Advanced SDN Applications using the P4 language

Bernardo Valente
bernardofvalente@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2021

Abstract

Software Defined Networking (SDN) aims to provide dynamic and programmable solutions for network management. In its original form, the SDN paradigm decouples the data and control plane of conventional network devices, centralizing all control functions of the SDN domain on a single server with a unified view of the network. Network devices were therefore simplified to a reduced set of data plane operations, programmed using a standard protocol. OpenFlow was possibly the most successful SDN protocol.

However, it soon became clear that most of the flexibility offered by the SDN model was limited by the capabilities and fields defined in the OpenFlow protocol itself. As a way of overcoming this limitation, an alternative model was proposed based on the concept of programmable data plane. P4 (Programming Protocol-independent Packet Processors) is a language developed for specification of how data packets should be processed and forwarded on compatible network devices. Since rules and packet actions are defined at the bit and byte level, P4 offers a more detailed control over network traffic than the one possible by OpenFlow solutions.

This work presents a novel load balancing solution using the P4 language which includes a production-grade SDN controller, a programmable data plane compatible with any number of connected servers, and a stateful load balancing algorithm with fault tolerance capabilities.
Keywords: Software Defined Networking. OpenFlow, Data Plane Programmability, P4, Load Balancing

1. Introduction

The conventional implementation of SDN networks assume non-programmable switches configured with a remote controller through a well-defined protocol. This is overall a good solution for some occasions; however, it has some known shortcomings that will be discussed in section 2.1. P4 was proposed [1], in the scope of data plane programmability, as a domain-specific language, designed to program network nodes, and overcome the limitations imposed by the traditional static SDN data plane architecture.

The P4 programming language offers many benefits, including high customizability of the data plane and control over the code running in the network nodes. One of the key aspects of P4 is to be protocol-independent. Switches can be loaded with large range of programs and each might have a different implementation of multiple protocols. Having one of these nodes in a network makes it very agile for future implementation of new protocols or network functions. The network administrator just needs to extend the P4 code in the already available and ready programmable-switches. Changes can be simple like changing how a protocol analyses a packet's header information, adding keys to be matched in a flow-table or adding new counters and sending the information to the network controller for statistics purpose either to have a better understanding of the network traffic, or to populate big data for network analysis and machine learning. Switches are also available to more complex network functions, like adding a firewall function to filter unwanted traffic or installing a load balancer in the network for better control of flows.

Shifting a network from fixed-function to programmable-switches is an investment for future network implementations, the network becomes agile and available to change how the data plane works. However, P4 still presents considerable challenges before it can be widely deployed. The motivation for this work was the need to explore and contribute to new P4 applications.

The objective of this thesis is to bring Data Plane Programmability in the form of the P4 language to Software Defined Networks, in order to have a better understanding of the capabilities that P4 can introduce in a Software Defined Network.

The implementation should provide the network with benefits from Data Plane Programmability such as high flexibility to include new protocols, as well as provide a base platform to build and install

new functions, making a transition to a future-proof, agile network.

To accomplish the work proposed, we have decided to develop a load balancing system. These are relatively easy to develop, while at the same time giving some room for improvements and creativity. Also, Load Balancers are crucial components of data centers, being therefore ubiquitous and fundamental to the day to day life of a modern technological society.

The goal of this work is to contribute to improve P4 solutions for LB applications. To accomplish that we propose to consult articles mentioning Load Balancing projects and develop an application based on these. The objective is to improve the solutions found by introducing new features and overcoming some of their obstacles an limitations. Later in this thesis we will make a further comparison between our work and the related work.

2. Background

In this chapter, we present an introduction to Software Defined Networking (SDN),Data Plane Programmability (DPP), and Load Balancing Algorithms.

2.1. Software Defined Networking

Traditional switches have the data and control plane in the same physical device. This allows for a fast and reliable operation since each switch is independent from all others in a network. However, distributed architectures are much harder to optimize since they rely on local algorithms that usually do not have a global network view. Moreover, in large networks, managing and configuring every switch manually can be tiring and cumbersome.

Software Defined Networking (SDN) offers an architecture that decouples the control and data planes from switches, centralizing all control decisions in a single computing platform. The control plane operates the logic and makes decisions about network traffic management and optimization, while the data plane focuses on forwarding the packets based on the control plane's decisions. The data plane uses flow tables that contain actions to execute when a match is made. Flow tables can be dynamically populated with rules by the control plane according to operation requirements. This architecture provides a strong foundation, since decisions depart form a single controller who has a global and unified view of the overall network, and the decisions are no longer distributed all over the network.

Separating the data and control planes from the same device originates a new network architecture. It consists of three layers vertically aligned. The central layer is the Control Layer, where network traffic is managed. It receives instructions from an Application Layer through the Northbound Interface. The bottom layer is called the Infrastructure Layer, it is built by network devices and contains the data plane. The Control Layer uses the Southbound Interface to communicate with the Infrastructure Layer sending instructions for how to manage network traffic. Figure 1 shows an illustration of the three layers present in a SDN architecture.

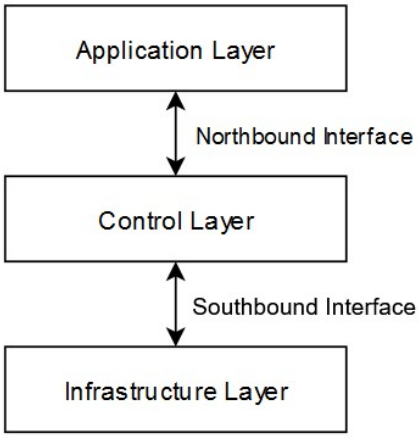

Figure 1: SDN architecture layers.

The SDN controller, also known as Network Operating System, is the key component for a SDN network to work properly. It can be found in the Control Layer and consists of an application that manages network flows [2].

The development of SDN was based on the decoupling of the control and data planes. The interface between these two planes is called the Southbound Interface (SBI). OpenFlow (OF) was the first widely adopted protocol for the SBI communication.

The OpenFlow protocol is managed by the Open Networking Foundation (ONF), a consortium leading the adoption of open network standards such as SDN and OpenFlow, as well as other technologies related to automatic network management. OpenFlow Development started in 2008 [3] at Stanford University and since its initial inception many versions have been launched. Each version introduced new network protocols which tried to complement previous versions. The first version only supported four common protocols, while the latest version of OpenFlow lets us add and delete forwarding entries for about 50 different header types [4].

OpenFlow can add, update, and delete entries in the flow tables of network switches. Each entry has a set of match keys and instructions to follow on matching packets. This can either be done proac-

tively, by populating tables before packet hits the network node, or reactively, in response to arriving packets that do not match any of a preloaded set of rules.

OpenFlow is a powerful protocol that gives the user full control of packet forwarding rules and the network data plane through the control plane.

However, OpenFlow has some known shortcomings. Supporting many protocols makes the protocol almost universal, since it can be used in nearly every situation possible. However, it increases the development complexity. In fact, each OpenFlow version requires new OpenFlow switches, since these too should implement every protocol addressed by it. This makes it harder for vendors to adapt, especially if some of the protocols are very specific and rarely used. This results in bloated network switches which customers may not want due to configuration complexity. Usually, users only need a limited amount of network protocols to run in their network.

2.2. Data Plane Programmability

Control plane programmability had a rapid evolution in the past 10 years and brought many benefits for SDN networks. However, data plane devices have remained nearly the same: a static platform that is essentially "dumb" and only forwards packets based on the input it receives from the control plane. This model has also contributed for limited data plane development.

In order to overcome these limitations, a new "top-down" model was proposed that introduces the concept of data plane programmability [4]. Instead of having a limited switch and adapting the control plane to its capabilities, as in a "bottom-up" model, this model suggests a dynamic data plane where the programmer can edit the forwarding plane with custom protocols and features. That way the control plane does not need to worry about the switches' or the SBI limitations.

Programming the data plane gives the user much more detailed control over how packets are processed. New protocols and features can be implemented and managed out of the scope of OpenFlow or other rigid SBI protocols.

The next section will focus on P4 and its features regarding data plane programmability.

P4 (Programming Protocol-independent Packet Processors) is a programming language used to specify how a switch's data plane processes packet forwarding. While being protocol independent, it is also target independent, which means that the same code can run in different hardware platforms, provided that they adhere to a common hardware specification [1].

The P4 programming language was initially proposed in [1] and later developed by the P4 language consortium, a non-profit organization formed by a group of network engineers. It has since joined the Open Networking Foundation [5].

Originally proposed in 2014, P4's objective presented three main goals:

- **Reconfigurability** Let programmers change how packets are processed.
- **Protocol Independence** Free switches from unwanted protocol integrations.
- **Target Independence** For hardware interoperability.

When OpenFlow was first engineered there was a big difference in processing speed between fixed-function ASICs (Application-Specific Integrated Circuit), and programmable switches. The latter were not viable for networks with fast paced traffic since they were much slower. Nowadays however, there are programmable chips that are as fast as fixed-function ones, called PISA (Protocol Independent Switch Architecture) chips. These are very versatile, and its flexibility allows for programmers to implement and try new network protocols as soon as they are designed [4].

Because P4 is used to program the data plane in software switches, it is usually used in SDN networks. It takes advantage of the fact that SDN decouples planes to program the data plane without worrying much about the control plane. It is an appealing addition for SDN networks that need data plane flexibility. P4 can be used to modify already existing protocols and add extra features or include meters for network statistics. P4 switches can also be programmed to implement new protocols, which is an advantage over fixed-function switches for developers that cannot wait for hardware manufacturing to test their algorithms.

Although P4 is commonly used with SDN, it can also be used in standalone mode with both the data and control plane in the same system. This provides great flexibility, P4 can be implemented independently of the network architecture to bring the benefits of Data Plane Programmability.

The language is very versatile and can adapt to most network protocols including OpenFlow [4]. This is useful for compatibility with conventional SDN architectures based on OF. However, to control custom protocols programmed in P4 a new communication protocol needs to be engineered.

2.3. Load Balancing

The great expansion of computer networks in recent years brought much more traffic than the conventional web servers could handle. Traditional services only had one server handling all incoming requests, server administrators quickly understood

that a new method was bound to be implemented to distribute the high load between more resources. Thus came Load Balancing.

Load Balancing refers to the technique to distribute load among a set of resources to get the best performance possible of the global system. In computer networks, load balancing can be viewed at two levels: to distribute network traffic through multiple paths, or to balance client requests to multiple servers. In this project we will focus on the latter.

There are many load balancing algorithms, which can be grouped in **stateless** or **stateful** algorithmsdesignated as Static or Dynamic algorithms [6]. StatelessStatic algorithms are those that do not take into consideration the state of the system, while StatefulDynamic algorithms, on the other hand, consider the load of the system and make real time decisions to distribute the load with the objective of achieving the best overall performance. StatefulDynamic algorithms are usually harder to implement but offer better load balancing results, these can take advantage of one or more parameters from the network as the decision factor for the algorithm. Some of the parameters can be associated with properties of the servers, for example: the CPU load time, the number of connections with clients, or the average request response time.

Probably some of the most common load balancing algorithms are the following [7]:

- **Round Robin** The Round Robin algorithm consists of ordering all the servers in a list, and distributing the client requests in rotation. If we have three servers, the first three requests will go, in order, to the three servers, considering that we have servers one, two and three. When the fourth request arrives, as there is no fourth server, the list will start at the beginning and the request will go to the first server.

- **Weighted Round Robin** Similar to the Round Robin algorithm, the Weighted Round Robin algorithm also orders the servers in a list and serves them in order and rotation. Unlike the previous algorithm, the requests are not distributed equally among the servers. The network manager may set different weights to each server. In an example with three servers and four requests, where the first server has a weight of 2 (two), and the others have a weight of 1 (one), the first server will handle the first two requests, while the third request will go to server two, and the fourth request to the third server. Like the previous algorithm, this process repeats when the list reaches the end. This strategy is an improvement of his predecessor since more powerful servers can be attributed with more requests.

- **Least Connections** As the name of the algorithm suggests, this algorithm is considered stateful and relies on the number of open client connections. The load balancer keeps track of how many connections each server has and, when it receives a new client request, assigns it to the server with least connections. Unfortunately this algorithm has a downside, it takes into consideration the number of active connections of each server, but some connections might be much heavier than others, making it imbalanced if a server is attributed too many heavy connections.

- **Resource Based** This algorithm listens for reports of agents installed in the servers that share the load status of the servers. This load is dependent of the load balance application and can take the form of one or several parameters, such as CPU load time, or the server memory. When the load balancer receives the report from all the servers, it processes the values and makes a load balancing decision.

2.4. Related Work

Recently in 2020, Chih-Heng Ke et al. [8] released a paper demonstrating a load balancing system with P4 switches in a Software Defined Network, using a centralized controller. The test topology consists of a client connected to a P4 switch that manages the flow to up to four servers. The motivation for this paper relies on the time consuming encapsulation and decapsulation of conventional load balancers like Linux Virtual Server (LVS) and HAProxy. It proposes the use of Data Plane Programmability with the P4 language to make load balancing faster, by moving the load balancing algorithm from the control plane to the data plane. During their work, a series of load balancing algorithms were used to test which has the best performance. The algorithms tested were: connection hash, random, round-robin, and weighted round-robin. They used the average request response time to compare the algorithms, and reached the conclusion that the best algorithms were the round-robin and weighted round-robin, depending if the CPU speed of the servers were equal or not, respectively. This work presents some interesting features like periodic health checks done by the controller, which can inform the data plane for server faults, and the ability for the load balancing switch to continue working if the control plane fails. However, after some inspection to the project's source code, we found that the algorithms performed are hard coded in P4 and very hard to alter in case the topology changes. In summary this pa-

per presents a project with four different stateless load balancing algorithms in a centralized network capable of fault tolerance.

3. Architecture and Implementation

The objective of this work is to develop a load balancing application with SDN and P4. Based on the article mentioned in section 2.4, we propose to develop a load balancing application that benefits from the best features of the mentioned article, while at the same time trying to overcome its limitations.

Considering the work by Chih-Heng Ke et al. [8], we propose some modifications. Firstly, by introducing a production-grade SDN controller with features like code modularity and configurability. Secondly, by improving upon the data plane P4 code, making it more generic and compatible to any number of servers in the system. Finally, by replacing the stateless load balancing algorithm, by a stateful one, which should offer better performance. In summary, this work shall consist of a stateful load balancing algorithm in a centralized network with fault tolerance capabilities.

The main scope is to produce a system with high quality that could be incorporated in a data center network. Since data centers are commonly found to work with centralized networks and load balancing systems, we consider that it would be a great achievement if our project could be ready for an environment like that. With that in mind, we aim to use SDN, since it is very common among data centers and has useful functionalities like a unified view of the network. Also, we want to incorporate a modern controller that is widely accepted and has modular applications, in order to ease the implementation and the change between multiple load balancing algorithms in a transparent way to the network.

In order to develop a load balancing SDN application, it is a good practice to separate the logic between the control and data planes. The control plane contains the part of the algorithm that is responsible for choosing the weight of each server, while the data plane has an ambiguous code that distributes the load between multiple servers considering the weight of each server, but is transparent to the decision algorithm itself.

With this approach, the data plane does not have to communicate with the controller for each packet it receives, since it already has a table indicating the weight of each server. In one hand, the data plane is responsible for having global control variables that manage the balancing flows independently, this way the decision algorithm is abstracted. In the other hand, the control plane is responsible to update the data plane every time there is a change in the decision algorithm. Additionally, if a modular controller is implemented, it should be easy for the user to change the load balancing decision algorithm in real time without the knowledge of the data plane.

To simulate a real world scenario, we propose that the servers run a HTTP socket listening for GET requests. This means that the switch handles TCP traffic. The data plane program needs to have a basic understanding of how TCP connections works, this means that the switch is not performing load balancing packet wise, but instead request wise. In other words, for each new packet inbound to the switch, this performs a hash calculation using the source IP address and the source TCP port, on the first packet of each connection it attributes a new server for the hash and saves the hash-server pair. That way, when a packet arrives and the hash calculation is already saved in switch memory, it loads what server is performing the request and sends the packet to that server. It is also important to note that in the end of each TCP connection, the header's FIN flag is active, when the switch detects this flag it proceeds to delete the hash-server pair from switch memory since it will no longer be used.

Since the application is balancing load between a set of servers, the internal server IP addresses need to be invisible to the clients. A strategy is implemented where the switch is attributed a public virtual IP address to be used by the clients. For each client request, the destination IP address is the switch's virtual IP address, instead of the physical IP address of the servers. The switch is responsible for modifying the packet headers of each request to change the destination IP address for the server's IP address that is assigned by the load balancing algorithm. Also, when a server responds to the client, the response packet also needs to be modified, the switch replaces the source IP address by its own virtual IP address. It is important to note that these header modifications make both the IP header and the TCP header checksums invalid. Therefore, before the switch sends the packet through the outbound interface, it needs to recalculate the header checksums. If this last step is skipped, the recipients of the packets mark them as invalid and discard the packets.

Regarding the load balancer algorithm, we decided to take the most benefit out of SDN and develop a stateful load balancer, since these often offer a better performance than stateless load balancers. The chosen algorithm is the resource based type, and the decision parameter to use is the average server request response time. Ideally the data plane should have a metric to calculate the average response time of each TCP connec-

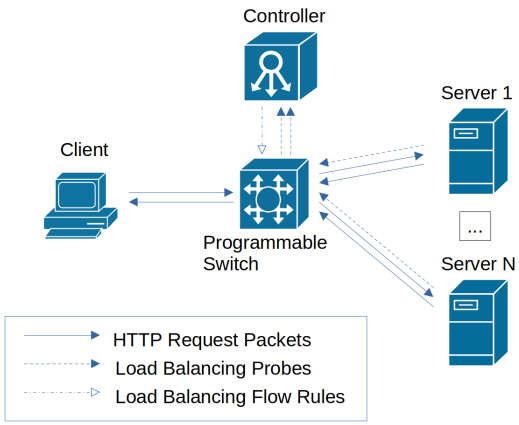

Figure 2: Proposed Topology.

tion, but since the P4 language does not support this functionality, we decided to use a probe based approach from the servers. In our approach, every server is responsible for sending a periodic report to the controller with the average response time of the their most recent requests. These probes are sent with the UDP protocol to the switch, that redirects them to the controller. When the controller has received a report from all the online servers, it processes them and makes a decision about how the weights should be distributed to each server. When the decision ends, the controller installs new load balancing flow rules in the switch, replacing the previous flow rules.

Lastly, our application is also capable of supporting fault tolerance. We use the advanced SDN capabilities of our remote controller to detect changes in the topology, these changes can be the addition or removal of servers connected to the switch. When the switch detects one of these changes, it proceeds to restart the load balancing algorithm. When the algorithm restarts, it forgets about the weights previously attributed to each servers and assumes that all the servers have the same initial weight. When the time passes, the controller receives the periodic server probes and adjusts the server weights accordingly.

A proposed topology for the proposed application can be consulted in fig. 2.

4. Results & discussion

In this chapter we demonstrate the tests done to evaluate our load balancing system. In section 4.1, we introduce the objectives that we aim to achieve in this chapter. Section 4.2 introduces the scenarios where the tests will be done. Lastly, in section 4.3, we present the results of the tests described in the previous subsection.

4.1. Test Objectives

With any algorithm developed in software, there must be a evaluation subsection to verify its operability as well as its performance. These are the objectives we aim to achieve with our tests.

- Prove that the algorithm developed is properly working by demonstrating connectivity tests.
- Test the algorithm with various types of variables to evaluate their impact on different network situations
- Place the developed system in faulty environments to assess its fault tolerant behaviour

4.2. Test Scenarios

To evaluate the good behaviour and performance of our algorithm, we used a set of topologies and variables to have the necessary tests to cover the minimum amount of possibilities tat we consider enough to evaluate our system for a near real world environment.

For every test performed the topology consists of one client, that simulates enough requests to act like a large set of clients, a load balancing switch connected to a SDN controller, and one to four servers, which will process the client requests assigned by the load balancer.

Since we are performing tests in a virtualized environment, we also decided to develop an artificial system for our servers in order to have a behaviour closer to the real world environment.

The load balancer itself has a series of parameters that will be tested in multiple topologies to test the algorithm and hopefully come to a conclusion about which are the best values for each parameter.

4.2.1. Artifical Server Load

Since we are working in a virtualized environment, it can sometimes be hard to evaluate a system with real world performance values. We decided to make a basic artificial load algorithm based on the work done in [9] and [10]. The resulting algorithm was developed in the python HTTP servers and has a response time distribution that can be seen in fig. 3.

Since our system can sometimes have more than two servers and multiple types of tests, the two distribution lines do not always correspond to servers one and two of the system. Instead, servers with an odd number have the performance of **server1** of the graphic, and servers with an even number have the performance of **server2** of the graphic.

4.2.2. Test Topologies

To test a server load balancing algorithm, the most simple case scenario is to have a system with one client, one load balancer, and at least two servers. We based our testing topology in this simple case, but extended the amount of servers connected to

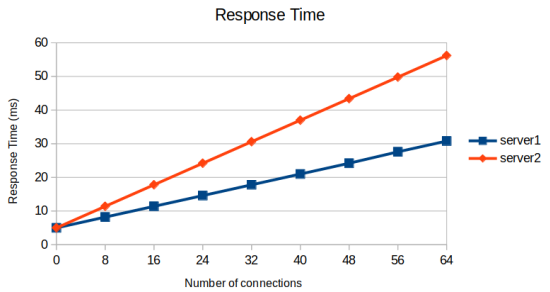

Figure 3: Artifical load, request response time by number of active connections

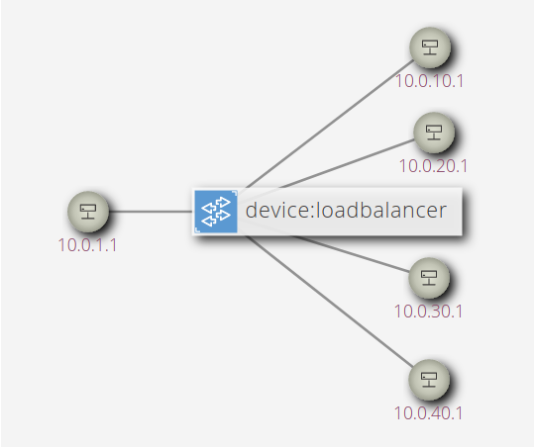

Figure 4: Load Balancing Testing Topology.
One client on the left, a load balancing switch in the middle, and four servers on the right.

the load balancer to have more broad tests. As such, our testing topology can be analysed in fig. 4, where the amount of servers connected to the load balancer can vary based on the tests being performed. The number of connected servers can be between one and four.

While the load balancing algorithm can not work with a topology that only includes one server, since there is only one path to where the traffic can go, we have this option because we are performing comparative case scenarios to prove that having two or more servers is better than only having one single server.

4.2.3. Test Variables

As explained in section 3, our application uses a load balancing algorithm that allocates weights represented as **flows** to each server based on their response time. These flows are controlled by the P4 program, as well as the ONOS controller, and are defined before compilation. The amount of flows that the program uses is defined by the programmer before starting the program, and can take any value in the form of a power of 2. We are testing these values for the total amount of flows with different topologies, each with a different number of servers, the tests are described in table 1 where each test is marked with an **X**.

Flows \ Servers	1	2	3	4
16				X
32				X
64	X	X	X	X
128				X

Table 1: Number of flows tested for each topology.

4.2.4. Test Script

To thoroughly test our application, we decided to develop our own script which will be executed by the client. Our script was written in Python and includes a library to perform HTTP GET requests to our servers, as well as a library to write the output of our tests in a Comma-Separated Values (CSV) file.

The client targets the load balancer virtual IP to perform the requests. It sends a set of 32 sequential batches of 128 requests, and between each batch of requests it waits either for user input, or for a *sleep* function. This wait between requests is unnecessary and would never happen in the real world, we decided to implement this feature so that between each batch of requests the servers can have time to send the load balancing report to the controller and this can evaluate if there are necessary adjustments to balance the system. With this behaviour we can have a more clear understanding of what is happening behind the hood of our application and show test results that are easier to read.

Lastly, we save all of the test results in a CSV file with the values gathered for each requests. The most relevant values are the response time of the request, and which server processed it. With the data successfully saved in a CSV file, it can be used to create Pivot Tables and plot the output with visual Charts.

4.3. Test Results

This subsection presents the results obtained in the tests described in section 4.2.

As previously stated, we designed our application algorithm with a couple of variables in mind, with this tests we aim to explore multiple values for these variables and reach a conclusion about which are the values that give the best performance. We will also test the overall performance of our system given a different number of servers to load balance.

4.3.1. System Tests

Our application balances the load between multiple servers using a weight variable, in our project we refer to this as *flows*. There are two important aspects to this component, the flows attributed to

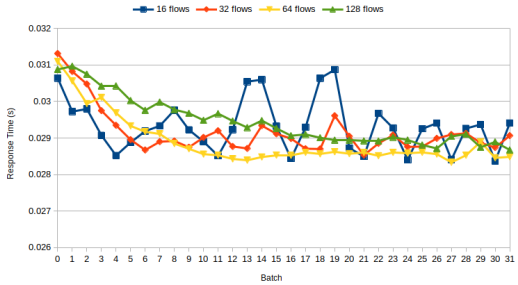

Figure 5: Average Response Time comparison between flows using 4 servers.

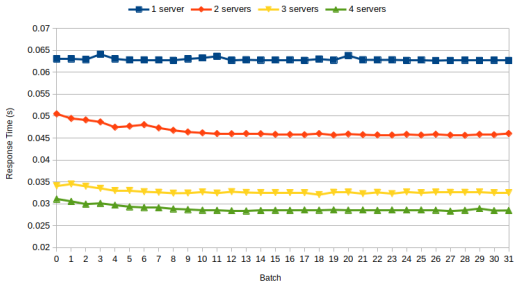

Figure 6: Average Response Time comparison between topologies with different number of servers.

one server, and the total amount of flows installed in the system. Since the P4 application is working with bitwise variables, we found both easier to developed and more performant to use values of the power of 2 for the total amount of flows in the system.

For testing the best values for the total amount of flows, we decided to use our standard load balancing testbed topology with four servers. The values that we will be testing will be 16, 32, 64 and 128.

In fig. 5 we can consult the Average Response Time for the four tests with different number of flows. We can reach two conclusions by analysing this chart. First, with less flows, the system has less stability. The tests with 64 and 128 flows have more precision than the tests with 16 and 32 flows. Secondly, with higher flows, the system takes longer to reach a stable value. Th test with 128 flows needs more batches of requests until it reaches the same value than the test with 64 flows, and the latter one also needs more requests to reach the same values of the tests with 16 and 32 flows.

Therefore we can conclude that, with 4 servers, the best outcome of our algorithm happens when we have a total of 64 flows. With 64 flows we have the best compromise between response time stability, and time necessary for the algorithm to find the best number of flows for each server.

4.3.2. Performance Tests

To test the overall performance of our application we used our system with 64 flows and tried multiple topologies that have between one and four servers connected to our switch.

In fig. 6 we can consult how the response time evolves based on the number of servers connected to the load balancer. We can easily understand that when there is only one server connected, the load balancer doesn't work. Either way this is a relevant scenario in order to have a response time value of reference.

When comparing the scenarios we understand that in fact the load balancing is working towards reducing the total average response time of the system. In all the cases were the load balancing takes effect we see a decrease in the average response time of the system, as well as a significant decrease in the maximum response time.

The expected outcome of a load balancing system is to reduce the average system response time when introducing new servers. We can verify that our application gives results in accordance with the expected in fig. 6. We can also verify that, with the increase of servers in the topology, the response time gain decreases. Therefore, it may not be worth the cost to introduce 5 servers if the increase in performance is very slow.

It is also important to add that all of these tests were performed with 64 flows installed in our application because that was the scenario with best overall performance for a topology with four servers. We then used the application with 64 flows and tried topologies with different number of servers connected to have a uniform testbed for all topologies.

4.3.3. Fault Tolerance Test

Software Defined Networks have the ability to have a unified view of the network. Advanced controllers such as ONOS have APIs prepared with entities that listen for a wide span of events, including events like the addition or removal of hosts.

With these kinds of entities we can create triggers that react to these events and change the behaviour of our load balancing application in real time.

We decided to use a topology with four servers and a total of 64 flows to test this functionality. During our test we used the capabilities of the Mininet cli to turn down, and later back up, the connection between the load balancer switch and the server4. The results can be seen in fig. 7, where the chart represents the evolution of the number of requests attributed to each server.

In this test we start with a topology consisting of four servers. As soon as the controller detects all servers, it distributes the 64 flows equally, meaning that every server is attributed 16 flows. In fig. 7 we can see that the servers start by receiving 32 flows

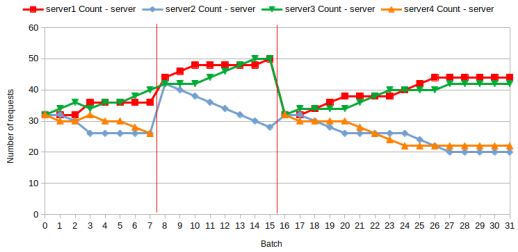

Figure 7: Fault Tolerance Test.
Server4 fails between batches 7 and 16.

each, that's because our test script sends 128 requests per batch, meaning that our algorithm loops two times and attributes 16 flows to each server twice.

The load balancer then starts the algorithm and begins to balance the load received until the batch 7, where the server4 crashes. At this point the controller detects the failure and reset the algorithm with three servers in mind, meaning that the previous load balancing evolution is forgotten and it start the algorithm from the start.

When the algorithm starts by the second time, it now only has three connected servers. It distributes the load again, but this time each server will be attributed with 21 flows (one server will randomly be attributed with 22 so the sum can give 64 flows). The algorithm then starts again and begins balancing the load with three servers online until the batch 15.

Server4 comes back online between batches 15 and 16. The controller detects again a change in the number of connected servers and decides to restart the algorithm. This time there will be no more interruptions and the program will balance the load until there it reaches a stable balance around batch 27, at which point the servers are considered to have requests with similar enough response times and the algorithm stops adjusting the flows.

4.3.4. Summary

The system was tested with different topologies and various kinds of flows.

We conclude that the number of total amount of flows can be adjusted considering the number of servers connected, in a topology with four servers, the best performance came with 64 flows, while with less servers a better performance may be achieved with less flows, as well as more servers may benefit from more flows.

Additionally, we reached the conclusion that adding more servers decreases the average response time, but the gap gets smaller with the addition of each server.

We also showed that the system has a near optimal behaviour upon the loss of one server, and that the adopted solution is fault tolerant.

5. Conclusions

In this chapter we present the conclusions about the project done in this thesis. Section 5.1 makes a brief summary of the paradigms addressed in this papers. Section 5.2 presents the achievements accomplished during the development of this project. Finally, section 5.3 references some relevant issues that may be addresses in future work.

5.1. Summary

SDN was developed to improve network configuration in traditional networks, it successfully decouples the control and data planes from the same network devices. The control plane was centralized in a controller that manages flow control by providing a global view of the network. There was a significant evolution in control plane technology, however, the data plane was left unchanged.

To address that issue and revolutionize the data plane, the P4 programming language was developed. It brings data plane programmability to switches by modifying how packet forwarding is processed. This can bring many benefits to SDN networking like custom defined network protocols and features.

This work aims to develop a load balancing application that benefits from the best features of both Data Plane Programmability, as well as Software Defined Networks.

5.2. Achievements

With our work we successfully improved upon the works mentioned in the Related Work, namely the article by Chih-Heng Ke et al. [8], by introducing a production-grade SDN controller, revamping the P4 code, and transforming a stateless algorithm, into a stateful algorithm.

Introducing ONOS in the application system bring a number of features like code modularity. This feature allows for the ONOS application to be higly configurable and interchangeable at runtime. With a generic load balancing scheme introduced in the data plane, the control plane can change the parameter of the Resource Based Load Balancer without the knowledge of the data plane. This is an improvement over the article by Ke since the work mentioned has the load balancing algorithms hardcoded in the data plane logic.

The project implemented in this thesis also supports an ambiguous number of servers trough configurability with NETCONF, while the related work has the servers explicitly in the data plane code, changing the topology by adding or removing a server is a heavy task.

Additionally, by introducing more control in the control plane, we were able to migrate a stateless

system to a stateful one, introducing the advantages of the latter.

However, the focus of Data Plane Programmability is to increase the control of the network in the data plane. With our work we are splitting the control in between the data and control planes.

One of the conclusions of the paper by Ke was that by providing the load balancing algorithms in the data plane, their system was independent of the controller. Our is more dependent of the controller since it is constantly trying to balance the load between the servers based on their average response time. If the control plane fails in our solution, we lose the functionality of realtime analysis of the system, but the state of the load balancer when the control plane fails is saved in the data plane, meaning that the control plane could fail and our load balancer would still work, while being stuck in the same load balancing state.

If we consider that most of the time the system is in a stable state, that it only converges between servers in a small window of time, we can assumes that work solution remains working properly even on a control plane failure. Instead, if the system is unstable, and the control plane is constantly changing weights in between servers, then we could assume that our soultion is not perfect, in a scenario where the control plane fails.

5.3. Future Work

On one hand, we defend the SDN paradigm which states that the control should remain in the control plane, where ONOS with a unified vision of the network has a lot of visibility and can manage the data plane by installing flow rules that work as ”guide lines” for the data plane to work independently. With the help of ONOS modularity we propose as future work to improve the control plane application to have a broader amount of load balancing algorithms which the user could chose and switch between at runtime. Also, we propose to improve the situation where the algorithm takes a long time to converge when a high number of flows is introduced in the system. This implementation could extend the ONOS GUI component to include custom buttons and commands in the web application.

On the other hand, we would also like to explore the capabilities of P4 to the maximum, and propose to try and bring all the algorithmic logic in the control plane to the P4 code. This would be a hard task, the hardest part would be to maintain a generic code for any arbitrary number of connected servers.

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming protocol-independent packet processors,” *Computer Communication Review*, vol. 44, pp. 87–95, 2014.

[2] F. Hu, Q. Hao, and K. Bao, “A survey on software-defined network and openflow: From concept to implementation,” *IEEE Communications Surveys and Tutorials*, vol. 16, pp. 2181–2206, 2014.

[3] 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, pp. 69–74, 2008.

[4] N. McKeown and J. Rexford, “Clarifying the differences between p4 and openflow - open networking foundation,” accessed 21 December 2020. [Online]. Available: https://opennetworking.org/news-and-events/blog/clarifying-the-differences-between-p4-and-openflow/

[5] P4.org, “The onf and p4.org complete combination to accelerate innovation in operator-led open source,” 2019, accessed 3 January 2021. [Online]. Available: https://p4.org/p4/p4-joins-onf.html

[6] O. Shyshkov, “Load balancing - oleksii shyshkov’s blog,” 2018, accessed 18 October 2021. [Online]. Available: https://oshyshkov.com/2018/07/20/load-balancing/

[7] “Load balancing algorithms and techniques,” accessed 30 September 2021. [Online]. Available: https://kemptechnologies.com/load-balancer/load-balancing-algorithms-techniques/

[8] C.-H. Ke and S.-J. Hsu, “Load balancing using p4 in software-defined networks,” *Journal of Internet Technology*, vol. 21, pp. 1671–1679, 2021.

[9] D. A. Menascé, “Load testing of web sites,” *IEEE internet computing*, vol. 6, no. 4, pp. 70–74, 2002.

[10] B. Boucheron, “An introduction to load testing — digitalocean,” 2017, accessed 7 October 2021. [Online]. Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-load-testing