POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Towards a tight and fast interaction of data and control planes

Supervisors

Prof. Guido MARCHETTO

Prof. Alessio SACCO

Candidate

Daniele SARCINELLA

July, 2022

Acknowledgements

Il primo dei grazie va a mamma Dolores e papà Massimo. Senza il loro supporto, che non è mai mancato dal primo respiro che ho prodotto nel 1995, non sarei neanche qui a scrivere queste righe. Non si può descrivere l'amore e l'ammirazione che provo per voi. Grazie.

Il secondo è per mia sorella Roberta, che mi è da sempre amica e compagna di giochi. La mia prima insegnante quando avevo ancora 4 anni, a tratti mamma e a periodi alterni anche coinquilina. Sei la migliore sorella che si possa desiderare. Grazie.

Il terzo è per gli amici, tutti quelli che in un modo o in un altro mi hanno portato fin qui. Siete stati la mia benzina, mi avete tirato le orecchie quando necessario, ci siamo sostenuti a vicenda, ci siamo sempre voluti bene e non siete mai mancati. Questa cosa non la dimenticherò mai. Grazie.

Il quarto è tale solo per ordine temporale, ma non per importanza. E' per Chiara, una persona meravigliosa che ho avuto la fortuna di conoscere negli ultimi sei mesi, ma che mi sembra di conoscere da sempre. Il tuo sostegno, il tuo sorriso, il tuo affetto sono state e saranno ancora (mi dispiace per te) la cosa più preziosa per me e li conserverò per sempre nel mio cuore. Grazie.

L'ultimo lo dedico a tutte le altre persone che mi hanno voluto bene e che nel corso della mia vita hanno sempre creduto in me: nonne e nonni, zie e zii. Quelli che ci sono ancora e quelli che non ci sono più, spero di avervi resi orgogliosi di me, anche da lassù. Vi voglio bene e ve ne vorrò sempre. Grazie.

Daniele

Table of Contents

1 Introduction 1
- 1.1 Data plane and Control plane . . . 1
  - 1.1.1 Data plane . . . 1
  - 1.1.2 Control plane . . . 2
- 1.2 Traditional networks and related issues . . . 3
- 1.3 Newer issues driven by virtualization . . . 4
- 1.4 Path toward Software Defined Networks . . . 5
- 1.5 Software Defined Networks . . . 6
  - 1.5.1 Architecture . . . 6

2 Background 9
- 2.1 P4 . . . 9
  - 2.1.1 Architecture . . . 10
  - 2.1.2 Differences with OpenFlow . . . 12
- 2.2 Deep Reinforcement Learning . . . 13
  - 2.2.1 Reinforcement Learning overview . . . 13
  - 2.2.2 Reinforcement Learning glossary . . . 14
  - 2.2.3 Bellman equation of optimality . . . 15
  - 2.2.4 Tabular Q-Learning . . . 16
  - 2.2.5 Deep Q-Network . . . 17

3 System design 21
- 3.1 Routing problem . . . 21
- 3.2 Mathematical Model . . . 22
- 3.3 DRL formulation . . . 23
  - 3.3.1 Key elements of DRL formulation . . . 23
  - 3.3.2 Reward function . . . 24
  - 3.3.3 Features encoding and neural network . . . 26
  - 3.3.4 Pseudo Algorithm during training . . . 28
  - 3.3.5 Pseudo Algorithm execution time . . . 29

4 Implementation 31
4.1 Architectural overview . . . 31
4.2 P4 Application . . . 32
4.2.1 Ingress processing . . . 33
4.2.2 Egress processing . . . 36
4.3 ML Controller . . . 37
4.3.1 Externs implementation . . . 38
4.3.2 ML Controller architectural overview . . . 38
4.3.3 Concurrent Circular Buffer of addresses . . . 40
4.3.4 PyModule . . . 41
4.4 DRL Module . . . 42
4.4.1 OpenAI Baselines . . . 42
4.4.2 DQN implementation . . . 43
4.5 Network Environment . . . 45
4.5.1 Reset function . . . 46
4.5.2 Step function . . . 47
4.5.3 Issues of N routers on the same machine . . . 48

5 Related work 50
5.1 Q-Routing . . . 50
5.1.1 Key Features . . . 50
5.1.2 Differences with ML Router . . . 51
5.2 Backpressure . . . 51
5.2.1 Key Features . . . 51
5.2.2 Differences with ML Router . . . 52
5.3 DQRC . . . 52
5.3.1 Key Features . . . 52
5.3.2 Differences with ML Router . . . 53

6 Evaluation 54
6.1 Test scripts . . . 54
6.2 The Static Router . . . 56
6.3 ML Router comparison with Static Router . . . 56
6.3.1 Retransmissions . . . 57
6.3.2 Throughput . . . 59
6.3.3 Latency . . . 61

7 Conclusion 63

A Hyper-parameters tuning 65
A.1 Hidden layers . . . 65
A.2 Lambda 1 . . . 66
A.3 Lambda 2 . . . 67
A.4 Lambda 3 . . . 68
A.5 Lambda 4 . . . 69
A.6 Action History length . . . 70
A.7 Future Destinations length . . . 71
A.8 Learning rate . . . 72
A.9 Total timesteps . . . 73
A.10 Exploration fraction . . . 74
A.11 Learning starts . . . 75
A.12 Buffer size . . . 76
A.13 Discount factor . . . 77

B Updates periodicity tuning 78
B.1 Periodicity equal to 1000 . . . 78
B.2 Periodicity equal to 5000 . . . 81
B.3 Periodicity equal to 10000 . . . 83
B.4 Periodicity equal to 20000 . . . 85
B.5 Periodicity equal to 50000 . . . 87

List of Figures 90

Bibliography 93

Abstract

With the birth of Software Defined Networking (SDN) a new impulse was given to networks innovation, making it possible to decouple data plane and control plane. Several technologies were born with the aim of speeding up the implementation of new features, exclusive task of hardware manufacturers for years.

One of them is P4, a solution capable of enabling the general-purpose programming paradigm in the networking world, thus telling the devices what exactly to do with incoming packets. While this language allows to easily instruct network devices how to handle packets (data plane), the network rules are decided by an external process (control plane), which is often centralized.

With this thesis, we propose a new concept of SDN control plane that breaks the main pillar of having one logical centralized one, providing each P4-enabled switch with its own decision capabilities. In particular, we consider routing rules to be decided individually by means of a Reinforcement Learning (RL) model.

This learning technique is a trial and error basing on feedbacks of the environment to the chosen actions. Each network device has its (limited) view on the topology and can adapt the routing strategy to the current network conditions (RL state) to perform congestion-aware decisions. Since the such an RL state can be high-dimensional, we apply a Deep Reinforcement Learning approach, based on the approach of a Neural Network (NN) for the process.

We empower the P4 language to interact with this external program running on the same device so that changes may be fast and in parallel with all the other devices of the network.

The results obtained on Mininet emulator validate that when the network congestion starts growing, our solution helps keep the throughput at a high level.

Chapter 1

Introduction

The way we intend networks has changed greatly during the last 20 years, thanks to new solutions that have given impulse to their evolution. Before introducing the significant change to our networking idea, we must clarify some simple concepts.

1.1 Data plane and Control plane

The distinction between **data plane** and **control plane** will be very important during the whole reading of this thesis. Not by chance, they both figure in the title of this work, so it's worth explaining what do they actually mean and how they come to be so important.

1.1.1 Data plane

Data plane is the piece of software that has to switch (or route) packets according to a given policy, which is not under its control. It's in charge of manipulating network packets as fast as possible. Packets handled by data plane usually travel **through** the device. It's meant to perform **elementary operations** at a **high frequency** and thus it needs very few lines of code (LOCs), but a great processing power.

It's usually implemented in hardware, not general-purpose CPUs as they are few programmable and plenty of useless features. They indeed are not designed for these execution patterns. Dedicated memories such as **BCAMs, TCAMs** or dedicated processing units such as **ASICs, NPU** and **FPGA** are the most widely known solutions.

Independently from the physical architecture adopted, the packet is processed

by the network device and can follow two possible paths:

- **fast path**: all the packets that need minimal processing will follow this path. Almost all packets belonging to the data plane will travel through fast path. It's optimized for performances, can also be implemented in software with a highly optimized code.
- **slow path**: all the packets that generate exceptions in the fast path will follow the slow path along with packets of the control plane. They may need a more complex and deep processing and the slow path can accomplish this task.

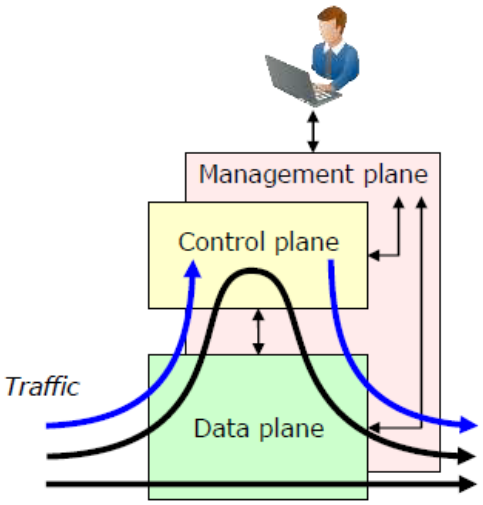

Figure 1.1: Slow path and Fast path [1]

A typical example that can make it clearer is the **load balancer**. To route packets properly, it must perform a lookup on the **session table**, which is a table too huge to be contained into an L3 cache. To reduce overheads due to memory accesses, it's a good practice to place common entries in the fast path and all the others in the slow path, after a proper traffic pattern analysis.

1.1.2 Control plane

Control plane controls the way data plane must behave. For example, it's in charge of configuring routing tables that network devices use to guarantee packets routing.

Packets handled by the control plane usually originate and terminate on a device,

it's really unlikely that they're going to traverse it. It performs much more complex operations at a low frequency (more LOCs needed, but also less processing power). It's always implemented **in software** with general-purpose CPUs and standard memories (DRAM).

1.2 Traditional networks and related issues

In the traditional networking model, control and data plane are **combined** in a network node. Control plane guides the paths used by the data plane, exchanging control packets to configure it. Once the policy of the forwarding has been defined, the only way to change it is via changes to the configuration of the devices. This is a **first limitation** network administrators have faced since from the very beginning of the internet as we know it today [2].

Computer networks are difficult to be managed also because of a variegate hardware with different roles and from different vendors. The control software run by these devices is usually **proprietary** and the configuration interfaces are very different from one vendor to another one and many times even between different products from the same vendor.

Last, but not least, this methodology of development of network devices, protocols and applications as a huge proprietary packet where each component is strictly dependent one to the others comes to be **closed to innovation**. It indeed operates in a totally different way with respect of general-purpose computing, where each vendor is specialized in only one of these components production, enabling competitiveness, dynamism and innovation. All these components have **clearly separated interfaces** with **different players** interacting to give the end user always **better products**.

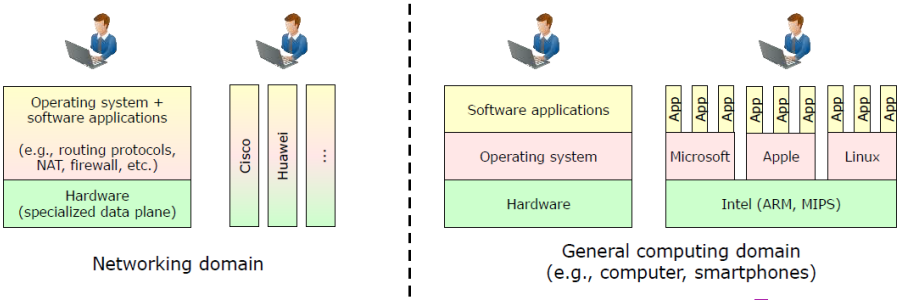

Figure 1.2: Network computing vs general-purpose [3]

1.3 Newer issues driven by virtualization

It's worth mentioning some new needs in networks administration led by **virtualization**. Virtualization can be defined as a flexible way to share hardware resources between different operating systems. It allows administrators to fully exploit the hardware at their disposal **consolidating** many application on one single physical machine and to enable **flexibility** as they can be migrated from one machine to another one.

Server virtualization has become really important when dealing with big data applications and to implement cloud computing infrastructures, but creates some problem with traditional network architectures. The **flexibility** coming from the adoption of virtualization techniques makes the administration of network configurations such as **VLANs** harder, as the control logic for each switch is co-located with the switching logic.

Another big change with respect to traditional networks is that **traffic** is no longer only "**vertical**", i.e. server to client with a very specific pattern in which clients send small requests and servers reply with huge amount of data. Data centers have started to increase their "**horizontal**" traffic due to virtual machines migration and the rise of **microservices** programming pattern which foresees smaller applications interacting with each other to provide a service to the external users. *"These server-toserver flows change in location and intensity over time, demanding a flexible approach to managing network resources."* [4]

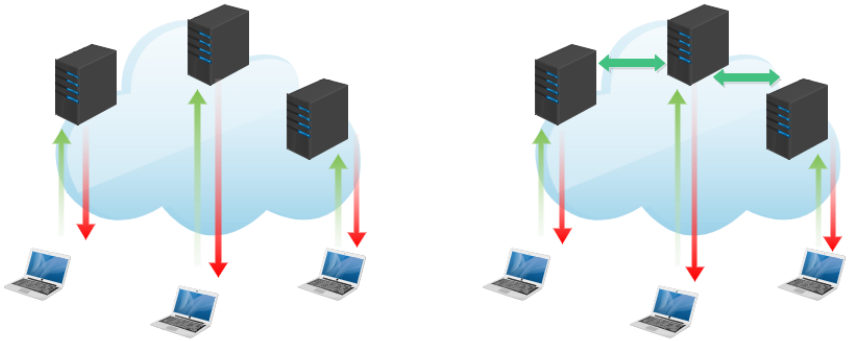

Figure 1.3: client-server pattern vs cloud pattern

1.4 Path toward Software Defined Networks

The big challenge for networking researchers was to make networks more programmable. Enabling **programmability** on the networks could have boosted **innovation** in their management and **new applications** development.

In an article written by Feamster and others [5], the path that led to the birth of Software Defined Networks (SDN) is divided into three **sub phases**. It's worth noticing that the problems cited in the previous section (1.2) arose yet at the mid of the 90s. The internet was taking over and to follow up on the development of their new protocols, researchers had to submit their ideas to the **Internet Engineering Task Force** (IETF) to standardize them. The process was **slow** and very often frustrating.

The response of many researchers was to try some innovative idea to make networking computing closer to **general-purpose computing**, trying to follow the path that led from mainframes to the general-purpose market as we know it today.

Active networking (phase 1) has been the first try made on this side, to envision a programming interface that exposed resources on individual nodes. In particular, among the two programming models pursuit at the time, it's interesting to focus on **programmable switch/router model**, that aimed at establishing the code to be executed by the device out-of-band [6, 7].

The motivations that gave impetus to active networking are very similar to those that led to SDNs [8, 9]: **timescales** necessary to deploy new services, more **control** needed on management, will to have an **experimentation-friendly** platform.

The concept of **separating control plane and data plane** will be one of the pillars of SDNs and appeared to be necessary even before their birth. In the early 2000s (phase 2), well known problems from the previous decade became tangible. **Reliability**, **quality of service** (QoS) and **predictability** was something that IP Networks as they are couldn't provide in any way.

Further, the **backbones** of the Internet were growing up in dimensions and became hard to be managed. The **speed of their links** grew fast and manifacturers started implementing data plane in hardware, thus separating it physically from the control plane (still software based).

These new trends led to innovations like **open interfaces** between control and data plane and logically centralized control of the network, thus opening up the

way to Software Defined Networks (phase 3).

1.5 Software Defined Networks

Software Defined Networks was born with the aim of:

- **influencing packets paths** on the network, by means of a new strategy which is different from those previously adopted (such as MPLS)
- **decouple hardware, OS and applications** in the world of networking, like in the general-purpose one to give market a new vivacity

The main **pillars** SDNs base their existance on are three:

- **elementar** physical devices
- **centralized** and **logically unique** control plane
- **context-based** forwarding

We say **logically unique** because physically we can implement a controller in each device to enable **robustness**, **scalability** and **less computational time**, actually contradicting someway the elementar physical devices pillar. Of course **consistency** in the information among all the phyisical control planes must be assured.

1.5.1 Architecture

The main actor of an SDN Architecture is, of course, the **controller**. It's like the Operating System of the network, it's in charge of decoupling the applications from the underlying hardware.

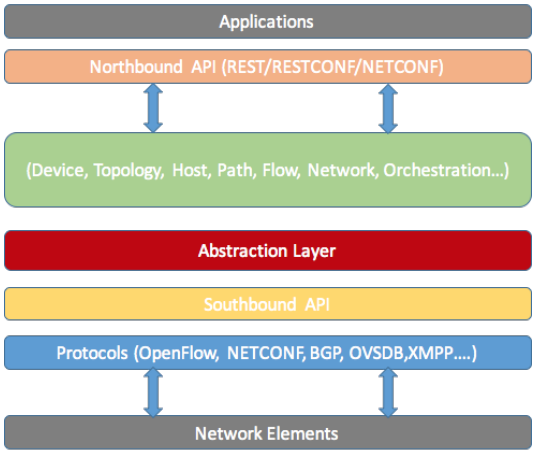

Figure 1.4: SDN Network

The controller starts from the **south bound interface**, endowed with protocols to talk with heterogeneus network devices and ends up in the **north bound interface**, which exposes APIs (REST, RESTCONF, NETCONF, ...) to the management applications that can exploit exposed services. From this interface, application can declare **high-level rules** that will be imposed at a lower level on the physical devices.

In the middle, there's a **service abstraction layer** which hides the complexity of the different protocols on the south bound interface and makes interaction with them **transparent** to the actual implementation.

The **heterogeneus network devices** are meant to be **simple packet forwarding hardware** that can be summarized as a lookup table based on the so called **context forwarding**. The match can be done on IP addresses (router like), MAC addresses (switch like) or a combination of several parameters involving L4 headers too.

If none of the table entries matches the packet, it can be sent to the controller which will perform a **runtime decision** which is going to be surely **expansive** in terms of computational time, especially in the case the controller is physically unique in the network.

Of course it's a technology which is not faults-tolerant and thus it's better to adopt it in static contexts such as **data centers** and **virtual networks** internal to the server, used to regulate the coexistence of different VMs.

Nowadays, SDNs lost those pillars they were built on and became a paradigm that lets network administrators handle in a dynamic and automatic way the control of a huge number of network devices, by means of third-party applications, high-level languages and APIs. As mentioned before, also devices often are not "stupid" as they were meant to be at the beginning due to **robustness**, **scalability** and **ability to react autonomously** to small-scale events need. As an obvious consequence, control plane is often only logically unique.

The only real pillar that has resisted over time is the **programmability** of the **control plane**. All started from there, after all.

Chapter 2

Background

The two main concepts that must be introduced before getting into the details of the work made in the context of this thesis are **P4** and **Deep Reinforcement Learning**. The opportunities coming from the adoption of P4 and Deep Reinforcement Learning techniques are at the very base of the proposed solution.

2.1 P4

P4 stands for **Programming Protocol independent Packet Processor**. It was born with the aim of decoupling the networking applications from the underlying hardware to enable a **general-purpose programming paradigm** in the networking field, i.e., given a packet, tell the fully programmable device (switchs, NICs, routers, filters, ...) what to do, at a higher level.

The three pillars P4 is based on are:

- **Reconfigurability**: the controller must be able to redefine the packet parsing and processing in the field
- **Protocol independence**: protocols can be inferred by the programmer as the parsing in under his control
- **Target independence**: programs are written at such a level of abstraction that they can be run on different devices, just like in general-purpose programming. The compiler will be in charge of translating these instructions at a lower level, depending on the specific target device [10]

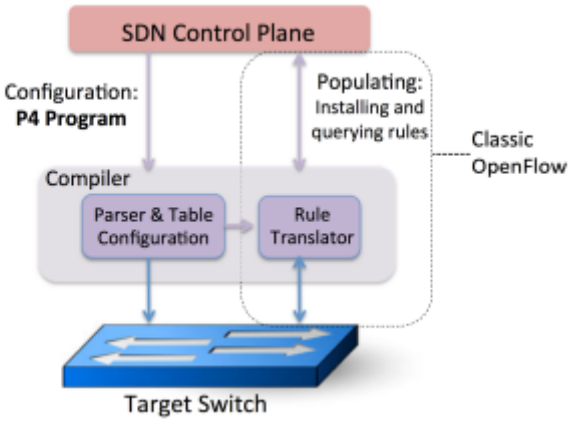

Figure 2.1: P4 is a language to configure switches. [10]

2.1.1 Architecture

P4-16 introduces the concept of an **architecture**, i.e., an API () to program a target (network device) [11]. The architecture basically defines which blocks compose the chain the incoming packet will pass through. The programmer can implement its own logic in each of these blocks, just paying attention to respect the given interface.

Each architecture defines the metadata it supports, including both standard and intrinsic ones and a list of **"externs"**, i.e., blackbox functions whose interface is known [11]. They will be very relevant with respect to the solution we have implemented. They are used mostly to implement complex operations that are not P4-native as P4 was thought to offer a very simple "instruction-set" to the programmer.

Finally, it exposes the interface the programmer has to take into account when coding a networking application that must fit that architecture. Below we can see an extract from the **v1model** architecture defined in the p4lang repository [12].

```
package V1Switch<H, M>(Parser<H, M> p,
                       VerifyChecksum<H, M> vr,
                       Ingress<H, M> ig,
                       Egress<H, M> eg,
                       ComputeChecksum<H, M> ck,
                       Deparser<H> dep
                       );
```

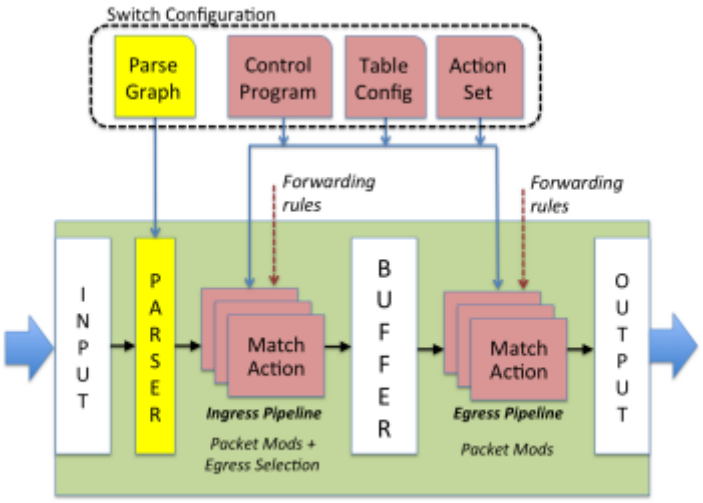

Figure 2.2: The abstract forwarding model. [10]

As we can see, the V1model architecture defines 5 blocks in its chain, with different capabilities and duties. Leaving aside checksum related operations, let's give a glance to the 3 key blocks of each architecture.

The first block in the chain is the **parser**. It handles first the incoming packet and uses a state machine to map packets into headers and metadata. *"The model makes no assumptions about the meaning of protocol headers, only that the parsed representation defines a collection of fields on which matching and actions operate"* [10].

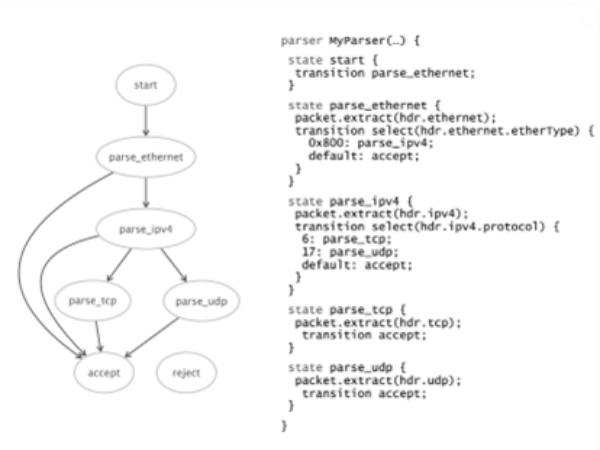

Figure 2.3: How a parser works. [11]

The extracted header fields will then be forwarded to the **match+action tables**. These tables are divided into **ingress** and **egress**, which both can modify the extracted fields. Plus, the ingress block has to determine the outcoming queue the packet must be appended to (i.e., it has to choose the out port and so the next hop).

The additional information that can be carried between stages is called **metadata**, which is treated identically to packet header fields extracted by the parser. For example, some metadata are ingress port, egress port, queuing metadata (that we have used to save queuing time, useful for our solution) and more [10, 13, 11].

Trivially, in a common router implementation, ingress processing will perform a **longest prefix match** lookup to select the right entry from the routing table, adjust extracted headers fields (MAC source, MAC destination of Ethernet header and Time-To-Live of IP header) and put the packet in the correct queue for egress processing.

Ingress and egress are also called **control blocks** and are made up of:

- **tables**: can match on one or multiple keys in different ways. Match types are specified in the P4 core library and in the architectures. They're populated by the control plane [11]
- **actions**: *"are code fragments that describe how packet header fields and metadata are manipulated"* [13], same meaning of functions in a common programming language
- **control flow**: *"expresses an imperative program that describes packet-processing on a target"* [13], it's like C control flow, but deprived of loops

Control flow enables interaction with tables, validation and computation of checksums, packet cloning or recirculating and interaction with the control plane.

2.1.2 Differences with OpenFlow

OpenFlow and P4 pursued different paths toward networks programmability. In particular, OpenFlow was born with the main aim of separating the control plane from the data plane, offering a common interface to different vendors devices, most of which implemented with fixed-functions ASIC circuits. OpenFlow adopts the traditional **bottom-up** approach: an OpenFlow controller adapts itself to the switch features, exposed in the feature reply message which comes, eventually, right after a feature request sent by the controller.

P4 tried instead to **revert** the typical bottom-up approach adopted when dealing

with data plane functions. *"Rather than have the switch tell us the limited set of things it can do, P4 gives us a way to tell the switch what it should do, and how it should process packets"* [14]. The **P4 Language Consortium** itself explains the reason of this new approach in its website's homepage: *"Before P4, vendors had total control over the functionality supported in the network. And since networking silicon determined much of the possible behavior, silicon vendors controlled the rollout of new features (e.g., VXLAN), and rollouts took years. P4 turns the traditional model on its head. Application developers and network engineers can now use P4 to implement specific behavior in the network, and changes can be made in minutes instead of years"* [15].

According to the Open Networking Foundation, there's no reason to think that P4 will make OpenFlow **obsolete**, as there are many fixed-functions switches that need OpenFlow as P4 can't be used in that case. They can rather **work together** when dealing with hybrid networks which may contain both fully programmable and fixed-functions switches. In such a scenario, **OpenFlow** would become just a **possible implementation** of a P4 program, so much so that an implementation of OpenFlow in P4 has been already provided [14, 16].

2.2 Deep Reinforcement Learning

Deep Reinforcement Learning (**DRL**) is a field of machine learning which combines the usage of **Reinforcement Learning** and **Deep Learning**. For our purposes, it's sufficient to say that Deep Learning is a methodology based on the usage of **Artifical Neural Networks** with Representation Learning. We're going to focus more on what Reinforcement Learning is and how it's integrated with Deep Learning techniques considering the chosen algorithm (Deep Q-Network).

2.2.1 Reinforcement Learning overview

"Reinforcement Learning (RL) is a subfield of machine learning which addresses the problem of automatic learning of optimal decisions over time. This is a general and common problem studied in many scientific and engineering fields" [17].

The key idea behind RL is to replicate something like the **Pavlovian conditioning** so that an agent (its meaning will be explained in a moment) can learn what to do or not to do in a certain situation [18]. It will just learn from its **experience**, thanks to a good reward received for a good behaviour. Just like dogs learn which one is the best thing to do, given a certain situation (a command, a gesture from their owners), thanks to a food prize that rewards the desired behavior.

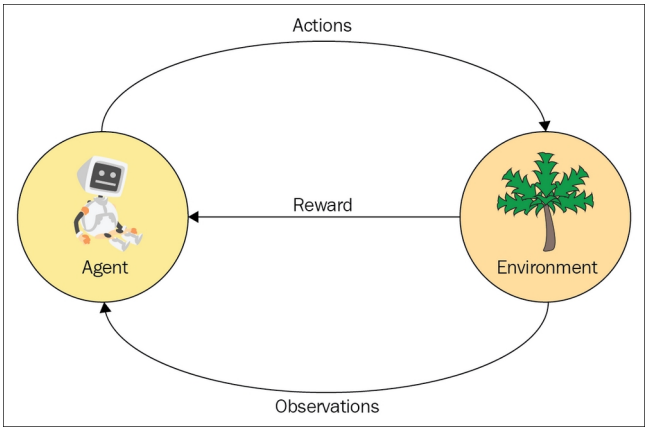

Figure 2.4: RL entities and their communication. [17]

2.2.2 Reinforcement Learning glossary

To understand what's behind RL functioning, it's crucial to give some definitions about its key features.

Agent

It's in charge of interacting with an environment performing some actions to get a reward in change. In a videogame the agent could be the character moving across the map to collect moneys trying to maximize his score.

Action

Mentioned right above, the action is picked from the set of all the possible actions. Performing an action means to move from a state to another one (moving left, moving right, jumping, crouching, ...).

State

"Concrete and immediate situation in which the agent finds itself" [19], it stores information about what's relevant about the environment surrounding the agent or about the agent itself (think about health value of a character in a videogame).

Environment

"The environment is everything outside of an agent. In the most general sense, it's the rest of the universe" [17]. It interacts with the agent taking actions as inputs and returning a new state and a reward as output.

Reward

Is a feedback given by the environment in response to an action performed in a given state. *"Reward is local, meaning, it reflects the success of the agent's recent activity, not all the successes achieved by the agent so far"* [17].

Policy

It basically maps a state to the action that is supposed to give the highest reward. It can be seen as a strategy.

Q-Value

Q-Value function maps a reward to any combination of state and action, under a certain policy π.

$$Q_\pi(s, a) = r \tag{2.1}$$

2.2.3 Bellman equation of optimality

To talk about **Q-Learning**, one of the most widely known RL algorithms, we need to introduce first the concept of **Bellman equation of optimality**.

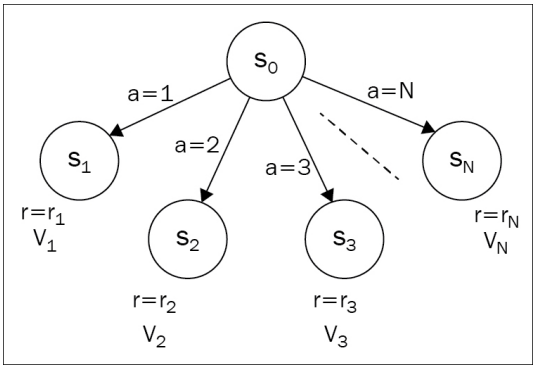

Figure 2.5: An abstract environment with N states reachable from the initial state. [17]

We assume that the agent observes state s_0 and, doing a certain move (action), it will **transit** to a state s_1 with 100% of probability. The value given to a certain action, starting from this observation would be:

$$V_0(a = a_i) = r_i + V_i \tag{2.2}$$

So the **best possible action** we can choose will be the one that maximizes this value:

$V_0 = \max_{a \in 1..N}(r_a + \gamma V_a)$

V_a in the previous equation stands for the **value of the next state**. It has been **discounted** by factor γ to give more importance to the immediate reward coming from action i.

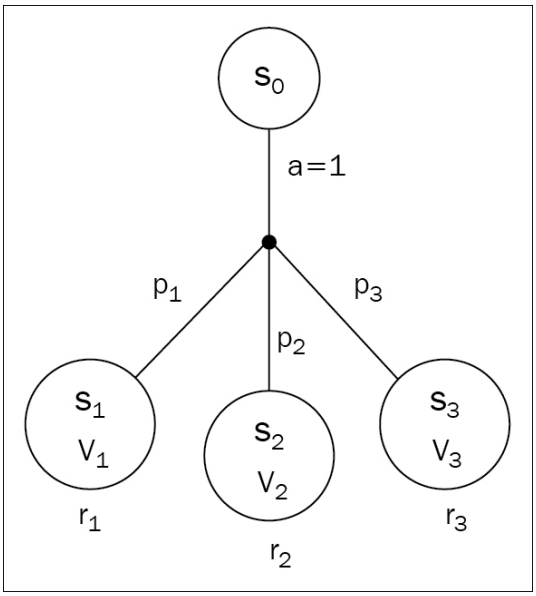

Figure 2.6: An example of the transition from the state in a stochastic case. [17]

As we can see in the picture right above, it's very unlikely that an action, whatever it is, always leads to the same state. It will rather give us different scenarios with different probabilities and **values**. We'll then better refer to V_0 as the maximum **expected value** of $V_0(a_i)$:

$V_0 = \max_{a \in 1..N} E[(r_a + \gamma V_a)]$

2.2.4 Tabular Q-Learning

Q-Learning is an algorithm that has been proposed by Cristopher Watkins in 1989 [20]. It introduced the concept of **value of an action**. This function is called Q and couples a state-action tuple to a value:

$$Q(s,a) = V_s(a) \Longrightarrow Q(s,a) = r_{s,a} + \gamma V_{s'} \tag{2.3}$$

So we can say that V_s is equal to the **maximum**, over all the possible actions, **Q-value** for that state. We can so write that:

$Q(s,a) = r_{s,a} + \gamma V_{s'} \implies Q(s,a) = r_{s,a} + \gamma \max_{a' \in A} Q(s',a')$

"Q values are much more convenient in practice, as for the agent it's much simpler to make decisions about actions based on Q than based on V. In the case of Q, to choose the action based on the state, the agent just needs to calculate Q for all available actions, using the current state and choose the action with the largest value of Q. To do the same using values of states, the agent needs to know not only values, but also probabilities for transitions. In practice, we rarely know them in advance, so the agent needs to estimate transition probabilities for every action and state pair."[17]

In the **Q-Learning** algorithm we don't iterate over all the possible states to find all of their Q-Values, but we rather use states coming from interaction with the environment.

Algorithm 1 Tabular Q-Learning algorithm

$Q(s,a) \leftarrow emptyTable$
$s \leftarrow initialState$
$convergence \leftarrow False$
while $convergence == False$ **do**
  $a = getAction(policy)$
  $(r, s') = makeAction(s,a)$ ▷ r is the reward, s' the new observed state after action a
  $Q(s,a) = (1-\alpha)Q_(s,a) + \alpha(r + \gamma \max_{a \in A} Q(s',a')$
  $convergence = checkConvergence()$
end while

What we can notice is that, updating Q(s,a) value, we don't replace it with the new value as the training may become unstable. We use a **"blending"** technique to update it with approximations, that means averaging between old and new values using **learning rate** α [17].

It's demonstrated that this algorithm will meet its **convergence condition**, as long as there are **finite number of states**.

2.2.5 Deep Q-Network

There are many scenarios in which state space is so big that it can't be considered finite, thus making the adoption of Q-Learning unreliable. What **Volodymyr Mnih** and others [21] presented in 2013 was a variation to Q-Learning algorithm

that integrates it with Deep Learning.

The proposed solution foresees the usage of a non-linear representation to map the state-action tuple into a continuous value, what in machine learning is called a **regression problem**, i.e. *"the task of approximating a mapping function (f) from input variables (X) to a continuous output variable (y)."* [22]

Exploration vs Exploitation dilemma

In this way, we don't need to store an almost infinite number of Q-values, but we can use a **Deep Neural Network** (in most of cases) to approximate them. The approximated Q-function can be the source of our moves so that, rather than picking random actions with low chances of succeeding, we pick the one with the **highest expected outcome**. The drawback of such an approach is that the agent could **stuck into local optimum** without exploring enough.

To overcome this possible issue, we must alternate **exploration** (attempting to discover new features about the world by selecting a sub-optimal action) and **exploitation** (using what we already know about the world to get the best results we know of). [23]

One of the possible methods that we can adopt is known as **epsilon-greedy** method. According to the hyperparameter $\epsilon \in [0,1]$, there's ϵ **probability** of choosing a **random action** and $(1 - \epsilon)$ probability of picking the best one found so far. In particular, the value of epsilon can be **annealed** over a certain number of timesteps so that the exploration can assume a variable importance over the whole duration of the training.

Replay Buffer

Pretending that we're solving a **regression problem**, as mentioned before, means that we should satisfy two main requirements:

1. training data must be **independent**
2. training data must be **identically distributed**

The issue is that our samples are **neither independent nor identically distributed**. Saving previous experiences into a buffer and sampling random data from it instead of using last experiences is the solution to this problem and the buffer is known as **replay buffer**. It *"allows us to train on more-or-less independent data, but data will still be fresh enough to train on samples generated by our recent policy."* [17]

Target Network

Another element that must be introduced before introducing the final form of DQN alforithm is the concept of **Target Network** $\widehat{Q}$. At each step we update Q(s,a) using the value of Q(s',a') (see Algorithm 1), but they may be very close to each other and the neural network may found difficult to distinguish between s and s'.

This scenario can cause **instabilities** in the algorithm so that Q(s,a) updates will influence Q(s',a') and vice-versa. *"To make training more stable, there is a trick, called target network, when we keep a copy of our network and use it for the Q(s', a') value in the Bellman equation. This network is synchronized with our main network only periodically, for example, once in N steps (where N is usually quite a large hyperparameter, such as 1k or 10k training iterations)."* [17]

Final Algorithm

Algorithm 2 DQN

$Q(s,a), \widehat{Q}(s,a) \leftarrow random$
$replayBuf \leftarrow empty$
$\epsilon \leftarrow 1$
$s \leftarrow initialState$
$nsteps \leftarrow 0$
$updateFreq \leftarrow N$
$convergence \leftarrow False$
while convergence == False **do**
  $a = getAction(\epsilon)$
  $(r, s') = makeAction(s, a)$
  $minibatch = replayBuf.sample()$
  $i \leftarrow 0$
  while $i \neq minibatch.size()$ **do**
    $y \leftarrow r + \gamma max_{a' \in A} \widehat{Q}_{s',a'}$
    $loss \leftarrow (Q_{s,a} - y)^2$
    update Q(s, a) using SDG algorithm
    $i \leftarrow i + 1$
  end while
  if (nsteps % updateFreq) == 0 **then**
    $\widehat{Q} = Q$
  end if
  $nsteps \leftarrow nsteps + 1$
  $Q(s,a) = (1-\alpha)Q_(s,a) + \alpha(r + \gamma \max_{a \in A} Q(s', a')$
  $convergence = checkConvergence()$
end while

Chapter 3

System design

In this chapter we're going to introduce the problem of **routing**, the **target network** adopted for testing purposes and all the elements related to the **proposed solution**. This section won't contain any implementation detail, as they will be stressed later in the next chapter.

3.1 Routing problem

The problem of routing belongs to the family of **optimization problems**, which aim at minimizing or maximizing a certain function, according to the goal of the target problem.

To find a clear example, we can go back in 1958, when **Richard Bellman** wrote in the abstract of a very famous article: *"Given a set of cities, with every two linked by a road, and the times required to traverse these roads, we wish to determine the path from one given city to another given city which minimizes the travel time. The times are not directly proportional to the distances due to varying quality of roads and varying quantities of traffic"* [24].

The most straightforward algorithm that can be used to approach routing problem would be a **shortest path** algorithm (e.g, Bellman-Ford [24], Dijkstra [25]). It picks the route with the lowest weight that brings from starting point A to the end point B. However this solution doesn't suit a real scenario like the one cited by Bellman. "Quantity of traffic" in particular is something that we can't prevent, but we must take into account adopting some countermeasures.

3.2 Mathematical Model

In the case of interest of this thesis, we don't have cities and roads. We rather have a **network**, which is modeled as a directed graph $G(N, E)$ where N and E are respectively the **set of nodes** and the **set of edges** linking the nodes.

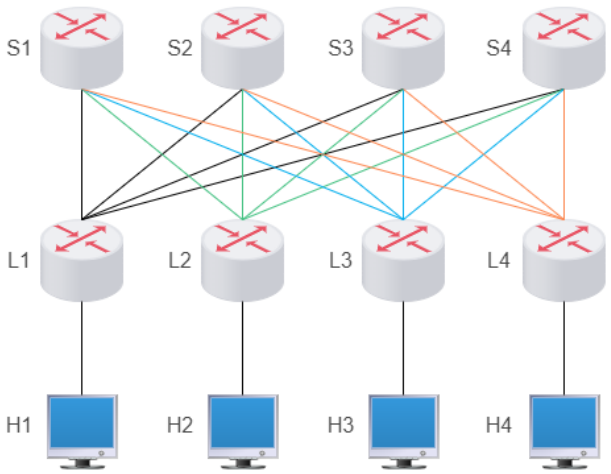

Figure 3.1: Adopted topology

Below we can see the topology adopted for the testing purposes. There are two layers of P4 Routers called respectively "S" for the **"backbone layer"** and "L" for how concerns the **"leaf layer"**.

Each router belonging to the backbone layer is connected with all those that belong to the leaf layer and vice-versa. Routers belonging to the same layer can't communicate with each other directly, but need to cross the layers boundary twice. Further, leaf nodes are connected directly to an **host** "H" numbered accordingly to the numeration of the L router.

The mission here is to **deliver packets** from one host to another, trying to **minimize the latency** and **maximize the throughput** over the network basing on some metrics that will be shown in the next section. Packets originating from one host reach the incoming queue of the leaf node directly connected to it. According to a **FIFO** (First In First Out) criterion, packets are then processed by the router and placed in one of the outgoing queues.

It's pretty clear that the longer the queues are (both incoming and outgoing), the longer will take the whole routing process for a specific packet. We will keep

this in mind for the design of the DRL solution.

Once out of the outgoing queue, the packet will reach another router on the network and so on, until it reach an endpoint (i.e., an host) of the network. At that point, the headers are stripped out and the actual payload of the packet can be processed by its recipient.

3.3 DRL formulation

As I've said in the introduction chapter, the aim of the provided solution is to break one of the pillars coming from the definition of SDNs: the **logically unique controller**. It means that there's not a "supervisor" external to the network which has a global view over the network and inject routes consequently (like an Openflow controller [26]).

The main idea is to treat each router as an **agent**, trying to accomplish its mission according to the metrics used in the **reward function**, in an environment where other agents are present trying to reach the same goal. This scenario is known as **Multi Agent Reinforcement Learning** problem (MARL) [27].

Routers are supposed to learn by trial and error how to route incoming packets and their decision will affect other agents decision too, in a **very dynamic context** where every little detail can change completely the behavior of the agents.

3.3.1 Key elements of DRL formulation

Let's introduce now the key elements of the solution, according to the glossary defined in the section 2.2.2.

Agent

The **agent** is each router in the topology. We have thus 8 agents, each one of them in charge of interacting with the environment to find the best output port for the packet that is being processed. The agent is a **P4 switch** (with routing functionalities though) that possibly communicates with the environment (according to the phase, will be explained later) to know the action that must be done.

Action

The **action** is a discrete number that can range from 1 to N, where **N is the number of ports used by the switch**. If the switch is asking the environments which action it has to perform, it's actually asking it for the outgoing queue to put the packet into.

State

The **state** S of the switch is composed by three main actors:

- **Current Destination**: the IP address of the current packet's destination, so basically the host that is meant to be the recipient for that packet
- **Future Destinations**: a list of the N ipv4 addresses belonging to the N next packets that have to be routed right after the packet being processed at that time
- **Action History**: a list of the last M actions adopted for the current destination

3.3.2 Reward function

The **reward function** is evaluated each time a given action for a certain state has been performed and its value will be used to update the Q-Value for that state-action couple. It takes into account two main factor:

- **queuing time**: the time the packet has spent in the output queue (chosen by the action)
- **distance**: of the chosen next hop from the final destination

While it's quite obvious that we want to minimize the time spent by the packets into the outgoing queues, **distance** of the next hop from the final destination is the only information that the agent knows about the global topology. It's not so important in the context of the chosen topology as it's small, but it may be necessary for **bigger networks**.

Distance is a fundamental metric for backbone routers to route packets in the **right direction**, for how possible. It isn't something that falls into a static routing case, since queuing time has the same importance of distance in the choice made.

And how do **leaf nodes** understand if they're routing packets properly or not? We actually didn't mention the other two actors of the reward function, which are two indicators:

- **Delivered indicator**: is set to 1 if the packet has been routed to the final recipient, to 0 otherwise
- **Dropped indicator**: is set to 1 if the packet has been routed to an host which is not the final recipient (dropped), 0 otherwise

Their value is always set to 0 in case of backbone routers, as they're not directly connected to any host. The final form of the reward function is then:

$$\begin{cases} R = rw_1 + rw_2 + rw_3 + rw_4 \\ rw_1 = \lambda_1 \cdot \delta_1 \\ rw_2 = -\lambda_2 \cdot q \\ rw_3 = -\lambda_3 \cdot \delta_3 \\ rw_4 = -\lambda_4 \cdot \delta_4 \cdot d \end{cases} \Rightarrow R = \lambda_1 \cdot \delta_1 - \lambda_2 \cdot q - \lambda_3 \cdot \delta_3 - \lambda_4 \cdot \delta_4 \cdot d \quad (3.1)$$

In the previous equation:

- $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ are **hyper-parameters** needed to give each piece of R the right weight, basing on the quality of the outcome of the training (i.e. **tuning** [28])
- δ_1 is the **delivered** indicator
- δ_3 is the **dropped** indicator
- δ_4 is the **backbone** indicator. It goes to 1 only when the router is a **backbone** one. Distance is indeed useless for leaf nodes as each of the backbone routers they're connected to is at the same distance from any destination
- d is the **distance** like it's been defined at the beginning of the section
- q is the **queuing time**

As we can see from the equation, the only positive member is rw_1, which is valid only when the packet has been correctly delivered. It means that λ_1 is the **maximum value** for R and can only be assumed for a **leaf router**. The other members of the equation rw_2, rw_3, rw_4 are all negative and the agent will try to learn how to make them as close as possible to 0.

In the case we're dealing with **backbone routers** indeed, the first component of R, rw_1, is always zero (as backbone routers can't deliver/drop packets) just like the third component, rw_3. In this scenario the agent will try to minimize the value of rw_2 and rw_4 and the maximum value of R will be **zero**.

Trying to summarize what said in the lines above, here there's a table which will make it more clear:

	leaf router	backbone router
rw_1	worst case: 0 best case: λ_1	always 0
rw_2	worst case: $-\lambda_2 \cdot maxqtime$ best case: $-\lambda_2 \cdot minqtime$	worst case: $-\lambda_2 \cdot maxqtime$ best case: $-\lambda_2 \cdot minqtime$
rw_3	worst case: $-\lambda_3$ best case: 0	always 0
rw_4	always 0	worst case: $-\lambda_2 \cdot maxdistance$ best case: $-\lambda_2 \cdot mindistance$

3.3.3 Features encoding and neural network

As we have modeled the problem as a MARL problem with DRL techinques, we expect that we're using a neural network to approximate Q values for a state-action couple.

Lists length

First of all, the number of incoming features, i.e. the ones that represent the state space, has to be determined. We said that the three components of the state representation are **current destination**, **future destinations** and **action history** for the current destination.

The length of future destination and action history lists has been object of study to identify the value that gives the best outcome in terms of **mean reward** (in the Appendix A at page 65 you can find data about testing phase). According to collected data, they've been both set to 2 so that we'll have the two next destinations to be reached and the last 2 decision taken for that destination.

What's actually almost not exact in what it's been said before is that we don't really give the neural network only the last 2 decision for the current destination. We rather provide each time the last two decision about **all the possible destinations**, otherwise those terms would change radically their meaning and it would be harder for the **neural network** to understand it.

Each **feature** of the state must **maintain its meaning** in every case, it can't represent two different things at different time (now it's the action history for destination d_1, now for destination d_2 and so on).

Lists encoding

Both action history's and future destinations items are **categorical features** [29]. It means that if the future destination is H1, then it's not H2, not H3, not H4. A good encoding practice for categorical features is the so known **One-Hot-Encoding**.

"The one-of-K or one-hot-encoding scheme uses dummy variables to encode categorical features" [30]. It means that if we have 4 categories an element can belong to, we will use 4 different **boolean indicators** (the dummy variables) and only one of them is set to `true`.

The actual encoding for those dummy variables is a 0, 1 encoding and, given that the two lists contain 2 elements each and that the maximum number of out ports in our topology is 5, while the number of possible destination is 4, we have all the elements to know the number of features incoming the neural network:

action history features $= ah\_f = hosts \cdot maxports \cdot ah\_length = 4 \cdot 5 \cdot 2 = 40$
future destinations features $= fd\_f = hosts \cdot fd_length = 4 \cdot 2 = 8$
current destination features $= cd\_f = hosts = 4$
features $= ah\_f + fd\_f + cd\_f = 40 + 8 + 4 = 52$

Neural Network

After different trials, the best structure for the neural network, given the test results, has been an architecture with **3 hidden layers** of respectively 128, 64 and 32 neurons, like in the picture below.

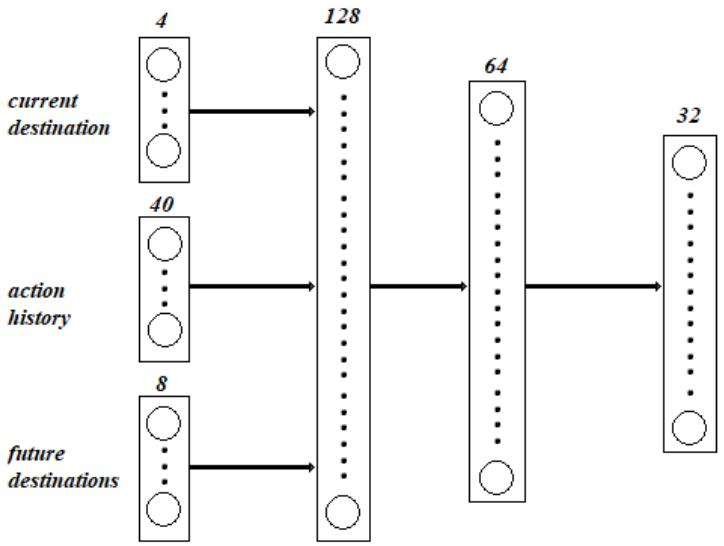

Figure 3.2: Neural network architecture

3.3.4 Pseudo Algorithm during training

Now that all the elements of the DRL modeling are known we have all what we need to introduce the **pseudo algorithm** used to train the agents.

The idea is that for each incoming packet the P4 code will have to save the **destination address** to populate its future destinations lists. Then, once it will be time for routing it, P4 will **always** ask the environment the port it has to be routed to, sending the information we spoke about previously in this section.

The **DRL algorithm** needs then to know what's the **reward** for that given action performed. While distance and possible deliver/drop actions are something that it can know by itself, **queuing time** is something that must be **communicated by the router**, again. For the moment, let's just say that the router notifies about the queuing time so that the reward can be stored. The actual **implementation** will be explained in the next chapter.

This is the procedure that will be repeated N times, where N is the number of **total timesteps** stated at the beginning of the training. This is another **hyperparameter** that has been object of tuning and it's been seen that 20000 timesteps are perfect for training agents.

This means that after having handled 20000 packets, routers are **supposed to have learnt** how to route them in each possible scenario they can face in the given network. Of course the hyper-parameters tuning process is strictly related to the adopted topology and their optimal values may vary in case we're going to change it.

Algorithm 3 Pseudo algorithm (training)

for agent in agents **do**
  $futuredst \leftarrow empty$
  $actionhist \leftarrow empty$
  $timesteps \leftarrow 20000$
  for `i in timesteps` **do**

```
p = readpacket()
futuredsts.add(p.destination)
outport = askport(p.destination, futuredsts, action-
hist[p.destination])
d = readdistance(outport, p.destination)
q = readqtime()
backbone = isBackbone()
delivered = isDelivered()
dropped = isDropped()
reward = makereward(d, q, backbone, delivered, dropped)
actionhist[p.destination].add(outport)
```

  end for
end for

3.3.5 Pseudo Algorithm execution time

Once the training is over we have a model that will let routers do their job just basing on the state they find themselves into, in terms of current destination, future destinations and action history.

After the training we must change something in the behaviour we've seen in the lines above. We can't think of asking the DRL algorithm for the action to do **at each packet received**. The resulting **overhead** would be too high and the solution couldn't ever be at least as performant as a traditional routing algorithm.

The resulting **tradeoff** is to adopt a **static routing** guided by the DRL algorithm by means of **periodical updates**. The aim is to let the router go at its full speed for great part of its execution time. This is another parameter that had

to be properly tuned in order to **maximize performances** and **adaptability** coming from the DRL guide too. The best results has been obtained updating the single route toward a specific destination each **10000 packets** directed to that host.

Of course it leads to a **sub-optimal policy**, the best scenario would be to have packets routed every time by the DRL algorithm at **zero costs**. But we know that it's **not feasible** in a real environment, also because each router has its own DRL underlying logic and very likely its **hardware** is very **simple** and not performances oriented as it's born to perform quite easy tasks (from the computational point of view).

Algorithm 4 Pseudo algorithm (execution time)

```
for agent in agents do
    futuredst ← empty
    actionhist ← empty
    timesteps ← 20000
    counters ← [0, 0, 0, 0]
    while EXECUTING do
        p = readpacket()
        counters[p.destination]++
        futuredsts.add(p.destination)
        if counters[p.destination] % 10000 == 0 then
            outport = askport(p.destination, futuredsts, action-
hist[p.destination])
            d = readdistance(outport, p.destination)
            q = readqtime()
            backbone = isBackbone()
            delivered = isDelivered()
            dropped = isDropped()
            reward = makereward(d, q, backbone, delivered,
dropped)
            actionhist[p.destination].add(outport)
            Update route for p.destination
        else
            Use routing table
        end if
    end while
end for
```

Chapter 4

Implementation

The implementation of the system has been the most **tricky** part of the solution. The target is to build a system capable of making a P4 application and a DRL algorithm **communicate** according to our needs to cooperate in making routing more performant.

4.1 Architectural overview

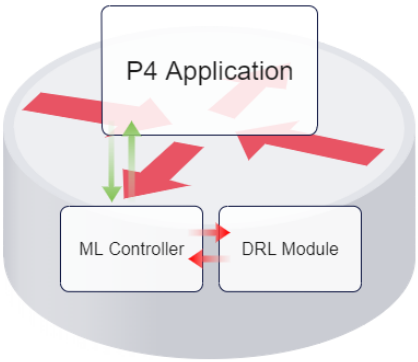

Figure 4.1: Overview of the ML Router

The architectural model is that **each router** is composed of **three main modules**:

- **P4 application**: logic of the network device
- **ML Controller**: exposes high-level methods accessible by the P4 application and data structures needed to communicate with the DRL module
- **DRL Module**: runs the learning algorithm to route packets

The most challenging implementation is the **communication** between these three modules:

- **P4-ML Controller**: we've exploited the existence of **externs** to declare high-level classes capable of acting as a **middleware** between P4 application and the DRL module

- **ML Controller-DRL**: these two modules, written in different languages, take advantage of **sockets** abstraction to establish a communication channel to exchange relevant data on, by means of a clear **communication protocol**

4.2 P4 Application

The P4 application is the core of the solution, as it represents the **logic** of the network device. It says what the device must do with incoming packets, according to the **pipeline** described in the Chapter 2 (page 9). It embodies the **higher level logic**, since it takes advantage of abstractions whose implementation can be found in the ML Controller.

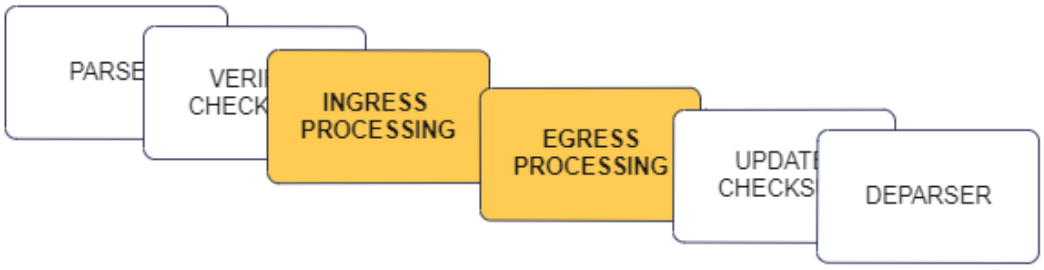

Figure 4.2: P4 Pipeline

As mentioned in Chapter 2, P4 exposes some functional blocks that can be programmed to obtain the desired processing logic. Let's go into details about **ingress** and **egress**, which are the most relevant ones. Let's just consider that **parsers** enable Ethernet, IP, UDP, TCP and ICMP headers, while we won't care about checksums verification and update for the purposes of the research. All the code listed by now can be found in my **ML-Routing repository** [31].

4.2.1 Ingress processing

The set of actions to be performed in the processing of a packet is wrapped into the **apply** control block.

```
apply {
    ...
    pushAddress();
    if (update_entry == 1) {
        choosePort();
        if (fw == 1) {
            getNeighbor(outP);
            if (fw == 1) {
                ipv4_forward_rl(dstMac, outP);
            }
            if (fw == 0) {
                mark_to_drop();
            }
        }
        if (fw == 0) {
            mark_to_drop();
        }
    }
    if (update_entry == 0 && hdr.ipv4.isValid()) {
        ipv4_lpm.apply();
    }

}
```

Listing 4.1: ML-routing/MLRouter/MLRouter.p4 [31]

Here basically you can find the actual implementation of huge part of the algorithm 4 at page 30. The first action you must pay attention to is `pushAddress()`.

```
action pushAddress() {
    ml_controller.pushAddr (
        hdr.ethernet.dstAddr,
        hdr.ipv4.dstAddr,
        meta.identifier,
        meta.valid_bool,
        update_entry
    );
}
```

Listing 4.2: ML-routing/MLRouter/MLRouter.p4 [31]

It's needed to **add the current destination address** into the **future destinations** data structure. This data structure is not under the direct control of the P4 application, but can be accessed by means of the interface offered by the **ML Controller**.

The parameters given to the method exposed by the controller are:

- **Ethernet destination address**: basically it's one of the router's MAC addresses. We use it to identify the current router since the implementation has been done running **all the devices** on the same physical machine. This means we needed a way to understand which device is running the code. In an implementation where all the routers are actually different physical devices this parameter is actually **useless**.
- **IPv4 address**: it's the address that must be pushed in the future destinations queue
- `meta.identifier`: it's a **metadata** (thus can be propagated through the whole pipeline, enabling information sharing among different blocks) that will be set equal to the **index** of the pushed address into the queue. Will be useful in the egress block to pop the right entry
- `meta.validbool`: it's another **metadata** that will be set equal to one only if the address is really valid
- `update_entry`: it's a **local variable** which is set to 1 only if the counter for the given destination has reached the established **update frequency**. In that case a call to the `choosePort()` action is done, otherwise the static table is looked up.

Let's dive into the `choosePort()` action:

```
action choosePort() {
    ml_controller.getOutputPort (
        hdr.ethernet.dstAddr,
        meta.identifier,
        meta.valid_bool,
        outP,
        fw,
        meta.id
    );
}
```

Listing 4.3: ML-routing/MLRouter/MLRouter.p4 [31]

This is the action responsible for getting an output port for the given packet asking the **interaction with DRL module**. Once again the action wraps an API exposed by the ML Controller. It takes as arguments:

- **Etherned destination address**: to identify the current router
- `meta.identifier`: to know which one of the addresses included in the future destinations queue is the current address (the same address may be included very likely more than once)
- `meta.valid_bool`: only if it's been set previously to 1 the method actually does the request
- `outP`: **local** variable that will contain the value of the output port to route the packet on
- `fw`: **local** variable that is set to 0 if, for some reason, the request had a bad response. In that case the packet is dropped
- `meta.id`: **metadata** needed to keep track of the identifier of the reward associated to the choice made (the mechanism behind the action-reward functioning will be explained later)

The last remarkable action that will be shown is `getNeighbor(outP)`:

```
action getNeighbor(bit<9> port) {
    dstMac = hdr.ethernet.dstAddr;
    ml_controller.getNeighborMac(dstMac, port, fw);
}
```

Listing 4.4: ML-routing/MLRouter/MLRouter.p4 [31]

It wraps the ML Controller `getNeighborMac(...)` API which needs as parameters:

- **Ethernet destination address**: will be used first to **read** the current destination address to identify the router and then to **write** the MAC address of the interface connected to the given port
- `port`: port whose MAC of the connected interface must be known
- `fw`: local variable set to 0 if the destination address couldn't be resolved

4.2.2 Egress processing

Let's take a look at the egress processing and the **two** relevant operations that characterize its apply block.

```
apply {
    ...
    popAddress();
    sendReward();
}
```

Listing 4.5: ML-routing/MLRouter.p4 [31]

The first of the two defined actions is `popAddress()` and is the dual function of `pushAddress()` seen in the ingress processing block. Once the packet is going to exit the outgoing queue, its address is popped out from the **future destinations** queue. Here there is its implementation:

```
action popAddress () {
    ml_controller.popAddr (
        hdr.ethernet.srcAddr,
        meta.identifier,
        meta.valid_bool
    );
}
```

Listing 4.6: ML-routing/MLRouter/MLRouter.p4 [31]

The wrapped API exposed by ML Controller is `popAddr(...)` and takes three arguments:

- **Ethernet destination address**: to identify the current router
- `meta.identifier`: is the **metadata** defined in the previous block needed to pop the right entry in the future destinations queue
- `meta.valid_bool`: is the **metadata** that states if the operations must be done or not, basing on the previous outcomes

The next important action is `sendReward(...)`, another wrapper for a method of ML Controller, which is needed to communicate to the underlying **learning** algorithm the value of the **queuing time**, i.e. the **time spent by the packet in the outgoing queue**. It's an important parameter to determine the outcome of the reward for the action chosen by `choosePort()`.

```
action sendReward() {
    ml_controller.sendReward (
        meta.valid_bool,
        standard_metadata.deq_timedelta,
        meta.id
    );
}
```

Listing 4.7: ML-routing/MLRouter/MLRouter.p4 [31]

As we can notice, it gets **three** parameters which are:

- `meta.valid_bool`: used with the same aim of the previous action
- `standard_metadata.deq_timedelta`: is a **metadata** proper of the chosen target (BMV2) and represents the queuing time, time elapsed from when the packet has entered the outgoing queue to the moment it enters the **egress processing**
- `meta.id`: is the user defined **metadata** in charge of carrying information about the correct id for the current reward. The importance of reward ID will be more clear in the next section

4.3 ML Controller

The **target** chosen for the development of the current solution is **BMV2**. It is a **P4 software switch** written in C++11, not meant to be production-grade. It's rather born for **development, debugging and testing purposes** [32].

The **architecture** is `simple_switch` and it comes in pair with `v1model`, which is its counterpart on the compiler side. It's very similar to the **abstract switch model** described in the P4 specification [13] (see image at page 32).

The biggest issue of making a Python script, such as a **Reinforcement Learning** algorithm, run and communicate with a P4 program is the simplicity of the last one. P4 was born to perform simple operations (as data plane operations are meant to be) in software with a **high degree of programmability**. They are still simple operations though and communicating on a **socket** with a Python application is not part of them.

4.3.1 Externs implementation

It's been exploited the **externs** feature declared by the P4 Consortium, to add some functionalities to the simple switch architecture to implement the model described in the previous chapter. In this way **complex operations** at any level can be written in a **high-level language** such as C++, enabling a full spectre of possibilities.

For how externs are part of the specification of the language, their implementation is **not trivial**, since support for the generation of their **JSON** lines, at the very base of P4 programs functioning, is not provided. BMV2 indeed, takes as input a **JSON file generated by the P4 compiler** and **interprets** it.

After some searches, we found a repository by Jeferson Santiago da Silva [33] which extended BMV2 to enable the generation of JSON code related to the externs defined by the user. The **drawback** is we had to adopt a **previous version** of the target architecture, with some newer features missing and a **lower computational power**.

Some of the lines needed to implement an extern will be shown in the context of ML Controller in the next subsection, but a complete guide can be found in the related repository [31].

4.3.2 ML Controller architectural overview

ML Controller is a C++ class implemented extending the `ExternType` class. It's meant to be a **unique instance** in the device, provided that we're using different physical devices for each router. But what happens in reality (in the context of this research) is that all the routers are **emulated** onto the same machine.

Let's think for a moment that we are facing the ideal scenario in which we are running only one **logical** instance of simple switch. I said logical because, as we've seen in the P4 program section, we use ML Controller both in **ingress processing** and in **egress processing**.

Instances of an extern, as **local variables**, can't "travel" between blocks so the only way to use ML Controller in both control blocks is to declare it **twice**. The obvious implication is that each data structure that must be accessed by the two control blocks must be implemented as a **singleton** [34] from the C++ side.

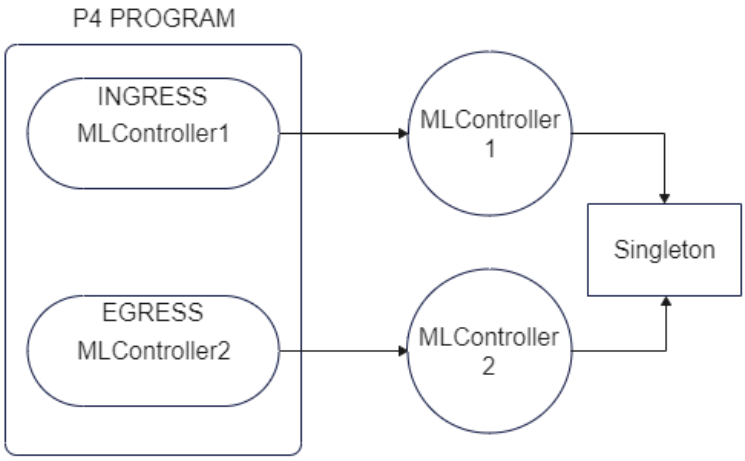

Figure 4.3: Ideal scenario

The adoption of singleton design pattern implies that the instance for that data structure will be shared among all the instances of the holding class, which is in our case **ML Controller**. But, let's get back to reality, we are not running just one logical instance of it on different machines. We are running **16 instances** of ML Controller on the same machine and they should share data structures in pair (two for each router).

So, keeping in mind that the ideal solution is to have singletons for those data structures, we are rather using **static maps** which pair routers with their data structure representing a certain resource to access. And here there's the why we need the **MAC address** of the current node in each method exposed by the class to access the right resource.

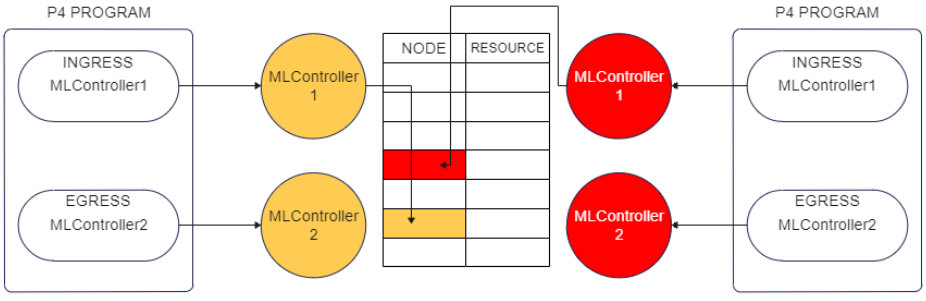

Figure 4.4: Real scenario: 2 nodes represented on the same machine

4.3.3 Concurrent Circular Buffer of addresses

This is the first remarkable data structure accessible by means of a **static unordered map**. Thanks to the MACd of the incoming packet the correct instance of this queue is picked and the IP address given is added to the structure.

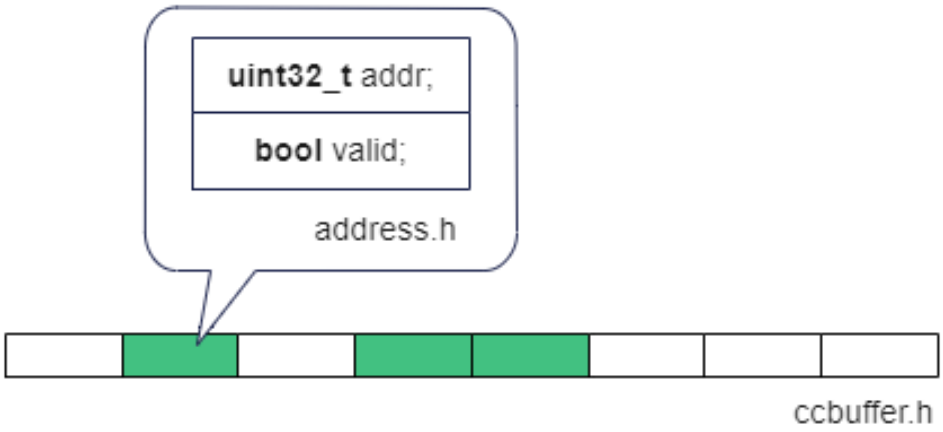

Figure 4.5: Concurrent circular buffer of addresses

It's implemented like a **circular buffer** whose methods are protected from concurrent accesses by means of a mutex that is acquired at the beginning of any read/write operation.

Each item stored is defined by the class `Address`. The `Address` class defines two properties:

- the **IP address** of the destination
- a **validity bit** needed to disable elements rather than popping them out, when the `pop` operation is performed. This let us preserve the order of the entries while guaranteeing a fixed positioning needed to access them from the ingress and the egress blocks

This class will be demanded of storing the **future destinations** at the time each packet is processed. Every time a new packet comes into the MyIngress block of simple_switch router, a push of its address is called and the buffer (virtual) size increases. Of course the element will be pushed at the first free spot **after the last valid element**. Once it reaches the MyEgress block, it's popped out.

4.3.4 PyModule

PyModule is an abstraction layer which offer methods to **establish, close** and **use** the wrapped socket connection.

The main method exposed by the class is `getPort(...)` which is responsible for the communication with the **DRL module** and takes as arguments:

- current destination
- concurrent circular buffer
- last queuing time value

```
int getPort (uint32_t address, uint32_t qTime, ConcurrentCBuffer& c);
```

Listing 4.8: ML-routing/bmv2/targets/simple_switch/pymodule.h [31]

While current destination and future destinations are needed to obtain the output port from the learning algorithm, it's most interesting the concept of **last qtime**. That value represents the queuing time due to the choice made in the previous request on the socket. The way this mechanism work will be clearer in the next section, when the **environment** for the learning algorithm will be presented.

On the same socket the Python module in charge of implementing the learning algorithm will answer with an **integer** representing the chosen port to route the packet on.

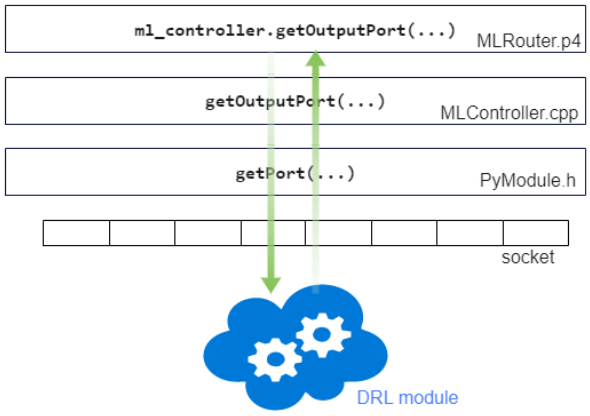

Figure 4.6: Different layers of abstractions in the get port request

4.4 DRL Module

The **Deep Reinforcement Learning** module is the one responsible for the implementation of the designed **DRL model** (see section 3.3 at page 23).

Previously we have seen it as a **blackbox** which takes an output port request as input and returns an integer number (i.e., the port) as an output. Now we can finally see how it's implemented.

4.4.1 OpenAI Baselines

The core of the DRL module is **OpenAI Baselines**, *"a set of high-quality implementations of reinforcement learning algorithms"* as it's presented in the associated GitHub repository [35].

Baselines was born with the aim of providing the community a set of **good implementations** for complex algorithms so that apparent RL advances are never due to **bad software** or **tuning**. Giving the community a shared platform that can be used to develop new solutions makes the results more reliable.

As said in **design section**, the chosen algorithm is **DQN** whose implementation has been released by OpenAI in 2017. Baselines provides an easy interface to interact with itself. It's sufficient to set a bunch of parameters to customize the test, such as the chosen **algorithm**, the target **environment** and all the hyper-parameters that characterize the algorithm.

The first landing point when calling baselines with custom parameters is the `run.py` module. In particular let's give a look at the `train` function.

```
def train(args, extra_args):
    env_type, env_id = get_env_type(args)
    ...
    learn = get_learn_function(args.alg)
    ...
    env = build_env(args)
    ...
    model = learn(
        env=env,
        seed=seed,
        total_timesteps=total_timesteps,
        **alg_kwargs
    )
```

```
    return model, env
```

Listing 4.9: baselines/baselines/run.py [35]

What we can notice is that the **train** function extracts the given parameters to **specialize** the algorithm with the right implementations of the declared methods.

The **learn** function is obtained invoking the **get_learn_function** with the given algorithm as a parameter. This means the function will retrieve the **learn** implementation from the module relative to the chosen algorithm. The environment is built basing on the name, but if we want to provide a custom environment implementation there are some steps to do that we will see in a moment.

4.4.2 DQN implementation

Now we can dive into the implementation of **learn** that will be called by the **train** function when the algorithm is set equal to deepq.

```
def learn(...):
    ...
    observation_space = env.observation_space
    ...
        for t in range(total_timesteps):
            var = int(cli.get("share_place"))
            if var == 1:
                break
            ...
            # Take action and update exploration to the newest value
            kwargs = {}
            if not param_noise:
                update_eps = exploration.value(t)
                update_param_noise_threshold = 0.
            else:
                update_eps = 0.
                ...
            action = act(np.array(obs)[None], update_eps=update_eps, **
   kwargs)[0]
            env_action = action
            reset = False
            new_obs, rew, done, _ = env.step(env_action)
            # Store transition in the replay buffer.
            replay_buffer.add(obs, action, rew, new_obs, float(done))
            obs = new_obs
            episode_rewards[-1] += rew
            ...
```

```
            if t > learning_starts and t % train_freq == 0:
                # Minimize the error in Bellman's equation on a batch
    sampled from replay buffer.
                ...
            if t > learning_starts and t % target_network_update_freq == 0:
                # Update target network periodically.
                update_target()
            ...
    return act
```

Listing 4.10: baselines/baselines/deepq/deepq.py [35]

Let's try to understand a little bit the key actions that we can see in this code.

```
var = int(cli.get("share_place"))
if var == 1:
    break
```

These lines have been **added** to coordinate the training of all the routers (agents). It's been used **Redis** [36] to share a variable that's set by the first agent which terminates its training so that all the other routers can end as well. The difference in the timesteps walked by each agent was **negligible** though.

```
action = act(np.array(obs)[None], update_eps=update_eps, **kwargs)[0]
```

Here, the action to be given the environment is extracted according to the given policy and the value of ϵ which regulates the **probability** of picking a random action rather than the best one found so far.

```
new_obs, rew, done, _ = env.step(env_action)
```

This is the core of the way the algorithm interacts with the environment. The algorithm calls a `step` on the given environment, which returns a set of parameters:

- `new_obs`: the state after the action has been done
- `rew`: value of the reward for the action performed
- `done`: flag that is true if the environment has reached a termination state

```
replay_buffer.add(obs, action, rew, new_obs, float(done))
```

Here you can see the usage of the **Replay Buffer**, one of the remarkable additions made to the algorithm to overcome the issue of pretending to be solving a regression problem (section 2.2.5 at page 18).

```
update_target()
```

The last point of interest is to see the usage of the `update_target()` function. This practice has been presented previously in section 2.2.5 to decouple $Q(s, a)$ from $Q(s', a')$.

4.5 Network Environment

The network environment, short net_env, is the environment implemented to interact with the **ML Controller**.

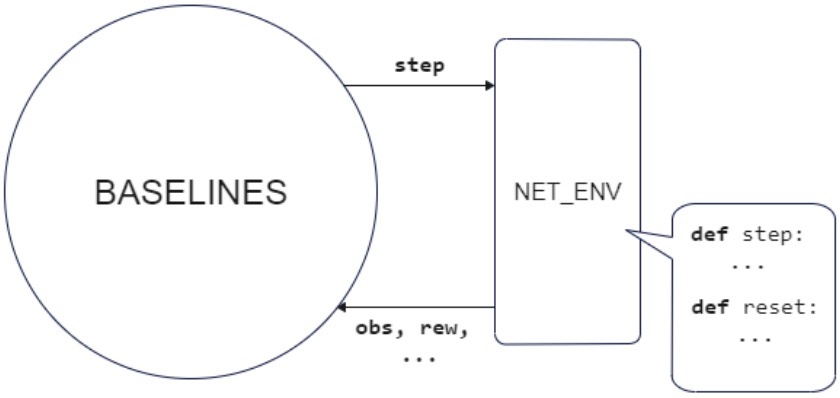

Figure 4.7: Overview of the interaction of Baselines with the custom environment

We built the environment on top of an **OpenAI Gym** environment, which proposes an easy interface to create them [37]. The two main methods that it must implement (besides init of course) are:

- `reset`: is called at the beginning of the algorithm to receive the first **observations**, i.e. the state the agent will start moving from

- **step**: will be called N times, according to the **total_timesteps** parameter set when calling baselines

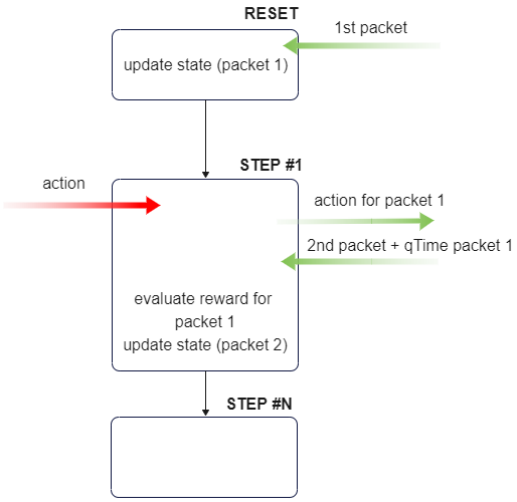

Figure 4.8: Basic functioning of the Net Env

4.5.1 Reset function

The idea is that **reset** is going to listen on socket, initialized by the **__init__** funtcion, for the first time. As soon as the first request for an output port arrives, it updates the current state with the fields extracted from the request and returns the observed state, without doing anything more.

```
def reset (self):
    # listen on socket
    if not self.resetvar:
        try:
            data = self.conn.recv(512)
            if not data:
                return
            # as msg arrives store fields in state, drop reward
            pkt = parse_req(data)
            self.state.setDsts(pkt.getDsts())
        except Exception:
            self.conn.close()
            self.s.close()
    return self.state.makeNPArray()
```

Listing 4.11: ML-routing/rl/net-env/net_env/envs/net_env.py [31]

Let's just clarify a couple of things about variables and the structures they belong to:

- `pkt` is an instance of `Packet` class, which stores future destinations (with the current one) and last queuing time (in this case it's meaningless) in two structured fields, populated by `parse_req` function
- `pkt.getDsts()` returns an instance of `FutureDestinations` class, which wraps the list of the future destinations and the current one

It's worth mentioning the last function `makeNPArray()` since it's responsible of the encoding of the state. It performs the **One-Hot-Encoding** for the future destinations, the current destination and the action history, returning an array which can be handled by **baselines**.

4.5.2 Step function

The step function will take as input the **next output port** to be sent on the socket. It basically:

- gets the **action** from deepq
- sends back the action on the socket
- listen for a **new request** on the socket
- extracts the **queuing time** for the action communicated right before and **updates the state** with the newer values
- returns the reward evaluated thanks to the queuing time too and the new state observed

Let's take a look at the code now:

```
def step (self, action):
    action = action + 1 # because action goes from 0 to N-1, while ports
    are counted from 1 to N
    info = {}
    # action contains the # of the port the packet must be forwarded to
    ret = action

    # store action in action history
    addAction(self.state.getCurDst(), action)

    # send action back on socket
    sendBack = struct.pack('I', ret)
```

```
    self.conn.sendall(sendBack)

    ...
    # here distance has been evaluated, dropped and delivered indicators
    have been set
    ...

    # listen on socket
        # as msg arrives store fields in state(t + 1) and reward(t)

    try:
        data = self.conn.recv(512)
        if not data:
            print("no data")
            return self.state.makeNPArray(), self.pkt.getReward(), True,
    info
        self.pkt = parse_req(data)
        self.state.setDsts(self.pkt.getDsts())
    except Exception as e:
        print("Exception occured:", e)
        self.conn.close()
        self.s.close()

    done = False
    isBackbone = "s" in self.id
    qtime = self.pkt.getReward()
    rw1, rw2, rw3, rw4, rw = makeRw2(distance, qtime, dropped, delivered,
    isBackbone)
    ...
    return self.state.makeNPArray(), rw, done, info
```

Listing 4.12: ML-routing/rl/net-env/net_env/envs/net_env.py [31]

4.5.3 Issues of N routers on the same machine

Of course the problems issued for ML Controller's data structures are also faced in this case. The **`code couldn't be specialized on each machine`** and there was the need to **`identify the router`** running it since we need it for evaluating the neighbor in the context of distance from the target node computation and to set the proper number of **`output ports`**.

This means that **`8 different environments`** have been created, each of them with a different port number (for socket communications) and a different **`id`** field to be set as class property by the **`__init__()`** method of net_env.py.

```
register(id = "net-v1", entry_point = "net_env.envs:NetEnv", kwargs = {'
    nports' : 4, "id":"s1", "port": 1401})
register(id = "net-v2", entry_point = "net_env.envs:NetEnv", kwargs = {'
    nports' : 4, "id":"s2", "port": 1402})
register(id = "net-v3", entry_point = "net_env.envs:NetEnv", kwargs = {'
    nports' : 4, "id":"s3", "port": 1403})
register(id = "net-v4", entry_point = "net_env.envs:NetEnv", kwargs = {'
    nports' : 4, "id":"s4", "port": 1404})
register(id = "net-v5", entry_point = "net_env.envs:NetEnv", kwargs = {'
    nports' : 5, "id":"l1", "port": 1411})
register(id = "net-v6", entry_point = "net_env.envs:NetEnv", kwargs = {'
    nports' : 5, "id":"l2", "port": 1412})
register(id = "net-v7", entry_point = "net_env.envs:NetEnv", kwargs = {'
    nports' : 5, "id":"l3", "port": 1413})
register(id = "net-v8", entry_point = "net_env.envs:NetEnv", kwargs = {'
    nports' : 5, "id":"l4", "port": 1414})
```

Listing 4.13: ML-routing/rl/net-env/net_env/___init___.py [31]

A guide to the installation of custom environments can be found in the associated repository [31].

Chapter 5

Related work

This chapter should have been part of the **evaluation** chapter, in order to compare the given solution to other similar existing works. The fact we need to reduce the impact of **ML Controller** to obtain surely better performances than a static implementation means the solution is not ready for being compared with them.

We will briefly introduce three algorithms which are proved to be better than an **OSPF** implementation, which are

- Q-Routing
- Backpressure
- DQRC

5.1 Q-Routing

Since the official article [38] which explains the algorithm is only available under payment, we don't have access to the detailed features of the algorithm. Anyway we've found another document where the key features of the algorithm are presented [39].

5.1.1 Key Features

Q-Routing is a **routing algorithm** which works thank to a **Reinforcement Learning** module running in each node, implementing a **Tabular Q-Learning** algorithm (explained at page 16).

The metric that its **reward function** is based on is **total transmission time**, for how it only uses information which are **local into the nodes**. The two big aims of the algorithm are:

- to minimize number of hops
- to avoid congestions

They also tried to implement a **neural network** to move it from the RL field to the DRL one, but the results have been ineffective.

5.1.2 Differences with ML Router

As we don't know anything about the implementation details, what's sure is that the learning algorithm adopted is different. Q-Routing uses a **tabular** implementation, while we're exploiting the **deep** adaptation of the algorithm.

The will of avoiding **congestions** is in common among the two solutions. We manage to reduce the queuing time of packets using it as a metric for the reward functions, but we don't have information about how Q-Routing does it or how it tries to minimize the number of hops just basing on **local information**.

5.2 Backpressure

On **Backpressure** we definitely have more documentation to get a little bit deeper into its features.

5.2.1 Key Features

The original Backpressure algorithm was developed by Tassiulas and Ephremides [40]. The algorithm consists of a selection stage of the **max-weight** link and a **differential backlog** routing stage.

Data is meant to reach the network in **precise discrete timeslots**. A data which is directed toward node i will be called **commodity i data** and will be stored according to its commodity, so that $Q_j(i,t)$ is the amount of **commodity i in node j at time t**, i.e. the **queue backlog** and can be measured in number of packets, bits to be transferred.

Now, at each timeslot t, the **Backpressure controller** observes $S(t)$ a function which represents the **state of the network**, capturing properties of the network on slot t and

- selects the **optimal commodity** for each link
- determines the **transmission rate** to use, choosing it from a matrix

- determines the **amount of chosen commodity** to transfer in the timeslot

The **optimal commodity** is determined basing on the state of queues for each destination in the **current node A** and the **adjacent node B**. The best commodity for **destination X** is the commodity that **maximizes** $Q_A(X,t) - Q_B(X,t)$, meaning that we want to have an **high amount of data** for that destination in the **source node** and a **small amount of data** for that destination in **adjacent node**.

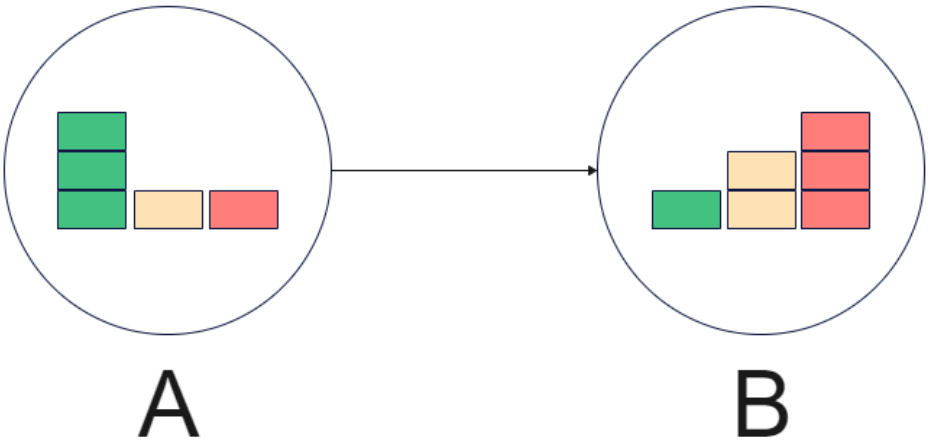

Figure 5.1: Example where green commodity is the best one

5.2.2 Differences with ML Router

The algorithm is **completely different** from the proposed one as it isn't built on RL techniques, but takes advantage of **congestion gradients**. It's proven that Backpressure performs better than **Q-Routing** in every scenario.

5.3 DQRC

This is the **design** which provides the **best performances** over all the related works presented. This is also the algorithm which looks more similar to **ML Router**.

5.3.1 Key Features

DQRC [41] is a **DRL-based** solution which implements a **DQN** algorithm. It's **multiagent** and each node concurs in building the optimal distributed policy.

Each node, given an incoming packet, collects some information which are

- **local**: current destination, future destinations and action history
- **shared**: queue lengths collected from adjacent nodes

These features are encoded to be the input of a **neural network** of **3-layers** (128 neurons each), plus an **LSTM** layer, to maintain an internal state and aggregate observations over time.

It's been demonstrated that DQRC is a better solution than both Q-Routing and Backpressure.

5.3.2 Differences with ML Router

The protocols are very similar, in the **design** of the solution the meaningful differences are:

- the **architecture of the neural network**
- the presence of **shared information** in DQRC
- the presence of **distances** in ML Router

The main goal of **ML Router** was to improve the efficiency of the algorithm thanks to the absence of any shared information, replacing them with distances knowledge. Of course they have to be set fixed *a priori*, which opens the discussion about the adaptability to changes. Anyway we know that this is a solution that suits fixed architectures such as **data centers** which are meant to be quite fixed in their structure.

Eliminating the information about **minimum length neighbor** was an interesting experiment aiming at evaluating the capability of the different agents to adapt to a context which is **influenced** by other agents. It's like the communication between agents of DQRC is mediated by the environment, which changes accordingly to each agent's decisions.

Another comparison that we could have made is about the **implementation details**, but we don't have information about DQRC implementation and about the **devices** it's been tested on.

Chapter 6

Evaluation

In this chapter we're going to show some **plots** about the performances given by the adoption of the designed solution with respect to a **static implementation**, i.e. a router which uses `static routes`.

The testbed foresees:

- links of **100Mbps**
- the topology introduced at page 22

6.1 Test scripts

Tests have been performed using the topology shown in Fig. **??** sending data from **H1** to **H2** by means of **iperf3** commands. iPerf3 is used for active measurements of the maximum achievable bandwidth on IP networks, supporting tuning of various parameters related to timing, buffers and protocols (TCP, UDP, SCTP with IPv4 and IPv6). For each test it reports the bandwidth, loss, and other parameters[42].

The two scenarios that have been created are:

- a **"no load"** scenario: it means that **no one is transmitting** on the network but H1 to H2
- a **"heavy load"** scenario: it means that **each host is sending** data toward all the other hosts

No Load scenario

The "no load" scenario is achieved running one single couple of commands in the network. The first one is the one which starts the **server** on H2:

```
iperf3 -s -p 5567 -J > outputfile.txt
```

The used **options** are:

- `-s` which means it's being used as a **server**
- `-p 5567` to listen on **port 5567**
- `-J` to enable **JSON** output format which provides more details needed for our tests

The other command is the one which enables H1 to send data toward H2:

```
iperf3 -c 10.6.2.2 -p 5567 -b 100M -t 60
```

The used **options** are:

- `-c` which means it's being used as a **client**
- `-p 5567` to send on **port 5567**
- `-b 100M` that sends the bandwidth to 100Mbps
- `-t 60` which means the client will send data for **60 seconds**

Heavy Load scenario

In the other scenario, the command given by **H1** doesn't change, while what changes is:

- each host runs a **server** instance on **three different ports** at the same time, to enable connections from all the other 3 hosts
- each host, but H1, runs a list of iperf3 client commands toward random destinations (fixed between all the testcases we will introduce later), possibly sleeping for no more than 3 seconds with a low probability

What we would expect to see is that, in case the network is issuing huge traffic loads, the **ML Router** will provide better performances than a **static router**, despite the **overhead** of updating routes basing on **baselines** and the usage of **extern functions**.

6.2 The Static Router

The static router is implemented in P4 in a **standard manner**. It means that it has a **longest prefix match** table, which is populated with static optimal routes:

- each **host** "H" sends packets toward any destination on the upper router, the only one attached
- the **leaf** routers sends packets toward any destination on the upper router (L1 to S1, L2 to S2, ...)
- the **backbone** routers sends packets to the **leaf** router that connects the network the packet is directed to (10.6.2.0/24 toward L2, 10.7.3.0/24 toward L3, ...)

The **key factor** which will make the DRL solution better or worse than a static one, will be the **updates periodicity**, i.e. how many packets must be processed before asking the **ML Controller** a new route for that destination.

6.3 ML Router comparison with Static Router

We better will show only how the **throughput**, **latency** and **retransmissions** values change tuning the **periodicity**. In **appendix B** at page 78 you can see the performances of the two implementation (throughput and latency) compared second by second, for each frequency of update adopted.

Further frequencies have been tested during the work, lower and higher ones. They got worse results than the lower peaks found so far and for this reason those frequencies have not been included neither in this chapter nor in the related appendix.

6.3.1 Retransmissions

Let's start seeing how the rentransmissions change tuning the periodicity, with respect to the **fixed** value given by the **static** implementation.

No load scenario

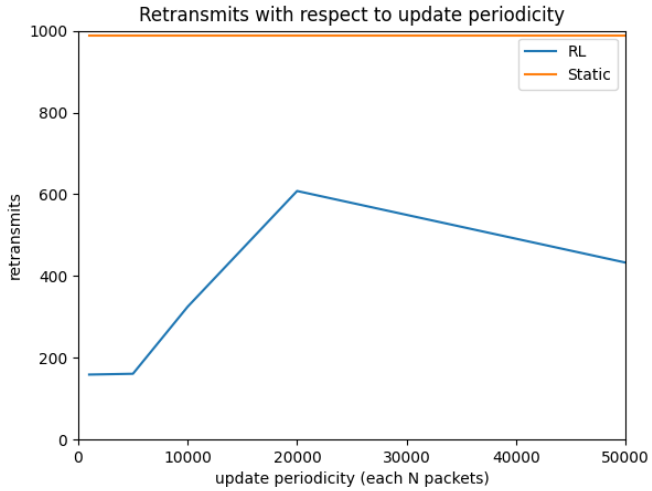

Figure 6.1: Retramsissions with no load

As we can see, the "no load scenario" remarkably changes between the two implementations, as the **static router** retransmits a **huge number of times**. The test has been performed several times to validate this result and it never changes by more than 100 retransmissions.

Periodicity	Retransimissions	Difference with static
1000	159	-830
5000	161	-828
10000	325	-664
20000	608	-381
50000	433	-556

Heavy load scenario

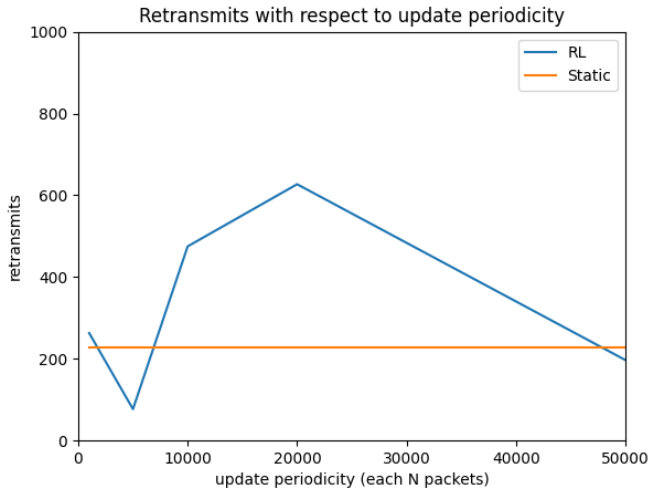

Figure 6.2: Retramsissions with heavy load

What changes now is that we can find only two values for the periodicity that gives **better results** than the static implementation. Both with **updates** made **each 5000** packets and **each 50000** packets, the **ML Router** produces a lower value for retransmissions than the **fixed value** of the static router, equal to 229 retransmissions.

Periodicity	Retransimissions	Difference with static
1000	263	+34
5000	77	-152
10000	475	+246
20000	627	+398
50000	197	-32

6.3.2 Throughput

Let's see now how the throughput changes tuning the periodicity, with respect to the **fixed** value given by the **static** implementation.

No load scenario

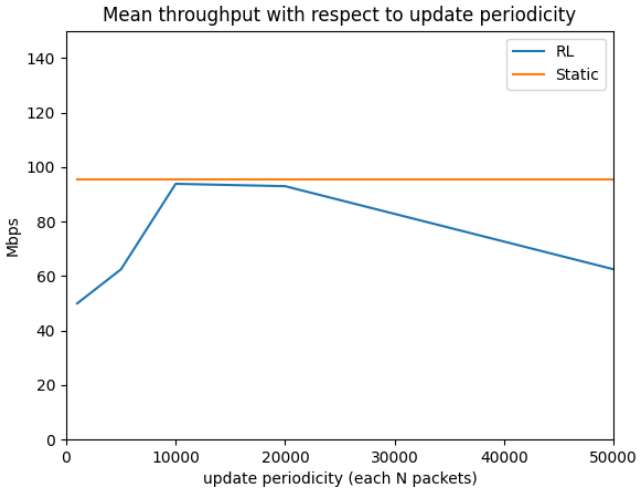

Figure 6.3: Throughput with no load

What happens "in the void" is that for values of **periodicity** between 10000 and 20000 the throughput of the two implementation is **comparable**, with the static one giving **better performances**, for how the difference is **negligible**. The **static implementation** produces a speed of 95 Mbps, while the **ML Router** as follows:

Periodicity	Mean throughput	Difference with static
1000	50 Mbps	-45 Mbps
5000	62 Mbps	-33 Mbps
10000	94 Mbps	-1 Mbps
20000	93 Mbps	-2 Mbps
50000	63 Mbps	-32 Mbps

Heavy load scenario

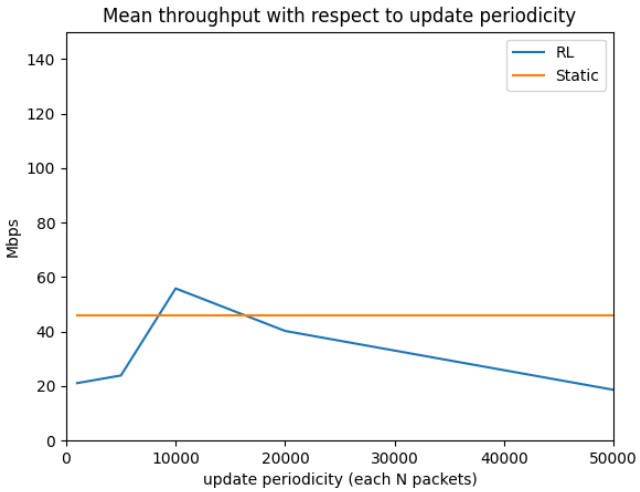

Figure 6.4: Throughput with heavy load

The interesting part is the "heavy load" scenario. Here there is **one single value** for periodicity that lets us overcome the static solution, with a mean throughput **10Mbps higher**. While the speed for the **static router** is fixed at 46 Mbps, here there are the expanded values for **ML Router**:

Periodicity	Mean throughput	Difference with static
1000	21 Mbps	-25 Mbps
5000	24 Mbps	-22 Mbps
10000	56 Mbps	+10 Mbps
20000	40 Mbps	-6 Mbps
50000	19 Mbps	-27 Mbps

6.3.3 Latency

Now it's the turn of the latency and its changesdue to the periodicity tuning, with respect to the **fixed** value given by the **static** implementation.

No load scenario

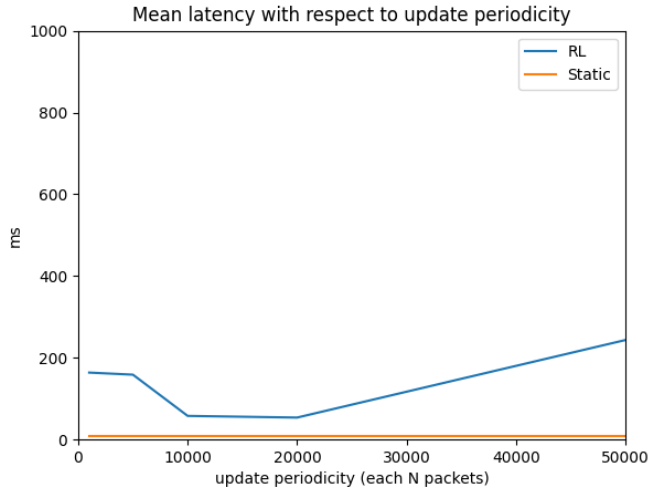

Figure 6.5: Latency with no load

Latency in the void is definitely better in the **static implementation**. It surpasses the **ML Router** by almost 50ms in case we tune **periodicity** to 10000 or 20000. Static router's **mean latency** is equal to 8 ms, while in the **ML Router**:

Periodicity	Mean latency	Difference with static
1000	164 ms	+156 ms
5000	159 ms	+151 ms
10000	57 ms	+49 ms
20000	54 ms	+46 ms
50000	243 ms	+235 ms

Heavy load scenario

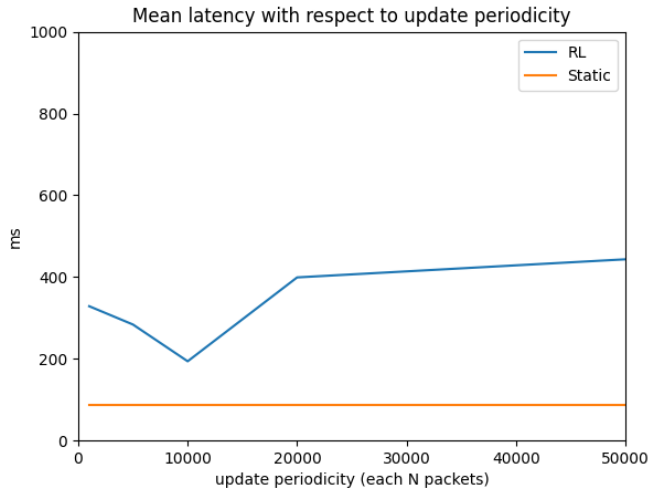

Figure 6.6: Latency with heavy load

The scenario gets even worse in case the traffic on the network increases, both the **mean round trip time** (rtt) values get higher, but the distance between the two implementations lower peaks increases by other 50 ms almost. The **static router** value for **mean latency** is 89ms, while for the **ML Router**:

Periodicity	Mean latency	Difference with static
1000	329 ms	+240 ms
5000	284 ms	+195 ms
10000	194 ms	+105 ms
20000	400 ms	+311 ms
50000	443 ms	+354 ms

Chapter 7

Conclusion

For how we've found values of the **periodicity** which give an higher throughput and a lower number of retransmissions, we **never** beat the **rtt** value of the static router implementation.

Basically this means that the computation given by the adoption of **extern functions** is still heavy with respect to the simple implementation which uses only a **routing table**.

It may also be due to the fact that the **queuing time** is not optimized as much as needed, since the updates are only periodic. By the way, we can clearly see that increasing the updates frequency leads to **higher latencies**, with respect to the lower peaks seen before.

Of course the **overhead** of updating the routes is high, since it involves a **socket communication** with the **DRL module** to get the new output port and the **injection** of the new routes by means of **bash scripts** called by the **ML controller**. It means that a tradeoff between the need of updating often the routes and the need of reducing the cost of this operations must be reached, but it's not 100% satisfying.

The fact we can reach **higher throughput** in conditions of huge traffic patterns in the network, while dealing with high latencies means the algorithm is working and even if it struggles with some **implementation limitations**, it's capable of finding better routes than optimal static ones.

Another relevant thing to say is that we tested the solution on an **emulator** (i.e., Mininet) using **BMV2** as a switch implementation which, it has been said before, is a test and debug oriented implementation of a software switch.

While performances of a static router are likely close to the real ones, it would be important to test the complex software of **ML Router** on a high performances router, as it would impact less on the transmission time [43].

The next challenges are to reduce as much as possible the impact of the **ML Controller** on the given solution, to implement it on a real device or a **production-grade** software switch and to test it on different topologies too, in order to verify the **scalability** of the solution, but they won't be part of this thesis.

Appendix A

Hyper-parameters tuning

In this appendix you will find a summary of the test performed to properly tune the **hyper-parameters** of the **DRL training**. All the hyperparameters have been tested in combination with the others and what we can see below is the **mean value** over all the tests performed.

A.1 Hidden layers

Hidden layers represent the architecture of the neural network. Over all the possible tests performed we can show 4 possibilities taken into consideration.

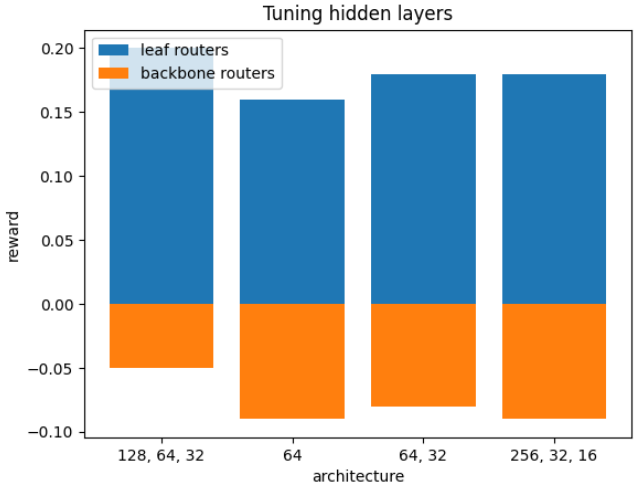

Figure A.1: Tuning of hidden layers

As we can see, the best performances both on **leaf** and **backbone routers** can

be obtained with the adoption of a **3-layers architecture** with 128, 64 and 32 neurons respectively.

A.2 Lambda 1

λ_1 is the hyper-parameter that provides the weight for the **delivered** indicator. That means, the higher is λ_1, the more reward function will be influenced by that indicator.

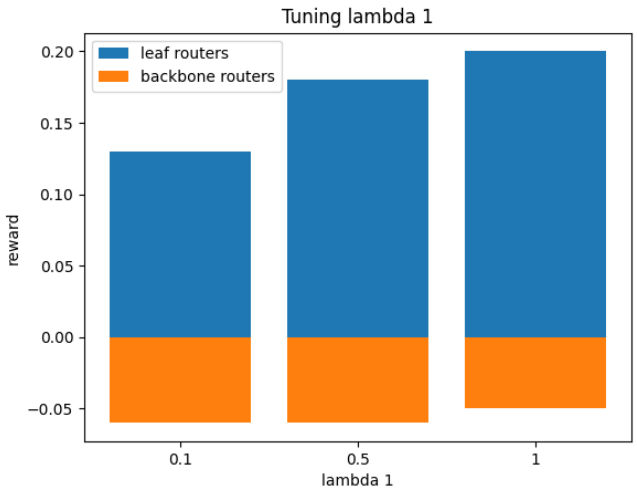

Figure A.2: Tuning of λ_1

What we can desume from the plot is that both **leaf** and **backbone routers** perform better with the adoption of λ_1 equal to 1. Actually we would expect that backbone routers are not affected by changes on this value, as the delivered indicator is always null for them. We can explain this little difference in the value of reward in the backbone with the randomness during the exploration phase.

A.3 Lambda 2

λ_2 is the hyper-parameter that guides the weight of **queuing time**.

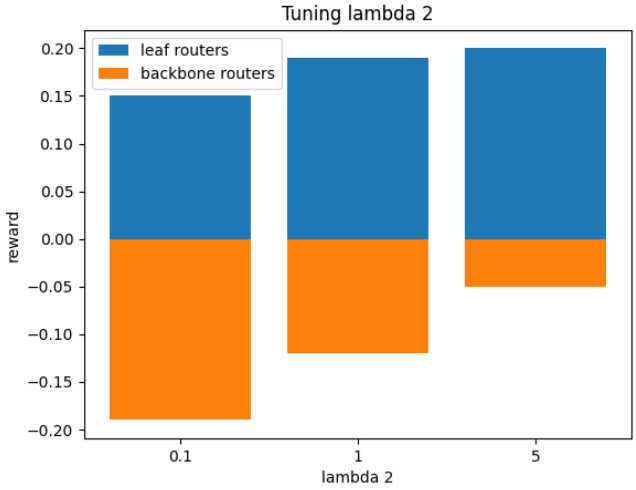

Figure A.3: Tuning of λ_2

Once again, is not hard to choose among these values, both **leaf** and **backbone routers** perform better with the adoption of λ_2 equal to 5.

A.4 Lambda 3

λ_3 is the hyper-parameter related to **dropped indicator**.

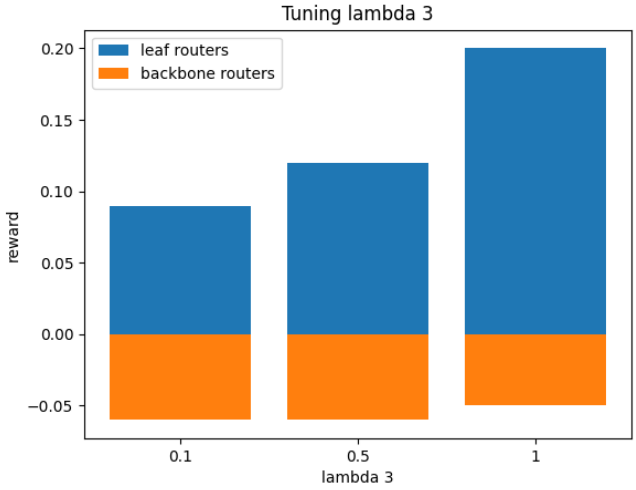

Figure A.4: Tuning of λ_3

This time we can notice that **backbone routers** are averagely not affected by changes on this value, while **leaf ones** have a better mean reward with value of 1.

A.5 Lambda 4

λ_4 is the hyper-parameter that multiplies **distance** in the reward function.

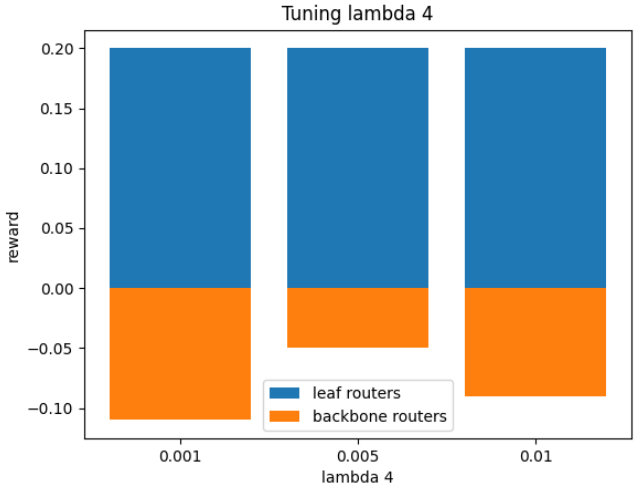

Figure A.5: Tuning of λ_4

It's worth noticing that **leaf routers** don't change their mean value of reward during the tuning process. This happens because they don't use distance to evaluate the reward value. **backbone routers**, on the contrary, change their behavior and give better performances with a λ_4 equal to 0.005.

A.6 Action History length

Let's see how reward values changed accordingly to the value given to the size of the list representing **action history**.

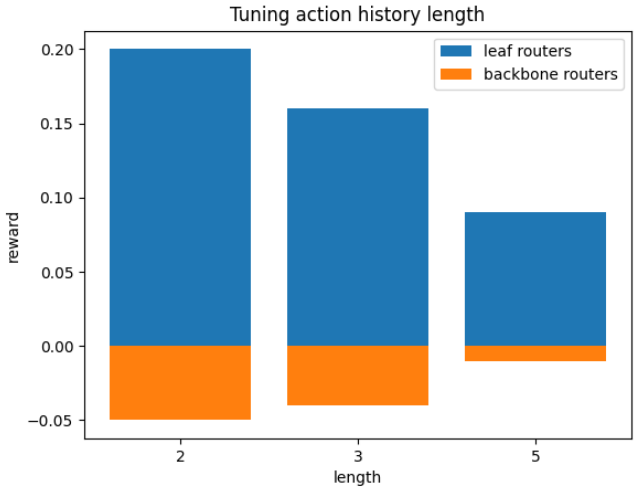

Figure A.6: Tuning of action history length

While **backbone router** give best results with a length equal to 5, **leaf routers** definitely work better with smaller lists. We give them much more importance as they're responsible for delivering packets to their attached destination and the **drop ratio** with other values of length would have been too high.

A.7 Future Destinations length

Future destinations is the queue that save the next N destinations that the router has to reach. It's another hyper-parameter that needs to be tuned.

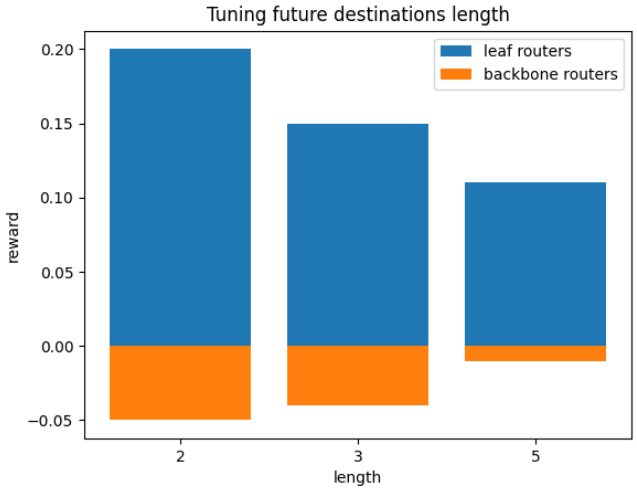

Figure A.7: Tuning of future destinations length

Exactly like in the previous tuning section (action history length), a bigger list helps **backbone** to route packets more efficently, but it's not sufficient for **leaf routers** to learn how to properly deliver and, much more, not drop packets. That's why, once again, a length of 2 seems to be the best choice.

A.8 Learning rate

Learning rate is the α of the algorithm 2 (page 20). The higher it is, the more Q-Values will be changed according to the last computation. The lower it is the smoother updates will be.

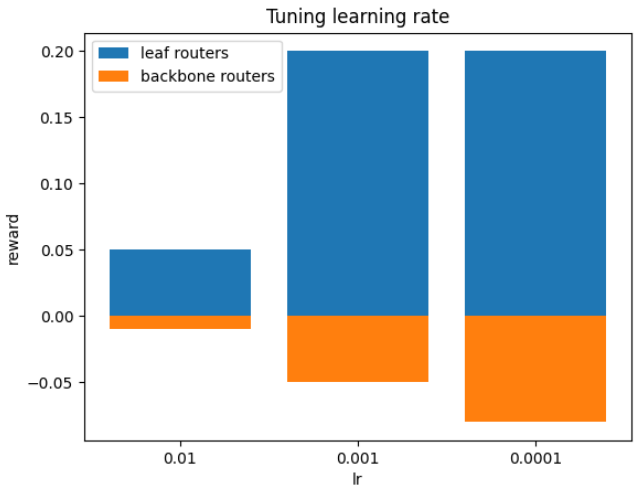

Figure A.8: Tuning of learning rate

Again, the behavior is different for routers of different layers. In the **backbone**, frequent updates give better performances. Queuing times are going to be very different and actions in a given state must be changed often. Such a behavior is source of instability for **leaf nodes**. They must be able to deliver and not to drop with a high precision. If a packet is directed to the host directly attached to me I must deliver it the packet, that won't change. That's why performances improve significantly at a lower order of magnitude. Since backbone routers are going to decrease even more their performances lowering even more the value of α, 0,001 seems to be a good tradeoff.

A.9 Total timesteps

Total timesteps should affect almost in the same way both routers in the **backbone** and in the **leaves**. Let's see how the reward changes with lower or higher number of timesteps.

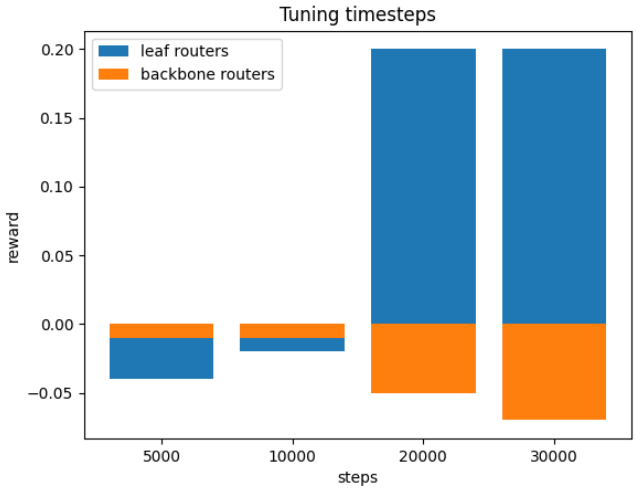

Figure A.9: Tuning of timesteps

While with lower values the quality of the reward decreases quite a lot (only in the case of **leaves**), we can see that starting from 20000 steps we have good values for both rewards and they're not going to change with a higher value of timesteps. 20000 steps, thus, look to be the best tradeoff for training steps.

A.10 Exploration fraction

Exploration fraction is the percentage of time during the training, over which is **annealed** the value of **epsilon** (page 18).

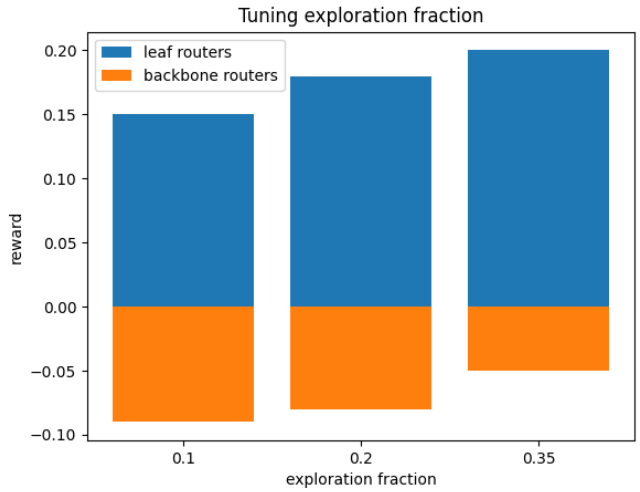

Figure A.10: Tuning of exploration fraction

Here we can't have any doubt: the two categories of routers both perform better with a value of exploration fraction of 0.35. Higher values didn't add any quality to the solution.

A.11 Learning starts

The value of **learning starts** says after how many steps the algorithm starts updating its values, thus using the gained knowledge.

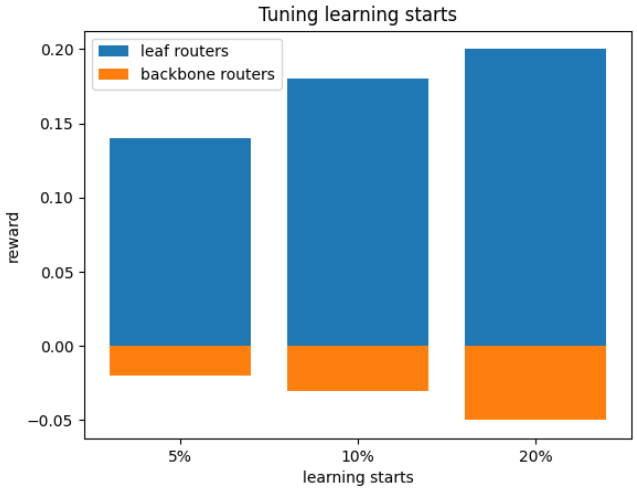

Figure A.11: Tuning of learning starts

Since a value of 20% of the total timesteps assured a high percentage of packets delivery that seemed to be a good value for the hyper-parameter. The reward in the backbone is still high too even if sub-optimal.

A.12 Buffer size

Replay buffer has been explained at page 18 and its size is another parameter to be studied in order to properly dimension it.

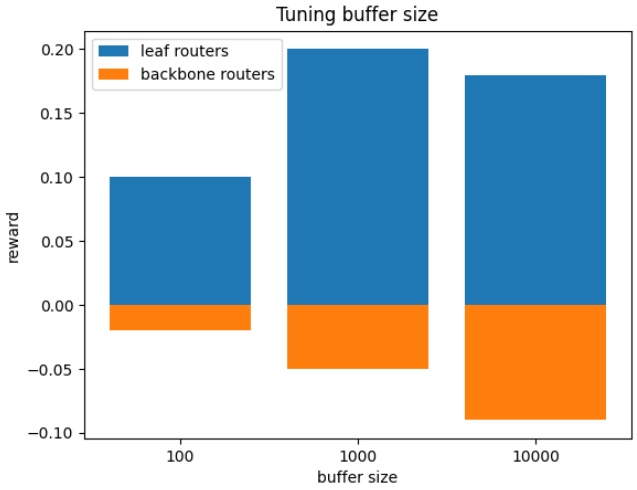

Figure A.12: Tuning of replay buffer size

The size of 1000 looks the best one over all the trainings. Both decreasing and increasing its value deteriorate the quality of the reward.

A.13 Discount factor

Last, but not least the **discount factor** γ. Its value gives more or less importance to the next rewards with respect to the immediate one (see algorithm 1 at page 17).

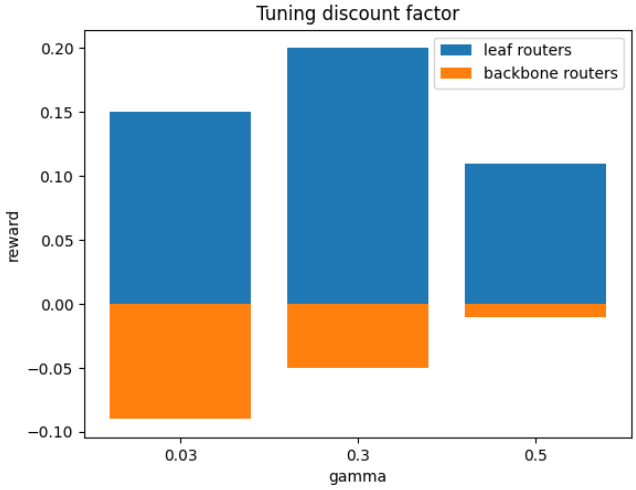

Figure A.13: Tuning of γ

A value of 0.3 is quite high and guarantees a good tradeoff for both the categories of routers. A high value of γ is great for the designed environment as the newer state at each step will always be almost impossible to be predicted as it's represented by the new incoming packet, along with other information about router. Its **random component** make the adoption of 0.3 fraction value ideal.

Appendix B

Updates periodicity tuning

B.1 Periodicity equal to 1000

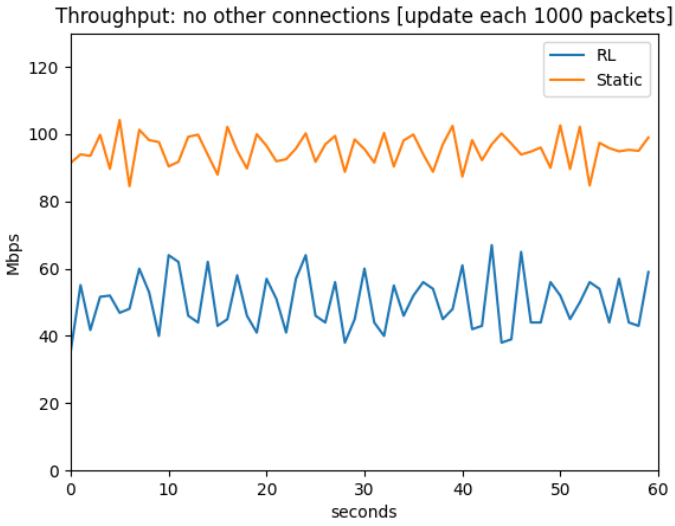

Figure B.1: Throughput with no load, periodicity = 1000

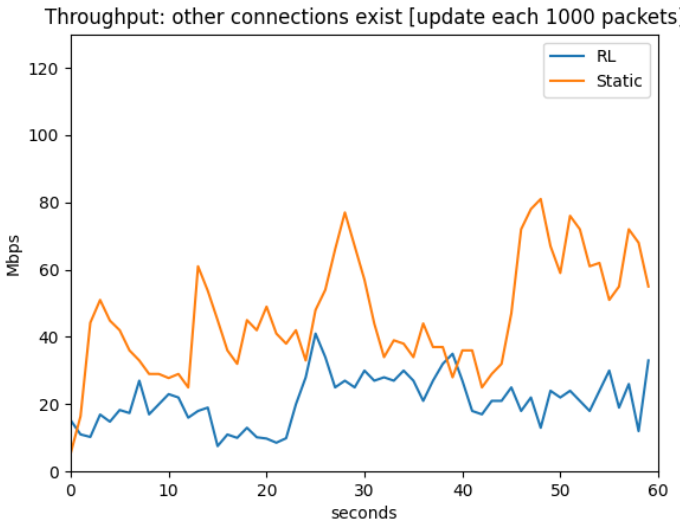

Figure B.2: Throughput with heavy load, periodicity = 1000

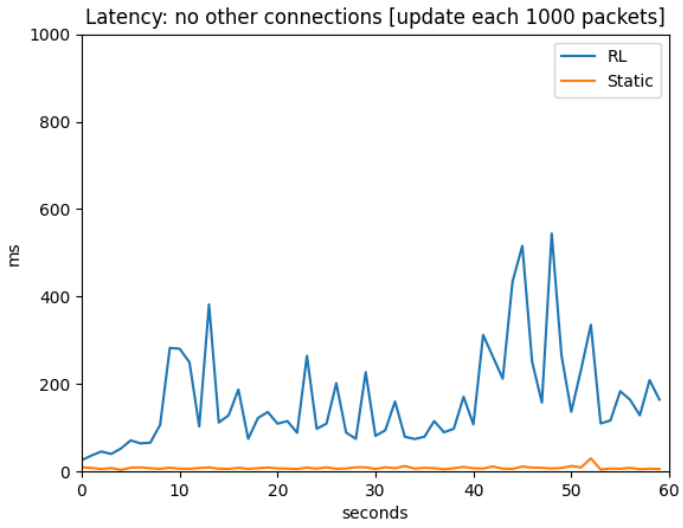

Figure B.3: Latency with heavy load, periodicity = 1000

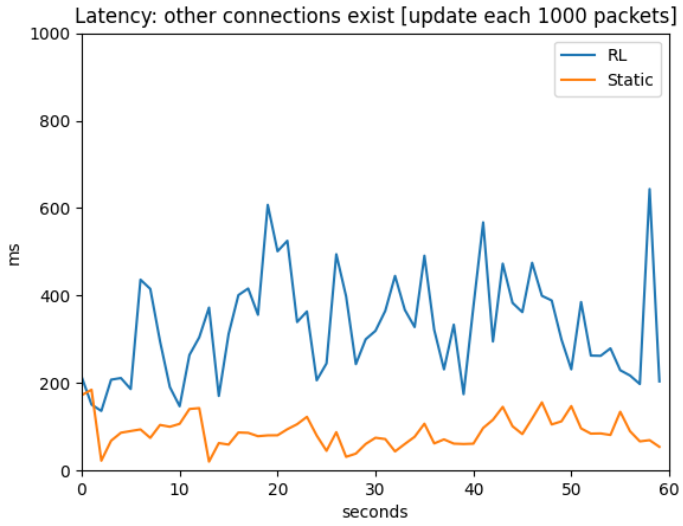

Figure B.4: Latency with heavy load, periodicity = 1000

B.2 Periodicity equal to 5000

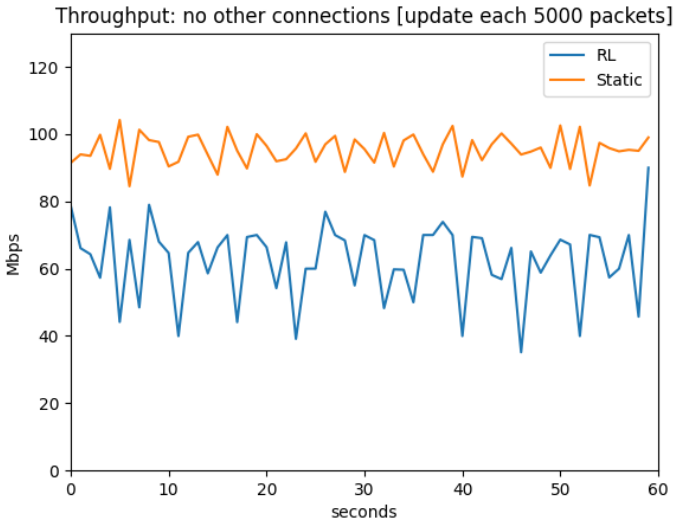

Figure B.5: Throughput with no load, periodicity = 5000

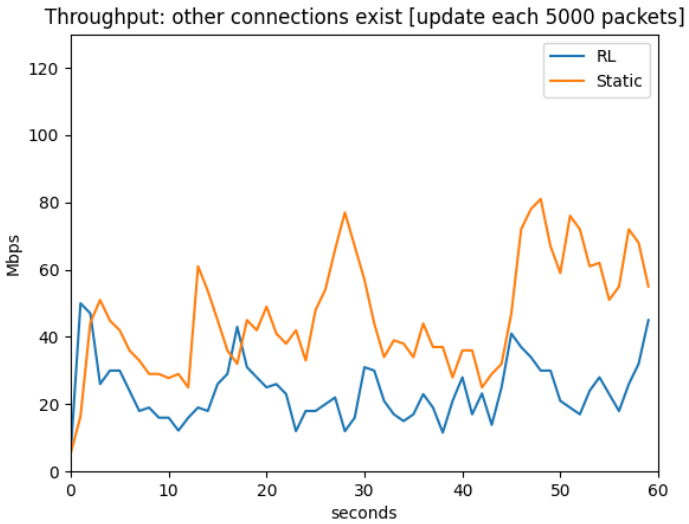

Figure B.6: Throughput with heavy load, periodicity = 5000

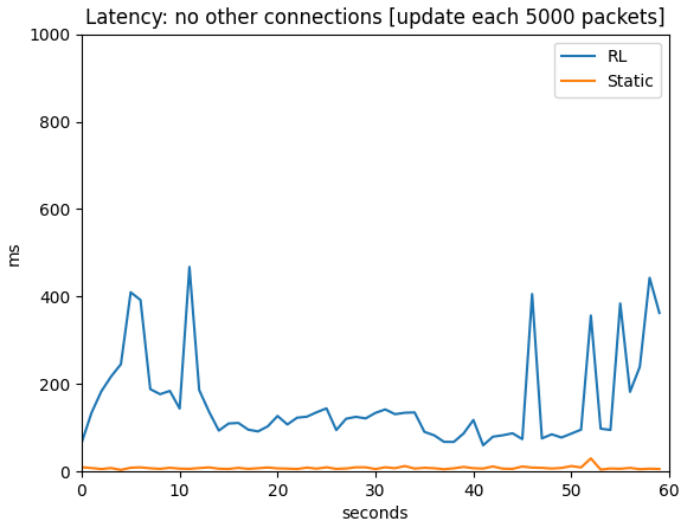

Figure B.7: Latency with heavy load, periodicity = 5000

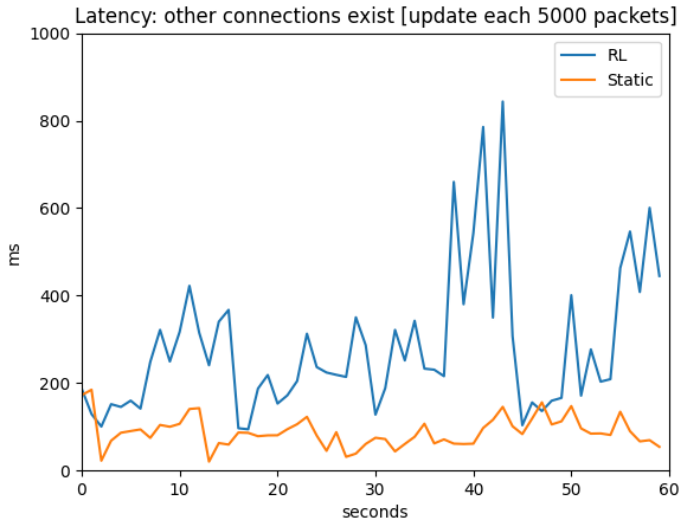

Figure B.8: Latency with heavy load, periodicity = 5000

B.3 Periodicity equal to 10000

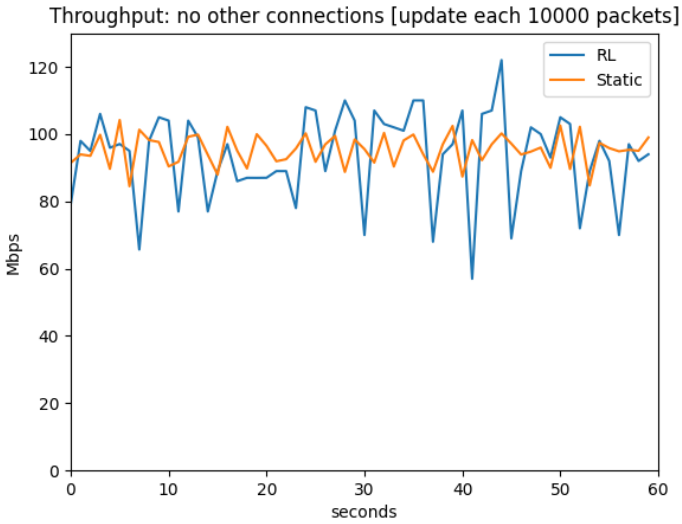

Figure B.9: Throughput with no load, periodicity = 10000

Figure B.10: Throughput with heavy load, periodicity = 10000

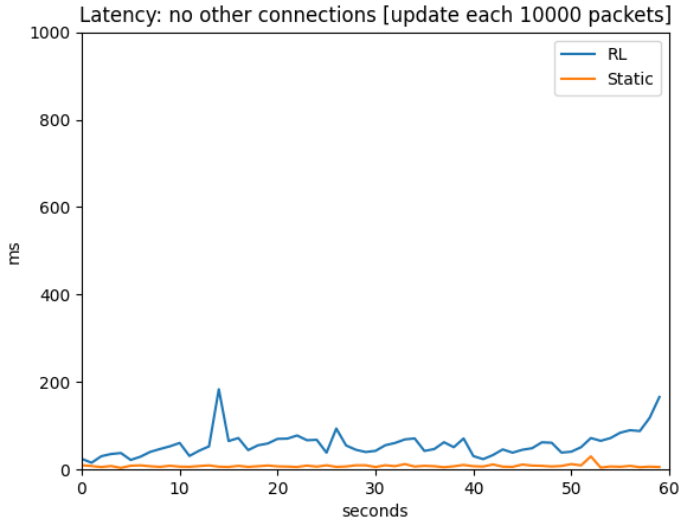

Figure B.11: Latency with heavy load, periodicity = 10000

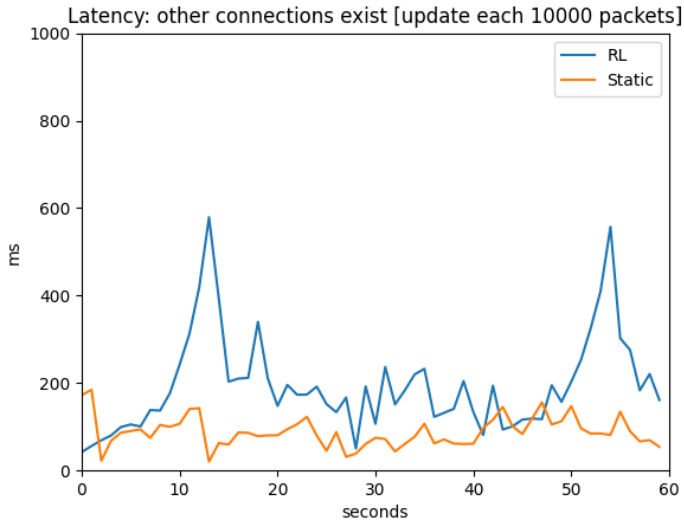

Figure B.12: Latency with heavy load, periodicity = 10000

B.4 Periodicity equal to 20000

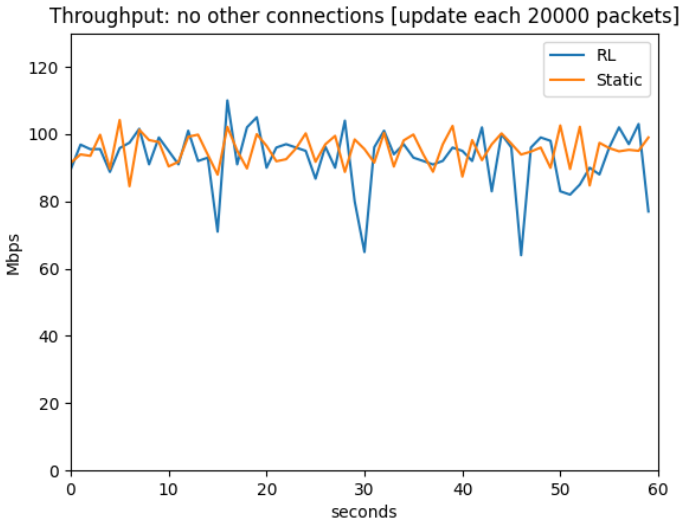

Figure B.13: Throughput with no load, periodicity = 20000

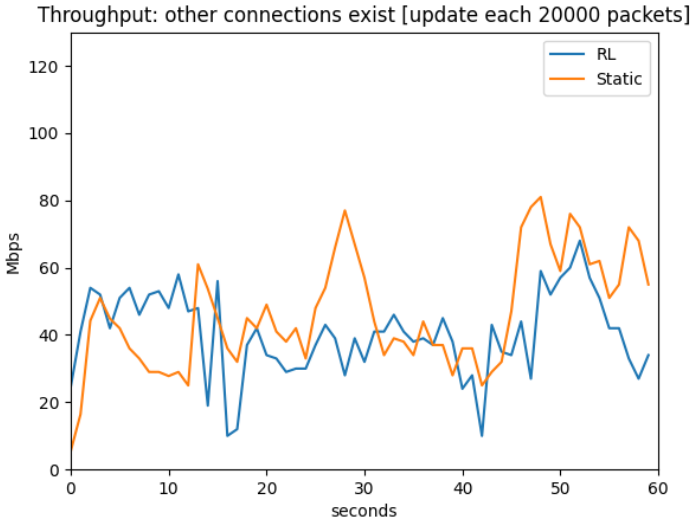

Figure B.14: Throughput with heavy load, periodicity = 20000

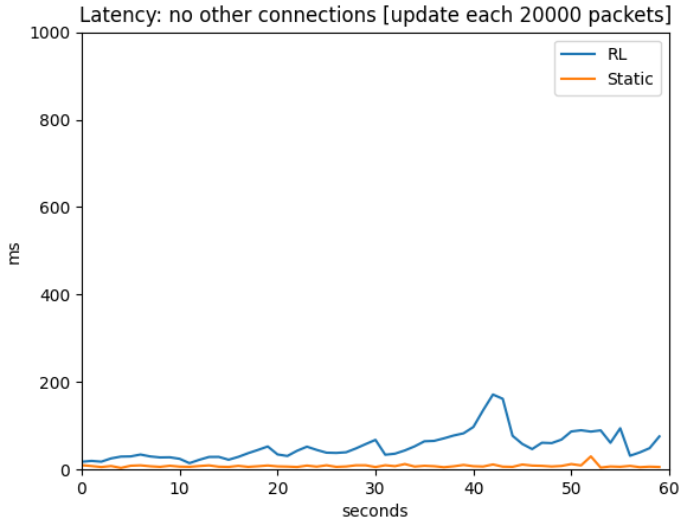

Figure B.15: Latency with heavy load, periodicity = 20000

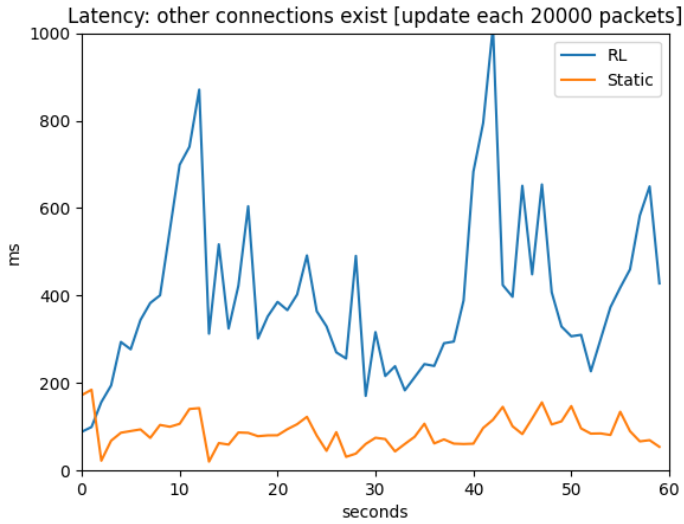

Figure B.16: Latency with heavy load, periodicity = 20000

B.5 Periodicity equal to 50000

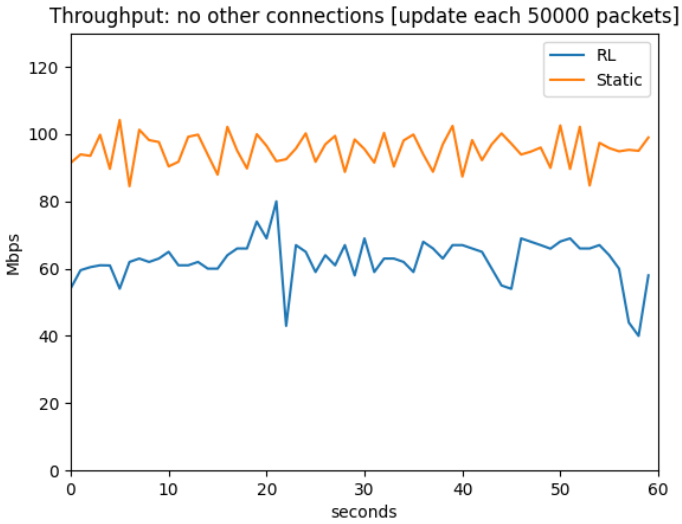

Figure B.17: Throughput with no load, periodicity = 50000

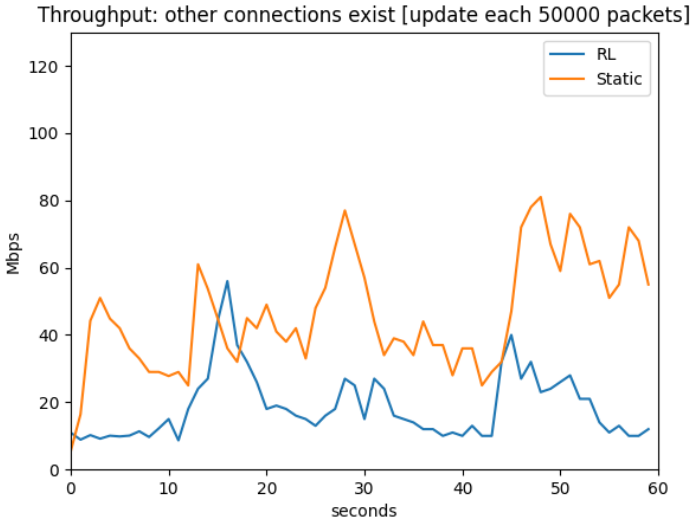

Figure B.18: Throughput with heavy load, periodicity = 50000

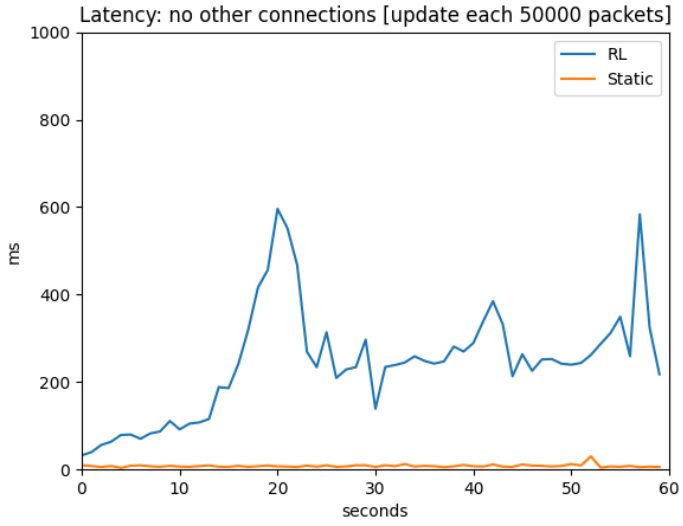

Figure B.19: Latency with heavy load, periodicity = 50000

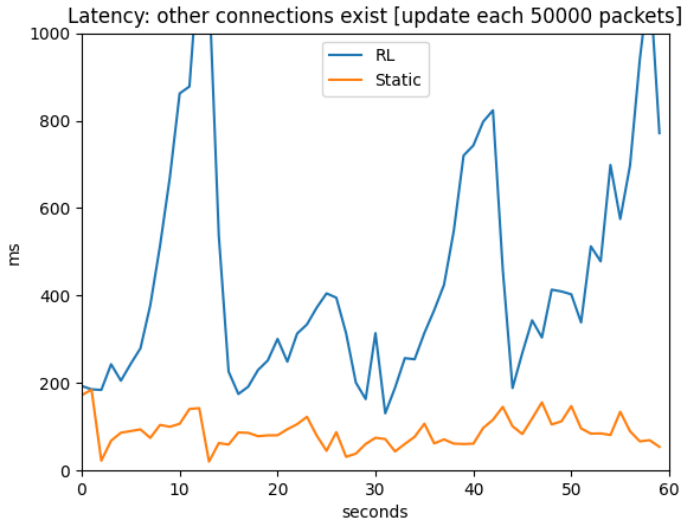

Figure B.20: Latency with heavy load, periodicity = 50000

List of Figures

1.1 Slow path and Fast path [1] . . . 2
1.2 Network computing vs general-purpose [3] . . . 3
1.3 client-server pattern vs cloud pattern . . . 4
1.4 SDN Network . . . 7

2.1 P4 is a language to configure switches. [10] . . . 10
2.2 The abstract forwarding model. [10] . . . 11
2.3 How a parser works. [11] . . . 11
2.4 RL entities and their communication. [17] . . . 14
2.5 An abstract environment with N states reachable from the initial state. [17] . . . 15
2.6 An example of the transition from the state in a stochastic case. [17] 16

3.1 Adopted topology . . . 22
3.2 Neural network architecture . . . 28

4.1 Overview of the ML Router . . . 31
4.2 P4 Pipeline . . . 32
4.3 Ideal scenario . . . 39
4.4 Real scenario: 2 nodes represented on the same machine . . . 39
4.5 Concurrent circular buffer of addresses . . . 40
4.6 Different layers of abstractions in the get port request . . . 41
4.7 Overview of the interaction of Baselines with the custom environment 45
4.8 Basic functioning of the Net Env . . . 46

5.1 Example where green commodity is the best one . . . 52

6.1 Retramsissions with no load . . . 57
6.2 Retramsissions with heavy load . . . 58
6.3 Throughput with no load . . . 59
6.4 Throughput with heavy load . . . 60
6.5 Latency with no load . . . 61

6.6 Latency with heavy load 62

A.1 Tuning of hidden layers 65
A.2 Tuning of λ_1 66
A.3 Tuning of λ_2 67
A.4 Tuning of λ_3 68
A.5 Tuning of λ_4 69
A.6 Tuning of action history length 70
A.7 Tuning of future destinations length 71
A.8 Tuning of learning rate 72
A.9 Tuning of timesteps 73
A.10 Tuning of exploration fraction 74
A.11 Tuning of learning starts 75
A.12 Tuning of replay buffer size 76
A.13 Tuning of γ 77

B.1 Throughput with no load, periodicity = 1000 78
B.2 Throughput with heavy load, periodicity = 1000 79
B.3 Latency with heavy load, periodicity = 1000 80
B.4 Latency with heavy load, periodicity = 1000 80
B.5 Throughput with no load, periodicity = 5000 81
B.6 Throughput with heavy load, periodicity = 5000 81
B.7 Latency with heavy load, periodicity = 5000 82
B.8 Latency with heavy load, periodicity = 5000 82
B.9 Throughput with no load, periodicity = 10000 83
B.10 Throughput with heavy load, periodicity = 10000 83
B.11 Latency with heavy load, periodicity = 10000 84
B.12 Latency with heavy load, periodicity = 10000 84
B.13 Throughput with no load, periodicity = 20000 85
B.14 Throughput with heavy load, periodicity = 20000 85
B.15 Latency with heavy load, periodicity = 20000 86
B.16 Latency with heavy load, periodicity = 20000 86
B.17 Throughput with no load, periodicity = 50000 87
B.18 Throughput with heavy load, periodicity = 50000 87
B.19 Latency with heavy load, periodicity = 50000 88
B.20 Latency with heavy load, periodicity = 50000 88

Bibliography

[1] Fulvio Risso. «Architecture of network devices». URL: `https://swnet.frisso.net/` (cit. on p. 2).

[2] Sakir Sezer, Sandra Scott-Hayward, Pushpinder Kaur Chouhan, Barbara Fraser, David Lake, Jim Finnegan, Niel Viljoen, Marc Miller, and Navneet Rao. «Are we ready for SDN? Implementation challenges for software-defined networks». In: *IEEE Communications Magazine* 51.7 (2013), pp. 36–43 (cit. on p. 3).

[3] Fulvio Risso. «SDN Intro». URL: `https://swnet.frisso.net/` (cit. on p. 3).

[4] William Stallings. In: *The Internet Protocol Journal* 16.1 (Mar. 2013) (cit. on p. 4).

[5] Nick Feamster, Jennifer Rexford, and Ellen Zegura. «The Road to SDN: An Intellectual History of Programmable Networks Nick Feamster Jennifer Rexford Ellen Zegura Georgia Tech Princeton University Georgia Tech». In: *ACM SIGCOMM Computer Communication Review* 44.2 (Apr. 2014) (cit. on p. 5).

[6] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. «An Architecture for Active Networking». In: *High Performance Networking VII: IFIP TC6 Seventh International Conference on High Performance Networks (HPN '97), 28th April – 2nd May 1997, White Plains, New York, USA*. Ed. by Ahmed Tantawy. Boston, MA: Springer US, 1997, pp. 265–279 (cit. on p. 5).

[7] J. Mark Smith, D., Farber, Carl A. Gunter, Scott Nettles, David C. Feldmeier, and W. David Sincoskie. «Switchware : Accelerating Network Evolution (White Paper) MS-CIS-96-38». In: Jan. 1996 (cit. on p. 5).

[8] K.L. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. «Directions in active networks». In: *IEEE Communications Magazine* 36.10 (1998), pp. 72–78 (cit. on p. 5).

[9] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. «A Survey of Active Network Research». In: *IEEE Communications Magazine* 35 (1997), pp. 80–86 (cit. on p. 5).

[10] Pat Bosshart et al. «P4: Programming Protocol-Independent Packet Processors». In: *ACM SIGCOMM Computer Communication Review* 44.3 (July 2014) (cit. on pp. 9–12).

[11] Laurent Vanbever. *Advanced Topics in Communication Networks*. 2019. URL: https://adv-net.ethz.ch/ (cit. on pp. 10–12).

[12] p4language. *v1model.p4*. Dec. 2021. URL: https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4 (cit. on p. 10).

[13] The P4 Language Consortium. *P4-16 Language Specification*. May 2017. URL: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html (cit. on pp. 12, 37).

[14] Nick McKeown and Jen Rexford. *Clarifying the differences between P4 and OpenFlow*. May 2016. URL: https://opennetworking.org/news-and-events/blog/clarifying-the-differences-between-p4-and-openflow. (cit. on p. 13).

[15] P4 Language Consortium. URL: p4.org (cit. on p. 13).

[16] p4language. *openflow.p4*. July 2016. URL: https://github.com/p4lang/switch/blob/master/p4src/openflow.p4 (cit. on p. 13).

[17] Maxim Lapan. *Deep Reinforcement Learning Hands-On*. 2nd ed. Jan. 2020 (cit. on pp. 13–19).

[18] Ruben Fiszel. *Reinforcement Learning and DQN, learning to play from pixels*. Aug. 2016. URL: https://rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN.html (cit. on p. 13).

[19] pathmind. *A Beginner's Guide to Deep Reinforcement Learning*. URL: https://wiki.pathmind.com/deep-reinforcement-learning (cit. on p. 14).

[20] Cristopher Watkins. «Learning from delayed rewards». PhD thesis. King's college, 1989 (cit. on p. 16).

[21] Mnih Volodymyr, Kavukcuoglu Koray, Silver David, Graves Alex, Antonoglou Ioannis, Wierstra Daan, and Riedmiller Martin. «Playing Atari with Deep Reinforcement Learning». In: (2013). URL: http://arxiv.org/abs/1312.5602 (cit. on p. 17).

[22] Jason Brownlee. Dec. 2017. URL: https://machinelearningmastery.com/classification-versus-regression-in-machine-learning (cit. on p. 18).

[23] Melanie Coggan. Research. McGill University, 2004 (cit. on p. 18).

[24] Bellman Richard. In: *Quart. Appl. Math* 16 (1958), pp. 87–90 (cit. on p. 21).

[25] Dijkstra E.W. «A note on two problems in connexion with graphs». In: 1 (1959), pp. 269–271 (cit. on p. 21).

[26] *Openflow Switch Specification*. v1.5.1. Open Networking Foundation. Mar. 2015 (cit. on p. 23).

[27] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. «Multi-agent Reinforcement Learning: An Overview». In: *Innovations in Multi-Agent Systems and Applications - 1*. Ed. by Dipti Srinivasan and Lakhmi C. Jain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 183–221. ISBN: 978-3-642-14435-6. DOI: 10.1007/978-3-642-14435-6_7. URL: https://doi.org/10.1007/978-3-642-14435-6_7 (cit. on p. 23).

[28] Jordan Jeremy. Nov. 2017. URL: https://www.jeremyjordan.me/hyperparameter-tuning/ (cit. on p. 25).

[29] Nicolas Vandeput. «22 Categorical Features». In: *Data Science for Supply Chain Forecasting*. De Gruyter, 2021, pp. 200–208. DOI: doi:10.1515/9783110671124-022. URL: https://doi.org/10.1515/9783110671124-022 (cit. on p. 27).

[30] Y. Liu. *Python Machine Learning By Example*. Packt Publishing, 2017. ISBN: 9781783553129. URL: https://books.google.it/books?id=0nc5DwAAQBAJ (cit. on p. 27).

[31] Daniele Sarcinella. *ML Router*. Version 1.0.0. July 2022. URL: https://github.com/danisrcnl/ML-routing (cit. on pp. 32–38, 41, 46, 48, 49).

[32] The P4 Language Consortium. *bmv2*. Version 1.7.0. URL: https://github.com/p4lang/behavioral-model (cit. on p. 37).

[33] Jeferson Santiago da Silva. *p4-programs*. URL: https://github.com/engjefersonsantiago/p4-programs (cit. on p. 38).

[34] *Singleton Design Pattern | Implementation*. Nov. 2020. URL: https://www.geeksforgeeks.org/singleton-design-pattern/ (cit. on p. 38).

[35] OpenAI. *Baselines*. URL: https://github.com/openai/baselines (cit. on pp. 42–44).

[36] Redis. *redis-py*. Version 4.3.4. URL: https://github.com/redis/redis-py (cit. on p. 44).

[37] Mate Pocs. Dec. 2020. URL: https://towardsdatascience.com/beginners-guide-to-custom-environments-in-openai-s-gym-989371673952 (cit. on p. 45).

[38] Alexis Bitaillou, Benoît Parrein, and Guillaume Andrieux. «Q-routing: From the Algorithm to the Routing Protocol». In: *Machine Learning for Networking*. Ed. by Selma Boumerdassi, Éric Renault, and Paul Mühlethaler. Cham: Springer International Publishing, 2020, pp. 58–69 (cit. on p. 50).

[39] Justin Boyan and Michael Littman. «Packet Routing in Dynamically Changing Networks: A Reinforcement Learning Approach». In: *Advances in Neural Information Processing Systems*. Ed. by J. Cowan, G. Tesauro, and J. Alspector. Vol. 6. Morgan-Kaufmann, 1993. URL: https://proceedings.neurips.cc/paper/1993/file/4ea06fbc83cdd0a06020c35d50e1e89a-Paper.pdf (cit. on p. 50).

[40] L. Tassiulas and A. Ephremides. «Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks». In: *IEEE Transactions on Automatic Control* 37.12 (1992), pp. 1936–1948. DOI: 10.1109/9.182479 (cit. on p. 51).

[41] Xinyu You, Xuanjie Li, Yuedong Xu, Hui Feng, and Jin Zhao. «Toward Packet Routing with Fully-distributed Multi-agent Deep Reinforcement Learning». In: *2019 International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOPT)*. 2019, pp. 1–8. DOI: 10.23919/WiOPT47501.2019.9144110 (cit. on p. 52).

[42] iperf. URL: https://iperf.fr/ (cit. on p. 54).

[43] The P4 Language Consortium. Nov. 2019. URL: https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md (cit. on p. 64).