destined to any of those ports, there is no necessity to redirect it to a *LBVip* associated with TLS protocol. It is forwarded to a pool of the *LBVip* with attribute *address* equal to the destination IP address of the packet-in message. After deciding the *LBVip* to forward the request, the *LBVip* picks a *LBPool* associated with it. Then, the *LBPool* chooses a member which is the final server that responds to the request. From here it is necessary to create bidirectional routes between the client and the member chosen. The load balancer uses another module services to get the available devices in the network. It is able to map the IP address of the client and the member to the corresponding MAC addresses. This way, it is possible to set the *macString* attribute of *LBMember* with the corresponding server MAC address. Knowing the devices of the client and the member, we create bidirectional routes between them. Using the data retrieved from the packet-in, we create matches and actions for the flow tables of the forwarding elements in the route. We set the matches with the IP address of the client, the port of the switch used to forward the message and the IP protocol. For the actions of the matched packets that are sent from the client to the *LBVip*, the switches change the destination MAC and IP addresses to those of the *LBMember*. For the actions of the matched packets that are sent from the *LBMember* to the client, the switches change the source of the packet to the MAC address of the *LBVip* and the source IP to the original destination address of the packet-in. The above actions are necessary to have a transparent packet redirection for the clients. Furthermore, the load balancer uses the static flow pusher module in order to have these routes established permanently. This guarantees connection persistence, as the same client will have the packets match the same fields and the forwarding elements will route them to the same member without the need of a packet-in message.

The load balancer is using the statistics module to get information from the forwarding elements. This information is used in the load balancing algorithms used to make the decision to direct a request to a particular server, it is used to get information about the ports of the switches connected to the members and it is used to collect switches flow information, in order to have pool statistics available through the NBI. The messages implemented in the statistics module are the following:

- **Port Stats Message**: The statistics returned by this message are used to calculate the bandwidth of a switch port. The port to get the statistics from can be specified to collect it individually. This message returns the received and transmitted bits and received and transmitted packets, among other information, of an interface. Due to the information referring to the time since the port was alive, we have to retrieve two of these messages before being able to calculate the bandwidth of a port. It would be ideal to have the switch report the bandwidth of a port directly through the message, but this is not the case. So, the calculation of the bandwidth will take into account the current and the most recent statistics reports to calculate it over the time of the collection of these messages.
- **Port Description Message**: This message is used to collect information regarding the description of a switch port. It returns the hardware address, the features supported by the port, the current features, the configuration and the port status (enabled or disabled). Then, this information is used by the health monitoring algorithm to decide what is the availability of a member.

- **Flow Stats Message**: The information delivered to the controller by this message enables it to understand the state of the configurations of the flow tables in the forwarding elements. The flows to get information from can be filtered by the match field, as well as the output port of the actions field and the table identifier. It is also possible to get all the individual flows without filtering. This message returns useful parameters as the number of bytes and packets matched by a flow entry and the total number of flow entries in a flow table.

Another important features implemented are the load balancing algorithms used to decide which member to pick when distributing the client requests. These algorithms rely on the OpenFlow protocol to get context from the forwarding elements to make a decision. Floodlight had one algorithm available to perform the distribution of client requests. It was a simple round-robin algorithm, in which the pool will simply iterate through the members list and pick the next element until the end, then, it goes back to the start of the list of members. To further improve the load balancer, we have developed two more load balancing algorithms. The first one is the WRR algorithm. In this algorithm a weight is given to the members and the algorithm picks a member according to the weight associated with it. The greater the weight of a member, the greater the chances for it to be chosen. The second algorithm implemented is based on the bandwidth of the port connected to a member. Using the statistics collection described previously, we are able to retrieve the bandwidth of the ports connected to the members. According to the bandwidth displayed, the algorithm chooses the member which has the least consumed bandwidth. Due to the way statistics collection works, there is a period where the bandwidth information remains constant. To avoid picking the same member over this period of time, a list of previously chosen members is kept and the algorithm ignores the members in it.

The last implemented feature discussed is the health monitoring system. It is needed to ensure there is no un-available member in the network able to be selected by the load balancing algorithms. The health monitors rely on the statistics module being enabled. It is through the context given by these statistics that the system is able to monitor the members. The statistics module sends the port description message to the forwarding elements, then the health monitoring system uses the collected statistics to determine if a member is available. If a port associated to a member is down, then it sets that member's status to inactive. If a port associated to a member is up, then the monitors send an ICMP request message to it, in order to further investigate its connectivity. When a member responds to this request, the member is deemed active, if the member does not respond to it, then it is deemed inactive until it responds. The status of the members are considered in the load balancing algorithms. If the health monitors are enabled and a pool is associated to a monitor, then the members of that pool need to have status set to active, to be available to the algorithms.

B. Northbound Interface

Floodlight has a module responsible for the REST server that powers the API, using *restlet* as the underlying structure to abstract the details of processing the HTTP requests for Floodlight. The REST server module is responsible for starting the server and providing an interface, so that other modules are able to map their resources using this server. Through this interface, the other Floodlight modules can create their own APIs in a simple way. An URI must be created for each resource, in order to be able to access it. Each module that wants its resources exposed, have to implement the REST server interface and associate URIs to its resources. The load balancer module has created resources defining its principal entities. The CRUD operations of the elements of the load balancer module, for instance, *LBVip* and *LBMember*, are functions which purpose is to add, remove and delete these entities from the hash maps where they are stored. The user can set an identifier, as the argument of the function and key to the hash map, to determine which objects are going to be created in the load balancer. We have provided the operations that can be used to manage the load balancer through the NBI, in the [12]. We have implemented two ways to access the NBI, using a python application through a CLI and a web GUI. The python application has a defined set of commands that are associated to the operations for the management of the load balancer. This application is enabled by *curl*, a command-line tool, which supports HTTP to communicate between network devices. It allows a user, that simply needs python and a CLI, to write commands and be able to manage the load balancer. Floodlight already has a web GUI that allows a user to manage the services provided by Floodlight. We added the load balancing services from the developed module to this interface using Hypertext Markup Language (HTML) and Javascript. This interface does not require the user to have python or a CLI to manage the load balancer. It can be accessed through a web browser and its made to be more user-friendly.

V. EVALUATION

In this chapter, we address the evaluation of the system implemented in this work. We start by stating the test environment. Then, we analyze the results regarding the most important features and performance of the system.

A. Test Environment

The goal of this evaluation is to assess the performance of the system in the real conditions that the solution was created to face. In our case, we want to recreate a SDN data center that is controlled by Floodlight. To test the solution in a more realistic environment, Floodlight will be placed in a remote machine, while the network is emulated in a Virtual Machine (VM), ran in another machine. These conditions are set to obtain test results that resemble a real networking scenario.

Our test bed consists of two machines, one running the Floodlight OpenFlow controller and the other running

Mininet with the emulated network. The specifications of the machines are as follows:

Controller machine:

- **OS**: Ubuntu 14.04.3 LTS (GNULinux 3.13.0-32-generic x86_64).
- **RAM**: 6 GB DIMM DRAM.
- **CPU**: Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz.
- **OpenFlow Controller**: Floodlight.
- **Network Bandwidth**: 20 MB/s.

Network VM:

- **OS**: Ubuntu 14.04.5 LTS (GNULinux 3.13.0-32-generic x86_64).
- **RAM**: 6 GB SODIMM DDR3.
- **CPU**: Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz.
- **Network Emulator**: Mininet version 2.2.1.
- **OpenFlow Switches**: OpenVSwitch version 2.3.90.
- **Network Bandwidth**: 400 MB/s.

B. Load Balancer Algorithms

We start by analyzing the test results of the load balancer module in Floodlight and its main components. The WRR algorithm is tested by tracking the amount of times a member of a pool is chosen by this algorithm, considering the total requests issued to the pool. In theory, the probability of a given member being picked is given by equation 1. The results of the tests performed to determine the practical probability of this algorithm showed that it is very close the to expected probability given by the theoretical equation.

$$P(Member\,picked) = \frac{weight\,of\,member}{sum\,of\,weights\,of\,pool\,members} \quad (1)$$

We tested the statistics algorithm to determine if the load balancer is working correctly, by attempting to stress a network with multiple clients establishing TCP connections with a pool running this algorithm. This algorithm works with statistics collection module, in order to obtain the bandwidth usage of the members of a pool and picks the members with the lowest score. We register the maximum and minimum bandwidth achieved by the members, while the request are being processed. We set twenty TCP connections per client, each sending 512 kilobits to the pool members, over a five second period, using *iperf*. In figures 3 and 4, we see the minimum bandwidth across the four figures is mostly zero, which means that at least a member of the pool is not responding to any request. This is not optimal, as the load should be spread across all the pool members, but it can be explained by the insufficient number of clients to reach the point where a member can no longer respond to a request before receiving another. This happens in this experiment with eight and sixteen clients, however only for a short period of time and with residual bandwidth values. Comparing the statistics collection period of ten and five seconds, we spot the consequences of this change in comparing the left side graphs to the right side graphs of both figures. The values of the bandwidth are constant for longer periods of time in the case of the statistics collection every ten seconds, because it takes more time to update the values and the algorithm is relying on older information. Although, this is not a limitation of this algorithm, as the bandwidth allocation is still balanced, as shown in table I. This is due to the previously chosen member list used by the algorithm, described in the preceding chapter. It is able to maintain the average bandwidth of the members, while processing the requests, fairly even. Considering the average bandwidth of the members in the same statistics collection period, we see that the values are close. Even comparing across the collection period, the values remain alike. This is proof that the algorithm is distributing the load across the servers uniformly.

Statistics Collection Period	Member Id	Average Bandwidth (bits/s)
10 seconds	1	10057
	2	91971
	3	80031
	4	95529
5 seconds	1	12 391
	2	82541
	3	83428
	4	92447

TABLE I

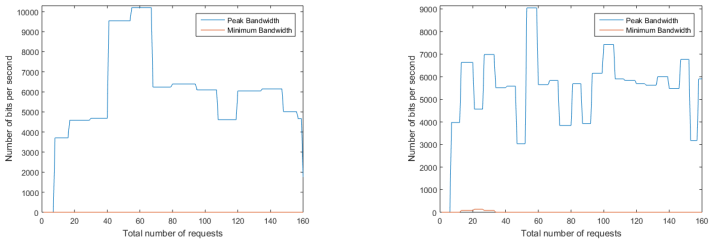
AVERAGE BANDWIDTH OF A POOL RUNNING THE STATISTICS ALGORITHM, WITH 4 MEMBERS HANDLING 320 TOTAL REQUESTS FROM 16 CONCURRENT CLIENTS.

C. Load Balancer Performance

In this test we will focus on the Floodlight elements bandwidth consumption. This will allow us to compare the load balancer with other Floodlight modules in terms of control plane communications. Firstly, we need to know the response time of Floodlight and throughput of these kind of communications. Secondly, we calculate the load on this channel and assess the bandwidth used. Note that these are measurements of the control plane efficiency of Floodlight, which is the base for the load balancer functionality.

The Floodlight control plane performance can be measured using *cbench*, a benchmarking tool for SDN controllers. It is able to emulate switch sessions with the controller and measure the throughput and latency of flow modification messages. The scenario used to perform the tests were three different tree topologies. The results of the Floodlight performance benchmark tests are shown in figure 5.

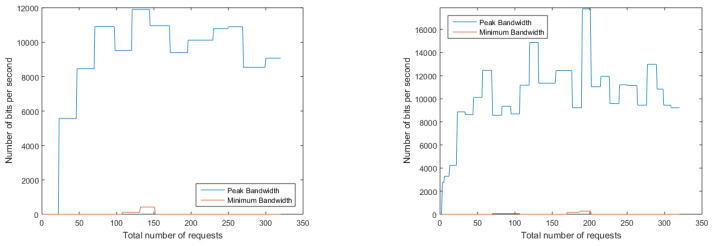
We further test the controller, by measuring the packets-out sent by the load balancer and statistics elements and compare them with other Floodlight modules that have a similar way of communicating with the forwarding elements. In figures 6 and 7, we can see the comparison between the packet counters used in the communication with the switches, by different Floodlight constituents in different tree topologies. We can observe a remarkable difference in total control plane messages from the forwarding module counters to the load balancer module counters. However, the number of packets-out sent from these modules is very close, as the number of flows installed is going to be equal, with regards



(a) Statistics period set to 10 seconds.

(b) Statistics period set to 5 seconds.

Fig. 3. Load balancer statistics algorithm tests with 8 members and 8 clients.



(a) Statistics period set to 10 seconds.

(b) Statistics period set to 5 seconds.

Fig. 4. Load balancer statistics algorithm tests with 8 members and 16 clients.

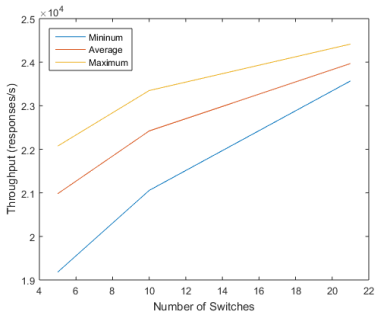


Fig. 5. Floodlight control plane performance test results with *cbench*.

to the same topology. We can also see the difference from the statistics module to the link discovery module is not extravagant. But it is still noticeable, with the latter having a few dozens more total control plane packets handled, in all of the tested topologies.

The results interpretation is not complete without further analysis of the size of the packets traveling through the communication channel. For this matter, we have used *wireshark* to determine the length of each message used by the communication between Floodlight components and the forwarding elements. Considering the throughput and the total packets to be processed, we calculated the time it takes Floodlight to process these packets. Knowing this, the number of packets exchanged between a module and the switches and the length of each message used in that communication, we are able to calculate the bandwidth used by each one of the Floodlight components tested. Following a scenario with a network with depth of three and fanout of four, we present these results in table II. Given these results, we reach the conclusion that the load balancer is using around 45% of the bandwidth of the forwarding module. Furthermore, as the statistics module is a core element of the load balancer solution, if we consider its bandwidth, it is still approximately 53% of the total bandwidth of the forwarding element in Floodlight.

Modules	Message Weights (kilobytes)	Duration (s)	Total Bandwidth (kilobytes/s)
Load Balancer	6234.84	2.38	2619.68
Forwarding	13950.00	2.38	5861.34
Statistics	1148.19	2.38	482.43
Link Discovery	37.25	2.38	15.65

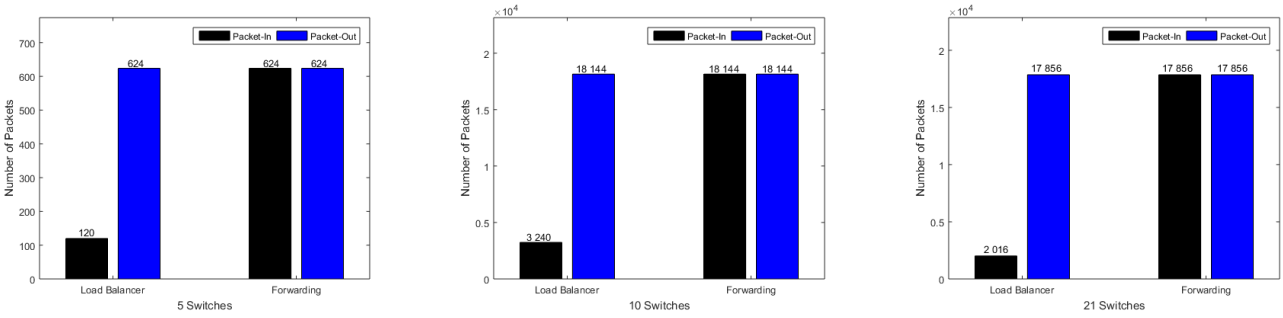
TABLE II

COMPARISON OF THE TOTAL BANDWIDTH CONSUMPTION OVER 2.38 SECONDS OF THE FLOODLIGHT MODULES.

Regarding this evaluation, we can conclude that the load balancer has a more efficient use of control plane resources when compared to the forwarding module, even when allied with the statistics module. Notwithstanding, the statistics module is contributing with more load than its reference element, due to its similar periodical control plane usage pattern, the link discovery module. This is mainly due to the greater length messages used to collect statistics.

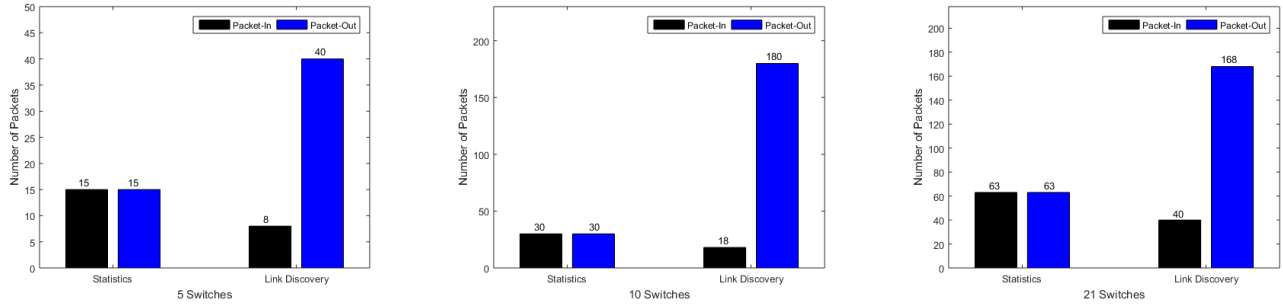
D. Northbound Interface Performance

The NBI is a crucial component of this work, as we want to make this interface ready for real environments with the possibility of intense communications between this and external applications. This test is to ensure the NBI performance is up to standards by testing if it is able to cope with a high amount of requests. To make this test possible,



(a) depth=2, fanout=4. **(b)** depth=2, fanout=9. **(c)** depth=3, fanout=4.

Fig. 6. Floodlight load balancer and forwarding modules packet-in and packet-out counters after achieving full network connectivity, for three different tree topologies.



(a) depth=2, fanout=4. **(b)** depth=2, fanout=9. **(c)** depth=3, fanout=4.

Fig. 7. Floodlight statistics and link discovery packet-in and packet-out counters over a period of 15 seconds, for three different tree topologies.

we used *bench-rest*. This tool is used to bench-mark the performance of REST interfaces, by sending high amounts of concurrent requests and outputting vital metrics about the state of the interface, when submitted to this pressure.

The scenario played in this test consists of Floodlight running in the cloud machine and the VM running a network connected to Floodlight, while sending 100 000 HTTP Get requests to the NBI, asking for the information about a members of a certain pool. Using *bench-rest*, we are able to set multiple threads to execute this requests and measure the interface performance in requests per second. The results of this experiment are show in table III, where we see that the requests per second are relatively constant. This leads us to believe that the NBI average performance of this interface is around 1236.97 requests per second.

Number of Requests	Number of Pool Members	Requests/second
100 000	100	1070.84
100 000	1000	1434.45
100 000	10000	1205.61

TABLE III

NORTHBOUND INTERFACE PERFORMANCE TEST WITH *bench-rest*, EXECUTING 100 000 HTTP GET REQUESTS ON A MEMBER.

In order to further analyze the performance of this interface, we test it with two consecutive HTTP commands. In this test, we will dispatch a HTTP Put message to create a member and then execute HTTP Get message over that same element. The results of these operations are shown in table IV. It is noticeable that with 200 and 2000 request, the interface is not being stressed. This is apparent when the number of requests increases and the requests per second also increases. Although, if we look at the bottom three lines of the table, we see that there was a decrease of performance, which makes us conclude that the maximum performance has been reached, and the system is pressured starting from 200000 concurrent requests. So, we can deduce that this interface is able to deal with 506.33 requests per second, in a stressed environment. With *bench-rest*, we are able to determine the errors given by the interface, when attempting to answer the requests and seen there were no errors running this test.

VI. CONCLUSIONS

Considering the SDN state of the art, we developed a REST API for load balancing applications, conductive to contribute to the definition of a standard. This API provides all the necessary functions to external applications in order to manage a load balancer, according to the load balancing principles. We contributed directly to the source code of Floodlight with new features, such as two new load balancing algorithms, two statistics collection methods and the pool statistics retrieval. Other features are pending approval by

Number of Requests	Requests/second	Errors
200	343.18	0
2000	309.84	0
20000	592.88	0
200000	525.47	0
2000000	506.33	0

TABLE IV
NORTHBOUND INTERFACE PERFORMANCE TEST WITH *bench-rest*, EXECUTING HTTP PUT FOLLOWED BY HTTP GET REQUESTS ON A MEMBER.

the Floodlight team, including the TLS handler and the health monitoring system. Thus, we improved this controller, making it relevant for load balancing applications, in the context of data center networks. Furthermore, we implemented two different interfaces, in which users have access to the NBI operations, simplifying the usage of Floodlight load balancing management functions.

In order to have a system ready to be deployed in a real data center, we performed an in-depth evaluation on the most important components of our solution. We used a test environment with conditions that are close to those that are expected in a data center. We reach the conclusion that the new Floodlight load balancing features are working as intended. The TLS handlers redirect the requests appropriately. The health monitoring system is identifying disconnected members. The balancing algorithms are distributing the load evenly across members. Furthermore, the performance of the controller is not degraded by these functions, it is actually using 53% of the bandwidth used by the forwarding component, due to the way the load balancer is installing the flows in the forwarding elements. Regarding the REST API, we concluded, after performing stress tests with 100000 concurrent requests, that it is suited to answer on average 1236.97 requests per second and 506.33 requests per second, on an extreme environment.

It is our hope that this thesis can help determine the functions, essential for the management of load balancers, to be defined in a standard. With this work, it is easier to identify and incorporate these functions into a future standard.

VII. FUTURE WORK

Although we see our solution as fitting for starting discussion about a standard, it comes with some limitations. In this section, we will propose a few ways to continue this work and fix these limitations.

The load balancing algorithms are working as intended, however they could be improved if there was server context information to better the distribution decision. This could be achieved by incorporating the Simple Network Management Protocol (SNMP) or similar protocols in the Floodlight controller.

Another aspect of our solution is that the flows installed by the load balancer in the forwarding elements have infinite time out. This is to ensure TCP session persistence between server and client. However, this will lead to the exhaustion of switch resources in the long term. There should be done more research concerning the optimal time frame to have when dealing with a TCP session establishing and termination, then implement it to the flows installation.

The CLI and web GUI interfaces developed to access the resources of Floodlight through the NBI were not user tested. So, a possible direction that should be exploited is user testing, thus increasing the quality and friendliness towards users of these interfaces. To further complete the range of applications available to access the NBI, it could be interesting to have an interface developed for mobile environments.

REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, April 2008.

[2] D. L. Sarwar Raza, "Northbound interfaces working group (nbi-wg) charter," Tech. Rep., June 2013, working for a standardized NorthBound Interface definition, Accessed: 2016-11-20. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/working-groups/charter-nbi.pdf

[3] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[4] W. Zhou, L. Li, M. Luo, and W. Chou, "Rest api design patterns for sdn northbound api," in *WAINA '14 Proceedings of the 2014 28th International Conference on Advanced Information Networking and Applications Workshops*. IEEE Computer Society Washington, DC, USA ©2014, May 2014.

[5] S. Rivera, Z. Fei, and J. Griffioen, "Raptor: A rest api translator for openflow controllers," in *Computer Communications Workshops (INFOCOM WKSHPS), 2016 IEEE Conference*. IEEE, April 2016.

[6] Open Networking LAB, "Introducing onos - a sdn network operating system for service providers," Tech. Rep., April 2015, Accessed: 2016-11-28. [Online]. Available: http://onosproject.org/wp-content/uploads/2014/11/Whitepaper-ONOS-final.pdf

[7] C. Janz, "Intent nbi – definition and principles," Tech. Rep., October 2016, Accessed: 2016-11-28. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR-523_Intent_Definition_Principles.pdf

[8] M. Pham and D. B. Hoang, "Sdn applications - the intent-based northbound interface realisation for extended applications," in *NetSoft Conference and Workshops (NetSoft), 2016*. IEEE, June 2016.

[9] C. Banse and S. Rangarajan, "A secure northbound interface for sdn applications," in *TRUSTCOM '15 Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA - Volume 01*. IEEE Computer Society Washington, DC, USA ©2015, August 2015, pp. 834–839.

[10] F5 Networks, "Load balancing 101: Nuts and bolts," Tech. Rep., May 2017, Accessed: 2017-7-15. [Online]. Available: https://f5.com/resources/white-papers/load-balancing-101-nuts-and-bolts

[11] K. Bilal, S. U. Khan, J. Kolodziej, L. Zhang, K. Hayat, S. A. Madani, N. Min-Allah, L. Wang, and D. Chen, "A comparative study of data center network architectures," in *ECMS 2012 Proceedings edited by: K. G. Troitzsch, M. Moehring, U. Lotzmann*, 2012, pp. 526–532.

[12] D. C. Coutinho, "Contributions for the standardisation of a sdn northbound interface for load balancing applications," Master's thesis, Instituto Superior Tecnico, Av. Prof. Dr. Cavaco Silva, 2744-016 Porto Salvo, October 2017.