POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

A collaborative and distributed learning-based solution to manage computer networks

Supervisors

Prof. Guido MARCHETTO

Dr. Alessio SACCO

Candidate

Doriana MONACO

December 2022

Abstract

The high programmability provided by Software-Defined Networking (SDN) paradigm facilitated the integration of Machine Learning (ML) methods to design a new family of network management schema. Among them, we can cite self-driving networks, where ML is used to analyze data and define strategies that are then translated by SDN controllers into network configurations, thus making networks autonomous and capable of auto-scaling decisions based on the network's needs. Despite its attractiveness, the centralized design of the majority of proposed solutions cannot keep up with the increasing size of the network. To this end, this thesis investigates the use of a multi-agent reinforcement learning (MARL) model for auto-scaling decisions in an SDN environment. Physically distributed controllers raise new challenges to be addressed: in absence of a central entity with global knowledge, controllers must cooperate sharing their information in order to make the best decision. In particular, we study two possible alternatives: a logically decentralized control plane and a fully distributed control plane. The former implementation is defined by multiple controllers that collect statistics over their own area and share them to feed the model with the same input; the latter deploys many controllers that collaborate to correctly perform routing but feed the model with their local information only. Results showed that both approaches can guarantee high throughput while minimizing the set of active resources.

I want to thank my family and everyone who believed in me.

Table of Contents

Chapter 1

Introduction

The presence of new requirements, e.g., high reliability, zero packet loss, and real-time interaction, posed by data-intensive applications, e.g., augmented/virtual reality, industrial 4.0, or healthcare, exacerbates the need for more performant, scalable, resilient, and self-adapting networks [1, 2]. To support such applications, there is a need to rethink the design of both networks and applications, creating more intelligent and autonomous networks. Next-generation networks are envisioned as the answer to network operators and service providers to replace existing infrastructures and to introduce a new platform able to support new telecommunication businesses and services. They are considered key enablers for delivering new services that are available to any place, at any time, on any device.

To provide such (artificial) intelligence, there is an increasing interest in equipping networks with autonomous run-time decision-making capabilities incorporating distributed machine learning (ML) algorithms, to foster automation in network configurations, network management, and network resiliency. While AI/ML technologies continue to evolve at a rapid pace, moving from a paradigm of supervised learning towards distributed self-learning requires solving several challenges in the design and deployment of wide-scale networks. Among those challenges, the scalability of AI/ML models for managing the Next Generation Internet is particularly critical.

Recent studies have partially addressed some of these challenges by employing model-free approaches to efficiently manage network resources. In particular, Reinforcement Learning (RL) finds profitable applicability given its ability to fit the network dynamics well without any prior knowledge [3, 4, 5, 6]. With such auto-scaling solutions, networks can deactivate idle resources that may increase unnecessary (energy) costs and provide redundant facilities to face workload peaks or unexpected failures.

However, current solutions fail to manage large-scale networks as they are limited by the single-controller architecture. In this thesis, we argue that a multi-agent approach is needed to overcome limitations. At the same time, distributed controllers raise new challenges to be addressed, for example, controllers must cooperate by sharing their information in order to make the best decision.

In this thesis, we propose and evaluate two different multi-agent designs. A first *collaborative distributed* solution, in which controllers share their local network information to achieve a common global view of the network state to make the finest decisions; and a second *individually distributed* approach, in which controllers autonomously manage their own network area and only share topology information or its changes. To share information among controllers, the adoption of a Distributed Hash Table (DHT) is investigated. However, this approach was found to be adequate for data that require weak consistency only. In addition to evaluating different control plane architectures, we also study the impact that different data plane technologies can have. In particular, we implemented a network based on P4-enabled switches and OpenFlow-enabled switches, where the network is emulated over Mininet.

Regarding the data-plane level, we experienced how P4 implementation is more efficient, while performance with OpenFlow are fluctuating. Regarding the control-plane level, we have observed that the *individually distributed* approach, despite its simplicity and less overhead, leads to similar results, thus being a valid solution in a real deployment.

1.1 Thesis structure

The second chapter of this thesis discloses about the research solutions to the problem of a single centralized controller in the context of Software-Defined Networking, including many examples of developing physically distributed controllers, from a logical point of view to an architectural point of view. We compare the two physically distributed architecture we implemented, e.g. logically decentralized and logically distributed, to the well-known described before.
Later in the second chapter, we also discuss distributed decision processes. We focus on how other researchers implemented auto-scaling networks using different methodologies, underlying why our solution based on Multi-agent Reinforcement Learning is innovative. Some successful MARL deployments in other domains are also investigated to give an overview of how the distributed system can be designed.

Chapter 3 introduces the crucial concepts and technologies to know in order to better understand the work presented in this thesis. We start defining the path towards Software-Defined Networking, its pillars and the main protocols deployed in this context, e.g. OpenFlow and P4; we later explain two possible technologies used to share information in distributed systems such as Distributed Hash Table and Raft; finally, we explain the concept of Reinforcement Learning and how Deep Reinforcement Learning works.

We investigate how to deploy a Distributed Hash Table in Chapter 4, through a heavy hitter detection use case. We first define the concepts of heavy hitters and counting bloom filter, secondly we describe the system architecture and functioning. The implementation of data plane and control plane is explained in detail. We created a software-defined network that links a controller to every switch in the network and promote collaboration among network devices to identify and kill heavy hitter flows. To do so, an overlay network of controllers is created, and a database containing data about flows is shared in the form of a DHT.

In Chapter 5 we present our autonomous network planning framework developed in the context of a multi-controller SDN adopting MARL in two possible designs. We start defining what self-driving networks are and what challenges they rise. Then, we explain how our network is designed and how all the components interact. Finally, we concentrate on the designs we propose, the collaborative distributed solution and the individually distributed. The first design proposed leverages the use of Raft protocol to create a consistent global state of the network to share among controllers, while the second one deploys controllers that individually manage their own subnetwork.

Chapter 6 provides the description of how the implemented solutions are tested. In addition, some aspects are clarified in order to understand the results obtained.

Then, several graphs and tables show the results of our tests. We mainly compare the two auto-scaling designs and two different data plane techonologies.

Chapter 7 ends the thesis summarizing the found results and suggesting a direction for possible improvements to the system.

Chapter 2

Related work

In this chapter, we deep dive into related research works defining two sections. In Section 2.1 we present the possible implementations of physically distributed controllers in terms of architectures and logical point of view, arguing how they are implemented in real frameworks and what problems they solve. In this regard, we compare our logically decentralized and logically distributed control plane implementations to the ones previously discussed. In Section 2.2 we focus on the concept of distributed learning applied in the networking domain, arguing that machine learning techniques are efficiently leveraged. Although other works related to the concept of autonomous networks employ ML techniques, as we do, we debate how our methodologies differ and how, instead, similar techniques are employed but in other domains.

2.1 SDN controller architecture

The SDN paradigm promoted a single centralized controller capable of a network-wide view that easily managed the whole network. In fact, this approach can easily guarantee that the network is in a consistent and optimal state. However, despite the attractiveness of this scenario, this centralized architecture is limited in terms of scalability, reliability, and availability. For this reason, researches turned their gaze towards **physically-distributed** control plane architectures, which can face big dynamic networks such as data centers and WANs. In this scenario, every controller manages a network area but still can reach a global view by communicating with other controllers. This solution overcomes single point of failure and bottleneck drawbacks and provides availability, partition tolerance, and eventual consistency (CAP theorem), but brings up new challenges. Distributed controllers must synchronize their virtual network topology views, an operation that introduces overhead and becomes arduous with the increasing size of the network.

We can classify physically distributed controller, based on controller instances organization, between **flat**, **hierarchical** and **hybrid** distributed SDN controllers. Another distinction we can do is based on how knowledge is partitioned among controllers so that we can have **logically centralized** or **logically distributed** SDN control.

The SDN control is defined as **flat** when controllers are horizontally distributed over multiple network areas, where each area is managed by one controller. Every network event or information is shared among all controllers. Since many controllers are deployed, this solution decreases latency and improves resiliency.

Typical examples of flat controller platforms are ONIX [7], ONOS [8], and HyperFlow [9], designed to improve scalability by deploying multiple controllers. In all these solutions, controllers operate on a global view of the network, and the platforms themselves provide state distribution services. The differences between these architectures rely on data structures they imply and their replication mechanisms.

ONIX network state is stored in the *Network Information Base* (NIB), a graph representing all the entities in the topology. Controllers extrapolate information about the underlying topology and send it to other controllers to build the global view, later stored in the NIB. Control logic is implemented reading/writing from/to the NIB, while ONIX takes care about distributing it: it offers two types of data store based on consistency needs. For information needing strong consistency, e.g. topology structure, a replicated distributed database is provided, while for more volatile data, e.g. link usage, a Distributed Hash Table can be used.

Similarly to ONIX, ONOS controllers are aware of a portion of the network only, that they will discover and share with other controllers to construct the global view. Applications make forwarding and policy decisions based on the global view, modifying it accordingly; changes are then propagated to physical devices. ONOS exploits Raft [10] algorithm to provide distributed primitives such as ConsistentMap, which guarantees strong consistency [11].

HyperFlow differs from the previous cited architecture because all controllers have the same global view of the network and run with the illusion to have control over all of it. It is event-based and uses a publishing/subscribe message model to communicate state changes to other controller instances, synchronizing their network view. The mechanism is based on a persistent storage of events for resiliency, but only local events that change the network state are published.

All the solutions we just presented, they all are physically distributed for scalability and reliability reasons, but practically work with a **centralized logical** view. Controllers maintain a global view of the network and share network information to collaborate in the management of the network. Every controller can take decision based on the whole network status. For ONIX and ONOS, the global view is obtained exchanging information with other controllers, while HyperFlow provides controllers with the global view at the same time they are instantiated.

In this regard, another example of logically-centralized architecture is provided by OpenDayLight [12] (ODL), a collaborative open-source platform developed by Linux Foundation with the intent to ease the spread of SDN in traditional network environments. Like ONOS, it provides full functionalities of a Network Operating System, but differently, all data shared to build the network state is handled leveraging Raft algorithm.

A big limitation of the previously discussed implementations can be the static mapping between controllers and switches. Elasticon [13] is another logically centralized solution whose goal is to address this problem. It is defined as an "*elastic controller architecture*" because it permits to migrate switches from one controller to another or increase the number of controller instances based on the workload. The system is composed of three main modules: Load Measure, Load Adaptation and Decision Module. Based on the measurement of the workload of the network, when a threshold is exceeded, the Load Adaptation Module redeploys the workload to balance controller instances and can also decide to expand/reduce the controller pool. Then, the Decision Module verifies that the new configuration can handle the network load and defines new thresholds. The centralized network view is granted by the Distributed Data Store Module while inter-controller communication relies on TCP channels.

Opposed to the concept of a logical central view, there are **logically distributed** architectures, such as the one proposed by DISCO [14]. The network is divided in domains, and every controller, in charge of a single domain, manages intra-domain and inter-domain activities separately. Intra-domain operations include traditional control plane job such as managing network and flows, reacting to failures etc. Inter-domain part manages the exchange of information about hosts, metrics and topology with other controllers through a publish/subscribe channel to guarantee essential services, e.g. routing. The main difference, with respect to previous solutions, is the fact that in distributed implementations, each controller takes decisions based on its local view. Having a per domain structure, DISCO provides a distributed control plane that can manage large and heterogeneous networks such as WANs.

As we cited flat architectures, we now explain the **hierarchical** structure. The hierarchical distributed controller is a vertical model that requires many layers of controllers. Low-layer controllers directly communicate with underlying devices, upper-layer controllers achieve non-local requirements, such as inter-domain connectivity.

Kandoo [15] is an example of a two-level hierarchical control structure to improve scalability. Low-level controllers manage their domain without having global network information nor interacting with each other. Upper-layer is composed of a root controller with a centralized view of the network to provide services like routing. This configuration can linearly scale in the number of local controllers and reduces central controller overhead.

We explained flat and hierarchical architectures, therefore we can present a mixed solution: the **hybrid** approach. The hybrid approach involves both control plane and data plane in networking decisions, optimizing the use of network resources, and tries to take the advantages of both flat and hierarchical solutions.

Orion [16] controller is representative of this approach. It has a per-domain structure, deploying a controller for each domain, as a flat architecture. Inside a domain, the control plane is implemented in a hierarchical manner: it is then vertically partitioned in two layers. The bottom layer, called *Area Controller*, has access to physical devices and abstracts network topology; the upper layer, called *Domain Controller*, synchronizes the abstracted global network view. Inter controller communication relies on the Horizontal Communication Module to synchronize information and build a network wide view; intra-domain communication leverages the Vertical Communication Module: a set of TCP connections that permit to send the abstracted topology of the physical layer. Many controller modules are used, such as routing module. This solution alleviates the complexity of the control plane by dividing the area in subdomains where a hierarchical architecture is then adopted.

In this thesis, two different controller designs are compared, here referred to as logically decentralized and logically distributed architectures. The former, similarly to ONIX and ONOS, deploys multiple controllers that share their network state through Raft protocol to achieve a consistent view of the entire network state, later used for the RL decision-making process. In the latter, similarly to DISCO, controllers feed the model with their local network state and are equipped with a TCP channel for inter-controller communications in order to trade topological information for routing.

2.2 Distributed learning

Machine learning techniques are increasingly being used to understand network behaviors, provide predictive analyses, as well as provide run-time fixes for network problems. These configurations make networks capable of taking remedial actions, or even avoid some network issues, autonomously.

A Reinforcement Learning approach to automate networking tasks has been proposed with DeepConf [17]. It applies Deep Reinforcement Learning (DRL) to automate some networking tasks and proposes an auto-scaling use case where the model is trained to learn which are the best links to activate at each step based on previous traffic. In particular, the model input is a double state space composed of a traffic matrix and a network topology matrix. The network topology matrix reflects changes caused by actions, in fact, the action space determines a list of links to activate at each step. The model is rewarded considering the throughput of flows and their duration.

Some studies applied RL to optimize power-consumption by consolidating traffic in the minimum amount of links, as done in SmartFCT [18]. With the goal of optimizing power-consumption in data center networks, they deploy a DRL agent on top of the SDN controller to consolidate traffic flows in a few links. The controller periodically collects statistics querying OpenFlow switches and computes per-link throughputs which is the input of learning agents. The DRL agent will output as action a list of margin ratio for each link, which is the reserved bandwidth in a link, considering a leeway to avoid congestion caused by unpredicted bursts. Considering power minimization and Flow Completion Time (FCT), the model is rewarded. Finally, the controller will update routes based on the decision of the model and will turn off all unused links or devices for power saving.

With the same goal of reducing energy costs, a deep learning model is employed by Chen et al. in [19]. The main idea is to collect metrics thanks to SDN controller, use a deep learning model to predict the traffic and activate sleep/wake up model for some switches and links based on current and predicted traffic. They maintain a virtual topology representing the current state of the network on which routes and energy consumption are recomputed after a decision, to analyze if it leads to an improvement; if it does, sleep/wake mode is then applied.

The main difference between our work and the solutions so far proposed relies on the controller design. We propose an auto-scaling framework that reduces unnecessary power consumption, but we employ multiple controllers and manage every network area with a different model instance. Although the usage of possible distributed architecture in auto-scaling networks is mostly unexplored, Multi-Agent

Reinforcement Learning (MARL) has been successfully used in other domains, with different possible designs.

Naderializadeh et al. proposes a solution for resource management in wireless networks with distributed MARL [20]. Specifically, they try to solve the problem of user selection and optimization of resources. User equipments (UE) make periodic measurements reported to Access Points (AP) with a certain delay; these measurements are then shared among APs. The distributed approach sees every AP equipped with a MARL agent that makes decisions on its own even though it also exchanges information with neighbors. They consider a *centralized training and distributed execution* which means that all agents train a single policy, thus have the same reward feedback. Since they want to jointly optimize resource and user scheduling, actions are defined as transmit power-UE pairs.

Mai et al. [21] deals with in-network load balancing and utilizes Multi-agent Actor-Critic RL to split traffic across multiple paths to minimize link occupancy and supporting size of flows. Each switch of the network is an agent and takes actions based on its local observation, while a *centralized critic* observes the global state of the network to give agents a reward so that they are globally coordinated. Reward is produced in order to minimize the maximum link utilization after a switch decides to send a fraction of flow across a link.

The collaborative approach proposed in the last solution is abandoned in [22], which focuses on routing with multiagent DRL in order reducing packet delivery time. Each router is also an agent and has its proper neural network with proper parameters that are not shared. Since it is difficult for a router to have a global view in real networks, routers exchange information. Each router has to decide who will be the next hop for a certain packet based on the state composed of the final destination, proper information and neighbor nodes shared information. The reward will be assigned minimizing the average delivery time.

A similar approach, with multiple model that don't share parameters, is adopted in Magnetic [23], which applies multi-agent machine learning to the problem of virtual machine (VM) consolidation With the goal of minimizing energy consumption in data centers. To choose the best host power mode, they rely on ML: they deploy one learning agent per host that takes decisions based on its state. In fact, no information about other agents is known. A central logic unit will take care of VM migration.

As we presented some examples of MARL application in networking environment, two possible design concepts can be distinguished. In some solutions, multi-agents cooperate on the achievement of a common goal, either sharing model parameters or training the same policy. Another possible approach deploys multiple agents that are unaware of the other learning units. This thesis investigates the use of a multi-agent reinforcement learning model for auto-scaling decisions in an SDN environment with two possible alternatives for distributing operations: a collaborative one, where controllers share the same observations, and an individual one, where controllers make decisions according to their own logic and do not share any model parameters.

Chapter 3

Background

This chapter will introduce the main concepts needed to better understand this thesis work. We will explain the Software Defined Networking paradigm and the main technologies used to enable it; we will describe what a Distributed Hash Table is, what Raft algorithm is and finally we will introduce the concept of Reinforcement Learning and Deep Reinforcement Learning.

3.1 Network device architecture

The architecture of many network devices is composed of three parts called "planes" which are in charge of different operations (Fig. 3.1).

1. **Data plane**: in charge of forwarding packets. It is designed to be as fast as possible to support modern network speed. It receives, processes and forwards traffic that traverse the device following predefined rules established by the controller. Commonly, a *fast/slow path* design is implemented: most of the packets follow a fast path because they match some configured rule and the output port is simply determined; other packets are sent to the controller for further elaborations and that's why this is called slow path. The data plane is hardware friendly because of the high speed required, simple tasks and rare updates, in fact it is mainly implemented through ASICs, FPGA or NPUs.

2. **Control plane**: it is in charge of configuring how data plane should forward packets, populating tables. Sometimes, it manages particular packets that don't fit any kind of rule. Its job depends on the purpose of the network device. For example, in case of a router device, the control plane executes routing protocols, performs connectivity management, adjacent device discovery, service provisioning. For firewall applications, it is programmed to react

dynamically to possible threats. It is implemented in software, because of its slow tasks, through general purpose CPUs.

3. **Management plane**: it manages all the devices, monitors traffic, thus its use is destined to the network administrator.

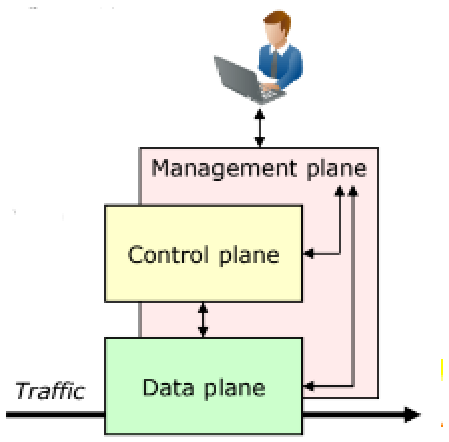


Figure 3.1: Network device architecture [24].

3.2 Software-defined networking

The tightly coupled control and data planes of traditional networks fostered vertically consolidated markets that were closed to innovation. The control over the network was limited by proprietary solutions, which caused difficulties in the management of the network because of different hardware with different functionalities.

This closed infrastructure couldn't provide control over the network or the applications, the attractiveness of programmable devices grew. This necessity opened the path for new innovations, like the separation of control and data plane and the introduction of open interfaces between them, thus posing the basis for the SDN paradigm.

Software-defined networking is a paradigm born to make networks more programmable in order to simplify management operations, increase flexibility and enable innovation. SDN paradigm key ideas were:

- **Decoupling data plane and control plane**: forwarding switches are conceived as simple devices that follow the logic established by the control plane through a software application.

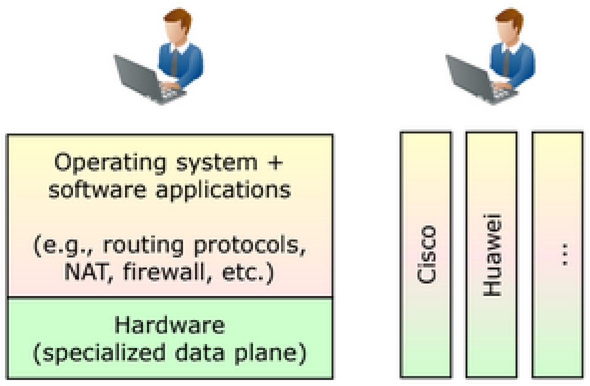


Figure 3.2: Traditional networking domain. Adapted from [25].

- **Centralizing the control** in one single object that has a complete view over the network to easily control it.

- **Context-based forwarding** enabled by the "big switch" view of the control plane that abstracts the network, allowing to set high-level rules.

The focus on separating hardware and software implementations, dynamic forwarding and high programmability led to the integration of machine learning approaches to collect data, manage the network and improve performance.

3.2.1 Data plane and control plane

The concept of separation emerged in the early 2000s because of the increasing volume in traffic and the will to manage flows controlling their paths. To sustain the higher links speed, equipment vendors started to program forwarding logic in hardware, thus practically separating it from the control plane in software. This new idea permitted the independent evolution of the two components and to two fundamental innovations: open interfaces between data and control plane and a centralized controller for the network.

In this view, the control plane logic is implemented in software and exports an open interface so that making changes becomes easier. The forwarding devices are simple switches that follow configured rules, so they become faster and cheaper. A standard southbound interface, like OpenFlow, guarantees interoperability among different vendors devices.

3.2.2 Controller

SDN paradigm promoted the idea of a centralized controller that could have a global view of the network to facilitate management and control, but this idea was found unfeasible to apply in real scenarios. In fact, SDN centralized controller was found to have scalability, reliability, availability issues and does not scale with large networks. One of the first solutions was to replicate the controller for failure recovery, but consistency issues arose, so the research focused on the idea of a **physical distributed controller**, which is more complex but more robust.

Depending on the needs, various design are available for physical distributed controllers:

- **Flat** SDN control: Controllers are horizontally distributed over multiple network areas, where each area is managed by one controller. Every network event or information is shared among all controllers. Since many controllers are deployed, this solution decreases latency and improves resiliency.
- **Hierarchical** SDN control: This vertical model requires many layers of controllers based on the services they provide. Low-layer controller directly communicates with underlying devices but do not have global view of the network nor can interact with each other. Upper-layer controllers achieve non-local requirements, such as inter-domain connectivity.
- **Hybrid** approach: The hybrid approach involves both control plane and data plane in networking decisions, optimizing the use of network resources.

Another distinction can be made inside physical distributed controllers:

- **Logically centralized**: Controllers maintain a global view of the network and share network information to collaborate in the management of the network. Every controller can take decision based on the whole network status.
- **Logically distributed**: Network is separated in domains, so we can distinguish intra-domain and inter-domain activities. Intra-domain operations require configuring network devices, retrieve metrics from them, flow engineering. Inter-domain part handles communication with other controllers for the exchange of information needed for services, such as routing. Every controller takes decision for his area based on local information.

3.2.3 Context-based forwarding

With this new configuration, the control plane can take decisions about forwarding at run-time, for example, based on the current traffic (the context). The switch can send a packet to the intelligent controller that inspects the packet and tell the switch where to forward it, potentially installing new forwarding rules.

3.2.4 Architectural overview

SDN architecture is composed of three main layers connected by southbound and northbound APIs:

- **Application layer**: control program designed to implement the network control logic. It may run locally or on a remote machine.
- **Northbound interface**: well-defined API.
- **Control layer** : also referred to as "Network Operating system", it represents the brain of the network. It translates commands from the application to the underlying layer and get information back. It provides a global view of the network to the upper layer.
- **Southbound interface**: standard open interface towards physical devices.
- **Data layer**: it identifies the distributed forwarding elements.

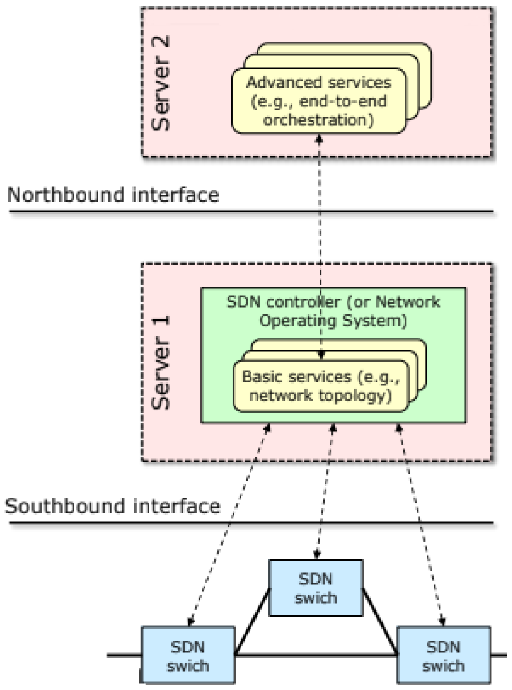

Server 2
Advanced services (e.g., end-to-end orchestration)
Northbound interface
Server 1
SDN controller (or Network Operating System)
Basic services (e.g., network topology)
Southbound interface
SDN swich
SDN swich
SDN swich

Figure 3.3: Software-defined network architecture [25].

Due to its wide adoption, OpenFlow became the de facto standard for the Southbound interface.

3.3 OpenFlow

OpenFlow [26] is a protocol that allows the control plane to access and control forwarding devices from a central point. It is an open interface for network abstraction layer, in fact, the controller doesn't need to be directly attached to the network devices. OpenFlow is just one way to achieve abstraction, but in 2011, Open Networking Foundation defined OpenFlow the de facto standard for SDN Southbound interface.

The OpenFlow architecture defines a centralized controller that communicates with switches to populate forwarding tables, to collect statistics or to learn the status of the network; an OpenFlow-enabled switch who does nothing but follows rules; a secure channel for the communication between them, see Fig. 3.4.

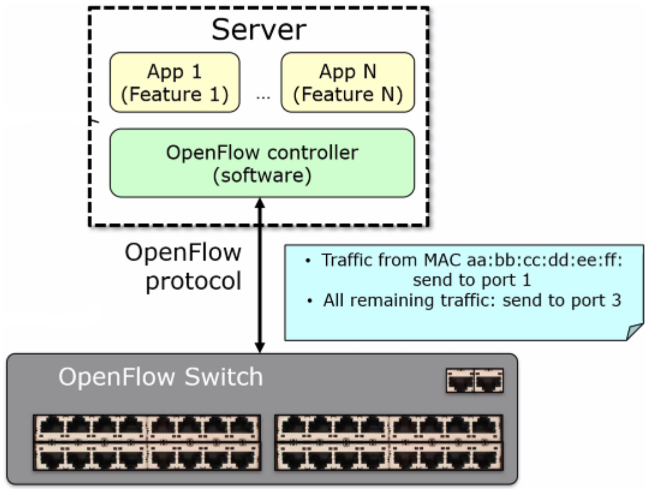

Server
App 1 (Feature 1)
...
App N (Feature N)
OpenFlow controller (software)
OpenFlow protocol
• Traffic from MAC aa:bb:cc:dd:ee:ff: send to port 1
• All remaining traffic: send to port 3
OpenFlow Switch

Figure 3.4: OpenFlow architecture [27].

Forwarding is based on rules (match-action) decided by the user and injected into the switch by the controller. They can be configured in two ways:

- **Proactive** mode: the forwarding table is populated a priori.
- **Reactive** mode: the only configured rule sends packets to the controller (table-miss entry), which will establish and write a new FlowMod.

The OpenFlow switch is composed of one or more *flow tables*, *group tables*, *meter tables* and some OpenFlow channels. The pipeline is divided in *ingress* and *egress*, where flow tables are queried in series, so if the match starts at the first flow table, it may continue in others in the pipeline.

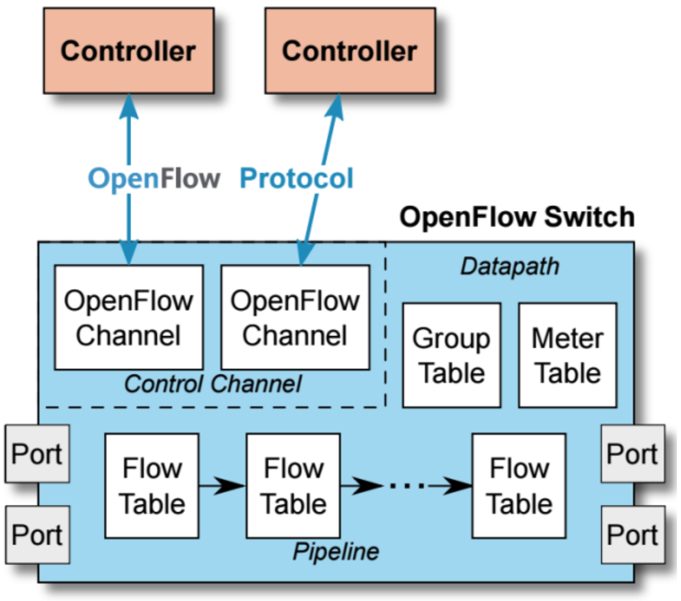


Figure 3.5: OpenFlow switch [26].

Every flow table entry is formed by:

- Match fields: values used for matching. Typically, involve input port and header fields.
- Priority: defines the importance of this entry over others.
- Counters: they are incremented whenever a packet matches.
- Instructions: actions to perform when the packet matches.
- Timeouts: indicates for how much time the rule is active.
- Cookie: used by the controller to grab statistics about the entry.
- Flags: used to distinguish the management of entries.

OpenFlow also offers the possibility to apply some behaviors to multiple entries through the configuration of a *group table*. Instead, The *meter table* is used to

implement QoS operations. The OpenFlow Channels present in the switch, are used to communicate with the control plane.
Although multiple connection can be established with multiple controllers, OpenFlow promote a single centralized controller with a global view of the network, and delivers multiple instances in a master-slave configuration for resiliency purposes only.

One of the most employed controllers is Ryu.

3.3.1 Ryu controller

The most adopted SDN Controller is Ryu [1], an open-source component-based software defined networking framework that exports well-defined APIs to easily create applications that can manage network devices. It provides some components that can be modified or extended to build SDN applications. It supports many protocols, among them, we find OpenFlow.

Ryu applications are single-threaded entities which implement various functionalities in Ryu. Events are messages between applications, and one example of such event sources is OpenFlow controller. A Ryu application can register itself to listen for specific events, so if it subscribes for OpenFlow messages, whenever a switch sends a message, this will be automatically decoded. Any OpenFlow event has at least a message object, a datapath identifier to clarify which switch is the source of the message, a timestamp. Based on the specific goal of the message, new attributes can be present. Typical events to subscribe to are: the connection/disconnection of a switch, the change in the status of a port, host registration.

Ryu controller can modify Flow tables, establishing new rules, deleting old ones, modifying match fields or actions etc. It is also used to collect metrics from switches, regarding the state of ports, tables, flows that can also be aggregated. Metrics are collected thanks to counters integrated in the tables, that measure bytes, packets or flows. The possible actions to perform upon the matching of a packet are: choose output port for the forwarding of the packet, change Ethernet source/destination address, change IP source/destination address, set TCP/UDP port, set a VLAN id, set the queue to which send the packet.

[1]https://ryu-sdn.org/

3.3.2 Conlcusion

In conclusion, the OpenFlow approach is flexible, enables context-based decisions through the use of a controller but is complex, suffers from scalability and performance issues and does not support new header definitions. To overcome these limitations, new protocols were proposed, such as P4.

3.4 P4

Programming Protocol-independent Packet Processors (P4) [28] is a high level programming language to describe packet forwarding and a protocol independent architecture. Usually, network devices are designed with a "bottom-up" approach, which means that they are based on fixed-function chip from third-party vendors. P4 promotes the opposite approach, which is "top-down". With this method, the programmer establishes the features set in a P4 program, which is later compiled and pushed into the network device to configure. The chips that support this approach contain programmable blocks.

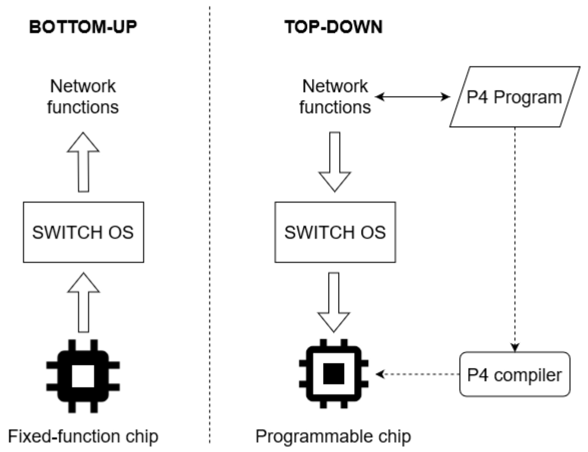


Figure 3.6: Bottom-up vs top-down approach.

The tree main goals of P4 are:

- **Reconfigurability**: controller should be able to redefine processing and parsing of packets
- **Protocol independence**: controller should be able to create and parse new headers with new fields and a set of match-action tables to process them.
- **Target independence**: the programmer doesn't need to know implementation details of the underlying switch, instead, specific compilers are used.

3.4.1 Abstract forwarding model

A generic P4 program is made of a parser, which is implemented as a finite-state machine, a match-action pipeline and a deparser. Differently to OpenFlow, P4 parser is programmable to create new packet headers and match-action pipeline stages can be in series or in parallel. When a packet arrives to the switch, the parser extracts headers fields from the packet and pass them to match-action tables. Two tables of this kind are present: ingress and egress. Ingress processing determines the output port of the packet in case of forwarding, but the packet can also be dropped, sent to the controller or re-inserted into the pipeline. Egress pipeline executes modification of the headers, e.g. remove some headers before sending the packet to the controller.

A P4 switch can maintain a state thanks to tables, registers, counters and meters: tables trigger some actions in case keys match, registers are used to store data and can be written/read both from data plane and control plane, counters and meters are mainly used to collect metrics and act accordingly. Communication between data plane and control plane can be established through digest or cloning.

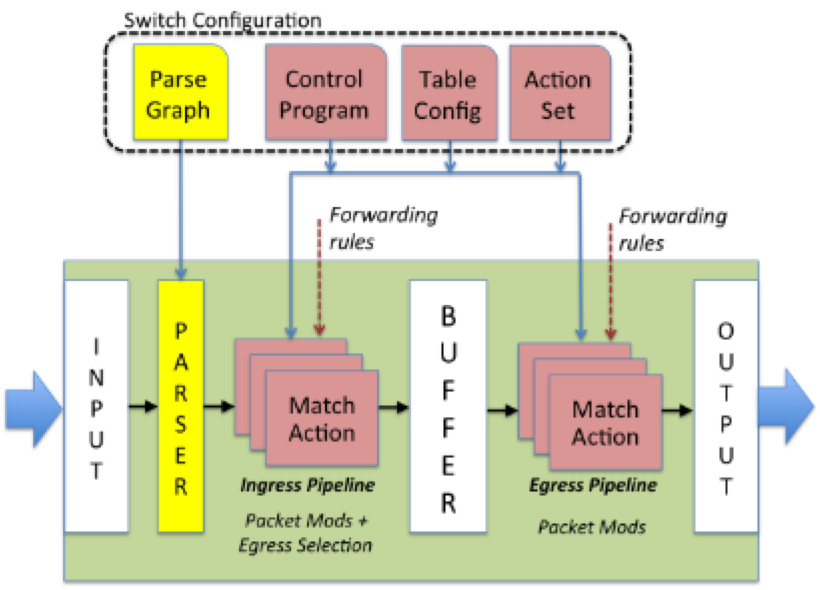


Figure 3.7: P4 abstract forwarding model [28].

3.4.2 Controller

We can use the Thrift API to control the switch establishing a connection with the server running on the switch. We can use the controller to populate tables, read registers, counters. Two methods are possible for switch-to-controller communication: digest and cloning.

Digest extern permits to send some small information to a controller. Receiving digested packets is not simple because the switch includes a control header that must be processed, furthermore the switch expects an acknowledgement message for each digested packet (used to filter duplicates). A packet's digest often has a lesser size than the packet itself, since it can only be used to deliver a portion of the headers or metadata related to the packet.

Cloning extern permits to clone a packet and send it to the egress pipeline or back to ingress pipeline for further elaborations. When a packet is cloned, a mirror ID can be configured to decide to which port the packet should be cloned, so it can identify the CPU port to send it to the controller.

3.5 Distributed hash table

Distributed Hash Table (DHT) [29] is widely adopted in large-scale distributed systems because it manages resources efficiently and develops a Peer-to-Peer (P2P) network, which offers better robustness and node's capability exploitation than a client/server model.

A Distributed Hash Table is a database that, as the name suggests, differently from a simple centralized version, is distributed among many nodes. Since it deploys a P2P network, where peers are nodes with the same functionalities, it doesn't need a central server, which usually behave as a bottleneck, and it also provides the advantages of hash functions, e.g., high efficient searching.

The peers in a DHT network are identified with an integer ID and store a part of the hash table, which means that are responsible for modifying, deleting or adding these entries. Elements of the hash table are treated as (key, value) pairs, and every peer can query it if it knows the key: the database will then locate the peer who has that entry and returns the (key, value) pair to the asking peer. The mapping between entries and peers is done through the use of a hash function, in fact, every non integer key is hashed to map it into a range of values used to identify the peers. The peer who is responsible for a key is the *closest* to it, so a distance function has to be defined.

One of the most successful DHT variants is Chord [30]. It is a distributed lookup protocol designed to efficiently locate peers and guaranteeing load balancing, availability, scalability and decentralization. Chord deploys a circular overlay network, denominated *Chord ring*, where peers are distributed clockwise in ascending order. The distance function, used to locate the node responsible for a key, is given by the Euclidean distance in the one-dimensional ring space. To efficiently address the joining/leaving of nodes, every peer maintains a list of successors. Chord heavily inspired our DHT design.

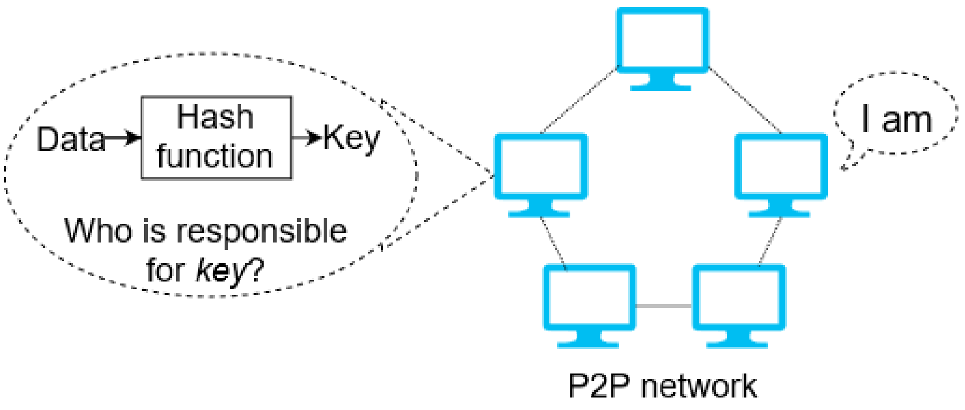


Figure 3.8: How to query a Distributed Hash Table.

We can summarize the properties of a DHT as:

- High efficiency. Inherited by the hash table, this property concerns the efficient location of data. It is very desirable in a large-scale network.
- Decentralization. The hash table is divided in portions, each assigned to a peer. There is no central entity, thus hot spots are avoided, instead, a good balance is achieved.
- Scalability. As a decentralized design, it can be applied to all type of systems, from small ones to very big ones. However, it must handle node removal/addition and node failures.

As a result of its properties, the DHT is widely adopted. It is adopted by search engines, for storage purposes and to efficiently implement file sharing. The traditional client/server system, where all data are stored in a server, lacks of scalability and charges the central entity with too many responsibilities. A first improvement was achieved with the distribution of the content to peers, charging the server to maintain the peers indexes only. One step forward is taken by BitTorrent [31], a self-scaling file-sharing P2P application. It leverages the DHT to build a distributed tracker. In BitTorrent, a *tracker* maintains a list of active peers interested in a file [32] which is associated to a *torrent identifier*. When a new peer joins the network, it can query the DHT with the torrent identifier used as key to find the peer responsible for it in order to track all the participating nodes for that file.

A DHT-based approach can achieve weak decentralized consensus, but when nodes join or leave the network, inconsistency issues may arise for some period. To achieve strong consistency, other protocols, e.g. Raft, may be needed.

3.6 Raft

Raft is a **consensus** algorithm used in distributed systems. Since some machines can be faulty or unreliable, nodes need to agree on single values to create a reliable large-scale distributed system. For a long time, Paxos was the most used consensus algorithm, but since it was difficult to understand and to apply in practical context, new approaches were developed, such as Raft. Raft uses similar ideas but breaks down the concept in different separated tasks:

- Leader election: the algorithm consists on the election of a ***leader*** who takes decisions about the state and replicates them among others. Nodes start as *followers* and can *candidate* themselves when leader fails, vote for themselves if candidates and send Request Vote RPCs. Whenever a new leader is elected, a new *term* starts. In case of draw, the process is repeated.

- Log replication: the leader communicates with the client and executes commands. Every command is appended in its logs and then sent to all other nodes through Append Entries RPCs to replicate the entry. Whenever the majority of nodes has committed the current entry, the leader updates its state machine and responds to the client. Persistent state of servers include: log entries with commands received for every term, candidate ID the server has voted for in this term, current state machine value.

- Safety: the goal is that every node must execute the same commands in the same order, and some restrictions are considered in leader election to guarantee it. For example, new leader must contain all previous committed entries. For this reason, if a leader discovers another node with a higher term, it will proceed to copy the missing entries into logs and become a follower.

Summarizing, Raft algorithm achieves consensus through the election of a leader, which takes decisions and replicate them among other nodes. To guarantee strong consistency, the state machine is changed only when every node in the network

has logged the new decision, to correctly handle eventual failures. In this way, the correct order of instructions is also preserved.
The whole state machine diagram describing Raft protocol is pictured in Fig. 3.9

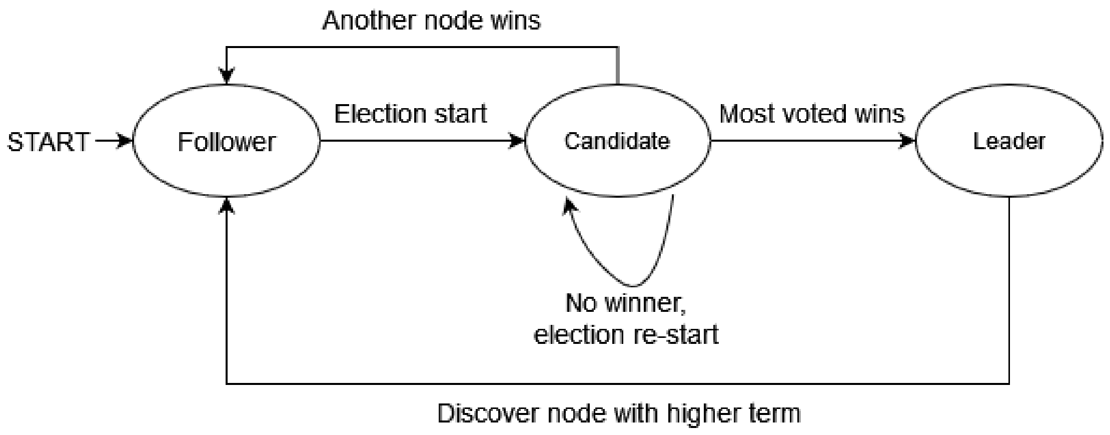


Figure 3.9: State machine diagram for Raft protocol.

3.7 Reinforcement Learning

Reinforcement learning is a part of Machine Learning whose principles are based on human learning. It is widely deployed in many fields, from healthcare, image processing, robotics, to the networking field. Here we illustrate the basic idea at the core of RL and explain the terminology used.

We first define some keywords:

- **Agent** : The agent is the operator that has to learn how to maximize the reward function, interacting with the environment.
- **State** : The state represents the environment in a certain moment.
- **Action** : An action is performed by the agent and modifies the environment.
- **Environment** : The environment is everything that surrounds the agent. When actions are performed, it goes into a new state e returns a reward based on the desired final state.
- **Reward** : Scalar value that represent how good is the action performed.

We can observe a graphical representation of how RL works in Fig. 3.10. The agent interacts with the environment, performing actions with some effects. The environment changes and returns the new state representation and a reward to the agent. This interaction goes on continuously until the agent has sufficiently explored his possibilities and, knowing the consequences of his actions, it now knows how to act to maximize the reward function, which represents its goal.

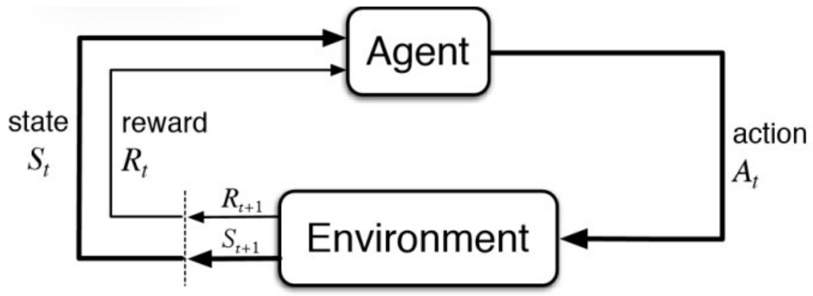


Figure 3.10: Basic diagram of RL [33].

3.7.1 (Deep) Q-Learning

Q-Learning [34] is a model-free reinforcement learning algorithm based on a value function, because it maps a value to the action performed in a particular state:

$$Q(s, a) = r + \gamma V_{s'} \tag{3.1}$$

The optimal action-value function has the highest value for all states, the optimal Q-function tells how good is to perform an action in a given state. There exists a relationship between these functions:

$$V^*(s) = max_a Q^*(s, a) \quad \forall s \in S \tag{3.2}$$

In other words, the highest Q-value over all feasible actions in state s is the maximum expected total reward. The Q-value for a pair action-state is computed through the Bellman equation:

$$Q(s, a) = r + \gamma max_{a'} Q(s', a') \tag{3.3}$$

where γ is a discount factor between 0 and 1. This tells us that the maximum reward is given by the reward obtained for arriving at this state plus the future maximum reward for the next state. The Q-Table is initialized with zero values and updated at each step using approximations between the old and new Q-Value:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma max_{a'} Q(s', a')) \tag{3.4}$$

where α is a tunable hyperparameter known as learning rate.

Since a Q-table cannot scale with bigger state and action space, we employ **Deep Q-Learning** algorithm [35] which incorporates the use of artificial neural networks to the idea of Q-Learning. Neural networks are function approximators, which means that we can train them on samples from the state space to learn to predict the action to perform when new states are encountered. For this reason, this is particularly useful when the state space is very large. The state is now entered into a Deep Neural Network, which generates a different Q-value for each action. Once more, we select the course of action with the highest Q-value. The iterative update strategy keeps the learning process constant, however here we update the weights in the neural network rather than the Q-Table to enhance the outputs.

Chapter 4

Distributed Hash Table

In this chapter, we study the deployment of a Distributed Hash Table as a way of sharing information among controllers in a multi-controller software-defined network. The use case presented concerns the detection of heavy hitters through a counting bloom filter. We will first explain the concepts of heavy hitters and counting bloom filter, then we present our solution in detail.

4.1 Heavy hitter detection

Heavy hitters can be defined as network sources sending unusual large traffic and can be categorized by source IP address or flow's 5-tuple. Since network measurements show that few heavy hitters account for a large percentage of the traffic [36], their detection can be very important for many application, e.g. routing optimization, traffic engineering, Dos detection.

4.2 Counting bloom filter

A bloom filter is a probabilistic data structure used to test if an element belongs to a set. It maps elements of a set into a bit array through the use of multiple hash functions, and can tell whether an element is present if all entries to which it is mapped are 1.

A **counting bloom filter** is similar to a bloom filter but uses counters instead of binary values and permits to query a dynamic set, because elements can be deleted. Every element is mapped through multiple hash functions to an array, whose values are updated according to the operation to perform. To establish if a certain item belongs to the set, all the cells to which it is mapped must exceed a threshold value. This is the data structure we employ in our solution, it is represented in Fig. 4.1.

The use of hash functions in probabilistic data structures enhances look-up performance and space efficiency. Despite this, because of hash collisions, false positives are possible, so an error rate is introduced. Bloom filters can be properly dimensioned to trade off between space and accuracy according to the application requirements.

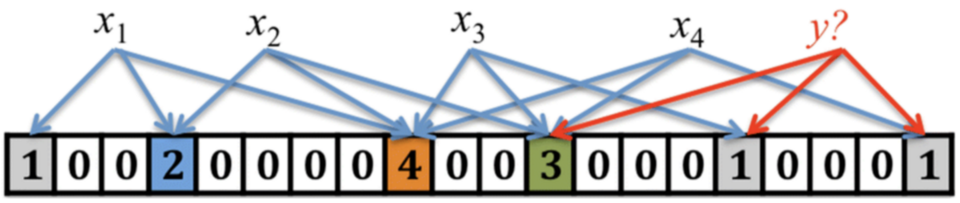


Figure 4.1: Counting bloom filter [37]. Every element is mapped to cells through multiple hash functions. When all the cells exceed the established threshold, the element is considered part of the set.

4.3 System overview

This implementation is based on a distributed SDN environment, where programmable switches are leveraged for efficient measurements, each associated to a controller. Controllers are nodes in a decentralized peer-to-peer network (*overlay network*), which is deployed to maintain a DHT. The DHT is a counting bloom filter which keeps track of heavy hitter flows in order to stop them. Fig. 4.2 represents the interaction among these elements.

4.4 Data plane

To build the data plane, the P4 technology is employed, which permits to program the switch device as needed. For this solution, we program a new table called *Heavy_hitter* that will contain the TCP 5-tuple of all flows that should be stopped. The 5-tuple is defined by:

- IPv4 source address
- IPv4 destination address
- TCP source port
- TCP destination port
- IPv4 protocol field

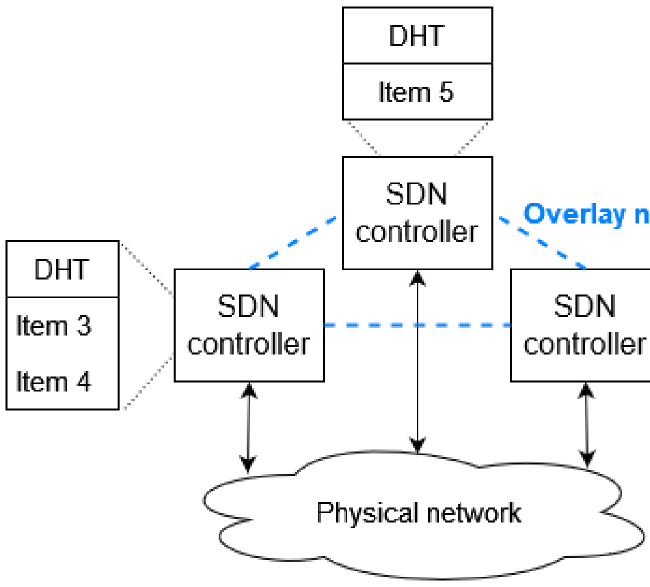


Figure 4.2: System design. Controller form an overlay network to exchange information about their portion of distributed counting bloom filter.

Whenever a TCP packet is received, the switch will check the *Heavy_hitter* table and in case of hit it will drop the packet. Otherwise, the packet will be sent to the controller, which updates the counting bloom filter, decides if the packet belongs to a heavy hitter flow and eventually populates the switch forwarding table accordingly. Two methods were tested for the communication with the control plane: digest and cloning.

The **Digest** extern function permits to send some small information to the controller, typically a subset of the headers or metadata associated with the packet, while the **Cloning** extern function permit to clone a packet and send it to the CPU port to reach the controller. We found that the digest method couldn't keep up with the high number of messages per second that our controllers should receive, due to the overhead associated with parsing operations, while cloning option was found to properly sustain the high rate of messages.

4.5 Control plane

The control plane is implemented in a distributed manner: every switch has its own controller. Controllers form an overlay network to share their information, since they own a portion of the distributed counting bloom filter. During the initial phase, every controller is instantiated and assigned with an integer ID, based on which a circular linked list of controllers is created. Integers IDs are used to associate keys to peers, since keys are mapped into integers in the range $[0,2^n - 1]$ where n is the number of bits used. The mapping between controller and keys is done through a **distance function**: given a key, the peer who must handle it is the closest one, here defined as the closest successor of the key based on its identifier. We can use Fig. 4.3 to clarify with an example. Controller with ID 12 is the closest successor, then responsible, for keys 7,8,9,10,11 and 12.

When a controller needs to update or read a table entry, it has to contact the responsible for that key, but keeping track of all the peers in the overlay network is not a scalable approach; keeping track of the predecessor and successor only, generates $O(N)$ messages per query, so a tradeoff is here implemented which consists on the use of shortcuts. The number of neighbors per peer and the number of messages per query are $O(logN)$ in a circular DHT-based network with shortcuts.

Every controller will put all CPU messages received by the switch into a queue and will eventually process them. When a packet is taken from the queue, TCP's 5-tuple is extracted to identify the flow and hash functions are applied in order to obtain its keys and update the bloom filter after having contacted the controller responsible for that entries. To check if a flow is in the bloom filter, which means that it is considered a heavy hitter, both entries to which it is mapped must have counted more packets than the threshold. When a threshold of packets is exceeded, an entry identifying that flow will be added in the heavy hitter table of the switch. The counting bloom filter here uses two hash functions, CRC16 and CRC32, therefore every TCP flow will be associated to two different keys used as indexes for accessing the bloom filter. Since the number of flows in our network is pretty limited, the use of two hash functions is sufficient for us to balance space occupancy and performance.

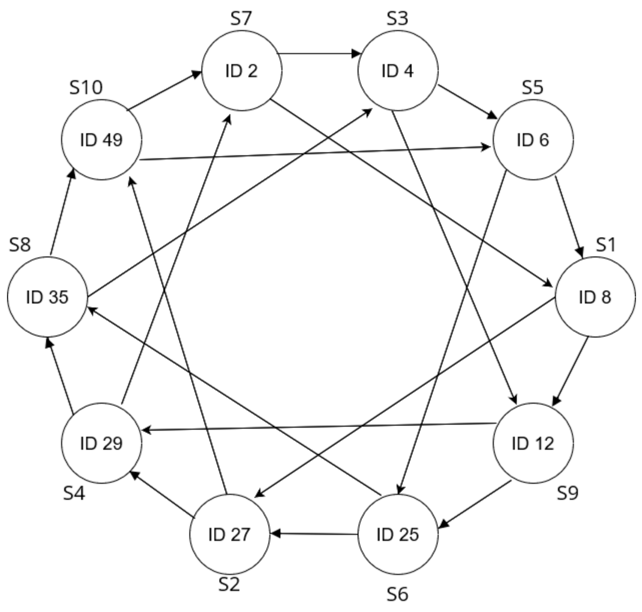


Figure 4.3: Overlay network with shortcuts. Every controller has an integer ID used in the distance function to determine who is responsible for a key.

Chapter 5

Auto-scaling decisions

In this chapter, we discuss the path toward our autonomous network planning framework, here presented in two different designs, collaborative distributed and individually distributed. The differences among these design choices are stressed in this chapter, while results are evaluated in Chapter 6.

5.1 Self-driving networks

The rapid expansion of the Internet led to over-provisioning and high power consumption. To face traffic spikes, failures and congestion, redundant devices and links are deployed which result in additional energy consumption not always well exploited. Backbone average link utilization is estimated to be around 30-40% [38] but energy consumption doesn't depend much on the work load, so sleep states for links and devices or dynamic resource adaption are required for power savings [39].

The SDN paradigm brought innovation, re-configurability and flexibility in the network management field. A possible way to exploit this flexibility is to move towards a fully automated network, often referred to as *self-driving* network. Self-driving networks can measure, analyze, and control themselves in an automated manner, reacting to environmental changes, e.g., traffic volume, while adjusting and optimizing themselves as needed [40]. Increased automation has the ability to not only make network operations simpler, but also make it possible to perform more precise improvements by utilizing all the network data available rather than just relying on established models. Any algorithm for network optimization must contend with the difficulty of the future's inherent uncertainty: there might be an unanticipated increase in demand, an edge failure, or any other unexpected behavior of a component inside or outside the network. As a result, a self-driving network should not only adapt to the current environment as best it can, but also be ready for any requirements that may arise in the near future. In other words, networks need to be robustly optimized, taking both current and potential future demands into account.

With this thesis, we attempt to build a completely automated network that dynamically adapts resources based on traffic need, with the goal of reducing power consumption.

5.2 System design

In particular, in this chapter, we propose an autonomous network planning framework that can dynamically scale up and down network resources as traffic volume changes to guarantee high applicative throughput in a multi-controller environment through the use of Mutli-Agent RL. The basic idea of our solution is to activate supporting switches that are typically unused, but that can be leveraged to deal successfully with congestion. On the contrary, to consolidate resources, they can be deactivated in order to save computing power.

We sketch the main operations of our solution in Figure 5.1. First, the multiple SDN controllers deployed collect and maintain traffic statistics from local switches and calculate the throughput of flows. Based on the chosen design, specific information are exchanged with other controllers and the network state is built up. The DRL model generates the best action to take based on the input received and gets rewarded. The action list is sent to the power optimization module that may change the status of supporting switches. In these circumstances, the virtual topology view is modified and re-routing is performed, preferring less congested paths.

5.2.1 DRL formulation

Our solution employs a Deep Reinforcement Learning (DRL) approach where the agent observes the state of the environment, our network, and generates an action that will alter the environment. Every action is rewarded with a scalar value to learn the best policy to actuate. The DRL agent receives the inputs, selects the best action, uses the reward value to evaluate the goodness of the chosen action, and proceeds with this process continuously.

We deploy the Deep Q-Learning algorithm because it is appropriate for a large state space, since it uses a deep neural network to approximate state values. Differently from vanilla Q-Learning, the neural network in this implementation maps input states to pairs of (action, Q-value), where the *Q-value* is the maximum estimated return for taking action into a given state, but, equivalently, updates the neural network weights conforming to the Bellman Equation.

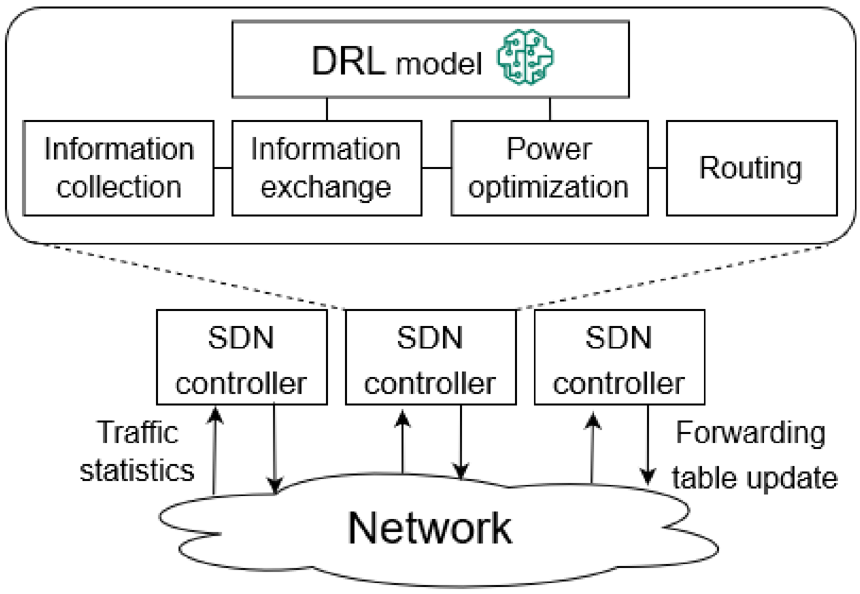


Figure 5.1: System architecture. The SDN controllers receive in input the network statistics and, using the DRL model, select the best set of network resources to deploy. According to this decision, routing tables should be appropriately updated.

During the training phase, we adopt the *Epsilon-Greedy* method in order to balance *exploration* and *exploitation*. The model initially explores the environment by taking randomized actions and stores the received reward values. As the training proceeds, the probability to take random actions is decreased, and the temporarily learned policy is leveraged to choose the action with the highest return.

To optimize the learning process, the RL agent periodically makes use of an *experience replay buffer*. The learning phase is separated from acting and relies upon randomly sampled batches of recorded data. This solution removes temporal dependency built into the data and treats our data as identically distributed.

Another improvement we introduced in the algorithm is the use of two different neural networks, *main* and *target* network. This approach is called Deep Q Network (DQN) and uses the target network to compute $Q(s', a')$ while the main network is used for the calculation of $Q(s, a)$. Their weights are updated with a different frequency, in particular, the target network is the copy of the main network in a previous instant. This solution overcomes possible training instabilities.

In our models' implementation, agents have two fully connected layers that elaborate the input, followed by a Softmax layer that converts output to a distribution of probabilities. When the model produces an output, the action with the highest probability is accomplished.

As in any other RL-based algorithm, we need to define the main variables of the framework: state, action, and reward. In our solution, we have the following mappings:

- **State space:** The agent state space is based on the load of links. The number of links considered as input depends on the specific distribution framework selected.
- **Action space:** For each supporting switch, the model defines if such a switch has to be powered on or off in the current step.
- **Reward:** The purpose of the model is to find the best resource allocation that maximizes the network-aware reward, where our objective is to detect the minimal set of active resources that can satisfy traffic demands. With this in mind, we construct the reward function considering the network performance and a penalization for resource over-provisioning to discourage a scenario where supporting switches are active but unhelpful to traffic. The reward equation results as the average network throughput deducted by the power associated with each activated supporting switch. It is here defined:

  $$R = \frac{1}{k}\sum_{i=1}^{k} B_i - \lambda \sum_{j=1}^{N} S_j \tag{5.1}$$

  where k indicates the number of active flows in the last t seconds, used to measure the average throughput, and λ is the power associated to the N active supporting switches.

5.2.2 Data plane implementation

The proposed framework is tested under two different data plane techonologies, P4 and OpenFlow. Therefore, we here describe the two implementations, underlying the differences between them.

Every controller we deploy in our network interacts with the underlying switches to collect metrics and to configure rules in the forwarding tables. These two operations generate sufficient differences in the two technologies we study.

OpenFlow-based switches are managed by the Ryu controller, which exports some APIs to create flexible applications that can manage network devices. Every OF switch maintains counters for each table to collect metrics about flows or entries. To interrogate the switch, Ryu exports multiple APIs for request with different granularity. In our implementation, we need to collect port statistics, e.g., the number of packets that traverse a switch, and flow statistics, e.g. the throughput. The request messages used are *OFPPortStatsRequest* and *OFPFlowStatsRequest*. Ryu simplifies a lot the work of the programmer, but this simple approach is the cause of some troubles, since, to collect individual flows statistics, we had to configure rules for every possible flow in the network, creating tables with many forwarding rules that increased the lookup time.

P4-based switches, instead, let us define custom counters to associate directly or indirectly to tables. In this case, we defined a counter for ingress packets, indexed by the port used, that is incremented in the Ingress processing as first operation, and a counter for egress packets that is incremented after the forwarding port is established. To compute the throughput of flows, the switch maintains a counter of bytes for every TCP flow, indexed by the hashed 5-tuple value. This configurability provided by P4 enables flexibility and efficiency.

5.3 Collaborative distributed

The first design choice we propose deploys multiple controllers training DRL agents that learn how to produce collaborative behaviors. To achieve this goal, controllers agree on a common input to feed their model, which represents the state of the global network.

In particular, the **state space** we use is the aggregate of subnetworks information on link utilization and the state of all supporting switches, i.e., whether they are active or not. In such a manner, agents can see the actions performed by other controllers and their effects on the network, then they can make a decision.

Actions about the planning of network resources of the SDN controller, instead, regard the single subnetwork. In other words, the model is logically centralized, but it runs in a multi-agent setting for reliability and scalability reasons.

Data consistency and consensus

Since we want our controllers to have a global view of the network, the information they collect needs to be combined and distributed among all instances. In our systems, agents must agree on the global state of the network, constituted by links utilization and helping switch status, which is later sent as input to the multiple instances of the DRL model. Moreover, it is fundamental that agents are synchronized in communicating the state to the model, since it chooses what actions to perform based on the current state of the environment. With this in mind, we can state that our system requires consensus and data consistency to perform as desired. Here we explain what are the differences among these two properties and how we decided to achieve them.

Consensus is the procedure to reach a common agreement in a distributed or decentralized multi-agent environment. **Data consistency** refers to whether the same data maintained in different places still matches, besides possible modifications that must be propagated to all the locations.

The study we conducted on DHT in Section 4 was preparatory for this. A DHT-based approach can achieve a weak decentralized consensus and weak consistency if node joining/leaving the network are well handled, but still there may be some instants where the ring is not stable. Since we want to obtain a consistent network-wide view, we choose to deploy the Raft algorithm. It is a leader-based algorithm that allows achieving consensus-based replication even in the presence of failures, guaranteeing consistency with the real-time ordering of operations.

We made some modifications to the Raft algorithm to adapt it to our scenario. Periodically, metrics are collected by controllers, which add information about the state of the supporting switch, i.e. whether it is active or not, and then send its information to the Raft leader. The leader, which is one of the controllers, aggregates the local states received and its own, builds up the global network state and replicates it. When nodes receive the updated state from the leader, they register the event in their log files and change their state machine accordingly.

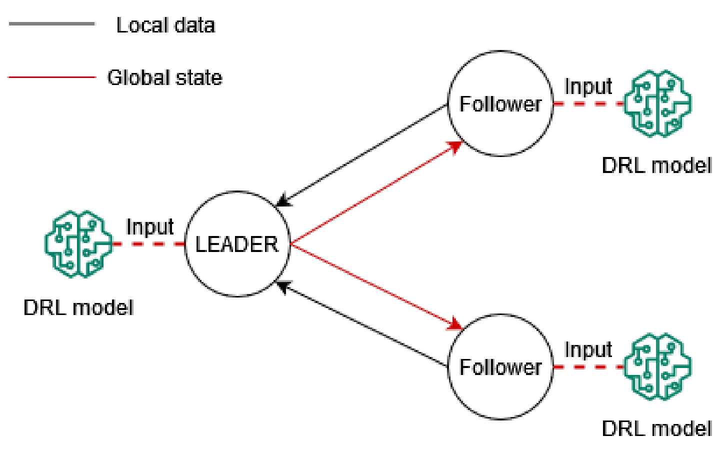


Figure 5.2: Use of Raft in the collaborative setting. Controllers send their local information to the Raft leader, which aggregate all the information received and builds the global state. This is replicated among all nodes and sent to DRL agents as input of the model.

5.4 Individually distributed

Opposed to the aforementioned design, we also present a fully distributed solution. The leading idea is to train agents independently of each other to make a comparison between the learned behaviors and reached performance of the two study cases.

To achieve the goal of independence, the **state space** is determined by local link utilization rather than global network utilization. Controllers collect metrics from the switches they manage, and send these exact information to their model.

Actions produced by DRL agents affect the local network and are taken for the sake of it, since controllers act as they don't have knowledge of other networks. In this scenario, agents are independent of each other, considering that they do not share network state information nor model parameters.

Despite the fact that networks are autonomous from each other, some information still has to be exchanged. In order to do routing and configure forwarding tables, controllers must be informed about other networks topology and hosts. We needed to design a protocol for the exchange of this knowledge. Controllers exchange JSON messages composed as follows:

- **Sender**: IP address of the controller in the cluster.
- **Graph**: graph representing the local network topology suitable for JSON serialization.
- **Edges**: list of edges to interconnect subnetworks.

Topology information is exchanged for the first time when controllers are instantiated to build a virtual network wide view. Later on, whenever changes in subnetworks occur, updated data is re-sent to correctly re-route flows. Changes relate to the removal of nodes from the graph, which represent the actions of sleep/wake up of supporting switches, for the sake of routing service only.

Chapter 6

Results

This section provides the results obtained after some experiments conducted on the proposed implementations. As far as concerns the application of a Distributed Hash Table in a heavy hitter detection use case, we will test the accuracy and response time of the implementation. Regarding our autonomous network planning framework, we will analyze the training phase, and later, we will test the learned behavior in an emulated environment to demonstrate that it is proficient in reaching the goal it was designed for. We will compare the two different designs we introduced in Chapter 5 in terms of performance and the two data plane technologies we employed.

All the experiments were conducted in a virtualized environment through Mininet[1]. Mininet is a network emulator widely used to test software-defined networks.It offers a simple and inexpensive testbed for network applications, it allows defining custom topologies without configuring a physical network and provides extensible Python API. The most important advantage surely is the portability of the code, because it runs real code that requires small changes to be run in real systems. Mininet employs virtualization to run hosts and switches in a single OS kernel, in particular, it leverages Linux namespaces, that provide separated environments for different set of resources, and virtual ethernet interfaces. However, the fact the Mininet depends on the Linux kernel is a limit and, anyway, it still remains an emulator that may produce misleading results.

[1]http://mininet.org/

6.1 DHT

We here evaluate the performance of our heavy hitter detector implemented with a distributed counting bloom filter. Starting from a real traffic capture, we reproduced many heavy hitter flows during the simulation time. The metrics we tested are *accuracy*, i.e. how many heavy hitters the program can identify, and *response time*, i.e. the time spent to block the traffic, measured as the difference between the insert in the forwarding table and the instant in which the 1000^{th} packet arrived.

6.1.1 Testbed

The reproduced topology is composed of 10 P4-enabled switches and 60 hosts. Every switch communicates with a proper controller to send TCP 5-tuple of the packet through CPU cloning. Controllers update the counting bloom filter cells belonging to that flow and if a threshold is exceeded, switches are reconfigured to drop future packets, adding a new entry into the *Heavy_hitter* table.

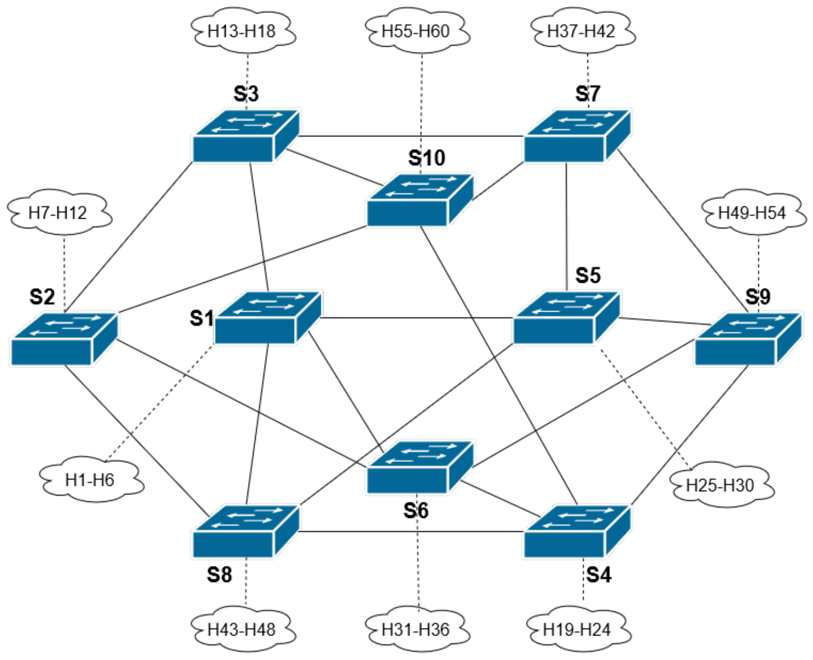


Figure 6.1: Network topology

6.1.2 Simulated traffic

We replicated real traffic from NETRESEC[2] collection of PCAP files. Particularly, we picked a PCAP file from "Hands-on Network Forensics" and selected 15 minutes of traffic to analyze.

I extracted all unique IP addresses using *tshark -r filename -T fields -e ip.src -e ip.dst* command and created a topology big enough to contain that number of hosts, but my hosts IPs are of the form 10.x.y.2 where x is the switch they're attached to and y is the ID of the host. For example, 10.1.6.2 is the IP address of host H6, which is attached to switch S1. The objective was to create a new PCAP file where host IPs belonged to my network in order to replicate the traffic. To edit the capture file to make it suitable for our network, we employed Scapy[3], a tool used both for packet manipulation and traffic replay. Scapy's *rdpcap* function reads a PCAP file and let us do per-packet operation, so for every packet of the capture file I substituted source and destination IP addresses (every "old" IP was always substituted with the same "new" one) with new ones belonging to my network and wrote every new packet into a new PCAP file.

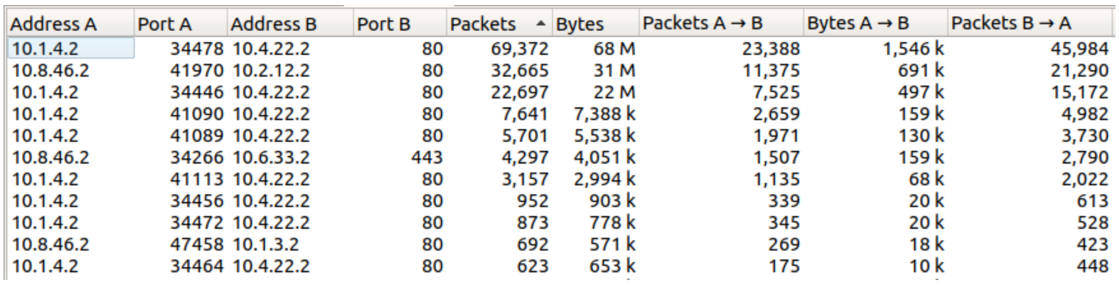
The final version of the PCAP file was analyzed with Wireshark[4], the most employed packet analyzer/sniffer. Considering a threshold of 1000 packets, 7 bidirectional flows are heavy hitters in the selected packet capture file, as shown in Fig. 6.2 (*Packets* field is the total number of packets exchanged by the two hosts).

The traffic was replayed in the network through a script, started by some hosts, that reads a pcap file and sends only the traffic with IP source address equal to its own. Replicated traffic will respect the timing of the capture in order to have a more realistic scenario and leverages the Scapy's library.

[2]https://www.netresec.com/?page=PcapFiles

[3]https://scapy.net/

[4]https://www.wireshark.org/



Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A
10.1.4.2	34478	10.4.22.2	80	69,372	68 M	23,388	1,546 k	45,984
10.8.46.2	41970	10.2.12.2	80	32,665	31 M	11,375	691 k	21,290
10.1.4.2	34446	10.4.22.2	80	22,697	22 M	7,525	497 k	15,172
10.1.4.2	41090	10.4.22.2	80	7,641	7,388 k	2,659	159 k	4,982
10.1.4.2	41089	10.4.22.2	80	5,701	5,538 k	1,971	130 k	3,730
10.8.46.2	34266	10.6.33.2	443	4,297	4,051 k	1,507	159 k	2,790
10.1.4.2	41113	10.4.22.2	80	3,157	2,994 k	1,135	68 k	2,022
10.1.4.2	34456	10.4.22.2	80	952	903 k	339	20 k	613
10.1.4.2	34472	10.4.22.2	80	873	778 k	345	20 k	528
10.8.46.2	47458	10.1.3.2	80	692	571 k	269	18 k	423
10.1.4.2	34464	10.4.22.2	80	623	653 k	175	10 k	448

Figure 6.2: Wireshark "conversations" extrapolated from a properly modified version of the original PCAP file from NETRESEC. There are 7 different bidirectional TCP flows that send more than 1000 packets.

6.1.3 Test 1: one flow

We first want to test our program against one heavy hitter flow at a time, selecting a minor portion of the capture file we chose. We want to see if the heavy hitter is stopped and after how much time.

A - H46 to H12

In this first experiment, H46 sends 11375 packets to H12. The flow is stopped after 0.22 s since the 1000^{th} packet is sent. The host H46 is directly attached to S8 and the keys generated by CRC16 and CRC32 for the flow directed to two cells owned by S8, which means that no other controllers were conducted to update the DHT.

No. Heavy Hitters	1
No. packets since stop	1139
Accuracy	100%
Response time	0.22 s

Table 6.1: Test 1 - A

B - H12 to H46

H12 sends 21290 packets to H46. The heavy hitter is directly attached to switch S2 and the generated keys are 21 and 27. According to picture 4.3, and the distance function we designed, S2 is responsible for keys 26 and 27. To update the cell indexed with key 21, S2 has to contact three other controllers until it finds the responsible, which is S6. To update the cell 27, no messages are required because it owns that cell. The overhead introduced by the DHT structure increases the response time registered in this experiment.

No. Heavy Hitters	1
No. packets since stop	1090
Accuracy	100%
Response time	0.36 s

Table 6.2: Test 1 - B

6.1.4 Test 2: more flows

In this advanced set of tests, we evaluate the behavior of our program in presence of multiple concurrent flows. Among these, multiple heavy hitters are present.

A - H22 to H4

The traffic contains two concurrent flows from H22 towards H4 that exceed the threshold. From port 80 to port 41090 are sent 4982 packets. From port 80 to port 41089 are sent 3730 packets. As far as concerns the flow towards port 41090, six controllers are contacted to update the bloom filter cells, while for the second flow 4 messages are enough. This causes the higher response time in the first case. Since the source host is the same for both flows, the same controller is in charge of handling packets coming from both flows, this means that the queue of messages between switch and controller is shared and vied.

	80:41090	**80:41089**
No. packets since stop	1097	1100
Response time	0.28 s	0.23 s
Accuracy	100%	
Average response time	0.25 s	

Table 6.3: Test 2 - A

B - Bidirectional H46-H12

We now test the program under a bidirectional flow between H46 and H12. H46 sends 11375 packets. H12 sends 22190 packets. H46 is directly attached to S8, H12 is directly attached to S2. Based on the keys generated by hash functions, the controller of S2 faces more overhead to find the responsible node for those cells and reacts slower. If we compare the responsiveness between tests A and B, we notice more latency in the previous test, this is caused by the fact that in this case the number of packets to handle to update the DHT is distributed among two different controllers, while in the previous situation there was one controller doing double the amount of work.

C - Multiple concurrent flows

In this more complex scenario, hosts H46 and H4 replay a capture file containing many small flows but also some big ones. In particular, there are 7 heavy hitters to be stopped. Table 6.5 contains the detailed results of this scenario. It is important

	H46 to H12	H12 to H46
No. packets since stop	1112	1070
Response time	0.07 s	0.12 s
Accuracy	100%	
Average response time	0.1 s	

Table 6.4: Test 2 - B

to notice that the last encountered heavy hitter flow, referred to as H4-H22:34478 in the table, is stopped after a large amount of packets over the threshold, especially compared to other flows. This is caused by the fact that controllers handle packets from switches through a FIFO queue. This creates more latency in case a single controller handles multiple flows, as in this case with many flows generated by H4, because it has to deal with previous packets, also over the threshold, before handling new ones.

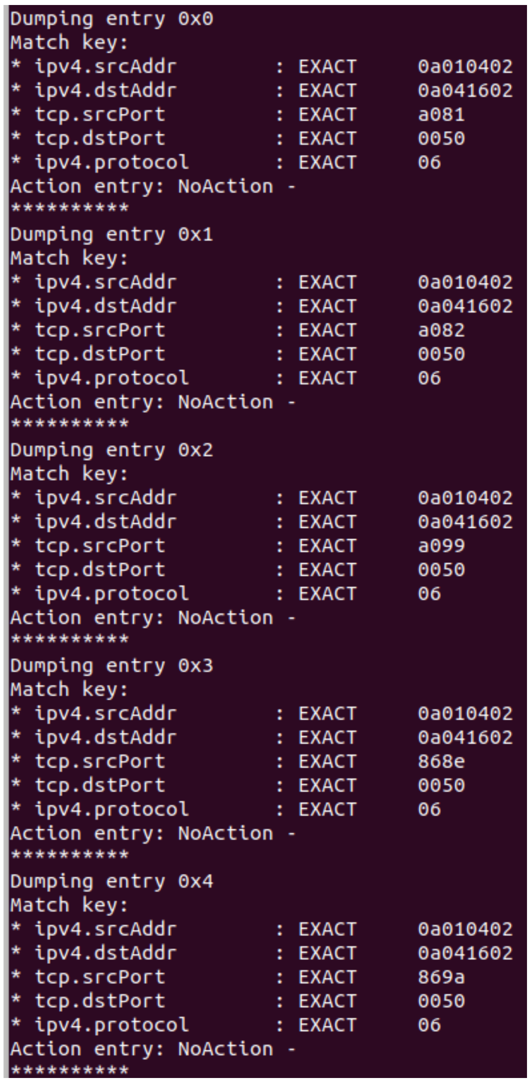
	Flow	No. packets since stop	Response time
	H46-H12	at 1119	0.14 s
	H4-H22:41089	at 1055	0.08 s
	H4-H22:41090	at 1070	0.20 s
	H4-H22:41113	at 1057	0.23 s
	H4-H22:34446	at 1037	0.36 s
	H46-H33	at 1080	0.15 s
	H4-H22:34478	at 1662	0.27 s
Accuracy	100%		
Avg. response time	0.20 s		

Table 6.5: Test 2 - C

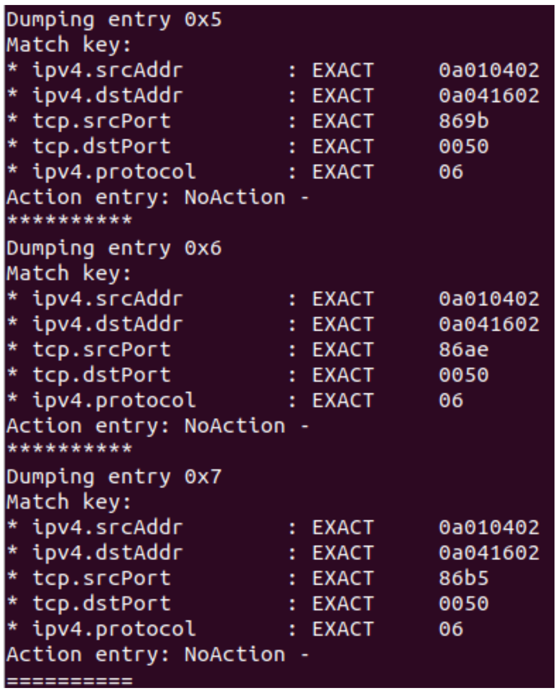
Fig. 6.3-6.4 show the entries registered in *Heavy_hitter* tables of switches S1 and S8. All heavy hitters were recognized and stopped, but three more entries are present in the switch S1 than expected. In fact, entries $0x4$,$0x5$ and $0x7$ are referred to three flows from H4 to H22 that were blocked that aren't actual heavy hitters. This is the effect of hash collisions.

6.1.5 Conclusion

This heavy hitter detector is 100% accurate, it can recognize and stop all flows exceeding 1000 packets threshold. Average time spent to make a decision is about 0.2 seconds, and it strongly depends on how many controllers are contacted in order to update the bloom filter. We can notice that some flows with higher response



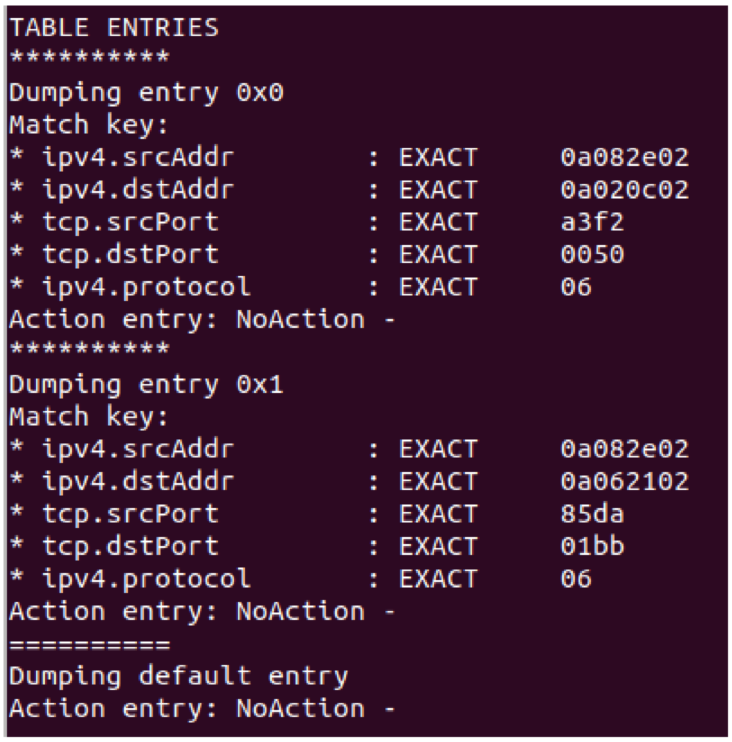
```
Dumping entry 0x0
Match key:
* ipv4.srcAddr        : EXACT     0a010402
* ipv4.dstAddr        : EXACT     0a041602
* tcp.srcPort         : EXACT     a081
* tcp.dstPort         : EXACT     0050
* ipv4.protocol       : EXACT     06
Action entry: NoAction -
**********
Dumping entry 0x1
Match key:
* ipv4.srcAddr        : EXACT     0a010402
* ipv4.dstAddr        : EXACT     0a041602
* tcp.srcPort         : EXACT     a082
* tcp.dstPort         : EXACT     0050
* ipv4.protocol       : EXACT     06
Action entry: NoAction -
**********
Dumping entry 0x2
Match key:
* ipv4.srcAddr        : EXACT     0a010402
* ipv4.dstAddr        : EXACT     0a041602
* tcp.srcPort         : EXACT     a099
* tcp.dstPort         : EXACT     0050
* ipv4.protocol       : EXACT     06
Action entry: NoAction -
**********
Dumping entry 0x3
Match key:
* ipv4.srcAddr        : EXACT     0a010402
* ipv4.dstAddr        : EXACT     0a041602
* tcp.srcPort         : EXACT     868e
* tcp.dstPort         : EXACT     0050
* ipv4.protocol       : EXACT     06
Action entry: NoAction -
**********
Dumping entry 0x4
Match key:
* ipv4.srcAddr        : EXACT     0a010402
* ipv4.dstAddr        : EXACT     0a041602
* tcp.srcPort         : EXACT     869a
* tcp.dstPort         : EXACT     0050
* ipv4.protocol       : EXACT     06
Action entry: NoAction -
**********
```



```
Dumping entry 0x5
Match key:
* ipv4.srcAddr        : EXACT     0a010402
* ipv4.dstAddr        : EXACT     0a041602
* tcp.srcPort         : EXACT     869b
* tcp.dstPort         : EXACT     0050
* ipv4.protocol       : EXACT     06
Action entry: NoAction -
**********
Dumping entry 0x6
Match key:
* ipv4.srcAddr        : EXACT     0a010402
* ipv4.dstAddr        : EXACT     0a041602
* tcp.srcPort         : EXACT     86ae
* tcp.dstPort         : EXACT     0050
* ipv4.protocol       : EXACT     06
Action entry: NoAction -
**********
Dumping entry 0x7
Match key:
* ipv4.srcAddr        : EXACT     0a010402
* ipv4.dstAddr        : EXACT     0a041602
* tcp.srcPort         : EXACT     86b5
* tcp.dstPort         : EXACT     0050
* ipv4.protocol       : EXACT     06
Action entry: NoAction -
==========
```

Figure 6.3: Table *Heavy-hitter* of switch S1

time were blocked before others, that is because more consistent flows (faster and bigger) fill the queue faster. The employment of a DHT is enough for small pieces of data but does not provide strong consistency, since data is not replicated.



```
TABLE ENTRIES
**********
Dumping entry 0x0
Match key:
* ipv4.srcAddr        : EXACT     0a082e02
* ipv4.dstAddr        : EXACT     0a020c02
* tcp.srcPort         : EXACT     a3f2
* tcp.dstPort         : EXACT     0050
* ipv4.protocol       : EXACT     06
Action entry: NoAction -
**********
Dumping entry 0x1
Match key:
* ipv4.srcAddr        : EXACT     0a082e02
* ipv4.dstAddr        : EXACT     0a062102
* tcp.srcPort         : EXACT     85da
* tcp.dstPort         : EXACT     01bb
* ipv4.protocol       : EXACT     06
Action entry: NoAction -
==========
Dumping default entry
Action entry: NoAction -
```

Figure 6.4: Table *Heavy-hitter* of switch S8

6.2 Two auto-scaling designs

We will now analyze and compare the two auto-scaling network designs proposed. We test performance when switches are programmed with the two most used languages: first P4-enabled and then OpenFlow (OF)-enabled switches. The network topology used contains 13 nodes, where 3 of them are supporting switches that can be activated/deactivated, they are remarked with red links in Fig. 6.5. We also employ 3 SDN controllers to manage different subnetworks, in particular each subnetwork contains one of the supporting switches. Half of the clients send iPerf3[5] traffic with custom scripts that, for each training episode, alternate highly rated traffic and weakly rated traffic. We also used iPerf3 to measure the throughput of flows at receiver's side during all the experiments.

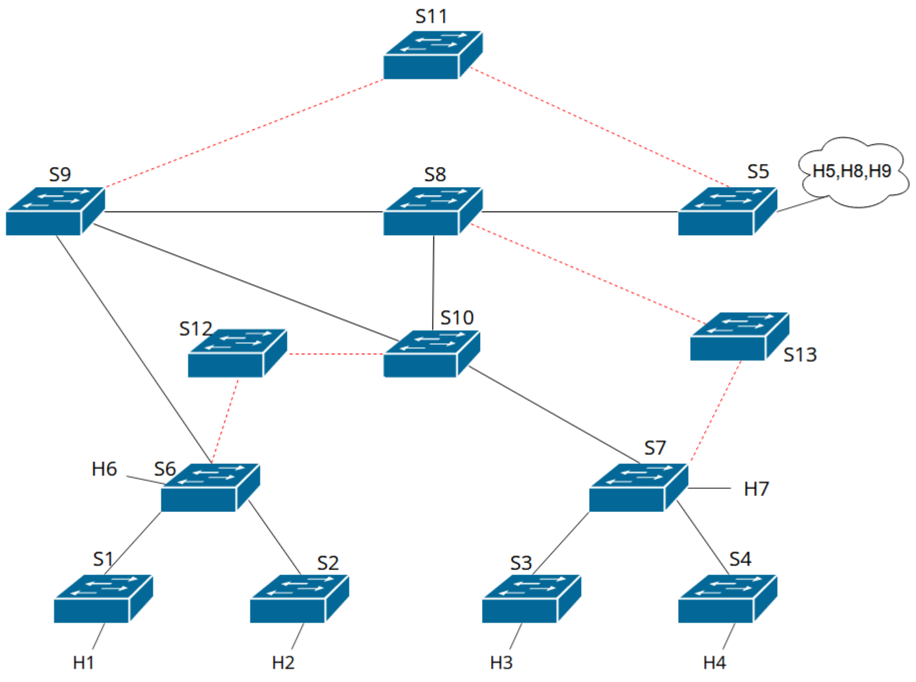


Figure 6.5: Auto-scaling framework topology. S11, S12 and S13 are supporting switches, they can be activated/deactivated by controllers. The goal is to power them on only when necessary to reduce congestion. The network is divided in three subnetworks, each managed by a different controller and each containing one support switch.

[5]https://iperf.fr/

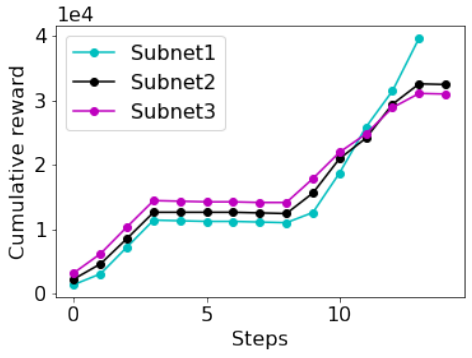
6.2.1 Convergence analysis

Firstly, we inspect the training process, analyzing the trend of cumulative reward and total reward to demonstrate that our model converged to an optimal policy. Then, we evaluate the reward over a testing episode.

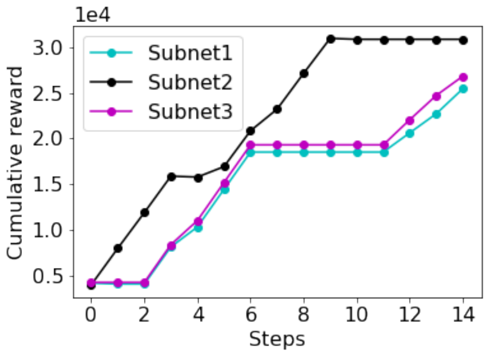
The cumulative reward is the sum of all the returns received so far, it indicates whether the model converged or not. Fig. 6.6 and Fig. 6.7 plot the cumulative reward obtained in the first and last episodes of training, respectively for the collaborative and individual solution. The different controller instances, that manage different subnetworks, are referenced as "Subnet1","Subnet2" and "Subnet3". Generally, a higher value of cumulative reward indicates that a model dominates another with a smaller value, but this is correct when the conditions over episodes are the same. Unfortunately, this is not the case for the first and last episodes of our training, because the traffic can be slightly different, and we know that the reward we designed highly depends on the throughput of flows. What is important to notice is that, in the first episodes, where a random policy governs, the cumulative reward decreases for some steps, which means that supporting resources are powered on, but no advantage is taken from that. Whereas, during last episodes this doesn't happen, which suggests that the policy learned by the model recognizes when additional resources can be helpful or not.

Another indicator of the convergence of the model is the total reward trend during the training phase, which is showed in Fig. 6.8. Typically, the total reward increases over episodes until the maximum is reached. Since our models perform experience mini-batch replay after each episode, which takes a variable amount of time, episodes face different traffic conditions, which determines different total rewards. For this reason it may not appear as obvious on our graphs, but the total reward across episodes is still increasing. The upward trend of the reward over episodes demonstrates that the model learned how to augment returns, and that it performs better than a random policy adopted at the beginning as a consequence of the Epsilon-greedy algorithm. It is also possible to see that each implementation converged in a different number of episodes, but most important we can notice that the collaborative framework took fewer episodes to learn.

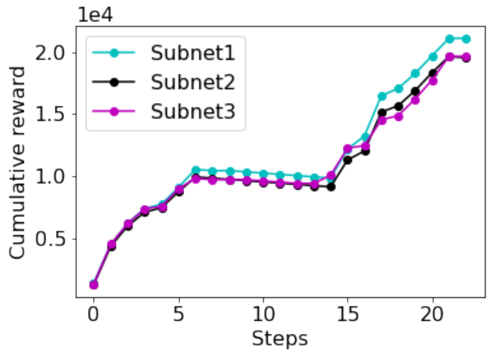
If we have a look at Fig. 6.9, plotting the reward over time in a single episode during the testing phase, it shows that the collaborative approach, supported by the efficient P4 technology, stabilizes returns around 4000 for all agents when the traffic is intense. OF based implementations show in both cases an inferior return, which is also unsteady in the collaborative solution, fig.6.9b.



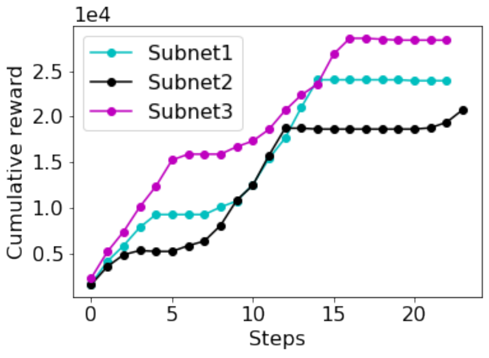
(a) Episode 0 (P4)



(b) Episode 7 (P4)

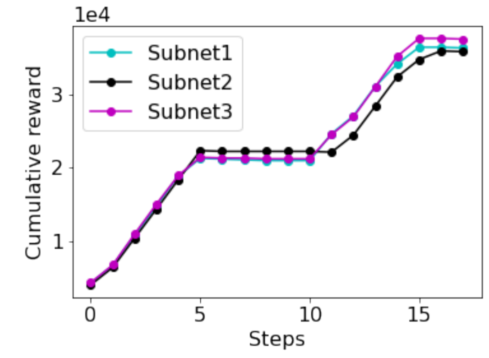


(c) Episode 0 (OF)

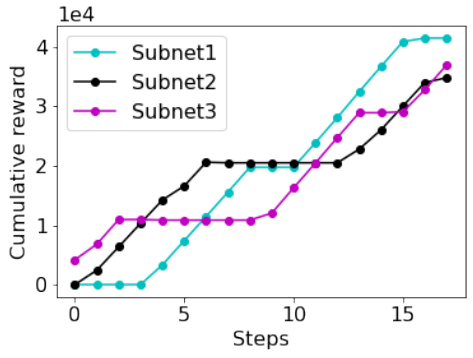


(d) Episode 14 (OF)

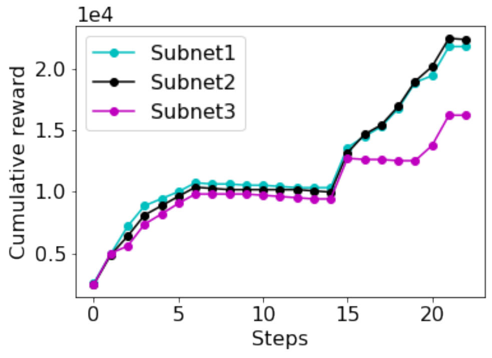
Figure 6.6: Cumulative reward over first and last episodes of training in the collaborative distributed framework.



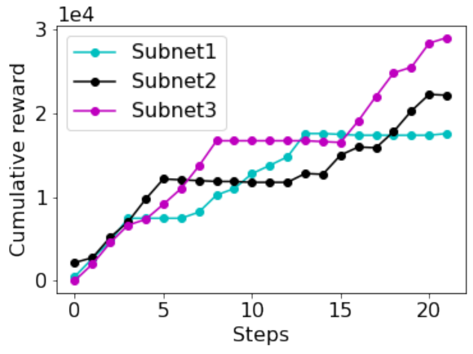
(a) Episode 0 (P4)



(b) Episode 19 (P4)

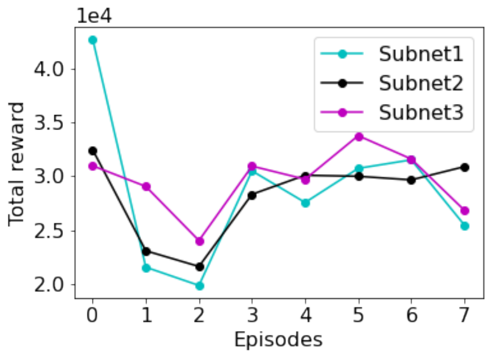


(c) Episode 0 (OF)

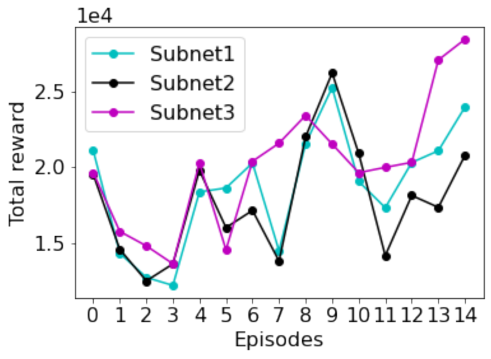


(d) Episode 19 (OF)

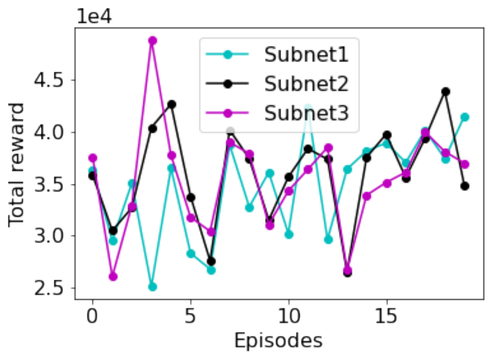
Figure 6.7: Cumulative reward over first and last episodes of training in the individually distributed framework.



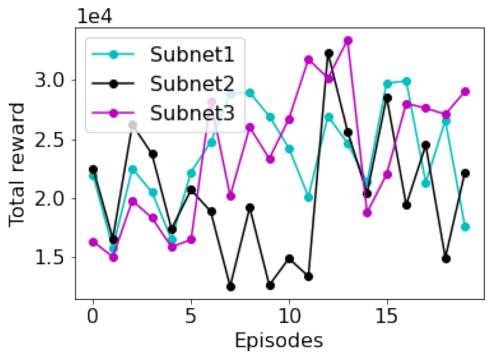
(a) Collaborative distributed (P4)



(b) Collaborative distributed (OF)

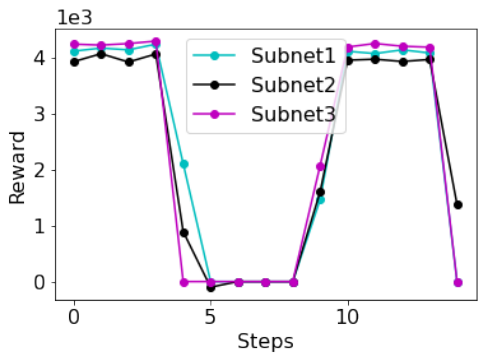


(c) Individually distributed (P4)

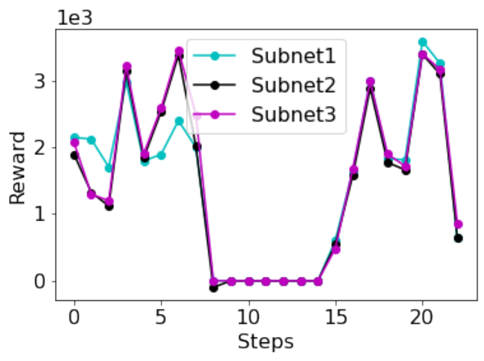


(d) Individually distributed (OF)

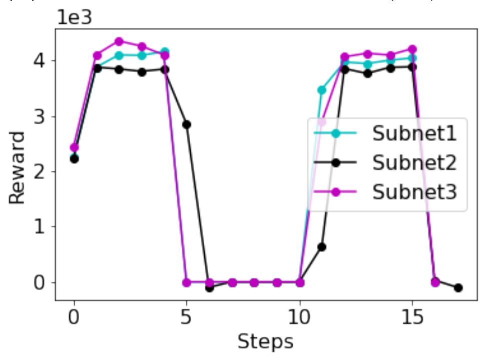
Figure 6.8: Total reward over episodes during training.



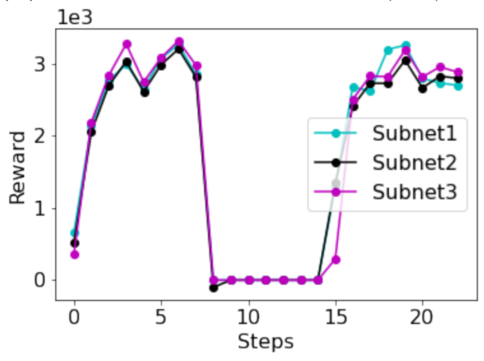
(a) Collaborative distributed (P4)



(b) Collaborative distributed (OF)



(c) Individually distributed (P4)



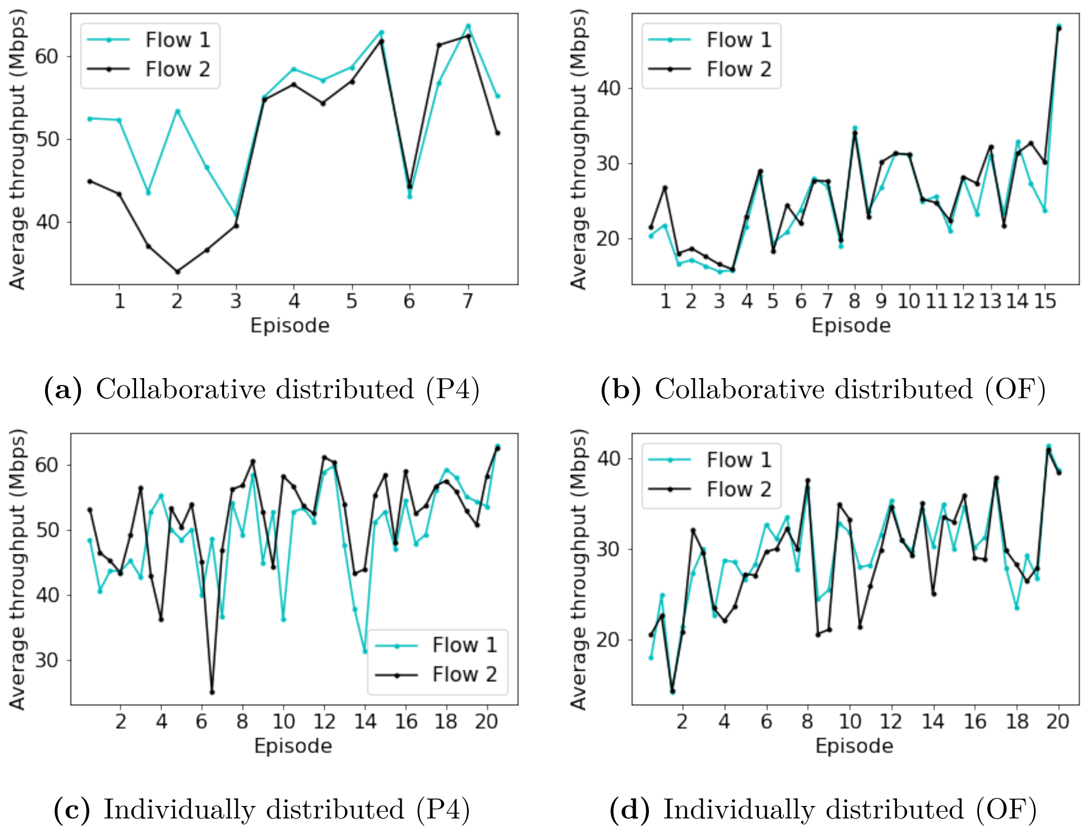
(d) Individually distributed (OF)

Figure 6.9: Reward over time during testing.

6.2.2 Network throughput

Now, we evaluate the network performance obtained with our framework in terms of throughput.

Fig. 6.10 presents the average throughput for two concurrent flows during the training phase of the agents in the two management designs and for both P4 and OpenFlow protocols. For each setting, we stop the monitoring when the reward converges. As training proceeds, the exploration probability decreases in favor of exploitation, and as our model learns the best policy, throughput is encouraged. This increasing trend highlights the fact that our program learns to maximize the designed reward function, and it reaches better network performance than a random policy. We can observe how, in the case of collaboration (Fig. 6.10a), the throughput is more stable. Moreover, in the case of P4 the throughput is notably higher than OpenFlow.

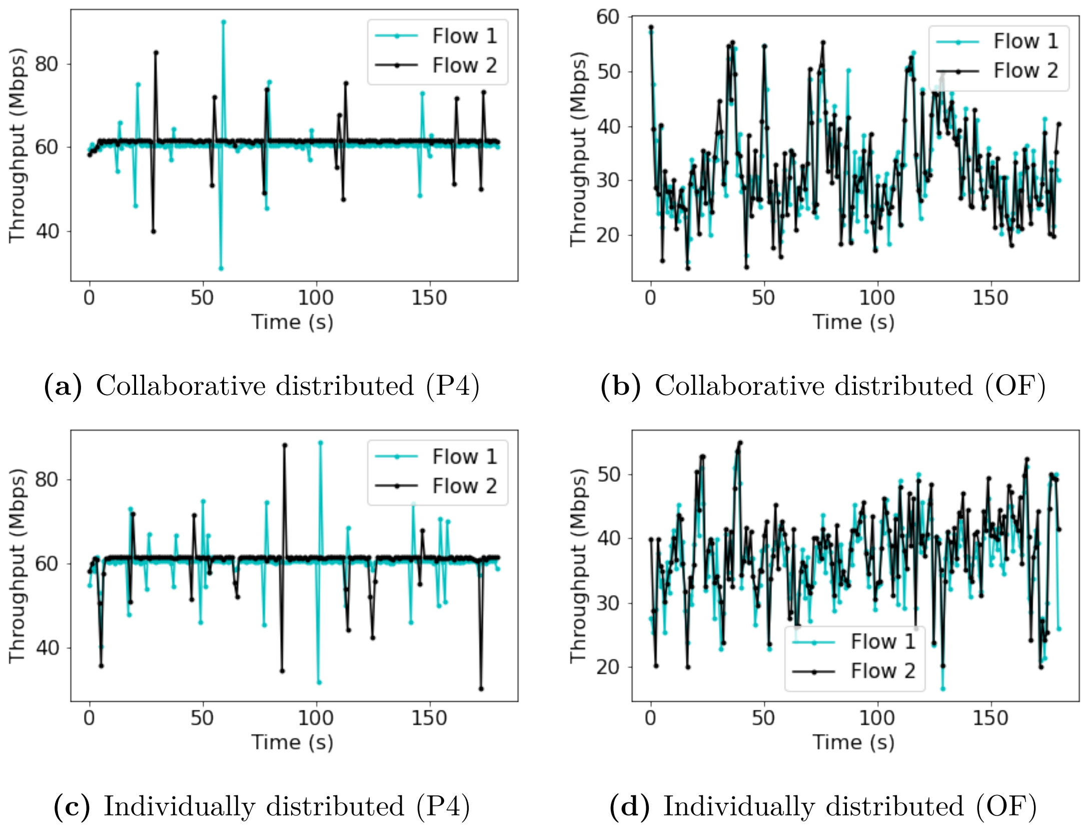


(a) Collaborative distributed (P4) **(b)** Collaborative distributed (OF)

(c) Individually distributed (P4) **(d)** Individually distributed (OF)

Figure 6.10: Throughput of flows during training. While differences between collaborative or individual is negligible, the throughput differ for P4 or OpenFlow (OF).

Fig. 6.11 provides a closer look at the throughput achieved with P4 and OpenFlow protocols during the test phase of the RL model. When using P4, the maximum throughput (as our network setup permits) is reached. All OpenFlow-based implementations face a lower and unstable throughput. This poor performance may be due to the higher number of rules configured in OpenFlow switches, which affect configuration and look-up time. If P4-based switches let us program one rule per destination address, this could not be possible with OpenFlow.This means that forwarding tables contain rules for every possible source-destination pair. Moreover, we notice a steady throughput in the P4-enabled collaborative distributed implementation in Fig. 6.11a.

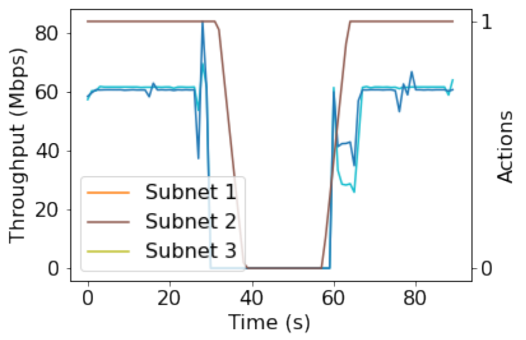


(a) Collaborative distributed (P4) **(b)** Collaborative distributed (OF)

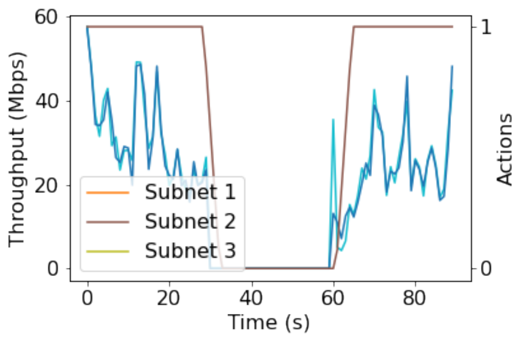
(c) Individually distributed (P4) **(d)** Individually distributed (OF)

Figure 6.11: Throughput of flows during testing. OpenFlow metrics are more jumpy.

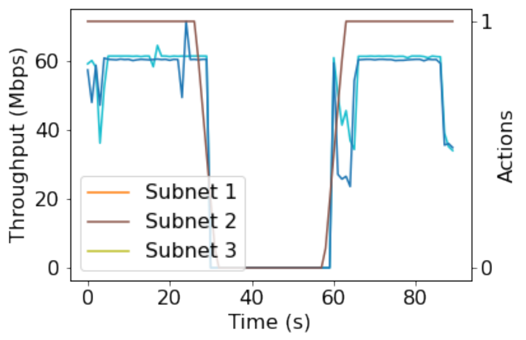
Since our model should also detect when resources are needless, we simulated congestion in the subnetwork 2, followed by less intensive traffic. We show in Fig. 6.12 the throughput after the model has been trained. It can be observed that the corresponding controller powers it off when the traffic decreases (around 30 seconds), and it also autonomously decide to power on the supporting switch to sustain the increment in traffic (around 60 seconds). As far as concerns the collaborative solution, even though simulated flows traverse the whole network and all controllers are aware of the congestion, our model can identify the best spot where additional resources overcome congestion. It is clear from the figure that other agents keep supporting switches powered off. Whereas, in the individually distributed design, controllers don't know about congestion in other subnetworks and learned to act independently for the sake of their local network.



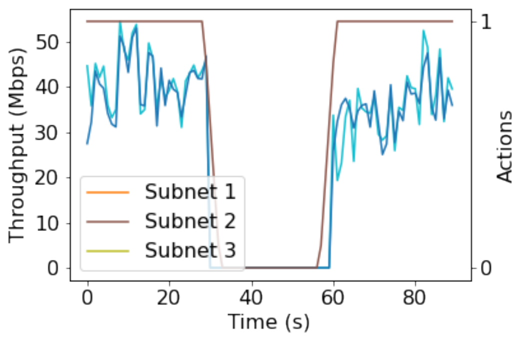
(a) Collaborative distributed (P4)



(b) Collaborative distributed (OF)



(c) Individually distributed (P4)

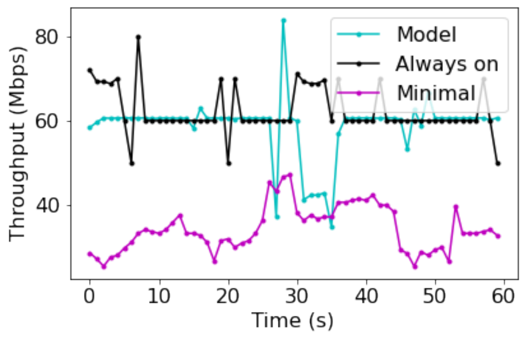


(d) Individually distributed (OF)

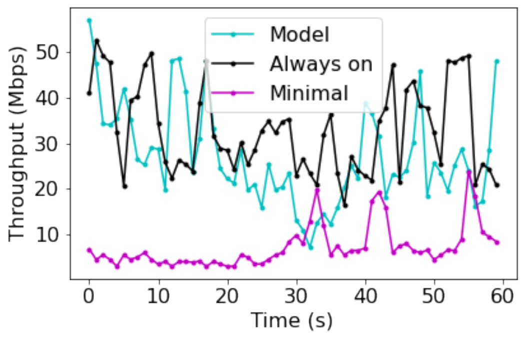
Figure 6.12: Evolution of throughput (left side) and actions (right side) in the three network regions.

From the graphs it appears how, despite the conceptual difference between the two implementations, the experienced behavior produces a similar result, with a slightly higher throughput for the collaborative framework. This result is particularly important since it can suggest that an individual distributed setting, which requires less exchange of packets among controllers, can still obtain acceptable performance. Regarding the switch languages then, with the P4 implementation, the achieved throughput is the maximum allowed by our network setup. As OpenFlow networks reach lower throughput in respect of P4, we can also observe a more stable and higher throughput in our collaborative distributed solution.

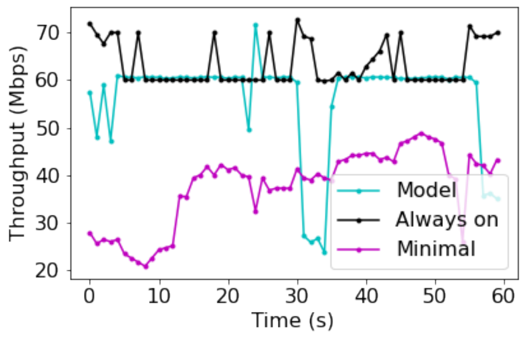
Lastly, to understand the convenience of an auto-scaling system, we compare our model performance against a network with supporting switches always powered off (minimal setting) and a network with always active resources (always on) that implement equal-cost multi-path routing (ECMP). Fig. 6.13 demonstrates that our independent version of logic distribution can achieve pretty similar performance to an always on case where switches are always active. However, in addition to this, our reactive logic can also reduce energy consumption powering off resources when they are not needed, as demonstrated previously.



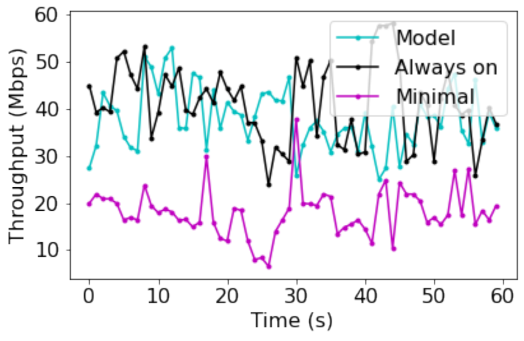
(a) Collaborative distributed (P4)



(b) Collaborative distributed (OF)



(c) Individually distributed (P4)



(d) Individually distributed (OF)

Figure 6.13: Throughput evolution for a minimal setting, always-on configuration, and our model.

6.2.3 Conclusion

We can conclude that our autonomous auto-scaling network dynamically manages network resources, without compromising network performance when supported by efficient data plane technologies. The multi-agent approach adopted make our framework scalable over large-scale networks, and two possible designs can be discussed. A collaboration among controllers can be established, but this solution brings new challenges to efficiently share information. We also experienced that the trained models can be interchanged among the multiple implementations.

Chapter 7

Conclusion

In this thesis, we addressed the need of deploying Reinforcement Learning with a multi-agent approach to help creating self-driving networks, in response to new strict requirements posed by data-intensive applications.

We first investigated how to possibly exchange information among multiple controller instances. We explored the use of a Distributed Hash Table through a heavy hitter detection use case where we distributed a counting bloom filter that stored the number of packets per flow. We concluded that a DHT can achieve weak decentralized consensus but doesn't provide strong data consistency. For this reason we decided to adopt Raft, a consensus algorithm based on the election of a leader that replicates data, making it highly-available and strong consistent.

We proposed an autonomous network planning framework that can dynamically scale up and down based on the traffic demand to guarantee high throughput and the reduction of unnecessary energy consumption. With the intent to achieve scalability, we presented two possible alternatives to manage multiple subnetworks: a collaborative approach and an individual one.

The collaborative solution leverages Raft protocol to build and distribute the global network state among all the controllers, which is later used by the RL model to make decisions based on the combined behavior.

The individual solution deploys multiple controllers and multiple RL agents, which operate independently of others. The only type of information exchanged are topological ones, used to offer routing services.

Experimental results showed that both solutions are capable of deploying appropriate reactive changes based on the traffic load and that an individually distributed approach, which adopts a simpler mechanism for data exchange among controllers and share less information, can still achieve good performance. In addition to diverse control-plane settings, we evaluated diverse data-plane languages experiencing how tests with P4-enabled switches are more stable and performant.

Future work will further investigate the behavior obtained with the two discussed solutions with more realistic traffic conditions to analyze deeper the effects of collaboration and individualism in this scenario. We also plan to study how to describe the network state in a more intelligible way for neural networks, in order to represent efficiently all the network features in a smaller state space.

List of Tables

List of Figures

Acronyms

SDN Software-Defined Networking

ML Machine Learning

MARL Multi-Agent Reinfrocement Learning

AI Artificial Intelligence

RL Reinforcement Learning

DHT Distributed Hash Table

NIB Network Information Base

ODL OpenDayLight

DRL Deep Reinforcement Learning

FCT Flow Completion Time

UE User Equipment

AP Access Point

VM Virtual Machine

OF OpenFlow

P4 Programming Protocol-independent Packet Processor

P2P Peer-to-Peer

DQN Deep Q Network

ECMP Equal-Cost Multi-Path Routing

Bibliography

[1] Adnan Aijaz, Mischa Dohler, A Hamid Aghvami, Vasilis Friderikos, and Magnus Frodigh. «Realizing the tactile Internet: Haptic communications over next generation 5G cellular networks». In: *IEEE Wireless Communications* 24.2 (2016), pp. 82–89 (cit. on p. 1).

[2] Alessio Sacco, Flavio Esposito, Guido Marchetto, Grant Kolar, and Kate Schwetye. «On edge computing for remote pathology consultations and computations». In: *IEEE Journal of Biomedical and Health Informatics* 24.9 (2020), pp. 2523–2534 (cit. on p. 1).

[3] Deze Zeng, Lin Gu, Shengli Pan, Jingjing Cai, and Song Guo. «Resource management at the network edge: A deep reinforcement learning approach». In: *IEEE Network* 33.3 (2019), pp. 26–33 (cit. on p. 1).

[4] Alessio Sacco, Flavio Esposito, and Guido Marchetto. «Supporting Sustainable Virtual Network Mutations with Mystique». In: *IEEE Transactions on Network and Service Management* 18.3 (2021), pp. 2714–2727 (cit. on p. 1).

[5] Doyoung Lee, Jae-Hyoung Yoo, and James Won-Ki Hong. «Deep Q-Networks Based Auto-Scaling for Service Function Chaining». In: *International Conference on Network and Service Management (CNSM)*. IEEE. 2020, pp. 1–9 (cit. on p. 1).

[6] Hang Zhu, Varun Gupta, Satyajeet Singh Ahuja, Yuandong Tian, Ying Zhang, and Xin Jin. «Network planning with deep reinforcement learning». In: *Proceedings of the 2021 ACM SIGCOMM Conference*. ACM. 2021, pp. 258–271 (cit. on p. 1).

[7] Teemu Koponen et al. «Onix: A distributed control platform for large-scale production networks». In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. 2010 (cit. on p. 6).

[8] Pankaj Berde et al. «ONOS: towards an open, distributed SDN OS». In: *Proceedings of the third workshop on Hot topics in software defined networking*. 2014, pp. 1–6 (cit. on p. 6).

[9] Amin Tootoonchian and Yashar Ganjali. «Hyperflow: A distributed control plane for openflow». In: *Proceedings of the 2010 internet network management conference on Research on enterprise networking*. Vol. 3. 2010, pp. 10–5555 (cit. on p. 6).

[10] Diego Ongaro and John Ousterhout. «In search of an understandable consensus algorithm». In: *USENIX Annual Technical Conference (ATC 14)*. 2014, pp. 305–319 (cit. on p. 6).

[11] *ONOS developoer guide*. `https://wiki.onosproject.org/display/ONOS/Distributed+Primitives`. Last edit: 2020 (cit. on p. 6).

[12] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. «OpenDaylight: Towards a Model-Driven SDN Controller architecture». In: *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*. 2014, pp. 1–6. DOI: `10.1109/WoWMoM.2014.6918985` (cit. on p. 7).

[13] Advait Dixit, Fang Hao, Sarit Mukherjee, T. V. Lakshman, and Ramana Rao Kompella. «ElastiCon; an elastic distributed SDN controller». In: *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2014, pp. 17–27 (cit. on p. 7).

[14] Kevin Phemius, Mathieu Bouet, and Jérémie Leguay. «Disco: Distributed multi-domain sdn controllers». In: *2014 IEEE network operations and management symposium (NOMS)*. IEEE. 2014, pp. 1–4 (cit. on p. 7).

[15] Soheil Hassas Yeganeh and Yashar Ganjali. «Kandoo: a framework for efficient and scalable offloading of control applications». In: *Proceedings of the first workshop on Hot topics in software defined networks*. 2012, pp. 19–24 (cit. on p. 8).

[16] Yonghong Fu, Jun Bi, Kai Gao, Ze Chen, Jianping Wu, and Bin Hao. «Orion: A hybrid hierarchical control plane of software-defined networking for large-scale networks». In: *2014 IEEE 22nd International Conference on Network Protocols*. IEEE. 2014, pp. 569–576 (cit. on p. 8).

[17] Saim Salman, Christopher Streiffer, Huan Chen, Theophilus Benson, and Asim Kadav. «DeepConf: Automating Data Center Network Topologies Management with Machine Learning». In: *Proceedings of the 2018 Workshop on Network Meets AI & ML*. NetAI'18. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 8–14. ISBN: 9781450359115. DOI: `10.1145/3229543.3229554`. URL: `https://doi.org/10.1145/3229543.3229554` (cit. on p. 9).

[18] Penghao Sun, Zehua Guo, Sen Liu, Julong Lan, Junchao Wang, and Yuxiang Hu. «SmartFCT: Improving power-efficiency for data center networks with deep reinforcement learning». In: *Computer Networks* 179 (2020), p. 107255 (cit. on p. 9).

[19] Xiangyi Chen, Xingwei Wang, Bo Yi, Qiang He, and Min Huang. «Deep Learning-Based Traffic Prediction for Energy Efficiency Optimization in Software-Defined Networking». In: *IEEE Systems Journal* 15.4 (2021), pp. 5583–5594. DOI: `10.1109/JSYST.2020.3009315` (cit. on p. 9).

[20] Navid Naderializadeh, Jaroslaw J Sydir, Meryem Simsek, and Hosein Nikopour. «Resource management in wireless networks via multi-agent deep reinforcement learning». In: *IEEE Transactions on Wireless Communications* 20.6 (2021), pp. 3507–3523 (cit. on p. 10).

[21] Tianle Mai, Haipeng Yao, Zehui Xiong, Song Guo, and Dusit Tao Niyato. «Multi-agent actor-critic reinforcement learning based in-network load balance». In: *GLOBECOM 2020-2020 IEEE Global Communications Conference*. IEEE. 2020, pp. 1–6 (cit. on p. 10).

[22] Xinyu You, Xuanjie Li, Yuedong Xu, Hui Feng, Jin Zhao, and Huaicheng Yan. «Toward packet routing with fully distributed multiagent deep reinforcement learning». In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* (2020) (cit. on p. 10).

[23] Kawsar Haghshenas, Ali Pahlevan, Marina Zapater, Siamak Mohammadi, and David Atienza. «MAGNETIC: Multi-Agent Machine Learning-Based Approach for Energy Efficient Dynamic Consolidation in Data Centers». In: *IEEE Transactions on Services Computing* 15.1 (2022), pp. 30–44. DOI: `10.1109/TSC.2019.2919555` (cit. on p. 10).

[24] Fulvio Risso. *Architecture of network devices*. `https://swnet.frisso.net/` (cit. on p. 13).

[25] Fulvio Risso. *SDN intro*. `https://swnet.frisso.net/` (cit. on pp. 14, 16).

[26] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. «OpenFlow: enabling innovation in campus networks». In: *ACM SIGCOMM computer communication review* 38.2 (2008), pp. 69–74 (cit. on pp. 17, 18).

[27] Fulvio Risso. *OpenFlow*. `https://swnet.frisso.net/` (cit. on p. 17).

[28] Pat Bosshart et al. «P4: Programming protocol-independent packet processors». In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95 (cit. on pp. 20, 21).

[29] *Distirbuted Hash Table*. `https://en.wikipedia.org/wiki/Distributed_hash_table`. Last edit: 2022 (cit. on p. 22).

[30] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. «Chord: A scalable peer-to-peer lookup service for internet applications». In: *ACM SIGCOMM computer communication review* 31.4 (2001), pp. 149–160 (cit. on p. 23).

[31] *Bittorent.* `https://www.bittorrent.com/` (cit. on p. 23).

[32] Giovanni Neglia, Giuseppe Reina, Honggang Zhang, Donald F. Towsley, Arun Venkataramani, and John S. Danaher. «Availability in BitTorrent Systems». In: *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications* (2007), pp. 2216–2224 (cit. on p. 24).

[33] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018 (cit. on p. 26).

[34] Christopher JCH Watkins and Peter Dayan. «Q-learning». In: *Machine learning* 8.3 (1992), pp. 279–292 (cit. on p. 26).

[35] Volodymyr Mnih et al. «Human-level control through deep reinforcement learning». In: *nature* 518.7540 (2015), pp. 529–533 (cit. on p. 27).

[36] Cristian Estan and George Varghese. «New directions in traffic measurement and accounting». In: *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications.* 2002, pp. 323–336 (cit. on p. 28).

[37] Denis Kleyko, Abbas Rahimi, Ross W Gayler, and Evgeny Osipov. «Autoscaling bloom filter: controlling trade-off between true and false positives». In: *Neural Computing and Applications* 32.8 (2020), pp. 3675–3684 (cit. on p. 29).

[38] Will Fisher, Martin Suchara, and Jennifer Rexford. «Greening Backbone Networks: Reducing Energy Consumption by Shutting Off Cables in Bundled Links». In: Aug. 2010, pp. 29–34. DOI: `10.1145/1851290.1851297` (cit. on p. 33).

[39] Raffaele Bolla, Roberto Bruschi, Franco Davoli, and Flavio Cucchietti. «Energy efficiency in the future internet: a survey of existing approaches and trends in energy-aware fixed network infrastructures». In: *IEEE Communications Surveys & Tutorials* 13.2 (2010), pp. 223–244 (cit. on p. 33).

[40] Nick Feamster and Jennifer Rexford. «Why (and how) networks should run themselves». In: *arXiv preprint arXiv:1710.11583* (2017) (cit. on p. 33).