POLITECNICO DI TORINO

III Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Deployment of programmable hardware in networking nodes

Data plane programmability in OSes

Relatore:
prof. Mario Baldi

Candidato:
Salvatore Pecoraro

Aprile 2018

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Abstract

Data-plane programmability is one of the most active trends today in the field of networking. The possibilities it is opening in the use of custom protocols and newer applications are generating a big effort in the implementation of its concepts. This introduces a new way of designing and producing network devices but also network OSes and protocols, which requires obviously new approaches in the various areas of testing in order to validate each component.
The objective of this thesis is offering original inspection methods which must be able to adapt the concepts of conventional testing to the new devices and their capabilities. The work is focused on two main cases: the functional testing for the porting of an existing OS on a new programmable chip; the emulation of an entire network of devices offering data-plane programmability, with the aim of defining their capacities in the execution of distributed algorithms. For the first case the proposed solution will be based on the use of an automation server to implement a light-weight Continuous Integration, while the second part of the thesis will suggest a way to use Mininet to launch and measure programs and protocols on a network of behavioural models.

Acknowledgements

I would like to thank anyone that has given his/her contribution, with any possible mean, to the writing of this thesis, even if I assume all the responsibilities for any potential error that could be present in the composition.

I intensely thank professor Mario Baldi, Supervisor of the thesis, not only for the essential help and guide throughout the entire thesis work, but also for having granted me the opportunity to enrich myself on a professional, cultural and personal perspective, living in a context that has the flavour of a dream.
I also thank Cisco System, Inc. for having offered me the chance to study deeply the problems and the topics that were the basis of this thesis.

A deserved thought goes to my entire family, for having supported me from the very beginning with all the means and in every difficulty and for being always part of my happiness, even when it brings me farther than what we can expect.
I thank all the friends that have always shown great affection, wherevere I was. I thank the long-standing friends of my hometown for being always present in my life and letting me feel at home even when I am miles away from Potenza, but also the ones I have met in Turin and France, with which I have shared the fear of new experiences in stranger places that has built an almost familiar relationship.
I infinitely thank my girlfriend, Vanessa, that has valorized and borne every part of my character, with an affection that has brought a new an deeper serenity into my life, which was fundamental in my most recent experiences.
In conclusion I thank all the people that left us before this important achievement, who guided me through the memory of them and their example.
Without all of you I would not be the man I am today, therefore I dedicate this work to all the people that inspired and enriched my personal development path.

Ringraziamenti

Desidero ringraziare tutti coloro che hanno contribuito, in ogni modo, alla stesura di questa tesi, pur assumendomi la responsabilità di eventuali imprecisioni ed errori.

Ringrazio profondamente il professor Mario Baldi, Relatore dell'elaborato, non solo per l'indispensabile aiuto e guida durante tutto il lavoro di tesi, ma anche e soprattutto per avermi dato la possibilità di arricchirmi a livello professionale, culturale e personale, vivendo un contesto che ha il sapore del sogno per chi lavora nel nostro ambito.
Contestualmente ringrazio Cisco Systems, Inc. per avermi offerto anche l'opportunità di approfondire i temi e le problematiche da cui sono partito per la stesura della mia tesi.

Un doveroso pensiero va poi alla mia famiglia tutta, per avermi sempre sostenuto con ogni mezzo e in ogni difficoltà ed essere sempre stata parte delle mie gioie, per quanto queste potessero allontanarmi momentaneamente più di quello che potevamo aspettarci.
Ringrazio inoltre tutti gli amici che, dovunque io fossi, non mi hanno mai fatto mancare il loro affetto e la loro vicinanza. Penso agli amici di sempre della mia città, presenti anche a chilometri di distanza e pronti a farmi sempre sentire a casa, così come coloro che ho conosciuto a Torino e in Francia con cui ho condiviso le paure di una nuova esperienza in posti sconosciuti cementificando così un legame divenuto col tempo quasi familiare.
Ringrazio infinitamente la mia ragazza, Vanessa, che ha valorizzato, e sopportato, ogni parte del mio carattere e ha saputo portarmi, con il suo affetto, una nuova e più profonda tranquillità che è stata fondamentale nelle mie esperienze più recenti.
In conclusione ringrazio chi non ha avuto il tempo di godersi e festeggiare questo traguardo assieme a me, pur avendomi guidato con il ricordo ed il loro esempio.
Certamente senza ognuno di voi non sarei quello che sono oggi e, per questo, dedico questo lavoro a tutti coloro che hanno ispirato e arricchito il mio percorso di crescita personale.

Contents

List of Tables

List of Figures

Chapter 1

General Introduction

Data plane programmability is, nowadays, the latest innovation in networking that, after having generated a lot of interest in scientific research, is trying to strongly penetrate the market of switching devices. This innovation is attracting a lot of attention for its potentialities in the field of Data Centers (DC), Cloud Computing and Virtualization that are getting more and more relevant today in a wide range of applications: IaaS, Machine Learning, Internet facing applications and so on.

This is due to the incredible power that Software Defined Networking (SDN) has given to this kind of infrastructures and paradigms, through its strong abstraction of control plane. Indeed, in this philosophy, switching devices are not involved anymore in the definition and construction of the state of the network, they are only responsible for the forwarding operation.

The management in the SDN paradigm was moved to a controller that provides the user with high-level commands, hiding all the details about how to impose the correspondent configuration to the network. To do this, the controller must acquire all the information about the topology and operational state of the network, and then translate these high level-commands in effective instructions for the devices. SDN paradigm allows this accentration and the definition of rules the controller can use to define a new forwarding state, depending on the view of the network it has and the commands it has received.

This is adding flexibility because switches do not run anymore all the protocols that defined in the past their behaviour during forwarding: they just use match-action tables to implement a decision that is taken at the controller. Therefore the switches will be instrumented only with some rules to install the proper forwarding state. Furthermore even the controller will implement only the logic that we need on a specific application, without all the algorithms that had to be supported by the switching devices that made network history.

All of this, together with the improvements in virtualization, permitted to easily create and remove, also on-demand, entire networks tailoring any aspect on the

service needed, wherever there was enough computational and switching power. It's quite clear why in environments like Data Centers, where orchestrators and virtualization systems have huge flexibility in spawning machines and nodes, this has represented a real breakthrough since it extended this flexibility also to networking.

These great advantages have inspired the search of an approach able to push these concepts one step further: programming also the data plane, and so the parsing and forwarding of packets themselves, rather than just the control plane with its management and configuration protocols. Indeed if SDN allows to manage the services and the resources in a very elastic and fluent way, it does this only for the protocols that are supported by the southbound APIs that we choose. This is where data plane programmability extends the revolution started by SDN, eliminating this fundamental limitation.

The main idea, embodied for example in the P4 language, is letting the programmer define everything about forwarding, with complete freedom in the context of the match-action table paradigm. The network programmer can now program the fields to be parsed, the way they are processed, the actions that the switch must take on every packet and the way we match over the defined fields to choose the right action. The big novelty is that we can get the switch to parse custom headers and treat the packet with custom actions, that combine additional processing to the usual forwarding. This concept expands our horizon, enabling the definition of every aspect in our networks from the forwarding state management and programming (control plane), to the effective forwarding process.

This also paves the way to accepting a big challenge: producing switches that implement this paradigm, offering P4-programmability in different flavours.

The development of such a product is where the project for this thesis finds its roots. Indeed, the use of a data-plane programmable chip will require the construction of new platforms comprehending also operating systems, whose characteristics will depend directly on the data plane. Every time the data plane is modified, the system needs to be tested and, consequently, the automation of this operation gains a vital importance.

For all these reasons, the thesis focuses on the creation of a testing framework offering a solution to these necessities in two main areas: automatic functional testing of a single device for the porting of a switching platform (Cisco NXOS) on a P4-enabled chip (Barefoot Tofino chip); the construction of a testing framework for the emulation of an entire network of P4-enabled devices, to inspect its performance in the execution of distributed algorithms.

The functional test, which preceeds the others also chronologically, focuses on the correctness of the code and the features it offers. It is a kind of inspection that is linked to the development phase of the product, to unveil the presence of bugs and eventually check the interaction with new platforms and interfaces. Undoubtedly, once the functional operation of the device has been verified, it is fundamental to

understand the performance it is able to assure in real scenarios. This introduces the second class of tests studied in the thesis: *performance testing*.
Both these types of inspection can be executed on the single device or on a network of devices and this is fundamental because the execution of distributed algorithms in a network will become the natural environment of the switch, once it is on the market.

Therefore if we talk about network device testing, we are referring to an umbrella concept that groups all these different shades. For each one of them a framework with specific characteristics is needed.
As a consequence, this thesis aims at building competitive platforms that combine the requirements of testing platforms with the challenges introduced by the use of a P4-enabled switch and its novelties. Great attention will be paid to the importance of having a testing framework able to offer reliable and correct results, but also a good grade of automation and integration in the context of the development process.

The thesis will focus on two testing scenarios and it is organized as follows.

- The first part will analyse the construction of a testing platform that can be used to test single network devices and their features. The project under inspection in this part will be the porting of a switch OS on the P4-enabled hardware, like Barefoot Networks Tofino chip. The main idea is using the *Packet Test Framework* (PTF), based on *Python* and derived from *unittest* and *Open Flow Test framework*, to emulate the interaction that characterizes the operation of the device.
  This must be intended as a kind of test inspired by the idea of Continuous Integration: we want the testing platform to support the work of the programmer throughout the entire development of the product. The platform must follow the production of the switch and its pace, giving the developer the chance to test every change in the code.
  The PTF framework uses for the tests the behavioural models offered by Barefoot Networks and the library functions that give an interface to these models. Testing is then integrated with the code development chain through the use of a Jenkins platform, which is able to recover the newest version of the code, build it, run tests and show results in an automated fashion.

- The second part will introduce a way of emulating an entire network of devices, in order to inspect distributed protocols and algorithms that are implemented through data plane programmability. The algorithm that will run on the devices is a deduplication algorithm that tries to eliminate duplicated packets that are originated in a traffic sample done through tap interfaces or Switch Port Analyzer (SPAN) ports.
  A first effort in this case, under the perspective of the testing framework, is

emulating the network that produces traffic, so that it is as similar as possible to real use cases. Then the logic continuation of the work will be the measurement of the performance of the algorithm on a network of P4-enabled switches running on top of the Tofino chip, to define its advantages and features.
The platform will be created using *Mininet* and its *Python* API, while the performance measurements will rely on the *tshark* tool and other *bash* utilities. Regarding the switching devices, instead, the production network will use some standard Open vSwitches (OVS) switch that are connected through some SPAN ports to the P4-enabled switches. These devices will run the analysed algorithm and are reproduced, in their behaviour, through the models offered by Barefoot Networks.

Chapter 2

Testing of a single device

2.1 Introduction to the platform

Testing the behaviour of a group of switches, in the terms of their mutual interaction and cooperation, is probably the most important evaluation to do on a network device. That will be indeed its usual operational context.
However, this is a step that can't be done without being sure that, at least when inspected alone, the switches behave correctly. This is the reason why the investigation on the behaviour of the single device, which chronologically preceeds the group/network tests, becomes central and pivotal in the development chain.

Generally speaking, this kind of inspection on the switch behaviour must check that the device is working efficiently and respects every customer requirement. It is a test on the switch itself, that is its entire operation and software.
In the specific case I had to face, instead, the situation was very different and more particular than a normal functional testing. Indeed the goal of my work is not the development of a completely new switch OS: the software and the switching logic of the device is already defined, but it is coded for a traditional forwarding hardware. The development is focused on testing the adaptation of this system on the Tofino chip, to offer the potentialities of data plane programmability. In this context, the target of functional testing must be checking the effectiveness of this porting. It will therefore become, in practice, a test focused on the interface between the operating system of the switch and the new programmable hardware. The APIs that will be used during this porting operation will consequently play a key role in our testing to work on this border.

Another important detail to consider, when we build platforms for this kind of inspection, is that nowadays functional testing must unfold together with the code production. This must be intended in the sense that testing has to be extremely integrated with the development to be able to inspect every modification as soon

as possible and become a sort of counterpart of code production as well as a real tool for who writes the code. Every change and addition to the potentialities of the device must be immediately stimulated and results must be as clear as possible in underlining possible bugs and errors, so that the inspection becomes an effective help and a way to support the recognition of the effects of the team work.

This is in line with the philosophy of Continuous Integration, which is becoming more and more popular today and is well explained in [1]: every change that is committed is checked for correctness, built and tested in order to confirm that the system is still respecting design requirements. In our particular case of study we are not trying to embrace completely this way of thinking, since we are not interested in committing as frequently as this ideas would require. Anyway checking the functional operation of the device and its interaction with the chip, in an automatic fashion and at every commit, is the main goal of this part of the thesis.
The improvement that this working method can bring is that, if we check the code at every commit, we will have a restricted source of bugs and unexpected behaviours for every test. If we know that functional testing was successful before, whenever some test fails only the new code is responsible for that. In this case, the advantage of knowing where to search represents the real convenience and, when we test the porting of an existing system on a new hardware, it becomes fundamental because this is an interface work. Consequently, knowing which modules create a problem makes the difference in understanding how the others are affected and solving it.
Moreover, it can also help workers build a new attitude while developing. In the idea of Continuous Integration, since all the code that is committed will be tested, the developer will start working with more responsibility: a build break or test failure will stop the entire team, since the tests will fail also for the following commits and, additionally, they will increase the amount of tested new code generating a wider potential source of the problems. For this reason, the porting will be coded with more attention to building restrictions and to reducing the perturbations in the system.

The necessity to follow the development requires that the framework or the platform, which is going to execute the functional testing, must be always up to date with the code that has been written. Therefore, it must be fully integrated with the process of code production and the tools that are used in it. This is necessary to make testing a part of the chain without requiring any specific additional operation to the programmer. The programmer must be able to run the tests as easily as possible, and in this sense our objective becomes making the system able to fetch the changes to the system automatically.
Moreover, all of this must be available both for on demand testing and automated testing, usually executed after commits and late at night to exploit periods of low load for resources.

Generally speaking, the main idea is that there must be a continuous integration

server (called also automation server) that is connected to the version control system that is used in the company, to recover the latest version of the system to test. Then this code must be built, installed and run on the device that we are using as testbed and, at that point, the framework can start the inspection to understand the behaviour of the target. This is done running particular programs on a machine that is controlled by the automation server and at the same time is connected to the switch. These programs will reproduce common operational situations and events to understand how the switch reacts.
These programs are our tests that usually use some specific library that give us the chance to write them with very high level instructions, which are then turned in the proper packet-level or low-level configuration command. Thanks to this, our scripts or executables are able to bring the switch in the right operational state and then check its answers to different impulses.

As we have said previously in the composition, our objective is proposing an approach that applies current testing technologies on the development of code for devices with a programmable data plane. The big challenge to put this in practice is that these networking concepts are not completely supported by all the tools and interfaces, because of their extreme novelty.
Anyway, some environments address this issue and, among these, Barefoot Software Development Environment (SDE) offered the possibility of running tests using the PTF framework. Notwithstanding the lack of clear documentation other than example code, this framework allows to write tests to check the compatibility of the OS with the chip through different APIs. Those APIs, laying on the border between OS and hardware, will help us in the inspection of this interface.

The objective of this part of the thesis will therefore be the description of the steps taken in order to adapt and put in practice these concepts, in the context of the development of P4-enabled devices. We will analyse the choice of tools and the different problems that we faced, explaining the reasonment behind every decision.

2.2 The continuous integration server

The continuous itegration server, for what has been said until now, is the fulcrum of the testing framework. It is connected to every part of it, it must coordinate and instruct the work of every other node and, in the end, it must present the results. It takes the role of the judge in our developing chain: the product of its work is deciding if the changes that were made have produced a system that is still satisfying the requirements represented by our tests ([1]). Therefore, the choice of the right tool becomes fundamental to get this server to integrate into our development chain, to get the best performance and the most straightforward indication of potential errors.

The choice for my system went on the server that is probably the most used today, which is Jenkins. The choice was due to various reasons that I'll try to resume hereinafter.
Jenkins is a Java-based automation server and this enables system administrators to run it on every OS where the Java Runtime Environment can be installed and run. The main idea that characterized Jenkins design and early releases was creating a tool with two souls: a strong central core and a high number of plugins. The core of the tool keeps together all the infrastructure but is not enough to satisfy testing necessities. That is, instead, the objective of plugins that give to Jenkins a great adaptability to a wide range of different situations, which was one of the reasons of our choice.
It is important to underline that this philosophy and this great separation was mitigated by the recent releases. Indeed now, when Jenkins is installed, the user can choose to select manually the plugins to install to increase the potentialities of the tool, or he/she can also decide to install a group of *default plugins*, which offers a good coverage on the services an average user needs.
In general, being Jenkins an open-source tool, lots of plugins were created by its community and this gave the possiblity to have different alternatives in every aspect of its operation, from the version control system it can access to several other building and testing options or interfaces.

Moreover, these great qualities are combined with different opportunities for installation, that help the administrator adapt precisely the automation server to his/her resource availability. In particular, for our tests, the chosen alternative was the docker version of Jenkins, because of its ease of installation but also for the isolation that this could assure us with respect to all the other processes running on the available servers. In this way, all the automation part was run on the docker avoiding any unwanted interaction with the rest of the server hosting Jenkins, which appeared as a separate machine to the docker. This can be very useful if we want to use the same server to run tests. Indeed, the separation we mentioned previously allows to use the rest of the server as a Jenkins Slave, keeping the orchestrating part well isolated and protected from the computational power used for the inspection.

As we slightly anticipated, Jenkins operation is based on the interaction between two categories of actors: the *Jenkins Master* and *Jenkins Nodes/Slaves*. In [2] it is possible to understand that the Jenkins Master is the process that coordinates all the build work and represents for the administrator the portal to the automation system, offering a web interface for the configuration of the platform.
The Jenkins Master can execute directly builds and tests but these are usually delegated to the Slaves, which are simply machines that are used as computational resources in the cluster. The administrator can choose to bind a project to a particular slave, in order to exploit its particular configuration or installed software, or let the Master select anyone of them, allowing the system to have a better load

balancing. Offloading work to Slaves enables the Master also to devolve all its computational power to controlling the jobs and offering an effective web interface to the users.
Even if ssh is the most used, the interaction of the Master with Slaves is implemented trough different types of connection. The connection is used by the Master to send jobs to the Slaves, but also to recover the status of the execution and of the workspace, which is then displayed on the web interface together with build and test results.

Consistently with the isolation choice that was made with the docker installation, the setup that was chosen in this phase was delegating the work to the Slaves in order to leave the Master free to care only for the interface and coordination work. Another reason to choose this setup is that, in general, the previously mentioned PTF tests can only be run if the Barefoot SDE, together with PTF dependencies, has been installed on the machine. Therefore, in order to keep the environment of the docker as clean as possible and avoid the SDE installation, the tests were run on other machines or, eventually, on the same server but outside of the docker.
Furthermore, when running tests with real hardware, there could be cases in which the testing machine must be directly connected to the testbed in the laboratory. In those situations, the delegation is forced if this machine can't host the Jenkins Master, like in our case.

For the great freedom that is given by all this installation alternatives and the versatility conveyed by the richness of plugins, Jenkins was the choice that best fitted the needs of our testing, since it adapts to a lot of situations and can be easily reconfigured and extended whether the necessities change over time.

2.3 Configuring the Jenkins project

Let me try now to delve deeper in the use that was done of Jenkins in this work. As we said, the objective is creating a Continuous Integration framework that is able to test code, for the porting of a network OS on P4-enabled devices, every time it is committed or on-demand. After explaining the details about the system installation, it is important to figure out also how the Jenkins project was configured to reach this goal.

Jenkins offers a lot of different types of project to automate building and testing. A basic and very popular one is the Free-Style project, in which the user can customize the execution of the build under many different aspects: how the code must be recovered, if we want to allow a concurrent build or bind the project to a particular machine, when to trigger the execution, which instructions should be done and how to handle the results.
The second very popular type of project is the Pipeline. It was introduced to allow

the use of Jenkins in the context of the Continuous Delivery paradigm, which extends the Continuous integration with the delivery of the verified code at the end of every build. The addition of the delivery after testing, together with the higher sophistication reached by builds and tests, brought Jenkins projects to become quite long in their execution and not so easy to manage.
That pushed the introduction of the Pipeline in which we can define a concatenation of build stages, which are similar to normal Free-Style projects, in order to divide the different steps and handle them better. This is perfect for long-running jobs that need to involve more than a slave, even because checkpoints can be created so that builds don't have to restart from the beginning after errors. Then other less popular kinds of projects can be chosen, depending on the needs, but these two are the ones usually taken into consideration.

For the case that is discussed in this thesis, the chosen alternative was the one of the Free-Style project, since the job remains quite straight-forward in its workflow. The tests run are quite heavy in resource consumption and slow in terms of execution time, but the instructions are still quite easy. That's the reason why the Pipeline could result, in the end, quite exaggerated in its complexity and was not used.

Once this decision is taken all that is left to do is defining the details and do the actual configuration.
Considering the use that was planned for this platform, I decided as said before to run the build commands on Jenkins slaves, so everything is executed outside the docker that hosts Jenkins. When tests are run using the behavioural models offered by Barefoot SDE, everything was run on the same server where the docker was runnning. When we moved to the inspection of real hardware, the run was instead bound to a machine directly connected to the testbed, to exploit some particular checks we could do with our tests.

The second thing to configure is the Source Code Management, which covers all the informations required by Jenkins to fetch the source code of the project. In this section, it is possible to specify the version control system that hosts the code and the repository that must be recovered from there: I exploited the space that Cisco has on Gitlab so that git could be used to fetch the code. Because of this, the only thing to do when a new part of the code is added is writing the proper test and push both the code and the test to this repository. In this way, everything is then pulled by the selected Jenkins Slave that will start the build and tests.

The other fundamental setting to program, which defines also how much we are embracing the Continuous Integration philosophy, are the events that trigger an automatic build. Different options are available, I decided to trigger a build after every commit and anyway at least once a day during the night, when the cluster has low load. However, the system allows also the execution of a build attempt on demand through the web portal.

After that, what's left to configure is the most intuitive part: the build instructions. In this step, it is possible to invoke many different type of scripts (bash, Windows batch commands, Gradle) or start tools like Maven or Ant. For the platform I was building, I decided to use a simple bash script. The important thing to remember is that the script will run on the Jenkins Slave and will be run in the folder that represents the root of the repository fetched from GitLab. The Slave is anyway a machine where Barefoot SDE was previously installed, so that dependencies and utilities needed in the OS and the kernel will be ready to run. In our case, when testing only Barefoot SDE or chip, the build is not even a real build: we just set some environment variables and start first the behavioural model, when the test is not involving real hardware, and then the test scripts.
When the code written by the developers for the interface between the OS and the chip is added to the repository, the building commands will be put right after the setting of those system variables in order to offer the new version of the software to the testing facility.

Finally, the last thing to define are the post-build actions: with this option I was able to give to Jenkins the location of some XML reports generated by the test scripts in the XUnit format. These reports can be used by the *Test Result Analyser* plugin to present the results to the programmer or to the team with hystograms and cake charts.

In conclusion, we have seen the configuration needed for the continuous integration server to have a platform able to test code in an automated fashion, showing the wide range of possibilities given by Jenkins. The next paraghraphs will expose the other face of the medal: we will discover the framework used to write tests and the generated outputs.

2.4 The PTF framework and the execution of the test

If we want to test the porting code every time it is committed, it's obviously necessary to find an easy way to write tests. It is important to have a good interface enabling us to write tests fast and at high-level, even if the system must give the possibility to check almost every detail.

The PTF framework is a Python-based framework that has been developed to offer this service for P4-enabled and Barefoot Networks devices. The framework defines a series of classes and functions that can be used to write the succession of events that represents the test and that define the way the platform will interact with the network device. In particular, tests are written as classes that extend the ThriftDataPlane class with the function *runTest*. This base class defines everything

that is needed to connect with the tested switch and its servers. In the case of Barefoot SDE, these classes have the responsibility to start the clients that must handle the sessions with the thrift RPC server, the Switch API RPC server and, if necessary, the SAI RPC server. When we run the tests indeed, after the scripts have recognized the group of tests that has been chosen, for each of them the system opens a connection with these servers on the switch through the ThriftDataPlane class. Then the function *runTest* is run and the test is executed.
The sessions opened with these servers are fundamental since they are needed to transfer to the switch the commands used to define tests. The servers have the responsibility to assure the execution of those instructions, translating everything into low-level commands that are comprehensible to the switch or its behavioural model. They will have to follow the execution and then return to the client the result, which will be used to continue the test.
To be more precise, the thrift server is the one that is responsible for the effective configuration of the switch, it is the same server that receives the commands when we use the CLI offered by the device. This means it is the server that works closer to the interface of the chip. The others are mainly dedicated to an interface work between the API used for the setup or the test writing and the thrift commands. It is quite clear, therefore, that these mechanisms and these RPC connections are pivotal for the execution of the tests and, additionally, are what truly allows the framework to offer such a straight-forward interface for their definition.

Obviously all these services are a bit beyond what is the true operation of forwarding. Indeed, when we run the tests with the help of the behavioural model, there is a strong separation between the model itself and the drivers needed to communicate with it. The former offers the emulation of the real chip in the execution of every command and instruction, while the drivers for the model are responsible for setting up all the processes needed to interact with the forwarding resource represented by the model. The drivers care for setting up the virtual interfaces that enable the connection to the model, start the servers needed to interact with the PTF framework and, consequently, they are in general what permits the interaction of the model with the external world. For this fundamental role they cover, they are part of the switch as much as the forwarding unit: indeed this separation is not visible in the real product, but only in the case of the emulation of the device through the behavioural model.

As we anticipated briefly, the real test is defined as the function that extends the ThriftDataPlane class and is written using one of the API supported by the clients launched by that class. All the APIs present a wide range of functions to configure the device and inspect its functional behaviour with the emulation of some events but, at the same time, each API assures different properties.

The switch API, as it is implemented in Barefoot devices and SDE, is targeting

common networking operation from different layers of the TCP/IP stack. The features required for a compliant device, indeed, go from basic L2 switching (learning, flooding, etc.) to L3 switching, L2 and L3 Multicast, different tunneling options, MPLS, mirroring with ERSPAN, Quality of Service, NAT and much more. We are therefore talking about the main technologies used in networking today: the PTF functions allow the use of this API to ask the switch the execution of the different commands needed to configure and use all these mechanisms.

The SAI API fulfills the same goal but it's very different in its intent and idea. Indeed the switch API is defined for a particular hardware suit, which is Tofino in this case, like it has always been in the past of networking. Before the definition of SAI API, every chip vendor and manifacturer defined a custom interface with the software that had to run on the produced hardware. This practice reduced the portability of the code written in any company using those chips, or at least made the code less readable and the work needed to have a portable code quite hard.
In this context, the need of a unique and manifacturer independent API emerged strongly to give to network developers at least a set, eventually restriced, of features and functions that are common to every possible ASIC. This gives the possiblity to write, for the ASICs that respect the SAI API, code that must not be modified in the event of a change or update of hardware suite. Additionally, the philosphy of ASIC design and production of compliant chips had to be reviewed, since new features now have to be added only as extensions of this new interface.
This is the reason why, during my internship at Cisco Systems, there was great attention in focusing the automated testing infrastructure on the operation of the SAI API. This is a choice that can be a bit restrictive on the short term, since other APIs offer more functions to interact with the chip during the tests, but on the long term this choice assures that all the tested code and functions would still be ready to run also on different compliant chips. It is a forward-thinking decision that defines the policy of our testing.

2.4.1 Writing the actual test

As anticipated in the previous paragraph, the PTF framework will give us the chance to start from the APIs we have at our disposal to run our tests with the proper configuration. Appendix A will give an example of the process.
It is important to understand the situation we start from in the tests: the idea is that we are interacting with a multi-layer switch and the initial configuration presents an L2 bridge with a list of ports that are its interfaces, including also the virtual ones. At the beginning of each test it is necessary to choose which of these ports will be used, since those are the ones we need to configure. As we said, the switch starts by default at layer 2 in the PTF tests, so if we want to test only the L2 features we just have to configure properly the interfaces: we can set VLANs as a port attribute,

define Link Aggregation Groups among ports and define entries of the flow table for forwarding purposes.
If instead we want to test L3 forwarding there is an additional step to do: a virtual router must be created and the ports that must support the L3 operation will have to be detached from the L2 switch to be used as virtual router interfaces. Then, with techniques very similar to what happens for L2 tests, interfaces can be configured in many different ways to have the router work in several conditions. It is possible to install routes, define multiple paths for the same destinations to check ECMP operation and much more.

Once everything is configured, the introductory part of the test is over and the real action starts. In this particular moment the machine running the test must create the events that trigger the reactions of the inspected part of the system, emulating the operation of the neighbours that are expected to interact with the device. In the vast majority of cases, at least in our testing, ad-hoc tcp packets are created in order to correctly stimulate the P4-switch: this operation is as easy as calling a function or creating an object, and it is possible to specify the value of every field of TCP/IP headers or just the major ones, leaving the default values for the others.

Then the packets are sent to the device and its behaviour is verified. It is possible to verify different things with a lot of variants. We can check that a packet is sent out from a specific port or, if we do not want this to happen, check the opposite. We can see if the packet is sent at least on a port in a group or to all of them but also any other boolean condition through assertions, like the balance conditions on aggregated links and multi-paths routes. Since the verifications launch exceptions if they are not satisfied, all packet transmissions and verifications must be written in a try-catch clause to handle potential failures. After that, all the resources are released to leave a clean environment to the next test.

A very interesting feature of PTF framework, as anticipated in other parts of this thesis, is that it is possible to ask to the platform to present the results of the tests in the XML Xunit format. In this case the success or the failure, with the exception that generated it, are packed in a format that summarizes the output of the inspection, including the traces recorded from the streams of the console. The production of reports in this standard shape is fundamental because enables a lot of tools to show with impressive graphical power the trends in test results, as done in our framework by the *Test Result Analyser* plugin of Jenkins.

2.5 Outputs of the platform

It is now important to talk about the outcome of the test execution: this is probably one of the biggest contributions given by this framework to the work of the developer.

Indeed, all the framework is trying to offer is a testing system that must not be just a detached and subsequent functional check. This requires the tests to start almost seamlessly with the commit of new code, in order to exploit the earlier cited advantages of the Continuous Integration paradigm, but it calls also for clear and significant outputs.

Indeed if we want the test to be a real weapon in the hands of the programmer, restricting the sources of bugs and errors is not enough. It is necessary to offer the results of the tests to anyone that is interested in them and with the most straightforward access. This is fundamental to identify easily the tests that have failed and the problems that have generated their failure. The objective can be achieved through some Jenkins plugins that are able to offer a complete insight into properly coded results through the web interface of the automation server.

The testing framework, for this reason, has been configured to generate XUnit reports, and this enabled the Test Result Analyser plugin of Jenkins to show multiple views on the outcome of the test. First of all, the platform gives a chance to graspe and sense the trends in the inspection at a glance, through its charts that are mainly dedicated to the overall statistics of the different run and build attempts. Among them we can see in Figure 2.1 the total number of successful and failed tests for the last twenty builds and in Figure 2.2 the time elapsed, in seconds, during the execution of the tests.

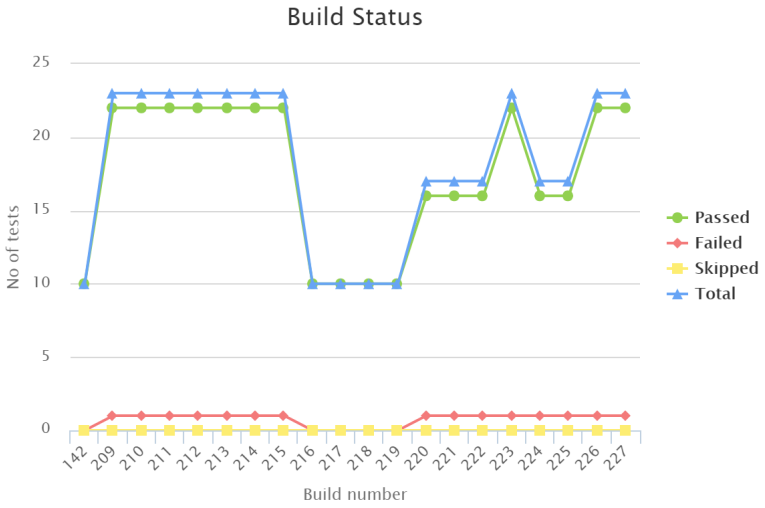


Figure 2.1: Jenkins Results chart.

Both of these results are not the effect of my work, since I was not directly involved in the development of the tested code. Anyway, for the sake of completeness, it is interesting to present this information to give a sense of the final result of the construction of the platform. We can see that the no test is being skipped, which

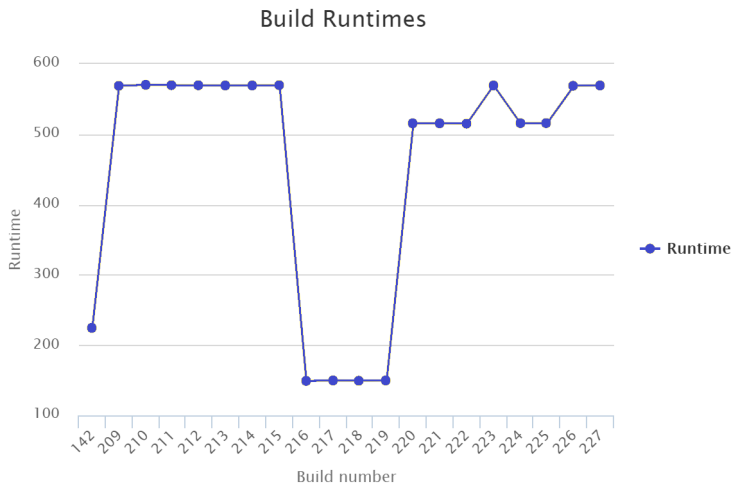


Figure 2.2: Jenkins Runtime chart. Times in seconds.

can be taken as an indicator of a good setup of the system, and the majority of tests are passed. The build time is quite high because of the delay generated by the access to the git repository.
The other thing I find very interesting to show with this charts, is the different types of runs that can be executed. In fact, it is possible to see a certain oscillation in the number of tests that are executed and, consequently, in the time required.
The builds with the highest number of cases are usually the ones programmed to start automatically, at least once a day, and address the widest available portion of the code. The others represent most likely the tests run on demand by the developer, which evidently wanted the inspection to focus on a specific service or network layer. In this sense it is relevant to show the results also to understand the use that is done of the framework in the context of code production.

The charts, anyway, are mainly useful to understand how the development of the code is proceeding and, for these reason, they are probably more helpful for managers' work. In the same panel, the web interface offered by Jenkins gives the opportunity to identify the failed test with a colorful and clear GUI shown in Figure 2.3, which then gives the access to the effective traces collected from the test during execution, which we can see in Figure 2.4.

The view shown in Figure 2.4 is much more oriented to the work of the developer. It gives the access to the recorded streams related to the standard file descriptors (standard output and error) and, in case of failure, to the stack trace and the message contained in the exception that has been launched.
Undoubtedly, this kind of information could be available to the programmer also without the Jenkins platform, just running the tests from its terminal. This is,

New Failures	Chart	See children	Build Number ⇒ Package-Class-Testmethod names ⇓	214	213	212	211	210	209	208	142
			sail2	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
			sail3	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	N/A
			L3EcmpLagTest	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	N/A
			L3IPv4EcmpLpmTest	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
			L3IPv4HostTest	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
			L3IPv4LagTest	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
			L3IPv4LpmTest	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
			L3IPv4MacRewriteTest	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
			L3IPv6EcmpHostTest	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
			L3IPv6EcmpLpmTest	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A

Figure 2.3: Access to the reports of the single tests in Jenkins.

in my honest opinion, the real achievement of the platform: even if the testing is completely automated and seamlessly integrated with the commit of the code to a repository, the programmer gets anyway all the low-level information he could get with a manual run of the tests. So we allow workers to launch tests with a click and present a high-level view to all the people that need to understand only the advancement of the project, but notwithstanding this the programmer is still able to find all the debug details he needs.

Error Message

```
Lag path1 is not equally balanced
```

Stacktrace

```
Traceback (most recent call last):
  File "pkgsrc/switch/ptf-tests/base/sai-ocp-tests/sail3.py", line 1118, in runTest
    "Lag path1 is not equally balanced")
AssertionError: Lag path1 is not equally balanced
```

Standard Output

```
{'11': 'Veth23', '10': 'Veth21', '13': 'Veth27', '12': 'Veth25', '15': 'Veth31', '14': 'Veth29', '16': 'Veth33', '1':
'Veth3', '0': 'Veth1', '3': 'Veth7', '2': 'Veth5', '5': 'Veth11', '4': 'Veth9', '7': 'Veth15', '6': 'Veth13', '9':
'Veth19', '8': 'Veth17'}
[101, 50, 33, 74, 81, 161]
```

Standard Error

Figure 2.4: The view on the traces collected by Jenkins.

This turns the platform in a very powerful development tool not only for the single developer, but also for the team work. In fact, the single worker will find in the web interface all the informations that are important to check the interaction

of its porting code with the P4-enabled chip and the OS, but he will also be able to present more easily its results to managers and co-workers. Indeed everything is shared on the same web interface at several levels of detail, so that everyone is able to recover the information he/she needs.

2.6 Conclusion

We have explained, with this chapter, every choice that was taken for the functional testing of a single device, which is the objective of the first part of the thesis.
Reassuming the main concepts, we have seen that the nature of the testing that was needed by the project is very specific: we wanted a functional testing that had to be able to explore the problem of porting a switch OS on a P4-programmable hardware and evaluate the effectiveness of the interface between them. This necessity required the PTF framework, to inspect the behaviour of the different APIs used in the development and explore every case.
In addition to this, we wanted the target to be pursued in the context of a "relaxed" continuous integration, to completely include testing in the development chain. This brought me to the construction of the platform based on Jenkins, with the explained setup.

It is important to underline that this kind of testing, due to its functional nature, and the infrastructural work it required produce no measure other than the number of successes and failures among the tests or the elapsed time. The situation is different for the network testing that covers the rest of the thesis because, in that case, my work was also creating an environment in which a distributed algorithm is able to run at its best. In this sense that work will find the final result of every effort in the presentation of the performance of the testing platform, while for this chapter the focus was more on the correctness check and the automation.

However, we will come back on this functional testing in chapter 6, when the accomplished goals and final achievements will be summarized.

Chapter 3

The testing framework for the deduplication algorithm

3.1 Introduction to the framework

Measuring the performance and the traffic of Data Center networks is becoming fundamental these days, in order to optimize resources and avoid congestions and/or losses in high load periods. New networking techniques have given infinite possibilities of adapting the network to necessities, even when they are temporary and very changing.
Anyway, in order to exploit these opportunities we should know our networks and their needs and, therefore, monitoring becomes central. One of the most common solutions for this need, is using a Cisco Nexus Data Broker (NDB) network to implement the collection of traffic.

When using this approach, we can identify two networks: the monitored network, which I will refer to as *production network* and generates the traffic that is under analysis, and the *collection network*, that is the actual NDB network and must collect samples of the traffic that traverses the other one. In particular, the task of the NDB network is obtaining traces and packets from some key locations in the target network and direct them to a particular machine, that is usually called *collector*, where all these data will be analized in various ways.
The two networks are tied together by the technical solutions that are used to sample the operation of the production network. The usual alternatives are tap interfaces and SPAN ports that are connected to the outer-most switching devices in the NDB network and will be used to obtain the packets needed for the analysis. Even regarding this operation, when we are using SPAN ports, different solutions are available: packets can be sent to the NDB networtk with a proper encapsulation, to help the NDB network push the monitored traffic towards the collector or to create

real tunnels between each switch in the production network and the collector.

The problem in these monitoring techniques is that often packets can traverse more than one observation point, where we have installed our tap interfaces or SPAN ports. Therefore, since every time this happens a replica of the packet is produced and sent to the NDB network, several instances of the same packet will arrive at the collector.
This is, first of all, an important waste of resources, especially if we consider the considerable amount of big packets that flow on Data Center networks: every transmission on the NDB network, after the first one, is completely useless because the packet has already been sent to analysis. Moreover, for some particular measurements, duplicated packets can produce also misleading results because they can alter the dataset on which the different analyses are done.
In particular, we could have a polarization of the dataset around the packets that have longer paths on the network or, in general, that traverse more observation points. The worst thing about this is that this phenomenon depends on the active paths on the network, which are determined by many unpredictable aspects. That is the reason why finding a solution that could assure a deduplication of the sampled packets becomes a necessity in this kind of applications.

All the current solutions, are based on deduplication algorithms that run on the collector machine. In this case the waste of networking resources remains, since duplicates are still forwarded uselessly, but at least the analysis will not be affected by polarized distributions in the datasets. Anyway, we have to consider that this approach charges the collector with the entire load linked to this operation, which is quite expensive both in terms of computation and storage since it has to build a state to remember the packets it has already received.

In this context, the big novelty due to the introduction of P4-enabled switches is the possiblity of using those devices in the NDB network to implement a distributed deduplication algorithm on the switches, while packets flow towards the locations of analysis. Indeed, the ability to define custom processing and the match-action paradigm, inherited by P4 from SDN, can give an infinite freedom in handling this packets on the NDB network, under a lot of different perspectives (e.g. flow identification, custom encapsulations, etc.).

The advantages and disadvantages of deduplicating traffic in the NDB network, and not directly at the collector, are the usual ones of every distributed algorithm. On the one hand, the computational load is spread among different devices, becoming lighter for each of them, and we also avoid waste of bandwidth eliminating the duplicates as soon as possible, without useless transmissions. On the other hand, it is necessary to find a way to generate in some sense a distributed, even if not necessarily shared, forwarding state of the network on the switches.

This is what is done by the deduplication algorithm tested in this thesis. The algorithm is based on identifying packets as belonging to a particular flow through

the following 5-Tuple: IP source address, IP destination address, Layer 4 protocol, Layer 4 source port, Layer 4 destination port. Once the packet is recognized as part of a flow, the main idea in the algorithm is associating, with a rule, the flow to an ingress port on each NDB switch. This port is usually the first that brings the packets of the flow on a switch that has no rule associated to it: the deduplication is given by the fact that the switch will then forward to the collector the packets belonging to that flow only if they come from that specific port. Therefore, since behind the ports of NDB switches we have tap interfaces and SPAN ports, we will forward only the packets sampled by a single observarion point.

This is the scenario that the testing framework, which is the objective of this part of the thesis, had to reproduce to introduce our approach in the estimate of the performance of a distributed algorithm involving P4-enabled devices. I had to face the following challenges:

- building the production network as a good miniaturized representation of a Data Center network, both in the topology and protocols;
- generating on the production network traffic flows that could be realistically present in a trace coming from a Data Center;
- finding a way to evaluate the number of surviving duplicates arriving at the collector to measure the efficiency of the algorithm.

The biggest issue to face is that the emulated devices and technologies are so innovative that there are very few alternatives and tools to build such a framework. In the following sections of this chapter we will go through the different aspects that were introduced now, to fully understand the operation of the software and the reasons behind every design choice.

3.2 Choice of the emulation tool

The newness of the devices used in the deduplication algorithm and network brought this work in a field that is still not completely covered by the emulation and simulation tools today. That's why, before even trying to approach the problem we are facing in this part of the thesis, a careful observation of the possible alternatives is necessary.

The major simulators, available on the market at the moment, are mainly stuck at Openflow-based switching when we speak about SDN. Indeed P4-enabled switches are not allowed in these tools and, accordingly, the algorithms that need to define custom headers and actions can't be adapted to run on them. In addition to that, the SDE of the Tofino chip offers a behavioural model that can match only few specific tools.

These are the reasons why we decided to use for our measurements the *Mininet* emulator, which is the prevalent solution in SDN world. This fame is due to the fact that this tool can be easily combined with a lot of different controllers, potentially external to the emulator, and it comes by default with behavioural models that mimic all the main devices needed to build a network, like basic Unix hosts and Open vSwitches (OVS). The hosts will be able to run practically every executable or script that we are able to run on the usual console of a Unix machine, while OVS switches will assure us the possibility to build efficient L2 networks.
Anyway the biggest advantage of Mininet, which is also the one that drove our attention on the tool, is the ease of integration it offers through its Python API. Thanks to this API, custom topologies and tests can be easily created with a script and new nodes can be added to the environment just declaring a Python class that offers the proper methods. This was fundamental to us because it gave us the chance to integrate the emulator with the addition of a new P4-enabled switch, through the Python classes provided by Barefoot Networks SDE.

The big drawback of Mininet is that it is an emulator and therefore, unlike a simulator, when we have to forward packets it replicates the execution of the instructions that are run on real devices. This means that it is spawning a certain amount of processes that mimic the behaviour of real devices, executing their effective instructions. The forwarding delays will be consequently increased by the time needed by this emulated execution.
In a simulator, instead, devices' reactions are reproduced only in their effects. So, if we know that a particular device reacts to an event with a specific packet on a specific port and a specific delay, the simulator will reproduce the dispatch of this packet in the virtual time-line it has created, just creating a packet on the right port in the right virtual moment. In other words,

"Unlike a simulator, Mininet doesn't have a strong notion of virtual time; this means that timing measurements will be based on real time, and that faster-than-real-time results (e.g. 100 Gbps networks) cannot easily be emulated." [3]

This means that the measurement of time and network's speed cannot be precise enough to have a scientific value. Indeed the delays that every operation will experience will be incremented by the emulation of the hardware and by the fact that we are emulating a quite big number of devices, all fighting for the same limited resources.

Therefore, the only analysis that can be meaningful is linked to checking the functional operation of the algorithm and the P4-enabled switches.
During this composition, we will consequently focus on the number of duplicated packets and the number of useless bytes that are received at the collector, with the aim of underlining the efficiency of the filter implemented by In-NDB-network

deduplication.

3.3 The topology in the testing framework

The first thing to define to build an emulation environment is, obviously, the topology. As mentioned before, the type of installations that we take as a model are the ones we could find in a Data Center, since these are the environments where P4 and more in general SDN, have the widest application. Because of this, before explaining the topology that was deployed in our platform, it is important to define what are the characteristics to reproduce.

The networks that inspire our work are usually characterized by the presence of two subnetworks: the production and collection network. To prove the correctness of our decisions, it is necessary to delve deeper into their nature.

- **Production network**: it is the monitored network, the one used by the Data Center to offer its services and run its applications. Usually one of the most common choices for its implementation is creating a big L2-network with a *spine&leaf* topology, like the one shown in Figure 3.1.
  This topology has the servers of the DC directly connected to the *leaf* switches, which play the role of access switches. Then those switches are connected with every *spine* switch: this allows the access network to reach the rest of the network. In this sense the spines represent the backbone of the network.
  The fact that every leaf switch has a connection with all the spine switches, gives high redundancy and allows every server to reach the others with almost constant and predictable delay, since we always traverse at most two leaves and a spine.
  These gigantic L2-networks can't afford the handling of a broadcast packet or the waste of resources, so a certain amount of countermeasures are used to avoid that ARP/Neighbour Discovery messages generate a flood on the network. Usually, these networks implement also a lot of tricks used to reduce the loss of capacity given by cycle-elimination, like LAG links, multi-path protocols etc. This is done so that no link is cut and the load is spread evenly on the available resources.

- **Collection network**: it is the network that is used to get all the measurements and traces on the operation of the production network and collect it to the collector(s). The topology and the characteristics of the network, in this case, can vary a lot: we can have full meshes, tree or triangle topologies and even more than one collector. The packets sampled from the production network can traverse the collection network as they are or modified with a wide range of encapsulation or tagging alternatives, like MPLS, ER-SPAN or

VXLAN, that is the most popular today. This means that this network can be traversed, in practice, by a very big number of tunnels going from the sampling points to the collectors.
I will often refer to this network as *NDB network*.

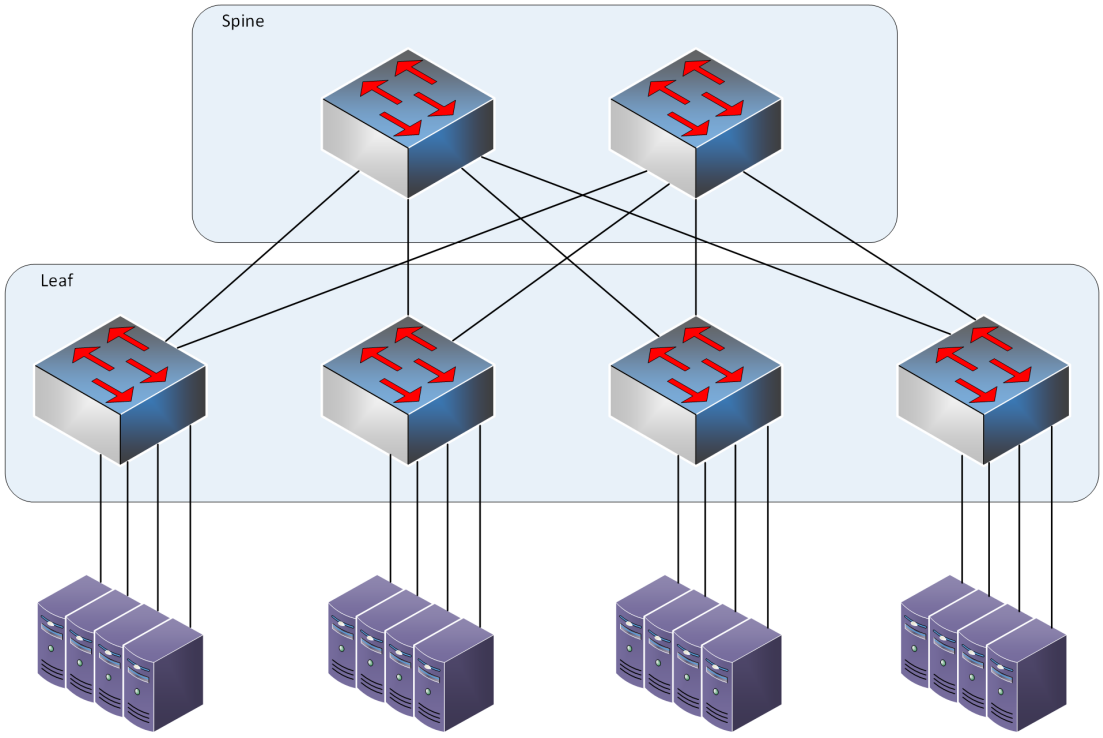


Figure 3.1: A classical spine&leaf topology.

The two networks are interconnected through the observation or sampling solutions that are used by the NDB network to get a copy of the packets that flow over the production network. The most common technologies for this task are SPAN ports and tap interfaces, which are installed and configured on links and switches of the observed network and then are linked to the NDB network where they forward the captured packets. Considering the properties mentioned above, the topology that was built for the emulation is the one depicted in Figure 3.2 and obtained through the code presented in Appendix B.

We can see, in the image, the production network in blue with a slight modification of the typical spine&leaf structure and the collection network, in red. The links added in the production network to the classical spine&leaf were useful to create a more general scenario: this was done to show that the algorithm works well for DCs, but also for other networks.
Every switch in the production network is the root of a small sub-network with 4

hosts connected to it. Those hosts will be used to generate the traffic that is inspected by the NDB network, so they play the role of the servers that would be connected to the production switches in a real Data Center. I decided to connect some hosts also to the two spine switches, even if in a real environment they wouldn't have any server connected to them but just leaf switches. Anyway, I wanted those hosts to mimic the traffic coming from other leaves that, for simplicity, are not emulated in the testing framework by real switches.
The device chosen for those switches, in order to create a unique L2 network, is the OVS switch: it allows us to have an L2 switch that is very used in real networks and even to configure easily a SPAN port to mirror all the traffic towards the collection network. The choice was also driven by the fact that the OVS switch is the device offered by Mininet as standard switch. This reduced the complexity of the emulation environment and, in some sense, made the network more realistic with the use of a traditional L2 device.

The connections with the NDB network are represented with the green dotted lines in the image: for the sake of clarity, they were not drawn as precisely as the other ones. Regarding the sampling techinque, my choice went on the SPAN ports, rather than on tap interfaces, because the former alternative is natively supported by the OVS switch and its configuration leaves space also to a lot of different encapsulation options.
Tap interfaces are instead not supported just as much by Mininet and, in addition to this, they were not bringing any significant advantage to our measurments that could motivate this bigger configuration effort. Indeed, the difference with SPAN ports is that they replicate the actual physical signal they get from the wire where they are installed or from the port they are coupled with. There is therefore no processing at layer 2 and therefore even malformed and error packets are forwarded more transparently.
This is what could make tap interfaces preferrable for network security applications but, for our measurments, this specific packets are not so interesting. For this reasons, tap interfaces were not used in the network.

If we move to the NDB network, it is clearly characterized by the presence of P4-enabled switches: those are the devices that will apply the decisions produced by deduplication algorithm.
The first two versions of the algorithm that were tested are runnning without the help of a real controller - even if the process resetting manually the registers of the first version could be considered a simple controller - because the algorithm uses only local information, which can be stored also on the switches. Indeed the previously cited association, between the flows and their incoming active port, is done by every NDB switch regardless of the decision of the others. Thanks to this, the information becomes switch-dependent and can be kept in some internal storage structures offered by the P4-language and Tofino chip. We can say that with these

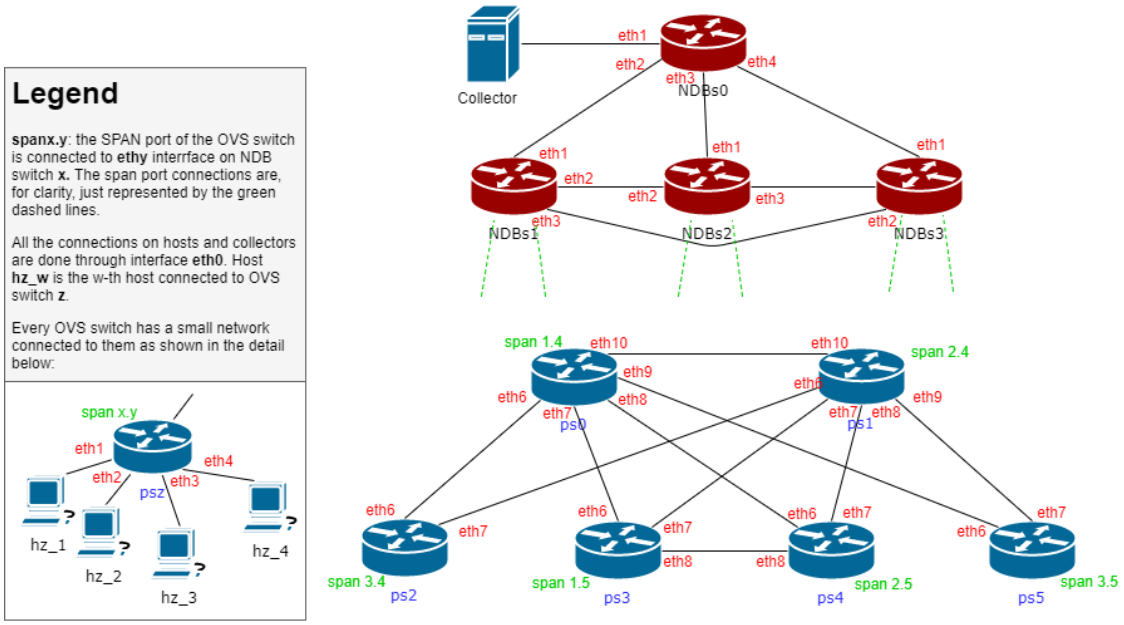


Figure 3.2: The topology implemented in the Mininet testing platform.

versions, the logic of the deduplication algorithm is still running over switches.
A third version, including the interaction with a controller to use the match-action table, was then tested. In this case the controller is, as in every other SDN application, the brain and the embodiement of the control plane in the network. As every other entity that is external to the algorithm, the controller is in truth part of the testing framework and was developed in this work using the utilities offered by Barefoot SDE to connect to the switch. That is the component that will effectively run the algorithm to take decisions and will be connected to each of the red switches in the figure, to set their forwarding state with rules, through their thrift server port. Every time a switch will receive a packet that is not matching any flow present in its rule-table, it will generate a digest containing its 5-Tuple and the ingress port. This digest will be recovered by the controller that is waiting on its connection with the switch for those events. Thanks to the information contained in the digest, the controller will be able to take a decision on how to handle the packet, following the algorithm.
The logic, illustrated in Appendix C, is very easy: if we have no rule for a flow, we just associate its 5-Tuple to the ingress port of the packet that generated the digest. There is no preference among the interfaces of the switch, consequently the first one that brings the packets of a flow to the device can be chosen as active port for it. Then, always through the thrift port, the controller sets the rule that will define the treatment of the following packets that are recognized as part of the same flow:

only the ones coming from the active port will be forwarded to the collector.

As said before in the discussion about the emulation tool, the models we are using in the NDB network are emulating a P4-enabled switch running on top of the Tofino chip provided by Barefoot Networks. The behavioural model used to reproduce the operation of such a switch was offered by Barefoot SDE itself, together with the definition of some classes and functions that build the interface with the Python API available in Mininet.

The last component that builds the network of the framework is the collector. In real applications it is the machine that executes the analysis over the packets that are sampled on the production network, here it is just attracting traffic since no analysis, other than recognizing duplicates, is done over the received traffic. There could be also more than one collector and, in general, we could have it also in other parts of the NDB network.

3.4 The traffic model

Another aspect that is very important to create a realistic environment in the framework, with the aim of increasing the scientific value of our measurements and results, is generating Data-Center-like traffic in our production network. This is required to prove that the algorithm and the P4 switches are able to handle a traffic that is similar to the one they could find in practice in the addressed network profile.

The first important requirement is finding a tool that can offer a wide range of possibilities to mimic the properties of the statistic processes that drive packet generation. In this case, the chosen tool is the Distributed Internet Traffic Generator (D-ITG [4]), developed at university Federico II of Naples. The reasons for this decision can be identified by the several alternatives that this tool offers in terms of the definition of the traffic characteristics and the fact it is a quite lightweight executable. Furthermore, thanks to the fact that Mininet hosts can run every program installed on the machine we are using, all we need to do to use it in emulation is running the executables of D-ITG on every couple of hosts that must share a connection/session (as we can see in the code in Appendix B).

Regarding the stochastic capabilities of the tool, I was very interested in the possibility of defining properly the processes that determine the inter-departure times (time between generation of two packets) and the size of packets. Indeed, one fundamental detail to consider is that our measurements over sampled traffic are focused on the granularity of packets, since the most important result of our test is showing the efficiency of the deduplication algorithm in eliminating duplicated packets. Therefore, I was interested in building a model that could represent precisely the distribution of the packet generation process in a Data Center, rather than focusing on the flow-level characteristics of the generated traffic or on the applications that

can produce it.

The definition of the traffic in Data Centers has been a central matter of study for a lot of papers recently. The ones produced by Microsoft Research ([5, 6]) can give a quite precise idea of how to obtain a realistic generation process. In these papers, empirical measurements were taken to analyse all the characteristics that Data Center networks' traffic has and, for this reason, they were taken as a reference for a lot of different tools and application designed for those networks. Even if those works analyse the traffic at every level, from the flow-level and packet-level patterns to the applications, we are mainly interested to the packet-level statistics, to understand which model can fit best the experimental findings.
The authors took into consideration the main types of DC that are in use all over the world today: university campus DC, enterprise DC and cloud computing DC. Their packet-level analysis was focused on the first two types, analysing the different applications that run on these infastructures nowadays, and proves to be very complete under this aspect.

A very interesting result comes out for the packet size distribution, that happens to have a bimodal distribution with peaks around 200 bytes and 1400 bytes. Quite similar results were found in [7], where the DC traffic was studied to understand the implications on the deployment of optical networks. This results show that the fluxes are mainly formed either by very small control packets or big packets that transfer chunks of the heavy files that are typical of Data Center applications. Specific protocols (HTTP, MSSQL, SMB) were suprisingly generating more small packets than big ones.
Considering our modeling objective, this distribution is not directly available in D-ITG but it can be built overlapping two Normal distributions with same standard deviation, one with the mean in 200 and the other one centered in 1400 bytes. Indeed a process that is generated in this way remains bimodal if the distance between the expected values of the two Normal variables is bigger than the double of their standard deviation. Since the peaks have a remarkable importance for the distribution, we can choose very small values for the standard deviation. In this way the majority of packets for the two variables will have a size that is very close to the expected value and, recalling the theorem that was cited about bimodality, we obtain a very good approximation of the distribution shown by the papers regarding the size of packets.

Again in [5, 6] we can also find important information about the inter-arrival time between packets on the switches. This measure can be used to define, eventually through rescaling, the distribution of inter-departure time of packets from the hosts, since we are not seeking exact precision but just a realistic distribution. In the papers comes out that the distribution has an ON/OFF behaviour: our main interest, to prove and test the algorithm in the worst case of load, will be reproducing the characteristics of ON periods. In those cases, a good approximation of the

experimental measurements is given by a Weibull distribution, whose parameters were initially interpolated to reproduce with our code the behaviour seen in the papers.
The Weibull distribution is a heavy-tailed distribution that represents a sort of variation of the exponential one, with the introduction of a time-dependent failure rate. The mathematical definition of its cumulative density function (cdf) is quite straightforward and defined by:

$$F_W(x) = \begin{cases} 1 - e^{-(\frac{x}{\lambda})^k} & x \geq 0 \\ 0 & x < 0 \end{cases} \quad \text{with} \quad k > 0, \lambda > 0, x \in \Re \tag{3.1}$$

Notwithstanding the easy expression of its cdf, the distribution is characterized by a great versatility, given by the two parameters that appear in the equation. In fact, the modification of shape parameter (k) and the scale parameter (λ) produces significant changes in the properties of F_W (e.g. for $k = 1$ the distribution describes in truth an exponential random variable).
This makes the distribution able to perfectly reproduce, for $k \neq 1$, the behaviour of a specific class of random processes: the processes with a time-dependent failure rate, where the probability that the modeled event happens in a fixed-width interval of time changes if we move this window on a timeline, i.e. if we shift the starting point of the interval. We are referring to processes that, in mathematical terms, have a memory, in opposition to the memory-less behaviour of the exponential distribution.
In truth, as we will see in chapter 4, the generation rate of the packets obtained from the papers had to be adapted in later experiments to the switching performance of the behavioural model used for the NDB switches. The throughput had to be reduced in order to avoid massive losses due to temporary congestions on the switches that would have compromised our measurements regarding duplicated and lost packets. For this reason I decided to keep the Weibull distribution, to be coherent with those results, but parameters were modified accordingly.

It is now possible to define the traffic generation process of the testing framework, which was inspired to all the scientific results cited until now. In this sense, the objective is generating flows of packets whose size follows the bimodal distribution described in the paragraph, with intervals of time between packets defined by a Weibull distribution adapted to the capacities of the available behavioural models. To do this, since D-ITG does not offer the bimodal distribution for the size of packets, each flow must be created as the combination of two sub-flows.
The idea is overlapping two sub-flows, with the same Weibull distribution for inter-departure times, that differ in the distribution of packet size. For each of them a normal distribution, with the mean respectively in 200 and 1400 bytes, will be used: as explained before, choosing the proper value for the standard deviation, this

assures the bimodality of the process and the desired characteristics. An example of how the traffic of each flow is generated, using D-ITG, is available at the end of Appendix B.

In conclusion, it is interesting to underline the differences with usual Internet traffic that are emphasized also in [5]:

- the size of packets is incredibly polarized towards very small or very large packet;
- for the inter-departure times we need positive skewed and heavy-tailed distributions, with the addition of the ON/OFF behaviour, that are in opposition to the long-tailed Pareto distributions that usually characterize Internet traffic.

All of this must be taken into account when designing models and networks for this particular applications.

3.5 The recognition of duplicates

We have explained that the duplicated packets received at the collector are our metric in the measurement of the efficiency of the In-NDB-network deduplication algorithm. The analysis of this performance is the purpose of this part of the thesis and the key element that must be defined, to achieve this objective, is how to recognize the duplicated packets as they arrive to the collector's interface.
The importance of defininig this process is due to the fact that this recognition cannot be done with the same methods used on switches by the deduplication algorithm.

In the algorithm, indeed, the principle that is used is the association of a flow of packets to the first port where it appears on the switch, when it has no rule for it yet. The flow is identified with the following 5-Tuple: IP source address, IP destination address, Layer 4 protocol, Layer 4 source port, Layer 4 destination port. The first packet of the flow that reaches the switch, when there is no rule nor information for it, brings the switch to a modification of its forwarding state that is supposed to enable the treatment of the following packets of the flow. The modification consists of introducing the previously mentioned flow-port association: from that moment on, every packet of that flux is expected to come from the port of that first packet, otherwise it is considered as a duplicate.
This principle is effective if we remember that the ports of the NDB switches are connected only to other NDB switches or to the sampling points in the production network. This means that, if we take one of the NDB switches directly connected to the production network, every port that receives a packet identifies an observation point in the analysed network. It is obvious, then, to observe that packets belonging

to the same flow and coming from two different ports will be for sure duplicates generated by the path followed by our traffic: they are the same packets, just observed from two different sampling locations on the network.
If we move to internal NDB switches, every port is instead associated to other NDB switches. In that case, packets in the same flow coming from two ports will be the packets sampled on two observation points connected to two different external NDB switch, so they will be duplicates as well. This shows why the idea of forwarding on every switch only the packets coming from a single port for each flow will assure to keep always one and only one copy of them.

The big problem of measuring duplicates from the perspective of the collector, to count how many errors are potentially done by the algorithm, is that this machine is receiving all the packets from a single interface. Accordingly, the policy of the flow-port association is not applicable any more and, in general, we cannot count any more on any metadata information related to the ingress interface or process. We have to rely exclusively on the information in the packet and, therefore, on the standard protocols, in order to add an other field to the cited 5-Tuple and make duplicates identification possible.

Indeed finding equal values in the 5-Tuple is not enough to say that two packets are duplicates, because all the packets going in the same direction in a TCP connection or UDP session will have the same values for those fields. The 5-Tuple permits only to recognize the membership of a packet in the flow: we need to build something bigger, adding some fields, whose repetition on two or more packets can be used as proof of duplication.

Considering this issue, the first idea that could come to our mind is using some sequence number, like the TCP sequence and acknowledgement number or a field added to the UDP payload. This choice could be useful to differentiate the pieces of information transmitted with packets of the same flow, but the repetition of the sequence number would make us recognize as duplicate also re-transmitted packets. In fact the sequence numbers in those cases wouldn't change and this would be an error case for the algorithm, both in the context of TCP reliable communication or an application protocol built on UDP.
The same argument deters from the use of the checksums in Layer 4 headers, since they cover that header and the payload of the packets that will be exactly the same on re-transmitted packets.

The other possibility that upper layers could offer is using transmission timestamps: TCP has one, and we could insert a timestamp also in the application layer running over UDP, as a lot of UDP protocols already do. This alternative could work since for re-transmissions these particular fields would change, therefore two packets with same values for the 5-Tuple and this field would be for sure duplicates. Anyway in this case the drawback is the fact that this approach is not unique for the two layer 4 protocols and, for UDP, we are also inserting a requirement for the

application layer. The addition of this dependency could reduce the applicability of the model and the use cases for our framework, so I decided to avoid also this solution.

The same consideration can be done if we add to the application layer a counter that is incremented at each transmission. In this case, this number wouldn't represent the position of the information transmitted with the packet in the session (as a sequence number does), it would just be a counter of the packets sent during the communication until that moment. If intended in this way, this could be another effective solution but we are adding again a dependency on the application layer.
The idea is anyway very good and it's worth trying to use something similar: if we delve a little bit deeper in network protocols, we can find something with an analogous behaviour in the IP header. The IPv4 header has indeed a field called IP Identification (IP ID) field in the part of the header that is reserved for the fragmentation information. This field is incremented each time a packet is sent, even on re-tranmissions, in order to respect the following rules, defined in the RFC 6864:

"» The IPv4 ID of non-atomic datagrams MUST NOT be reused when sending a copy of an earlier non-atomic datagram.
» Sources emitting non-atomic datagrams MUST NOT repeat IPv4 ID values within one MDL - Maximum Datagram Lifetime - *for a given source address/destination address/protocol tuple."* [8]

Since that part of the IP header is not modified during forwarding and re-tranmissions will not be mistakenly recognized as duplicates, the repetition of this field and the 5-Tuple can be used as a good proof of duplication.
Moreover adding this field to the 5-Tuple in this research, so that we form a sort of 6-Tuple, is the best solution we could find, since it allows us to have correct measurements with a complete, layer4-independent and self-sufficient solution.
The implementation of this solution is extremely simple and is represented in the following code:

Listing 3.1: "Duplicates recognition"

```
tshark -Y "ip and (udp or tcp)" -T fields -e ip.id -e ip.src -e ip.
    dst -e ip.proto -e udp.srcport -e udp.dstport -e tcp.srcport -e
    tcp.dstport -e frame.len -r input.pcap |
     sort |
     uniq -c |
     sort -n -r |
     awk '$1>1 {print $0}' > duplicated
```

In practice what is done is a simple extraction of the fields that build the 6-Tuple we have just defined, using the *tshark* tool. The code extracts the source and destination port both for UDP and TCP because for each packet only two of those fields will

have a value, while the others will be blank. We can say that the couple that is present and the blank positions in the output could represent alone, without the use of the IP protocol field, the information about the layer 4 protocol for the packet.
Once this is done, every packet, represented by its 6-Tuple, is reduced to a line in the output of the command. If, as we have just explained, equal values of the 6-Tuple represent a proof of duplication, all it is left to do is isolating the repeated rows in the text produced by tshark to identify duplicates. This is what the rest of the commands in the pipe do. In this case, the *uniq* command is used with its counting option to keep track of how many replicas are recognized and the *awk* command selects the packets with a count bigger than one.
From the code arises that the use of the IP ID field gives us a universal solution that works perfectly both with TCP and UDP traffic, and transitively with every application running on them. All of this without requiring any modification or ad-hoc behaviour to upper-layers.

Furthermore this can also be easily extended to IPv6 since the fragmentation information is still part of the protocol, even if it has been moved to the Fragment Extension Header. This is one of the optional headers whose processing is forbidden on routing devices and allowed only on end-hosts. This is not a problem at all because we are using it only for the recognition of duplicates on the collector, which is in truth an end-host, while the deduplication algorithm running on the P4 switches completely ignores this field, as it is necessary to respect the IP procotol.
In this work we will not address directly IPv6 traffic in the code, and it will not be used for tests. This choice was due to the fact that, in the end, supporting IPv6 wouldn't have added any value to the built platform, since none of the innovative features of IPv6 is used in the algorithm or testing. Notwithstanding this, I want to underline that, personally, I have had great attention in proposing IPv6-ready solutions, since I believe that this is an inescapable duty for everyone today to help the transition process as much as we can. In this sense, to support IPv6 traffic, the only thing to be changed in the duplicates recognition code is extracting the IP ID field from the optional header instead of the IP header.

Taking everything into consideration, it is easy to conclude that the creation of the described 6-Tuple, including the original 5-Tuple of the flow and the IP Identification field, gives the most complete solution in terms of correctness of duplicates recognition and versatility in the different contexts that networks can face. That's why the repetition of the 6-Tuple on different packets has been chosen as recognition method in the scripts that are responsible for producing test results, which I present in chapter 4.

3.6 The different versions of the algorithm

All the discussion presented in the chapter, until now, regards issues related in some sense to the algorithm itself and its concepts. Once the main theoretical details of our inspection have been explained, it is important to tell something more about the alternatives offered by the P4 language and the chip for the implementation of the algorithm.
We have seen that the key idea of the algorithm is associating the flows, sampled on the production network, to a port on the switches that execute the deduplication. The various versions of the algorithm that are tested in this work differ in the methods used to store, treat and renew this information.

The first big difference is given by the storage resource used to keep this association, since the hardware offers two possibilities: registers and match-action tables. Registers are simply stateful locations of memory that can be accessed for reads and writes in the custom actions defined with the P4 language. In practice they are very similar to arrays of integers and give a good freedom, with respect to simple counters, in terms of the operations a programmer can execute on their elements. The number of elements is fixed and must be specified in the moment of the declaration of the register.
The match-action tables are instead very similar in their use to the ones available in Openflow. They contain rules that connect a set of matching fields to an action. The difference with OpenFlow lays in the way this is implemented and in the fact that the set of fields, the type of matching and the actions taken on the packets can be completely customized.

The first two versions of the algorithm that are tested in the thesis use a register to save the active port of each flow. In particular, when the switch must process a packet, the 5-Tuple that identifies its flow is extracted and used as input for the CRC16 hash function. The output of the hash determines the index of the cell of the register to use for that flow.
This means that, every time the active port of a new flow must be saved or the one of a known flow must be checked to decide how to treat a packet, the hash is computed to know where to access the register. Therefore its dimension is defined by the size of the output of the hash and, obviously, if a collision occurs two or more flows will end up using the same cell. If the flows do not have the same active port this causes losses for some of them. The handling of this case is the first difference between the two register-based versions. The first one is not implementing any countermeasure to avoid the effects of collision, because of the rarity of the event. The second one is instead using three-bloom filters to handle the collisions and avoid losses due to them.

Moreover, the information about the active port must be updated regularly, to adapt to possible topology changes in the sampled network. The first version of the

algorithm, the one without collision-handling, is requiring also a manual reset of the register that stores the active port information, while the other one implements an automatic reset using timestamps to recognize stale information.

These two versions are running without a real controller, even if the process resetting manually the registers of the first version could be thought as a extremely basic controller. The algorithm indeed does not need the switches to share any information, since the associations between ports and flows are done independently from other devices. Each switch will therefore run all the logic of deduplication and forwarding, including the recognition of known flows and the initialization of the active port for new flows.

The table-based version is instead offloading part of these operations to the controller and brings the solution much closer to the ideas of the SDN paradigm. In this version, when a switch processes a packet, it is responsible only for the identification of the flow and, if its forwarding information is already installed on the switch, the application of the proper rule in the table to treat the packet accordingly. If the flow is being handled for the first time the switch creates a digest, carrying the value of the 5-Tuple and the ingress port of the packet, that is then pulled by the controller, which is running as a generic process with a connection to the thrift port of the switches.
As soon as it recovers the digest, the controller creates a rule using the information contained in it. The rule associates the 5-Tuple of the flow to the ingress port read from the digest and it is installed, using its CLI, in the match-action table of the switch that produced the digest.
To avoid the generation of an excessive delay in installing the rules, due to the latency in the communication with the controller, the algorithm is running the register-based version without collision control while no information for the flow is present. Then, as soon as the proper rule in installed, the register cell is reset and not used any longer.

With respect to the previous versions, the last one is surely the most sophisticated. Indeed it uses tables to store the forwarding state of the switch and presents an important interaction with a controller. Additionally, the operation of the controller is not requiring any hash computation, since we don't have to dimension in advance any structure to treat flows. Rules are installed as soon as new flows are sampled and no information must be kept on the controller: the only memory limit is the dimension of the tables on the switches.
Consequently, if no hash is computed, flows can't generate collision and we avoid all the additional handling and errors linked to that. For this reasons, and because it gives to our algorithm the shape of a real SDN distributed algorithm, it can be considered the definitive and most complete version of the algorithm.

3.6.1 The effects of hash collisions

As briefly anticipated before, specific conditions can bring, during the register-based operation of the algorithm, to collisions in the computation of the hash of flows' 5-Tuple. In order to understand what can be the impact of these collisions on the performance of the system, it is interesting to estimate the amount of erroneously dropped packets they cause depending on the parameters of our simulation. To do this, I define:

- n: the dimension, in bits, of the output of the hash function;
- k: the number of flows concurrently active on the production network;
- π: the number of ports of each NDB switch that are eligible to be active port for some flow, i.e. the ones that are not used to reach a collector;
- S: the number of packets totally generated and sampled on the production network.

If understanding how many packets are lost because of a collision is our goal, we need to find a way to connect this to the probability of a collision. If we consider each packet treated indipendently of all the others, we can see the treatment of a packet as a Bernoulli trial. In this case we will have that the number of lost packets will be distributed as the sum of Bernoulli(p) random variables, with p defined as the probability that we lose a packet due to a collision.
So our effort now moves on the search of a possible expression for p. To do that, I take the perspective of a packet because p is the probability that I, as a packet, reach the switch and get dropped because of a collision.

The value of p is defined for a packet by two events:

- E_1: one or more of the other $k-1$ flows generate a collision with the flow of the packet under analysis;
- E_2: the colliding flows install on the shared register cell an active port that is not the port of arrival of the packet we are considering.

Regarding E_1, I decided to consider all the $k-1$ flows sampled from the production network because some of the NDB switches must deal with the entire set of 5-Tuples characterizing our traffic. This is the case of the ones connected to a collector or an observation point that is particularly central in the forwarding process of the production network (e.g. the SPAN port installed on a root bridge).
Furthermore, the packet can traverse many NDB switches before reaching the interface of the collector: on each of them, during processing, it will find a different set of potentially colliding flows. Therefore, on its way to the collector, the packet

can eventually meet and cross the entire set of flows of the production network. For these reasons, I decided to define E_1 taking into account the likelihood of a collision with all the possible flows of the production network, to have a discussion as exhaustive as possible.

If we assume we are using a good hash, which has a uniform distribution of the inputs in its output space, the probability that the hash of a flow is the same we compute on the packet is $\frac{1}{2^n}$. In this sense, every flow represents an independent trial where the "success" is represented by the collision. Therefore the number of flows colliding with the packet can be modeled by $F \sim \text{Binomial}(k-1, \frac{1}{2^n})$.
However if we use a good hash function, since the likelihood of a collision with another flow is already extremely low, the probability that F takes a value bigger than one is several orders of magnitude smaller than $P(F = 1)$. For this reason, we can consider $P(F > 1)$ negligible with respect to $P(F = 1)$ and approximate $P(E_1)$ as:

$$P(E_1) = P(F = 1) = (k-1) \cdot \frac{1}{2^n} \cdot \left(\frac{2^n - 1}{2^n}\right)^{k-2} = \frac{k-1}{2^n} \cdot \left(\frac{2^n - 1}{2^n}\right)^{k-2} \tag{3.2}$$

Considering the semplification we have done for E_1, E_2 becomes the analysis of the case in which the colliding flow has a different active port and manages to install it on the shared register location. This event involves many different concurrent phenomena that are too complex to be modeled, especially in terms of their mutual interaction. Indeed the set of eligible ports depends on the path of the flow and consequently on the destination address and a lot of other unpredictable processes. Then timing conditions define which of them becomes active, since the first port to bring packets of the flow will cover this role.
Because of the impossibility to render all of this properly, I will assume that the two flows choose their respective active port with a uniform distribution from the total of the π available ports on the switch. This is coherent with a scenario in which both the capture points and flows are uniformly distributed across the production network. In that case, the probabilty that the two flows end up having different active ports is equal to the probability that the colliding flow chooses one of the $\pi - 1$ ports that are not active for the considered packet:

$$P(\text{different port}) = \frac{\pi - 1}{\pi} \tag{3.3}$$

Additionally, in the operation of the algorithm the flow that will define the value remaining in the register is simply the latest one to arrive to the switch. Because of the lack of the details needed to do a proper approximation of this race conditions, we can assume that each of the flows has half of the probabilty of installing its active port in the register and this is independent on the events that define P(different

port). This means, in the end, that:

$$P(E_2) = \frac{1}{2} \cdot P(\text{different port}) = \frac{\pi - 1}{2\pi} \tag{3.4}$$

We can now finally give a definition of the initial p we were looking for regarding the single packet. To do this, I consider that E_1 and E_2 are so slightly correlated that can be modeled as independent. Indeed their correlation is linked to the fact that the destination IP address of the flow influences both the events. Anyway, for E_2 the destination IP address is just one of the many more phenomena that are involved in the choice of the flow path on the production network and, consequently, of the active port. This, together with the difficulty in modeling all these random process both alone and in their interaction, makes the correlation between the two events negligible and too complex to be taken into account. Additionally, a good hash function should make the probability that two flows with the same destination address hash to the same value very small. Therefore we can approximate p as:

$$p = P(E_1) \cdot P(E_2) = \frac{k-1}{2^n} \cdot \left(\frac{2^n - 1}{2^n}\right)^{k-2} \cdot \frac{\pi - 1}{2\pi} = \frac{(k-1)(\pi - 1)}{2^{n+1}\pi} \cdot \left(\frac{2^n - 1}{2^n}\right)^{k-2} \tag{3.5}$$

and the number of packets lost because of collisions as:

$$L = \sum_{i=1}^{S} X_i \qquad \text{with} \quad X_i \sim Bernoulli(p) \tag{3.6}$$

The most interesting metric about this random process is obviously its expected value, that gives us a prediction of the losses we can suffer from. It is defined by:

$$E[L] = S \cdot E[X_i] = S \cdot p = S \cdot \frac{(k-1)(\pi - 1)}{2^{n+1}\pi} \cdot \left(\frac{2^n - 1}{2^n}\right)^{k-2} \tag{3.7}$$

The predictions obtained from this expression are shown in Figures 3.3. We display the relative expected losses $\left(\frac{E[L]}{S}\right)$, depending on the variation of the number of flows k and, respectively, of the bits in the output of the hash n.
In both cases, the plots show different combinations also for the value of non-varying parameters. The ranges and values for every variable in Equation 3.7 were chosen, for the creation of these scenarios, as follows:

- n: I consider n starting from the hash output dimension effectively used in the algorithm, which is 16 bits, and I increase it until the difference of performance is sensible;
- k: I referred again to the empirical results of [5], where the number of flows characterizing the operation of an enterprise DC is estimated between 1000 and 5000 flows for the 90% of the time. Therefore I decided to use values from 3000 to 5000;

- π: the number of ports eligible to become active ports for the various flows was chosen looking at the specifications of real switches on the market. So I decided to consider alternatively 256 and 512 total ports on NDB switches, in order to assume the use of relatively accessible devices.

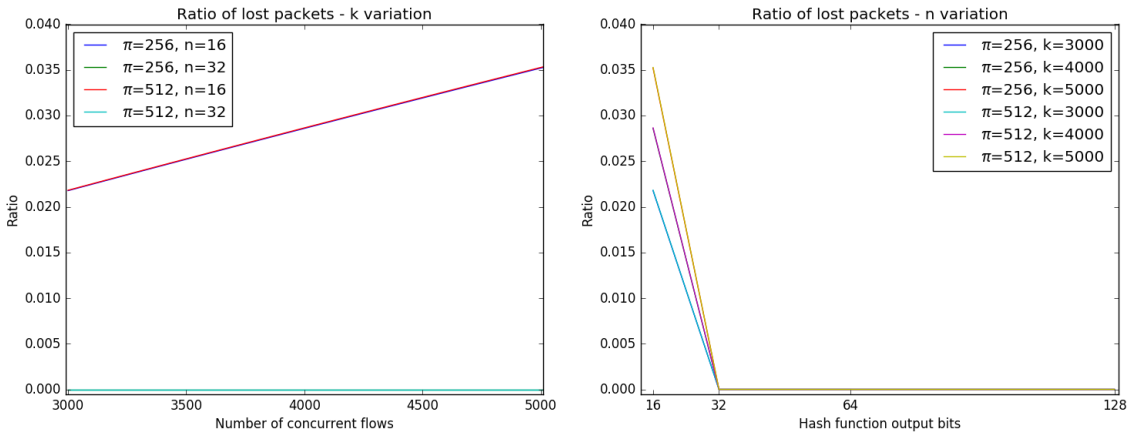


Figure 3.3: Relative expected losses due to collisions.

The first thing to notice in the images is that we always see half of the lines announced by the legend. The computation that was done for the plots, reveals that his happens because the number of eligible ports π is not affecting in our model the performance of the algorithm, or at least it is not as important as other parameters. The lines regarding simulations that differ only in the value of π are practically indistinguishable not only for the chosen values of this variable, but also for other realistic alternatives.

Furthermore, the dimension of the hash proves to have a much larger influence than the number of concurrent flows. It is indeed quite evident that, every time we increase the dimension of the output of the hash, the expected number of collision-related losses decreases by several orders of magnitude, while the change of k gives no significant effect for $n > 16$ and a light linear variation for $n = 16$. In other words, if the output space of the hash function is properly chosen, the number of active flow treated on the NDB switches has an intensely minor effect on the losses we predict.

Generally speaking, the percentage of the information that gets lost on the collection network due to a collision, is predicted under the 4% with whatever combination of parameters. This can explain the choice made for the first version of the algorithm, where no explicit countermeasure was taken to handle collisions and the consequent errors. The event is quite rare, therefore in some scenarios it is acceptable to sacrifice perfect correctness to simplify the code, especially in the context of a preliminary

version.
This was the last relevant aspect to be explained under the theoretical perspective, so we can finally move to the analysis of the realized measurements and their collection. At the end of the performance analysis of the first version of the algorithm, which is the only one that experiences the problem of collisions, it will be interesting to understand if this probabilistic reasonment is realistic enough to be confirmed by the emulation results.

Chapter 4

NDB Emulation and results

4.1 Testing Setup

The previous chapter explained thoroughly all the theoretical reasons behind the design choices that were made for the testing framework and all the possible scenarios for the emulation. This chapter is instead dedicated to the explanation of the tests and measurements that were done: we put in practice all the ideas previously exposed, explaining all the compromises that were needed.

The first thing to point out is the type of traffic I decided to generate on the production network. As we said in chapter 3, our solution for the recognition of duplicates is quite universal, it does not put any limit on this aspect: we just need something running over IP.
Notwithstanding this, our effort is modeling the traffic as realistically as possible, controlling even the packet inter-generation time through D-ITG. In this sense the way TCP sends informations on the network, its congestion control and its transmission window can generate additional delays that change the chosen distribution depending on the conditions of the network. This is the reason why I chose to use UDP in the production network, in order to avoid all this uncertainty and keep the generated traffic as close as possible to the desired model.

Moreover, when we talk about a network testing framework, it is imperative to be able to reproduce some topology changes during the emulation. Indeed, regardless of what kind of application we are testing, a network algorithm or protocol must prove to be able to react to possible route and path changes caused by inevitable failures and disruptions of network devices.
To fulfill this necessity, the framework launches, right after the generation of traffic begins, a subprocess which is reponsible of generating random topology changes in the sampled network. In particular, the process has an array that points to different bash scripts, each one installing a different route configuration on the production

network switches. A cycle then chooses randomly one of these configurations after a random time, defined by a uniform random variable whose value remains between 0.5 and 1.5 seconds.
All of this is done to test the capability of the deduplication algorithm to adapt to the changes in the network that generates the analysed traffic. It will be important to see how the cyclic reset of the structures, that enables this adaptation, interacts with the modification of paths in the production network.

Coming back on the traffic model, we have to register a compromise that had to be found with the performance of the behavioural model provided by Barefoot SDE. Even if I was able to reproduce the bimodal distribution reported in [5, 6] for the size of packets (as shown by the code in Appendix B), the same can't be said about the Weibull inter-departure time distribution.
Indeed, when I used for it the parameters obtained from the interpolation of the results of the cited papers, the emulation showed too many losses on the NDB switches. Logs and captures showed that the losses were due to a congestion of those devices, revealing that the behavioural models at our disposal couldn't support that throughput.
This forced me to re-adapt the approximation of the inter-departure times, whose cumulative distribution function had to be shifted to favour higher values for the intervals between two transmissions. Indeed, if too many packets are lost, the value of measurements over duplicates is reduced: it would not be assured that the missing duplicated version of a packet has been discarded by the algorithm, since it could also have been lost in switches' buffers in the collection network.

The application of all the reasonments explained in chapter 3 and the setup adaptations just exposed brought to the test scenarios illustrated in the next paragraphs. In particular, for the first version of the algorithm, which was register-based with a manual reset of registers, two sets of tests came out of the taken decisions:

- in the first scenario three fluxes with fixed sources and destinations were created, on a machine with 4 cores and a 16 GB RAM. The measurements were done both with and without the path change in order to analyse its effect. The idea is reducing the sources of variability to study the difference made by this route change;

- with the use of a more powerful machine (8 cores and 48GB RAM), the second set of tests aimed at creating the most realistic possible scenario. The throughput of the single fluxes was reduced in order to increase the number of contemporary sessions, which were randomized in terms of the involved hosts. The number of sessions was increased to forty and, again, the test was repeated with and without route change.

The limited variation of parameters is justified by the fact that this was almost a

preparatory and introductory version of the alrgorithm. It was meant to be a first attempt that had to test the correctness of the basic concepts of the distributed deduplication and show the implementation problems to be solved with following versions.

For this reason, the analysis of the second version of the algorithm, which includes the automatic reset of registers and represents a more complete version, is characterized by a wider oscillation of all the variables defining the emulated traffic and conditions. To be precise the number of flows and the registers' refresh interval were varied to study the reactions of the algorithm and its performance to them.
As we will see later on, the biggest issue identified by the analysis of the results given by the first version is the high amount of congestion-related losses. The problem about these losses is that they are not due to the algorithm itself, they are the effect of the mismatch between the performances the algorithm would require to the emulator and the limited resources at its disposal. Therefore, in order to have an analysis that is completely focused on the algorithm, in the tests that focus on the lost packets the amount of generated traffic was reduced until the risk of transient periods of high load for the switches was null.
The machine used for this tests is the same that ran the second set of tests for the first version of the algorithm and, since its effect is already revealed and studied by the previous inspection, the path changing script was always active during these measurements.

Finally, the table-based version of the algorithm that interacts with a controller has been tested with the same methodologies used for the one with the autoreset. In order to do a coherent comparison of the different results, including both the measure of duplicates and losses in the reasonment, the tests on this third version were done again with the traffic model that avoids the generation of transient congestions.

All these adjustements were made with the aim of finding a configuration which could assure, on the one hand, a realistic and exhaustive emulation and, on the other hand, some relevant results. In this sense, the idea was trying to obtain the highest possible throughput within the limits of a reasonable amount of lost packets due to congestion to preserve the quality of our measurements.
In other words, it was necessary to implement those compromises to have some valuable results, even if the emulation could result less fitting with respect to real use cases.

4.2 Collecting the results

I will try now to present what I consider to be the relevant results in this kind of emulation, in order to explain what has driven the choice of the metrics in this

work. As we have seen, the testing framework I have built tries to measure the performance of a distributed deduplication algorithm. The algorithm is based on the idea of identifying and eliminating the duplicates during their forwarding on the collection network, instead of filtering traffic at the collector.

The purpose of the algorithm and its nature suggest the first metric, which is obviously the number of duplicates that are erroneously received by the collector. This information can be assumed as the indicator of the effectiveness of the algorithm in the elimination of replicas from the traffic sampled on the production network.
In particular, what I decided to underline, together with the simple number of duplicated packets, was the relationship with the total number of packets that reached the collector. Indeed these packets represent, in some sense, a sort of measure of the error of the algorithm, but they are also the measure of how many packets are forwarded uselessly to the collector. For this reason, it was important to compare the number of duplicated packets and their dimension in bytes with the total number of packets and bytes received by the collector, including duplicates. This gave an idea of the percentage of the total traffic reaching collector's interface that is due to duplicates.

The measurement was done through a network capture executed on the collector. With the *tshark* tool, the udp traffic was filtered and the 6-Tuple we have defined in the previous chapter was extracted. Then the duplicates were extracted as explained in section 3.5 and the number of packets, with the sum of their dimension, was computed with *bash* utilities and a simple *awk* script.
The same commands were used to produce the same statistics on the entire capture, to give the term of comparison regarding the total traffic that touched the collector's interface.

Anyway, this result is just estimating the efficiency of the algorithm in terms of absence of duplicates: the other important thing to sense is the correctness of the deduplication. It is fundamental, to state the relevance of the results of the framework, to check that the filtering operation executed on the switches is effectively discarding only duplicates.
This idea is connected with the final objectives of every network on the planet: networks bring information from one point to one or many other points. Therefore the correctness of the filtering operation is defined by the percentage of the information, originally generated on the production network, that is able to reach the collector with or without duplication. It is a measure of how many packets are lost on the collection network, which is important to assure that all the statistics and the analysis, executed in real applications on the collector, work on a representative sample of the targeted traffic.

To have a sense of this correctness, the idea is comparing the number of packets received at the collector, with their size, to the ones generated by the hosts on the production network and sampled by the SPAN ports. Regarding the collector, in

this case we are interested in extracting the information about the fact that a packet has reached its interface or not: we don't care if it was duplicated. We can get this metric from previous results, as I did, or use almost the same code as before. This time the *uniq* command should be used without options and should be the last one in the extraction pipe, so that all the received packets could be extracted in a file without repetitions.
Obtaining instead all the packets sampled by the SPAN ports was slightly more complicated: the first thing to do was capturing on all the interfaces connected to the SPAN ports. Every capture was then manipulated as usual with tshark to reduce every packet to its 6-Tuple representation, which was then saved in a file. Finally, all these files were concatenated and treated as we just explained for collector's traffic, to obtain every sampled packet with no replicas. Then, the same scripts as before can be used to produce the actual numbers from the groups of 6-Tuples obtained from the collector and the SPAN ports respectively.

No other metrics could be functional to the study of the performance of the deduplication algorithm, because of the explained issues with virtual time. Notwithstanding this, what we have said so far gives us one more reason to choose Mininet as the emulator for our testing framework. It is indeed quite clear that the capability to run every bash command on the hosts and switches gave an incredible versatility in the definition and computation of whatever statistic.
In our case, this was useful to define the right metric to represent not only the efficiency of the filter, but also the way to affirm the scientific value of our results.

4.3 Register-based implementation with manual reset

I will now show and comment the results obtained with the described testing framework. The following paraghraphs summarize the measurements done in each scenario with three plots, representing respectively:

- the absolute number of duplicates received by the collector (above on the left in figures);
- the ratio between duplicates and total packets received by the collector, in terms of number of packets and cumulative dimension (above on the right in figures);
- the ratio between the amount of information arrived at the collector and the one sampled from the production network, in terms of number of packets and cumulative dimension (at the bottom of figures).

Test results will be commented at the end of the chapter.

4.3.1 Three fixed flows register-based scenario

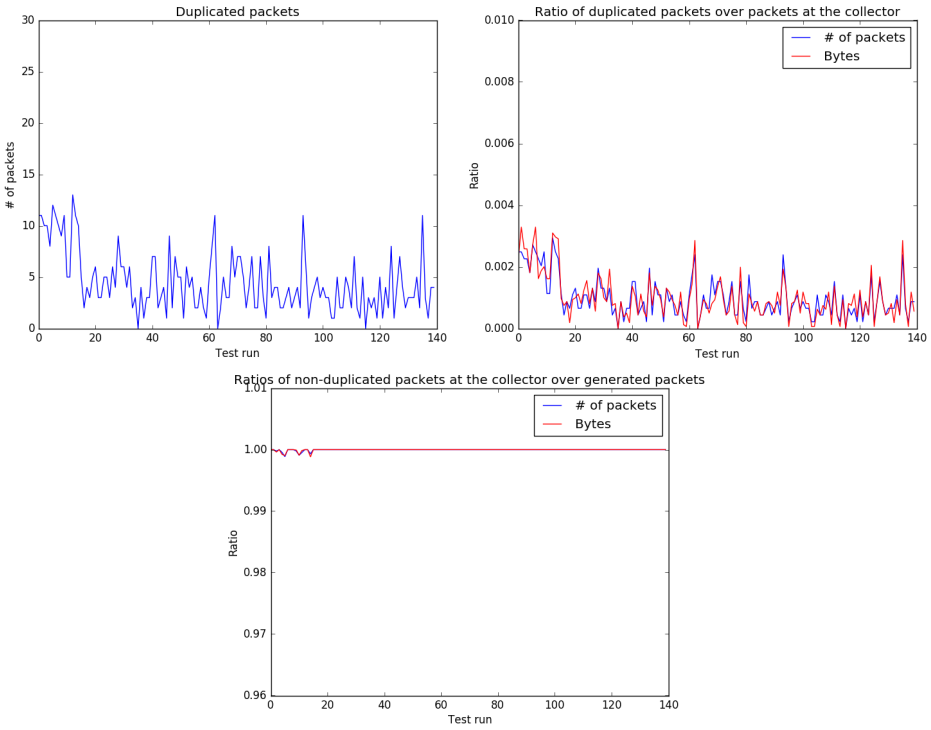


Figure 4.1: Results without route changes.

Figure 4.1 shows the statistics collected in the case without route changes and a total generated throughput around 3Mbps. It is evident how the number of duplicates is always under twenty packets, representing a negligible portion of the packets received by the collector. The reason of the presence of duplicates even in this scenario, which is the most stable one, is linked to the reset of the internal structures of the switch. This reset restores to a neutral value the data structures that are used to identify flows and their active ports. This neutral value is modifiable only after the end of the entire reset operation, so it can happen that if a packet comes in the middle of the reset operation the switch has the information to recognize its flow but the corresponding active port is set to the neutral value. In this case the decision taken is to forward the packet towards the collector, accepting the presence of some duplicates instead of the losses generated by the opposite decision, which could compromise the value of measurements. Moreover, the third plot on Figure 4.1 shows that no packet is lost in this scenario, assuring the absolute quality of duplicate count and demonstrating the effectiveness of the algorithm in the elimination of duplicates. This gives us the first answer we were searching for.

We can now analyse the situation of a slightly more realistic scenario, where the

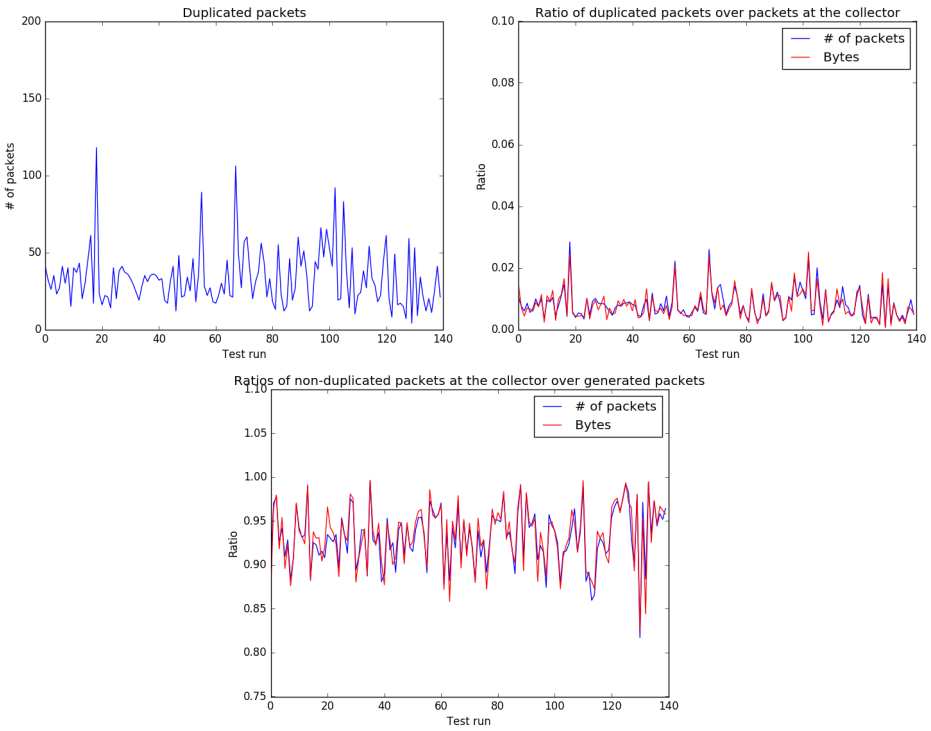


Figure 4.2: Results with route changes.

routes installed on the switches in the production network are cyclically changing. The plots in Figure 4.2 show that the number of duplicates is increased. A possible explanation for this increase is that the different topologies, which randomly follow one another, can bring the traffic fluxes to pass through a higher number of switches. This will create more replicas for each packet, which could end up arriving in the middle of a reset of the internal structures and therefore create more duplicates. However, we are still speaking about a small percentage of the total traffic received at the collector, since we are under 3% in terms of number of packets and bytes.
Much more interesting is the information given by the third plot, where we see a decrease in the portion of the total generated traffic which is able to reach the collector. This phenomenon is easily explicable considering a particular situation that can happen during the operation of the deduplication algorithm. Let me assume that an NDB switch has chosen port y as active port for a flow x. We know that the choice of a port on an NDB switch represents the choice of an observation point on the production network, and therefore of a production switch. If during the period in which y is active for x a topology change cuts out of the path the switch that produces the copy of the flow coming from y, the packets of that flow will now arrive only from the other SPAN ports of the network. These will be anyway inactive ports,

therefore packets will be discarded until the next reset of internal structures with the consequential loss of that portion of the flow.
However the percentage of traffic arriving at the collector is still around 90%: the reliability is still quite high.

4.3.2 Forty randomized flows register-based scenario

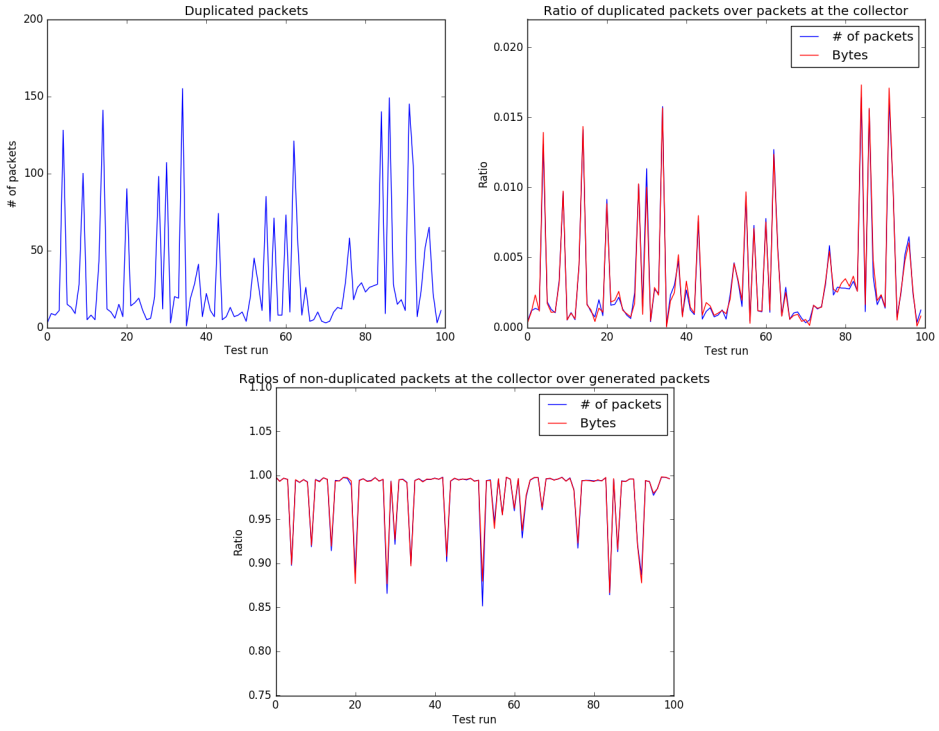


Figure 4.3: Results without route changes.

As we have anticipated in the previous paragraphs, this scenario is characterized by the increase in the computing power of the emulating machine and the randomization of source and destination hosts for the 40 flows. The first results are obtained without the path change and shown in Figure 4.3.
A fundamental premise is due in this case, before analysing Figure 4.3: to generalize as much as possible I decided to increase the number of flows. Even if the higher power of the machine enabled to double the average total generated traffic (from 3 to 6 Mbps), this required a reduction of the traffic of single flows because of the cited limits of our behavioural models. This forced a further modification of the Weibull distribution defining the inter-generation time of packets, which made the random process very unpredictable.

That is the reason why this scenario has witnessed wide oscillations in the number of generated packets. In some cases the volume of traffic was too high for the capabilities of the NDB switches and caused situations of heavy congestion with relevant losses. The sudden decreases that this phenomenon has produced in the portion of packets reaching the collector are visible in the third plot of Figure 4.3.
Anyway, coming to the results, we can see that duplicates show peaks in the tests that are influenced by this cases of congestion, marked by the decreases in the third plot. This is still reasonable if we think that those were the runs with the highest number of generated packets, therefore were also the runs with the highest number of packets arriving during structures' reset. The number of duplicates is higher than than the one seen with no path change in the case of the three fluxes, but this is still acceptable considering the increased number of collected packets. Indeed the portion of duplicated packets over the total traffic received at the collector is generally lower than what we have recorded for the path-changing scenario analysed before, as shown by the second plot of Figure 4.3.

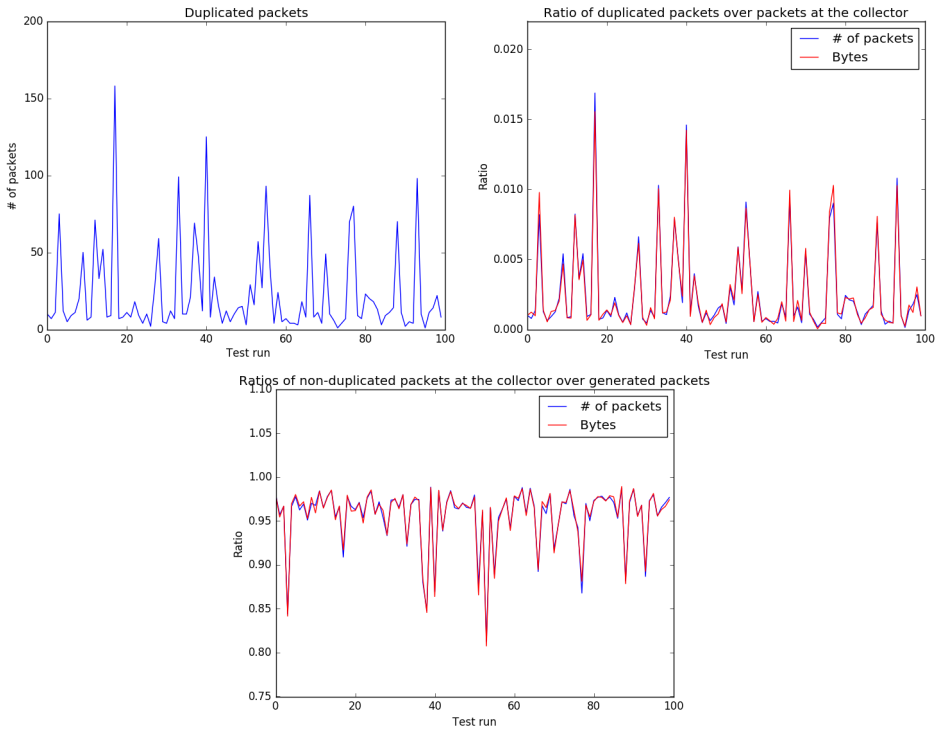


Figure 4.4: Results with route changes.

Figure 4.4 is instead showing the results obtained from the runs with forty randomized fluxes and the route changing subprocess enabled. We can see that the behaviour is almost the same that we have seen for the case without the topology

modification, in terms of the oscillation of generated packets and congestions.
Plots show also some interesting behaviour with respect to what we have observed with 3 fluxes. In that case we have seen how some particular conditions created by the path change could cause some losses on the way to the controller. This is confirmed with the activation of route changing process even in the forty-flows scenario where, combined with the explained congestion, these circumstances make us reach the lowest value for the percentage of generated information reaching the collector. Looking at the duplicates the situation is almost the same as we have seen in this paragraph without the path variation, and it could even seem a little bit better. However in this case the measurements of duplicated packets in correspondence of peaks could be influenced by the high amount of losses that are registered, which in this case get close to 20% of generated traffic. Therefore the lower values registered in the runs that are characterized by those peaks can be a bit misleading. Apart from those cases, the situation seems to remain almost the same in terms of duplicates, even with respect to the total traffic that arrives at the collector. A possible explanation for this is that the randomization of source and destination hosts produces an high variability for paths even without the topology modification. Therefore there is no big difference in the number of replicas created when we activate it. If we combine this with the oscillation in traffic production, the load on NDB switches is likely to be very similar in the two cases, especially in the moment of the reset. Accordingly, a comparable amount of duplicates is generated in the two scenarios.

Taking all the measurements in consideration, this version of the algorithm gives us a first confirmation of the quality that characterizes the key ideas of the algorithm, but also a first glance of the problems yet to face. The presented results are indeed encouraging and demonstrate that the execution of the deduplication on the switches is possible and effective. Nevertheless, the emulations have remarked the effects of congestions and of the use of manual reset for registers. In particular, the time needed for resetting entire registers is not constant and makes the interaction between the various events very unpredictable. Additionally, it forces to the decision of forwarding packets between the reset of the different registers without a real selection, generating the performance and duplicates shown in the plots.
In other words, the first version of the algorithm underlined the problems given by its external and atomic reset. This was taken as a suggestion to develop the other versions, which focus on implementing some alternatives to solve this issue starting from the positive outcomes regarding the idea of the flow-port association.
Finally, the impressive amount of packets lost in the input buffers of the switches, because of congestions, has made the measurement of the reactivity of the algorithm to the topology changes in the production network very unreliable. Indeed, the lost packets due to congestions are caused by the limited resources of the machine used for the emulation, but they can't be separed in our results from the ones generated

by the operation of the algorithm itself. For this reason, if we want to have a clear idea of how many of these losses are given by a delayed reaction of the algorithm to a topology change, the volume of generated traffic must be reduced so that congestions are practically absent.
These are the two lessons learnt from the performance analysis done in this paragraph that will be put in practice in the ones that follow, for the test of the other versions of the algorithm.

4.3.3 Empiric results and probabilistic prediction

To conclude the discussion regarding the first version of the algorithm, it is essential to close the loop with what was said in chapter 3 regarding the losses related to potential collisions of the hash function.

First of all, we need to define the scenario whose results will be used for this analysis. It is important to create a scenario that has the highest number of active flows, to have the widest input set for the hash function, and has the collisions as the only cause of erroneus packet drops. Therefore, I decided to run an additional set of tests with forty flows and without the modification of paths on the production network, since we have seen that this feature is a potential cause of losses.

In this particular setup a packet can be dropped erroneusly for two main reasons:

- a different flow installs the wrong active port in the register because of a collision on the hash;
- the packet is lost at the input of the NDB switches, due to a temporary congestion on the devices. In this circumstances the packet is not treated at all.

Considering that our objective is measuring the impact of hash collisions over hashes, we want to reduce the likelihood of situations of high load for the switches. Consequently, I modified the traffic model to obtain lower volumes and a less bursty traffic: the new model keeps the same distributions for packet generation but reduces the total throughput to avoid congestions. The same stochastic process, which is generating on average 2.2Mbps, is recreated for the tests that will follow, because of the problems that these congestions have generated in the measurement of the performance of the first version.
In this context I measured the packets that were not able to reach collector's interface to compare the probabilistic prediction with the performance observed with the emulator. Figure 4.5 summarizes the outcome of this measurement. If we look precisely at the data collected during the emulation, we can see that this model produces 18.7 lost packets (0.5% of the total traffic) on average.

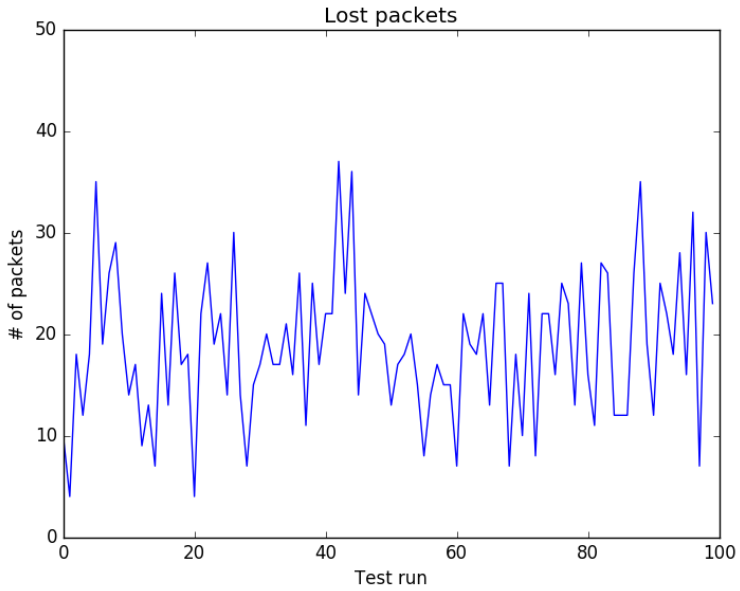


Figure 4.5: Lost packets.

These results are the term of comparison that must be used, to evaluate the quality of the predictions made previously in the thesis about the number of packets dropped because of a hash collision. If we consider the value of the parameters defined in §3.6.1 that characterize the scenario we have chosen (number of flows $k = 40$, bits in hash output $n = 16$, number of eligible ports $\pi = 2$), the relative expected number of lost packets given by Equation 3.7 is around the 0.02%.
This prediction tells that if we generate ten thousand packets, according to our probabilistic model, the switches will drop by mistake two packets because of hash collisions. In the case of the simulation shown in Figure 4.5, 4000 packets are generated on average in each run. Therefore, the expected number of packets lost as an effect of collision, in this particular scenario, is 0.8 packets. The comparison of this prediction with the empirical measurements previously illustrated shows that the probabilistic model defined in §3.6.1 produces a mistaken expected value. Indeed, considering the amount of packets that are lost in the majority of runs, the difference with the prediction is quite large.

The reasons for this error in the probabilistic forecast can be found in the aggressive assumptions I was forced to do, because of the unpredictable phenomena and timing interactions dominating the operation of the algorithm. The hypothesis of homogeneity that had to be applied to many distributions was very strong and led, in practice, the probabilistic model to misleading results that underestimate the number of losses experienced by the algorithm.
In addition to this, the error could have been increased by the low values of k and

π. Indeed in §3.6.1 we have seen that, with the use of more realistic values for these parameters, the expected amount of lost packets reaches much higher values (3-4% of the total generated traffic) and, accordingly, the precision of our forecast could be improved significantly.
In conclusion, the limitations of the assumptions done in the model and the value of the parameters describing the testing scenario have produced a prediction that is far from the performance measured by the framework. Anyway only the creation of a more realistic scenario, with higher values for k and π, could get the situation better, especially if we consider that there is no element available to correct the model and improve the approximation.

4.4 Register-based implementation with automatic reset

The first version of the algorithm made us understand the major problems the emulation framework must face in the measurement. Undoubtedly, the biggest issue is the amount of losses that are generated by the congestion of the switches. Indeed, the packets lost because of a temporary congestion cannot be isolated in the results with respect to the ones due to the operation of the algorithm, which are our main concern. In other words, those losses represent the error of the measure of the capacity of the algorithm to adapt to topology changes.
Considering that our objective is creating a framework able to measure the performance of the algorithm itself, I decided to reduce the amount of generated traffic when measurements included losses. This was done in order to eliminate any impact that is external to operation of the distributed deduplication.
Notwithstanding this, since the measurement of duplicated packets is not directly affected by this phenomenon, some runs were done also at the maximum throughput supported by the framework to have the most realistic scenario. In that case, the discussion will focus only on the efficiency of the filter and the eliminations of duplicates.

Generally speaking, the second version was tested with a wider range of combinations of parameters, to underline the behaviour of the algorithm with respect to the trends individuated by the previous paragraphs.
I decided to focus the study on three parameters:

- the number of flows concurrently active on the network during emulation, which influences the amount of traffic generated on the network;
- the packet generation rate of the single flows, which was modified in the scenarios that allowed this to change the traffic model and volume;

- the interval of renovation/refresh of the flow-port association in the registers, that determines both the reactivity of the network to topology changes and the efficiency in the recognition of duplicates.

The behaviour of the algorithm will be presented with the same statistics that were illustrated in the previous section, but we will focus on the two ratios defined in §4.3 that I consider to be more representative. All the emulations were run with the path changing script activated, since its effect is now well known, after the precedent study, and there is no further interest in doing runs without topology modifications. I will analyse together the sets of tests run with the same number of flows and throughput, to understand the effect that the refresh interval has on the performance given by the same traffic model. Anyway, during our discussion, all the different combinations will be compared to identify some trends and their causes, in order to prove the reliability of our measures and reasonments.

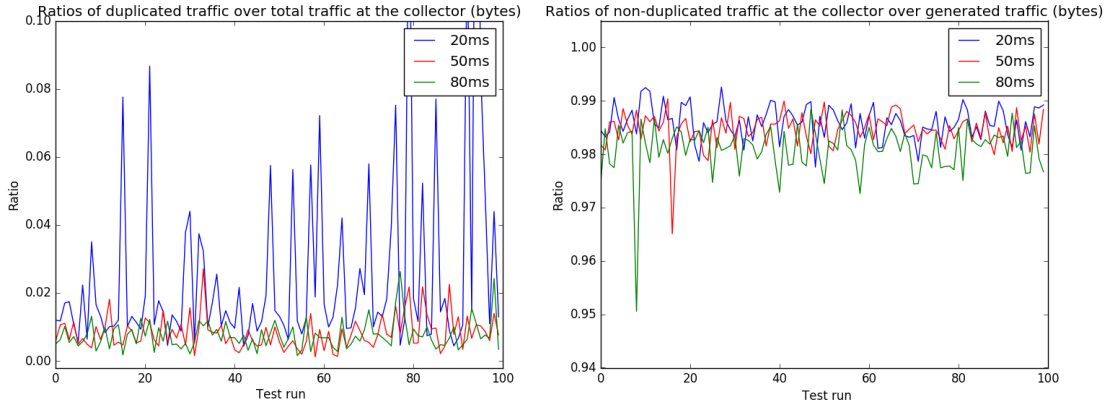


Figure 4.6: Results with 40 flows.

In Figure 4.6 we can see the measurements related to the tests run with forty flows and the traffic model that avoids congestions on the switches. Indeed in this case the total throughput generated on the production network was limited to 2.2Mbps (on average) to reduce the load and measure reliably also the amount of information that gets lost on the collection network. The impact of the refresh interval is quite evident: the longer is the interval, the lower is the measured number of duplicates and the higher is the amount of packets lost. The explanation resides obviously in the operation of the algorithm in this new version with the automatic reset.
Specifically, duplicates can be generated when a flow is renewing the information about its active port, i.e. when the refresh interval expires. Considering a packet sent right before the expiration of the information regarding its flow in the register, an NDB switch could receive two copies of the same packet, sampled by two different SPAN ports, before and after the end of the interval. If the copy arriving after the

end of the interval comes from a previously inactive port and manages to install this as new active port for the flow, both the copies of the packet are forwarded to the collector and we have a duplicate that survives our filter. Therefore, the smaller is the interval the higher is the number of the refreshes made during the emulation, and therefore the number of opportunities for this event to happen increases.
Regarding losses instead, the algorithm can generate erroneus drops when changes of topology happen, with the same mechanism we have seen in the case of the first version of the algorithm. If the change cuts out of the path of the flow the production switch connected to the current active port, all the packets that will follow are dropped until the next refresh because they will come only from inactive ports. The longer is the interval, the longer are these windows of time between path changes and the refresh and, consequently, the higher is the number of packets lost. Figure 4.6 shows how the first phenomenon is much more sensitive to the change of the refresh interval, even if also the effect on losses is quite evident although it is less intense and stable. Considering the two statistics as a whole the two runs with 50 ms and 80 ms are practically the best runs, even if the lower amount of erroneous drops increases the quality of the flter implemented with 50 ms and makes that alternative preferrable.

We can now move on a scenario with higher throughput, that recreates a traffic model almost comparable with the one that was used with the first version. In Figure 4.7 we can see the measurements related to the tests run with forty flows generating on average 4.8Mbps of total throughput, that is the scenario of highest load for the framework in the emulations with the autoreset. As anticipated before,

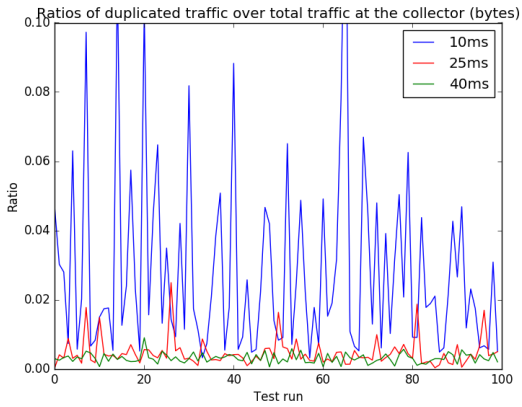
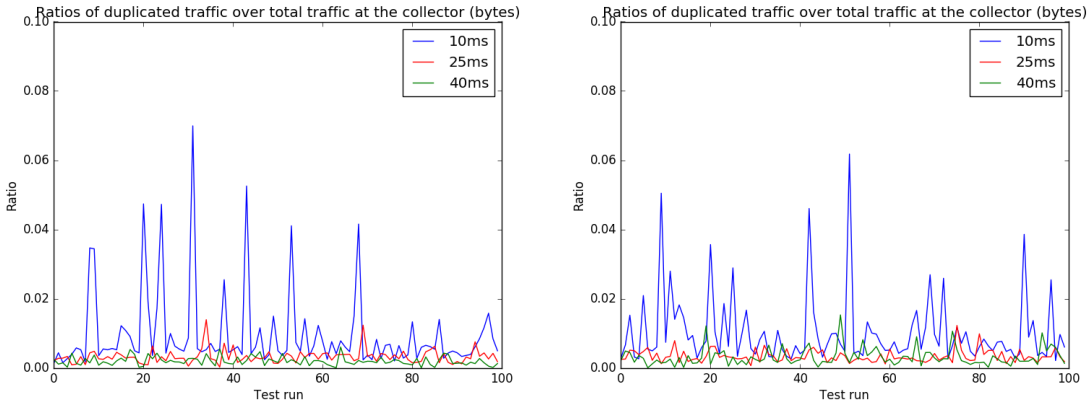


Figure 4.7: Forty flows with autoreset.

this results focus only on the duplicate count. It is quite evident that the effects analysed previously, regarding the variation of the refresh interval, are confirmed and even emphasized by the higher frequency of packet generation. Indeed, even if the trend is the same, in this scenario the choice of an excessively small refresh

interval is much more disruptive for the performance of the deduplication filter. For this reason, the best runs are by far the ones using 40 ms and 25 ms.

In this scenario, it is quite interesting to measure the modification given to the behaviour of the algorithm by a reduction of the active flows going over the production network. In fact the traffic generated with forty flows is too close to the limit supported by the platform, therefore it is not possible to increase the throughtput of the single flows. In an emulation with thirty flows we have more freedom in varying the packet generation rate of the fluxes to analize the perturbations related to this parameter. In Figure 4.8 is illustrated the situation obtained with thirty flows and



(a) Total throughput: 3.4Mbps

(b) Total throughput: 4Mbps

Figure 4.8: Thirty flows with autoreset

two packet rates for the sigle flow: the first is the same packet rate seen before with forty flows (3.4 Mbps in total), the latter is a slightly higher rate (4 Mbps in total). With respect to the results of Figure 4.8a, the performance of the algorithm is somehow showing the same characteristics seen in the case with forty flows. The effects of change of the refresh interval are less marked, but at the same time they are stabler than before: a smaller value durably generates more duplicates. This is probably due to the fact that this scenario is operating with lower volumes of traffic. The amount of traffic is always within the capabilities of Barefoot behavioural models and we have fewer congestions generating sudden bursts of duplicates or losses. Then if the pressure on the switches and the frequency of packets reduces, with respect to the case with forty flows, the likelihood of having errors of the algorithm becomes much smaller, considering that they are mainly due to timing conditions.

The last interesting scenario I wanted to analyse is shown in Figure 4.8b, where I increased the throughput of the thirty fluxes. This was done to bring the frequency of traffic back to values that are closer to what we have seen with forty flows. The difference is that now the frequency is increased even for the packets belonging to

the same flow that use the same entry in registers.
The behaviour underlined with previous cases of study is in general confirmed: with the increase of the refresh interval the number of duplicates decreases, apart from some peaks with 40 ms. In this case the load on the NDB switches is close to the one seen with forty flows, but it is a little bit lower. This generates results that show the oscillations we have seen in that previous case, that are typical of the scenarios that work close to the limit of throughput supported by the models of the framework. This is due to the runs that bring the NDB switches in conditions of temporary congestion with a massive volume of traffic, that suddenly increases duplicates and losses and produces the peaks we see in the chart. Notwithstanding this, the measurements show more stability than the scenario with forty flows.
Even with thirty flows, 25 and 40 ms prove to be the best values for the refresh interval, with no big difference.

In conclusion, taking all the results into consideration, we have seen the effects of the framework parameters on the behaviour of the algorithm in different scenarios. Undoubtedly, we can say that the frequency of the refresh must not be too high, whatever are the traffic conditions, if we do not want the number of duplicates to grow too much. If the interval is big enough instead, the performance depends a lot on the load put on the NDB switches. If the load is light, a lower frequency of renovation of the active port generates very few duplicates with no remarkable impact over losses. Instead, if the traffic is closer to the limit of the NDB network, an intermediate value is probably the best choice in order not to suffer too much from losses.

4.4.1 Results reliability

If we compare the results obtained with forty flows by the two versions of the algorithm when using a comparable amount of traffic, the second version presented no big improvement of performance notwithstanding the absence of the problems of the manual reset of the active port information.
It is therefore important to understand if these results, obtained with the highest load by the version with the autoreset, are reasonable with respect to the theoretical causes of duplicates and losses: this paragraph will illustrate the computation of the duplicates and losses generated in the worst possible case, depending on the parameters of each run, to compare this with the observed measurements and prove their quality.

We have explained that the second version of the algorithm can generate duplicates when packets are sent very close to the end of the refresh interval of their flow. As we said, a duplicate is generated when an NDB switch receives a copy coming from the active port before the end of the interval, and another one after the end from a port that was inactive. If the second copy manages to install its port as new

active port, then both the copies are forwarded to the collector.
To analyse the worst case regarding this phenomenon, I find quite convenient to take the perspective of a single switch. The worst possible case for a switch is having two interfaces connected to the farthest SPAN ports touched by the packets of a flow. If we assume that the two SPAN ports are separated by the longest path of the production network, the second copy of the packet will arrive with a delay that is equal to the maximum latency of the network. All the packets generated by the flow in a time window like the one shown in Figure 4.9, which is as big as the latency of the network and ends in the moment of the expiration of the active port, will be duplicated in this worst case scenario. So we can multiply the maximum rate seen during emulations by the maximum latency in the network to know how many packets we are talking about, and then we multiply by the number of refreshes executed every second to have the total.
Therefore, the number of duplicates generated per second, switch and flow in the worst case scenario is given by:

$$C_f = D\frac{R_f}{T_f} \tag{4.1}$$

where:

- D is the maximum latency in the network;
- R_f is the maximum rate for a flow;
- T_f is the refresh interval.

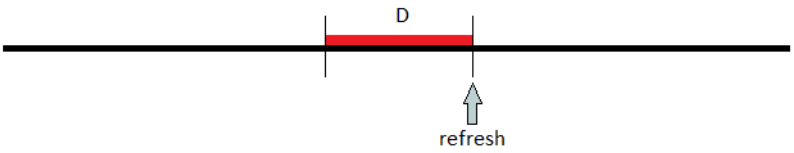


Figure 4.9: The window for the generation of potential duplicated packets.

Multiplying C_f by the number and the duration of active flows in the production network and the number of NDB switches, we have the amount of duplicates generated by the worst case in the framework. Table 4.1 shows the comparison between the maximum number of duplicates measured during emulation and the worst case

scenario, with each combination of the usual parameters. It is quite evident that all the recorded values respect the upper bound set by the worst case computation, confirming the reliability of our results.

Traffic \ Refresh	10ms	25ms	40ms
40flows	1319 / 2720	130 / 1088	63 / 680
30flows	346 / 2040	76 / 816	34 / 510
30flows high throughput	388 / 2040	88 / 816	103 / 510

Table 4.1: Maximum recorded number of duplicates (on the left) and worst case (on the right) for every scenario.

Regarding lost packets, the most important cause in our system is the interaction of path changes with the renovation of the flow-port association. Indeed, as said many times in the thesis, if a change cuts out of the path of a flow the production switch that is mirroring its traffic to the active port, than all the packets of the flow will arrive only from inactive ports and will be dropped until the next refresh.
The worst case for this statistic is obtained when the packets are generated at the maximum rate and the change happens right after the update of the active port. In that case the entire refresh interval will see erroneus drops. So we have, that the number of packets lost per switch, topology change and flow in the worst possible case is:

$$L_f = R_f \cdot T_f \tag{4.2}$$

If we project this on the number of switches, flows and topology modifications done by the framework, the number of lost packets reaches volumes that are extremely bigger than the total generated traffic.
The values are really high even if we decide to take $\frac{T_f}{2}$ as the average interval for the erroneous drops, because of the random timing of topology changes. The predicted number of losses in this case is so high that, if it had been reached during our emulations, we would have seen at most the 15-20% of generated packets arriving at the collector, which is never happening in emulation.

4.5 Table-based version

The last set of tests run in this thesis regards the third version of the algorithm, which is using match-action tables as storage support for the association of each flow to its active port. For this reason, this version requires the interaction with the controller presented in Appendix C. As done for the version implementing the autoreset of registers, the inspection was done using a traffic model properly chosen to avoid the risk of congestion of the switches. This was done to make a comparison

with the previous results that could be as coherent as possible, but also because this version of the algorithm heavily suffered from losses when handling more than 2.2-2.3 Mbps of traffic from the production network.
This additional problem, that was not encountered with previous inspections, pushed the test to focus only on the lightest traffic model seen in the thesis, which is the first seen for the autoreset register-based version. Also in this case, the route changing script was always active to measure the performance in the most realistic scenario.

Figure 4.10 shows the results obtained with this setup. The first chart in the

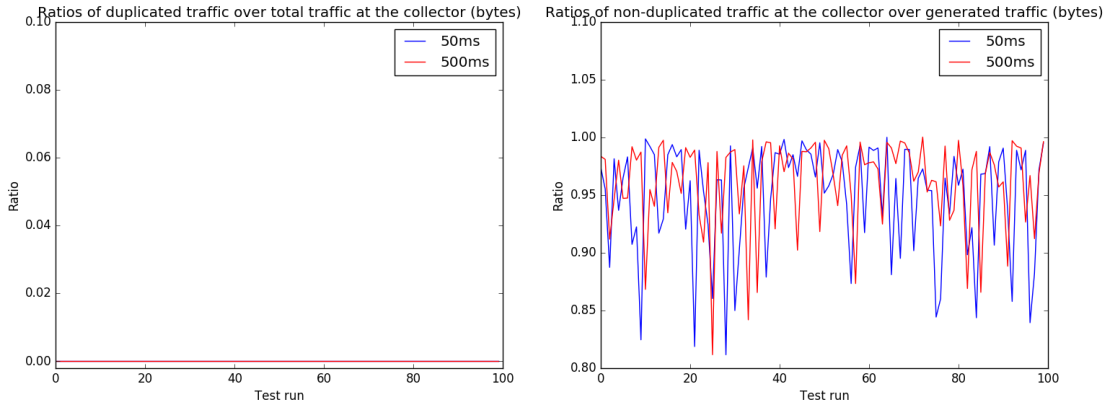


Figure 4.10: Table-based version with forty flows.

image, gives us the first big difference with previous results. Indeed the number of duplicates received by the collector is permanently equal to zero: this represents obviously the best performance obtained by the algorithm in terms of duplicates filtering.
The other important result is linked to the fact that, in opposition to what we have seen in the other analysed cases, the changes done to the lifetime of table entries had a quite negligible impact on the overall performance of the algorithm. Indeed, even if the phenomena generating losses are the same explained for the other versions, it is quite evident that there is no stable difference in the trends individuated for the two metrics by the different values of the frequency of table flushes.
This is in line with what was observed in the set of tests run with the first version of the algorithm. In both the cases indeed the expiration of the flow-port association is renewed cyclically through the CLI. It is an action that is external to the algorithm: its frequency has almost no influence because of the time needed by the system to pass through the CLI to give the command to the switch, which proves to be dominant.
Anyway, since the flush of the table is much more efficient and the algorithm, in its third version, handles better the transient period in which the switch has no rule for the flows, this external reset is not ruining the performance of the algorithm in

terms of traffic deduplication.

Therefore we can say with enough certainty that, as anticipated in other paragraphs, the use of tables and the interaction with the controller makes the third version the most complete and sophisticated among the tested ones. It is the one that is closest to the standard way of programming with the P4 language and, additionally, it is producing the most impressive results for deduplication, even if they are slightly ruined by an higher amount of losses.

4.6 Final comparison of recorded results

We have now all the elements to compare the performance obtained with the different versions of the algorithm and identify the one that fits best the original objectives of the algorithm. I will keep out of the discussion the register-based version with the autoreset since, as explained before, it must be seen an introductory version that was meant to approach the difficulties of the measure and prove the quality of the kernel of the algorithm.

Regarding the other two alternatives, which are by far the most complete and definitive ones, the discussion is quite challenging and uncertain.
If we consider both the statistics collected during emulations, probably the register-based version with the automatic reset of registers could be the one that satisfied best the necessities of the scenarios implemented by the framework. Indeed, with respect to the last version, it is assuring a remarkably lower amount of losses on the collection network, even if some duplicates manages to survive its selection. In other words, even if the filter is slightly less efficient, it is much more accurate because only duplicates are dropped.

However, if we try to abstract from the specific scenario of our framework, this preference could be inverted. If we think to the cause of losses on the collection network, once congestion is not a factor, it is quite unlikely that a real network produces, in its normal operation, topology changes as frequently as our production network does. It was an extreme setup, designed to stress the differences between different versions of the algorithm and different setups of the same version.
This could suggest that, once we get out of the testing framework and bring the algorithm on real devices, the table-based version could be the most performant considering that the difference in losses could become less relevant, giving to the complete elimination of duplicates a stunning importance.

In this sense, we can say that the alternative to choose is, as often happens in engineering, defined by our needs. On the one hand, if we want the algorithm to adapt to any condition, because we do not know the characteristics of the scenario we will face, the best solution is the register-based one with the autoreset. On the other hand, if the setup of the monitored network is known and regular, without

anomalous path modifications, the efficiency of the filter could guide our choice on the table-based version.

This concludes the analysis of the results collected in this work for the algorithm. Notwithstanding the limitations of behavioural models, the framework was anyway able to run the algorithm in challenging scenarios to automatically test its quality in terms of performance. Furthermore, the entire study of results showed the main implementation problems that the algorithm could have to face in the practic usage.

Chapter 5

Related and future works

5.1 Related works in literature

The topics covered in this thesis are, as we have said many times, something that has generated a lot of interest in the scientific community, therefore they have been object of several papers and studies. Notwithstanding this great attention obtained in the field of research, these technologies are still trying to enter the market and their concepts are finding a practical implementation only with recent projects, like the one that inspired this work.
Therefore, even if the set of papers talking about these concepts is very wide, finding something that could be related to the type of testing that was addressed here is still quite hard.

Anyway, concerning the functional test done for the porting of traditional OSes over the Tofino chip, one work that could be considered an inspiring model in terms of automation and integration of testing is for sure [9].
The paper explains perfectly all the concepts regarding Continuous Integration that were briefly summarized in the previous chapters, and then explains the solution used to apply them to the regression testing. The work represents, in some sense, the maximum level that this philosophy can reach, since it includes pre and post commit tests and a complex analysis on dependency graphs to identify which modules must be tested starting from the committed one.
In this thesis the level of test sophistication was much lower, because the needs were different, but for sure the cited paper represents a reference point for every work embracing the ideas of Continuous Integration.

Regarding the network-emulation test, the examples of use of Mininet to test distributed algorithms and simple protocols are particularly numerous, especially in the SDN domain.

In [10], for instance, it is possible to encounter an environment that is very similar to the framework designed in the thesis. Indeed, as for the case of our algorithm, the authors are addressing with their paper the world of Data Centers, proposing a solution to the problem of the high Flow Completion Time (FCT) experienced in those networks by short and lightweight flows.
The solution proposed is a library that acts also as a middleware between the application and the transport layer which is called *RepFlow*, whose main idea is generating a replica for every small flow with the aim of exploiting two different paths to reach the destinations. Since the congestion is due to random circumstances over these networks, it is likely that only one of the flows gets slowed down and therefore the replication of the traffic could reduce the FCT.
The use of Mininet done in the paper is extremely similar to what was done in this thesis: a topology with a 4-pod Fat-Tree was created and the RepFlow light protocol was installed on the hosts of the network created in Mininet. The traffic was generated with some socket-based sender and receiver processes, implementing the Repflow idea for the replication of short flows.
The first difference with our experiments lays, first of all, in the absence of P4-enabled devices, which is in some sense the real added value of our work. Then, the algorithm under inspection is running on the hosts and not on the switching devices of the network. Moreover, its operation requires also a slight modification of applications, even if the network stack remains the same for the networking nodes. However, it is a very good example of the versatility of Mininet in the inspection of the overall performance of a program running distributedly over different devices.

Another interesting work that is more related to the SDN concepts is [11], which focuses on the problem of balancing the load among the different instances of a distributed controller with respect to the number of switches to handle.
The solution described in the paper is a module that must be run on every instance of the distributed SDN controller, called *DALB*. This allows each controller instance to interact with the user for the setup, collect data on the operation of other instances and even implement different decisions and elections when it is necessary to offload some switches to another replica of the controller. What is interesting to us in the testing is the interaction of Mininet with external softwares: in this paper indeed the tool interacts with an external distributed instance of the *Floodlight* controller and tests the performance of the DALB module with the use of real benchmarks.
The points of contact with our case are several. First of all the traffic is, like in our case, generated by a general-purpose software which was not created ad-hoc for the simulation. Then, the use of an external controller is the only solution available for the future table-based implementation of the deduplication algorithm, considering that Mininet is not offering a P4 controller nowadays.
The difference with our work lays for sure in the use of a sophisticated controller, which is additionally the application under inspection and plays the role taken by

the deduplication algorithm in this thesis. However the cited work gives another alternative in the inspection of distributed algorithms on an emulated environment, showing a way to extend Mininet to bend it to specific needs.

5.2 Future works

The field of future works is highly polarized towards the part of the thesis devoted to the network testing, since the platform for the functional inspection is quite complete in its operation.

Speaking about the PTF testing, the possible modifications to the actual system are mainly related to changes in the nature of tests we want to run and the analysis we are interested to do on their results.
Indeed, if the tests require more steps or the presentation of results becomes more sophisticated, the adoption of Jenkins pipelines can be an attractive solution to exploit the presence of milestones and ad hoc setup for each build step.

Instead in the case of the deduplication algorithm a possible improvement to consider is the distribution of the emulation over different virtual machines. This could reduce the load of emulating each behavioural model and allow us to overcome the limits we have experienced about the supported throughput.
The other big problem in our framework is the absence of time measurements, due to the limits of Mininet. An interesting extension of the work could be the creation of something equivalent on a simulator, rather than on an emulator, to analyse the performance of the algorithm also in terms of speed and introduced delays.

Chapter 6

Conclusions

The objecitve of this work was proposing innovative frameworks for the addressed testing categories, with the additional difficulty of dealing with extremely new devices supported by few tools. Generally speaking, this thesis is not giving universal solutions, but it has reached the goal of offering a new view on the problems that were faced and some starting point for future extensions.

Indeed, in the case of the functional testing on the single-device, the first need to be fulfilled was creating a platform able to analyse the behaviour and the operation of the interface between the OS and the hardware. This brought to the necessity of being able to work on the different APIs used in the described porting operation, to assure the inspection of its effectiveness in offering seamlessly usual services and data-plane programmability.
For this reason I have chosen to test the code through the PTF framework, which offered library functions to invoke the primitives belonging to a wide range of APIs representing the border under analysis. The framework enabled me to work at different layers with great fluidity and an intuitive set of library functions, although the documentation regarding it was practically null. This easiness in the definition of test cases is particularly useful if we consider what was the nature of the platform I wanted to build: apart from the area of the code under inspection, the target was creating a testing platform that could be completely integrated in the work of the programmer.

To achieve this I managed to apply, in the limits given by the interests of the thesis, the philosophy of Continuous Integration in order to turn the framework in a real instrument in the hands of the developer to understand the behaviour of its code in the smoothest possible way. With the help of an automation server, obtained with the explained setup of a Jenkins instance, the system is in fact able to offer information about the status of the project on different layers and at different levels of detail. The programmer can choose and execute the test cases for its code literally with a click and, at the same time, comfortably access every debug-related

information from a web interface. The Jenkins panel also enables managers to follow the evolution of team work and simplifies a lot sharing results, bugs, reports of various type.

Consequently we can say that, notwithstanding the fact that Continuous Integration is far from being achieved, the initial objectives that were inspired to that philosphy were reached. Moreover, All of this was done creating a system that is also extendable and very adaptive to potential new needs, thanks to the plugin-based soul of Jenkins.

Regarding network testing, the considerations to do are more complicated. Even in this case we can say that this work managed to accomplish the task of giving a new approach for the test of distributed algorithms on P4 devices using Mininet, to measure the performance of the algorithm itself, the devices and the emulation tool. The algorithm is probably the component that comes out of this work with the best results. It was indeed possible to see how the effectiveness of the filter operated by the collection network is always very high, witnessing the quality of the conceptual kernel of the algorithm. Some problems arose under the perspective of the losses related to the adaptation of the algorithm to the changes of topology in the production network. Anyway, regarding this aspect, it is important to consider that the implementation of the path changing script was quite extreme. Indeed the likelihood that a network suffers from failures that force its devices to route changes every 0.5 seconds is very low and this suggests that the reduction of packets arriving at the collector, in the end, is not something that weakens the value of our measurements or of the algorithm.

The reasonment must be different if we move on the analysis of the performance of the tools themselves, which didn't match completely the expectations. Indeed, even if Mininet confirmed its incredible versatility and extendability once again, the behavioural model provided by Barefoot Networks didn't manage to deal with the traffic modeled as the cited papers suggested. This gave the impression that these behavioural models were designed more for tests on the conceptual correctness of algorithms, to check that the right packets reach the right destinations, than for tests on their performance in fully realistic environments.
In this sense, the only regret I have is not having had the chance to try a distributed emulation, which could have helped splitting the load on more machines and alleviate the limitations given by this model. Eitherway, if we consider that the objective was creating a testing framework offering the means to evaluate a distributed algorithm, the final product is effective and the initial objectives were met with the addition of a good grade of automation and attention to practical implementations.

Taking everything into consideration, we can say that, with the limitations given by our necessities and tools, the work has introduced a reproducible way of approaching the topic of testing under many different aspects, giving tools and ideas for the implementation of future works or extensions in the field of P4-enabled switching

devices.

Appendix A

PTF tests example

In this Appendix I wanted to give an idea of the work that is needed to write a test to check a particular behaviour with the PTF framework using SAI API. The example I am proposing is meant to check the operation of broadcast forwarding in the context of VLAN use: we are therefore talking about a test that is checking a layer 2 functionality.
As anticipated in chapter 2, the necessary step to add a test to the platform is the creation of a class: the definition of the class *L2VLANBroadcastTest* will therefore be the object of our discussion.
With the aim of having a clear and dynamic discussion, I will try to alternate the code and the comments on it.

Listing A.1: "Initialization"

```
class L2VLANBroadcastTest(sai_base_test.ThriftInterfaceDataPlane):
    def runTest(self):
        print
        switch_init(self.client)
        ports=[]
        ports.append(port_list[0])
        ports.append(port_list[1])
        ports.append(port_list[2])
        ports.append(port_list[3])
        v4_enabled = 1
        v6_enabled = 1
        mac_list = []
        mac_action = SAI_PACKET_ACTION_FORWARD
        vlan_members =[]

        vlan_id1=10
        vlan_id2=11

        sai_thrift_vlan_remove_all_ports(self.client, switch.
   default_vlan.oid)
```

```
        vlan_oid1 = sai_thrift_create_vlan(self.client, vlan_id1)
        vlan_oid2 = sai_thrift_create_vlan(self.client, vlan_id2)
```

To understand these first instructions, it is important to remark the situation the programmer must start from to define a test. In the initial configuration, the forwarding element is an L2 switch which operates with the virtual interfaces generated on the machine by the SDE.
The first thing to do, for this reason, is saving in some data structure or variables the ports that will be used in the test. In the case of a test that operates within the borders of L2 forwarding, this is just a comfortable habit to write readable code. Instead, in the case of L3 testing, this is necessary because the inspection requires the creation of a virtual router and the detachment of the chosen ports from the initial L2-switch, to configure them as the interfaces of the virtual router.
After this choice, in our case all the ports are removed from the default VLAN and the ID of the two VLANs involved in the test are defined.

Listing A.2: "VLAN configuration"

```
        vlan_members.append(sai_thrift_create_vlan_member(self.
    client, vlan_oid1, ports[0], SAI_VLAN_TAGGING_MODE_UNTAGGED))
        vlan_members.append(sai_thrift_create_vlan_member(self.
    client, vlan_oid1, ports[1], SAI_VLAN_TAGGING_MODE_UNTAGGED))
        vlan_members.append(sai_thrift_create_vlan_member(self.
    client, vlan_oid2, ports[2], SAI_VLAN_TAGGING_MODE_UNTAGGED))
        vlan_members.append(sai_thrift_create_vlan_member(self.
    client, vlan_oid2, ports[3], SAI_VLAN_TAGGING_MODE_UNTAGGED))

        attr_value1 = sai_thrift_attribute_value_t(u16=vlan_id1)
        attr1 = sai_thrift_attribute_t(id=
    SAI_PORT_ATTR_PORT_VLAN_ID, value=attr_value1)
        self.client.sai_thrift_set_port_attribute(ports[0], attr1)
        self.client.sai_thrift_set_port_attribute(ports[1], attr1)

        attr_value2 = sai_thrift_attribute_value_t(u16=vlan_id2)
        attr2 = sai_thrift_attribute_t(id=
    SAI_PORT_ATTR_PORT_VLAN_ID, value=attr_value2)
        self.client.sai_thrift_set_port_attribute(ports[2], attr2)
        self.client.sai_thrift_set_port_attribute(ports[3], attr2)

        for i in range(2):
            mac_list.append("00:00:00:00:%02x" % (i+1))
            sai_thrift_create_fdb(self.client, vlan_id1, mac_list[i
    ], ports[i], mac_action)

        for i in range(2,4):
            mac_list.append("00:00:00:00:%02x" % (i+1))
```

```
            sai_thrift_create_fdb(self.client, vlan_id2, mac_list[i
    ], ports[i], mac_action)
```

The following step in our test, illustrated in this listing, is the association of hosts and ports to the different VLANs. The configuration starts from ports, which are associated to the group of ports of the right VLAN. The most important information we set for the port, in this phase, is the type of traffic we expect from it, which defines also the operation of the port and the tagging of traffic. In this particular case the traffic expected from every port is declared to be untagged traffic, that is the reason why it is important to set the VLAN ID as a port attribute, to enable the switch to operate in the port-based tagging mode. If instead the traffic was expected to arrive to the switch already tagged, this step would not be necessary.
The last cycles of the snippet are simply introducing the entries in the flow-table of the switch that associates hosts, and their MAC address, to the port we use to reach them. This is, in practice, instructing the testing framework with the information a normal switch takes from backward learning.

Listing A.3: "Emulation and verification"

```
        pkt = simple_tcp_packet(eth_dst='ff:ff:ff:ff:ff:ff',
                                eth_src='00:00:00:00:00:01',
                                ip_dst='10.0.0.1',
                                ip_id=101,
                                ip_ttl=64)

        exp_pkt = pkt #since we are not crossing any L3 network
    even the ttl remains the same

        try:
            send_packet(self, 0, str(pkt))

            verify_packets(self, exp_pkt, [1]) #same vlan of the
    sender
            verify_no_packet(self,exp_pkt, 2) #other vlan
            verify_no_packet(self,exp_pkt, 3) #other vlan
        finally:
            sai_thrift_flush_fdb_by_vlan(self.client, vlan_id1)
            sai_thrift_flush_fdb_by_vlan(self.client, vlan_id2)

            attr_value = sai_thrift_attribute_value_t(u16=1)
            attr = sai_thrift_attribute_t(id=
    SAI_PORT_ATTR_PORT_VLAN_ID, value=attr_value)
            for i in range(4):
                self.client.sai_thrift_set_port_attribute(ports[i],
     attr)
                self.client.sai_thrift_remove_vlan_member(
    vlan_members[i])
```

```
            self.client.sai_thrift_remove_vlan(vlan_oid1)
            self.client.sai_thrift_remove_vlan(vlan_oid2)
```

We have now reached the moment where the real action starts. In the first lines of this piece of code, the packet that will be sent to the ports of the switch and the one we expect the switch to send out are created. It is possible to see the easiness with which the PTF framework functions enable us to produce packets for our inspections: creating a packet is as easy as calling a function, and the availability of default values for most of the fields lets us specify only the ones that are directly involved in the test.
In our case the test is focused on checking that a broadcast packet belonging to VLAN 1 is forwarded only to the ports associated to this VLAN, while the others don't see the packet. We are practically testing the virtual separation of the L2 domain that must be enforced by the VLANs and, to do this, a broadcast packet is sent on one of the ports belonging to VLAN 1 to see the reaction of the switch.
As we exposed already in chapter 2, the PTF framework gives the chance to check that a packet is forwarded on a particular port, but also that no packet is sent out of other ports as well as any other condition that we are able to specify in the form of an assertion. All the verifications anyway, if not satisfied, launch an exception that carries the information about the error that caused it. For this reason, the instructions that run the actual test must be executed in a try-catch clause.
The *finally* block is then dedicated to the cleanup: the flow-table is flushed and the configuration of the switch is restored to the original state. This is done to leave the switch ready to be used for other tests, since tests can be grouped to be executed in sequence through some Python annotations.
This is a very straight-forward example, but it is already significant to show how checking the operation of specific elements of the switch becomes easy in the platform. The PTF framework indeed offers, both for setup and verification, functions that are very effective in the inspection of several networking functions, but also immediate and self-explaining.

Appendix B

The code behind the network emulation

In this Appendix I will try to explain the code I have written to build the emulation framework described in Chapter 3. To avoid boring the reader with too many useless details, I decided to list and briefly comment only the most relevant parts of the code, where the logic and the main contribution of the work reside. With the same objective I will alternate the snippets of code with their explanation in order not to create two big blocks (code and explanation) that would be heavy to follow.

Listing B.1: "Topology definition"

```
class NDBDeduplicatorTopo(Topo):
    """NDB deduplicator sample network with product_num switches in
    the production networ and a mesh in the NDB network"""
        def __init__(self, sw_path, json_path, thrift_port,
   product_num, single_cell_hosts, ndb_number, **opts):
        #product_num is the number of switches (and cells, see
   below) we want in the production network
        #single_cell_hosts is the number of hosts we want to
   connect under each production switch
        self.product_num = product_num
        # Initialize topology and default options
        Topo.__init__(self, **opts)
        # Create a mesh to model the NDB network
        self.ndb_number = ndb_number
        NDBRoot=self.NDBMeshBuild(sw_path, json_path, thrift_port,
   ndb_number)

        # Extract the switches of the NDB network that will be the
   switches connected to the production network's SPAN ports
        lastlevel_switch = self.NDBswitch[1:]
        #Production network creation
```

```
        prod_switch = [self.NDBSingleSwitchSubTopo(s,
    single_cell_hosts) for s in xrange(product_num)]
        num_lastlevel= len(lastlevel_switch) #number of switches
    connected to the production network
        for s in xrange(product_num):
            # We add the link towards the NDB switch
            self.addLink(prod_switch[s], lastlevel_switch[s%
    num_lastlevel], addr1="00:00:06:00:%02x:%02x"%(s, s),intfName1="
    ps%d-span"%s, addr2="00:00:06:00:%02x:aa"%(s))

        # We now create an almost spine and leaf topology among
    production network switches
        for s in xrange(2,6):
            self.addLink(prod_switch[s], prod_switch[0])
            self.addLink(prod_switch[s], prod_switch[1])

        self.addLink(prod_switch[3], prod_switch[4])
        self.addLink(prod_switch[0], prod_switch[1])
```

This first piece of code is the constructor of the class that defines the topology used in the framework. Following the sequence of commands in the function, we can see clearly that this constructor starts building the mesh which represents the NDB network, maintaining a list of switches. The list is then used to group all the devices that will have to be connected with the SPAN ports of the production network (*lastlevel_switch* list).
Then the production network itself is created, initially with the definition of the cells that represent its elementary blocks and then with their connection (at the end of the snippet).
In the middle, between these two operations, every production switch is connected with an NDB switch, in order to create the link on which all its traffic will be mirrored. The choice of the switch as *lastlevel_switch[s%num_lastlevel]* is meant to try to balance the load on the NDB switches, with the introducion of a sort of round robin in this association.

Listing B.2: ”Mesh and Single switch sub-Topologies”

```
  def NDBMeshBuild( self, sw_path, json_path, thrift_port, ndb_num
   ):
        self.NDBswitch = []  #we keep a list to access the switches
    later on
        NDBcount=0
        for s in range(ndb_num):
            self.NDBswitch.append(self.addSwitch('NDBs%s' % (s),
   mac = '00:00:00:00:00:%02x' % s, sw_path = sw_path, json_path =
   json_path, thrift_port = thrift_port + s))
            if s==0:
```

```
                # Connect the collector to NDBRoot
                collector = host = self.addHost('collector', cls=
    P4Host ,ip = COLLECTOR_IP + "/24", mac=COLLECTOR_MAC)
                self.addLink(self.NDBswitch[s], collector)
            NDBcount = NDBcount + 1
            for s2 in range(NDBcount -1):
                self.addLink(self.NDBswitch[s], self.NDBswitch[s2])
        return self.NDBswitch[0]

     def NDBSingleSwitchSubTopo(self, switch_id, hostnum=2 ,**opts):
        switch = self.addSwitch("ps%d" % (switch_id), cls=
    OVSKernelSwitch)
        for h in range( 1, hostnum+1 ):
            host = self.addHost( 'h%d_%d' % (switch_id, h), ip="
    10.0.%d.%d/16"% (switch_id, h), mac="00:00:00:00:%02x:%02x" % (
    switch_id, h) )
            link= self.addLink( host, switch)
        return switch
```

The previously seen constructor uses the functions in this listing, which are quite straightforward to understand and were created with very slight modification of some function already present in mininet Python API. What is important to point out, for these functions is the use of the *thrift_port* parameter in *NDBMeshBuild*. In order to configure properly each of the NDB switch, it is important to have their thrift servers listening on different ports. Each one must be created with a different value of the thrift port, therefore the parameter passed to the function is used as the start of the sequence of ports to be used.

Listing B.3: "CLI switch configuration"

```
def configure_dp(commands_path, thrift_port):
    cmd = ["/home/user/sde/install/bin/tofinobm_CLI",
           "--thrift-port", str(thrift_port)]
    with open(commands_path, "r") as f:
        print " ".join(cmd)
        sub_env = os.environ.copy()
        pythonpath = ""
        if "PYTHONPATH" in sub_env:
            pythonpath = sub_env["PYTHONPATH"] + ":"
        sub_env["PYTHONPATH"] = pythonpath + \
                                "/home/user/sde/build/tofinobm/
    thrift/gen-py/"
        subprocess.Popen(cmd, stdin = f, env = sub_env).wait()
```

This is the function used to configure the dataplane of the switch. The method used for this configuration is creating a subprocess that runs the runtime CLI of the Tofino chip, setting its standard input to a file that contains the proper commands. It is needed to intialize some tables and structures.

The same idea is used for the manual reset of some of these structures, which is done later on when testing the versions of the algorithm that need it.

Listing B.4: "Network launch and SPAN mirroring"

```
    net.start()
    net.get('c0').start()
    for i in range(PRODUCT_NUM):
        p_switch = net.get('ps%d' % i)
        p_switch.start([ProdController])
        p_switch.cmd("ovs-vsctl \
                -- --id=@p get port ps%d-span \
                -- --id=@m create mirror name=m0 select-all=true
   output-port=@p \
                -- set bridge ps%d mirrors=@m" % (i, i))

    sleep(1)
    for i in range(NDB_NUM):
        configure_dp("./switch-commands/tofinobm-commands.txt",
   thrift_port+i)
    sleep(1)
    print "Ready !"
```

This listing brings us in the middle of the *main* function, we can see how the network is started and production switches are configured.
In the two register-based versions of the algorithm, the NDB switches are not needing any control plane management from a controller, since they are using the so called *registers* as internal storage resource to keep track of their state.
Then, even in the table-based version, the digest-based operation of the controller did not require a connection through Mininet Python classes. The controller just connects to the thrift servers of the P4-swiches as a process that is external to the emulator. In that scenario, this connection is done right after the configuration of the NDB switches: for each of them an instance of the process shown in Appendix C is started and connected to the right thrift port.
This is the reason why the network is always intialized with no controller and the production network switches must be connected to their controller individually to emulate an OVS switch. Then, the mirroring through the SPAN port is configured and the NDB switches are initialized as seen before with *configure_dp*.

Listing B.5: "The actual emulation"

```
    net.get('h3_1').cmd("ITGRecv &")
    sleep(1)
    net.get('h5_1').cmd("ITGSend -D -T UDP -a 10.0.3.1 -n 200 1 -W
   13.503 1190 -t 10000 -l h5_1.log -x h3_1a.log &")
    net.get('h5_2').cmd("ITGSend -D -T UDP -a 10.0.3.1 -n 1400 1 -W
    13.503 1190 -t 10000 -l h5_2.log -x h3_1b.log &")
```

```
    fnull=open('timestamps', 'w')
    subprocess.Popen("/home/user/sde/NDB_deduplicator/route_changer
.py", stdout=fnull)
    subprocess.Popen(["/home/user/sde/NDB_deduplicator/
reset_register.py", str(NDB_NUM), str(thrift_port)], stdout=
fnull)

    sleep(15)
```

This is the last snippet of code that I find interesting to present in terms of how the emulation is run. It is probably the most important among the ones I have showed here, since it illustrates how the traffic model explained in Chapter 3 and 4 takes life in one of the generated fluxes. We can see indeed how the code uses D-ITG commands and options to overlap two sub-fluxes, characterized by a normal distribution for the packet size, and obtain the bimodal distribution desired for that parameter. This is then combined, for both the sub-fluxes, with a Weibull distribution that instead models the inter-departure times of packets.
In the last lines of the listing, instead, it is possible to observe the part of the code that is responsible for the emulation of perturbations in the production network and for the reaction of the algorithm. The first subprocess that is created runs a script which chooses randomly one of the several configurations prepared for the paths in the production network, and imposes this to its OVS switches. This operation is meant to reproduce the case of a failure of a link or a switch, with the consequent modification of the paths over the network, and this is repeated at random intervals of time to be more general.
The second subprocess is instead resetting the registers that keep track of the different flow-port associations. In the table-based version of the agorithm an analogous script is used to ciclically flush the rules installed on the switches, while in the register-based version with the auto-reset no script is obviously needed. Anyway, regardless of how it is done, this reset is fundamental to adapt to changes in the production network, since it creates a cycle of renovation for the information that tells which replica of the flow is accepted. In other words, it enables the switches to choose a new active port if flows do not pass anymore through the switch which provided the chosen replica.
This last part of the code is therefore the most important to assure that the enrivonment is valuable in terms of testing distributed algorithms in real-life conditions: we could say they are what gives relevance to our measurements.

Appendix C

The code of the controller

I will now expose the code that implements the key aspects of the operation of the controller for the deduplication algorithm. The code I will present implements the rule installation on a single switch, to handle more than a switch the controller or the framework must start more instances of this process with the proper connection parameters. In our case this is done directly from the script seen in Appendix B. As for the other Appendixes, I will alternate listings of code with their explanation, to have an agile discussion.

Listing C.1: "Connection to the switch"

```
    client = utils.thrift_connect_standard(args.thrift_ip, args.
   thrift_port)

    # retrieve notifications socket and json config
    info = client.bm_mgmt_get_info()
    socket_addr = info.notifications_socket
    assert socket_addr is not None
    json_cfg = utils.get_json_config(standard_client=client)

    # connect to notifications socket
    sub = nnpy.Socket(nnpy.AF_SP, nnpy.SUB)
    sub.connect(socket_addr)
    print "connected"
    sub.setsockopt(nnpy.SUB, nnpy.SUB_SUBSCRIBE, 'LEA')
```

In this first part of the the code, the process connects to the switch it will have to interact with. The library functions offered by the Barefoot Networks SDE enable the connection with the thrift port of the switch and return the client that must be used to handle the session. Once we have the client, the script can obtain the address of the socket allocated for the transmission of notifications on the switch and retrieve the JSON file containing the P4 program run on the switch.
The notifications are the message conveying the digests to the controller. They are

not sent on the connection with the thrift port, therefore the controller must connect to the proper server on the switch as it is done in the last lines of the listing.
The JSON file can be used by some other library functions to learn automatically the format of the digests. Since I have written a manual parsing of them, this is not needed.

Listing C.2: "Reception and parsing"

```
    while True:
        print "Waiting for notifications"
        msg = sub.recv()
        # we read the header of the notification
        msg = msg[4:]
        s = struct.Struct("<IIBHHB")
        sw_id, cxt_id, list_id, buffer_id, num_samples, pad = s.
    unpack_from(msg)
        print "Received one notification"

        msg = msg[28:]
        i=0;
        while len(msg)>(14*i):
            #we first read the digest
            index = 14*i
            current_digest = msg[14*i: 14*(i+1)]
            ip_src = socket.inet_ntoa(current_digest[0:4])
            ip_dst = socket.inet_ntoa(current_digest[4:8])
            s1 = struct.Struct("<BHHB")
            ip_proto, udp_sport, udp_dport, ingress_port = s1.
    unpack_from(current_digest[8:])
            print "Fields: ", ip_src, ip_dst, ip_proto, udp_sport,
    udp_dport, ingress_port
            i = i + 1
```

After the creation of the socket the controller enters its operative cycle that starts with the wait for a message from the switch. The message, as we said, will be a notification carrying an header and then all the digests produced one right after the other.
The code therefore parses the header and reads the different parts of the every digest carrying the information of a new flow: IP source and destination addresses, Layer 4 Protocol, Layer 4 source and destination ports and ingress interface on the switch.

Listing C.3: "Rule installation"

```
            cmd="table_add flow_port_table retrieve_active_port %s
    %s %d %d %d => %d"%(ip_src, ip_dst, ip_proto, udp_sport,
    udp_dport, ingress_port)
```

```
            os.system("echo '%s' | /home/mario/bf-sde-6.0.0.16/
    install/bin/tofinobm_CLI --thrift-port %d --json /home/mario/bf-
    sde-6.0.0.16/install/share/tofinobmpd/ndb/ndb.json >> switch%
    d_rules.log" %(cmd, args.thrift_port, args.thrift_port-9100))
```

As we said in Chapter 3, the switch creates a digest every time it receives a packet belonging to an unknown flow. Therefore, the controller receives the information cited above only for the flows that have no rule in the table of the switch. Since there is no reason to prefer one of the interfaces of the switch in the choice of the active port, the first one bringing a packet of the flow can cover this role.
This is why, right after the parsing of every digest, a rule is installed in the *flow_port_table* to associate the 5-Tuple of the flow to the ingress port of the packet related to the treated digest. From that moment until the next refresh of the forwarding state, the packets of that flow will be forwarded to the collector only if they arrive from the ingress port contained in the digest. The installation is made in the easiest possible way: we exploit the commands of the CLI offered by the switch through its thrift port, using the system shell.
The logic of this controller is extremely simple, but this basic example shows the main interactions a controller can have with a Tofino-enabled switch. Indeed, apart from the different elaborations needed to produce a decision and a rule, the digest-based communication is practically recognized as the standard way to implement the control-plane management in the P4 paradigm and the presented example fully illustrates this techinque.

Bibliography

[1] M. Meyer. "Continuous Integration and Its Tools". In: *IEEE Software* 31.3 (May 2014), pp. 14–16. ISSN: 0740-7459. DOI: 10.1109/MS.2014.58.

[2] Jenkins. *Jenkins wiki*. URL: https://wiki.jenkins.io.

[3] Mininet. *Introduction to Mininet*. 2010. URL: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet.

[4] Alessio Botta and Alberto Dainotti and Antonio Pescapè. "A tool for the generation of realistic network workload for emerging networking scenarios". In: *Computer Networks* 56.15 (2012), pp. 3531–3547. URL: http://dx.doi.org/10.1016/j.comnet.2012.02.019.

[5] Theophilus Benson, Aditya Akella, and David A. Maltz. "Network Traffic Characteristics of Data Centers in the Wild". In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. IMC '10. Melbourne, Australia: ACM, 2010, pp. 267–280. ISBN: 978-1-4503-0483-2. DOI: 10.1145/1879141.1879175. URL: http://doi.acm.org/10.1145/1879141.1879175.

[6] Theophilus Benson et al. "Understanding Data Center Traffic Characteristics". In: *SIGCOMM Comput. Commun. Rev.* 40.1 (Jan. 2010), pp. 92–99. ISSN: 0146-4833. DOI: 10.1145/1672308.1672325. URL: http://doi.acm.org/10.1145/1672308.1672325.

[7] C. Kachris and I. Tomkos. "A Survey on Optical Interconnects for Data Centers". In: *IEEE Communications Surveys Tutorials* 14.4 (Fourth 2012), pp. 1021–1036. ISSN: 1553-877X. DOI: 10.1109/SURV.2011.122111.00069.

[8] J. Touch. *Updated Specification of the IPv4 ID Field*. RFC 6864. http://www.rfc-editor.org/rfc/rfc6864.txt. RFC Editor, Feb. 2013. URL: http://www.rfc-editor.org/rfc/rfc6864.txt.

[9] Sebastian Elbaum, Gregg Rothermel, and John Penix. "Techniques for Improving Regression Testing in Continuous Integration Development Environments". In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM,

2014, pp. 235–245. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635910. URL: http://doi.acm.org/10.1145/2635868.2635910.

[10] H. Xu and B. Li. "RepFlow: Minimizing flow completion times with replicated flows in data centers". In: *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. Apr. 2014, pp. 1581–1589. DOI: 10.1109/INFOCOM.2014.6848094.

[11] Y. Zhou et al. "A Load Balancing Strategy of SDN Controller Based on Distributed Decision". In: *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. Sept. 2014, pp. 851–856. DOI: 10.1109/TrustCom.2014.112.