POLITECNICO DI TORINO

Master of Science
in Computer Engineering

Master's Degree Thesis

Supporting TCP decisions in challenged networks via in-network telemetry

Supervisors
Prof. Guido Marchetto
Dott. Alessio Sacco

Candidate
Matteo Di Mauro

Academic Year 2021-2022

Summary

Nowadays, networks are facing several challenges in developing solutions to solve intrinsic problems like achieving acceptable levels of latency, bandwidth, faults, and loss of packets that guarantee responsive applications and services together with a better user experience.
TCP is still the most adopted transport protocol in network communication for its reliability, but it performs poorly in heterogeneous or wireless networks because it has no mechanism to distinguish between congestion and stochastic packet losses. Thus, the incoherence of the congestion window (cwnd) value could lead to performance and throughput degradation.
For this reason, there is a recent attempt to let the TCP learn the best cwnd updates via Machine Learning (ML)-based approaches. A specific class of ML, namely Reinforcement Learning (RL), has been proved to bring advantages to TCP and on top applications, having the ability to learn and adapt to the network environment. However, despite the improvements, none of these solutions can exploit network intelligence fully.
This thesis aims to understand how it is possible to give support to TCP decisions by means of data collected on networks and processed by RL algorithms. Our solution is constituted by two components: a modified instance of TCP that runs on the end-hosts, and intelligent network devices to empower network telemetry. Regarding the latter component, our choice is to use P4 (Programming Protocol-independent Packet Processors) language for network programming given the vast gamma of metrics available and the fast processing time. Switches, programmed by means of P4 language, have complete control of the traffic through the network. Then, we consider two different in-band ways to carry these metrics to the end host, and we study the advantages and disadvantages in terms of complexity and performance. The first proposal is the integration of data on IP Options field, which consists of IP packet utilization without any format change or bytes addition, just Option field exploiting. The second one is the BPP (Big Packet Protocol) protocol integration which allows controlling packets flow over the network through information encoded in packets themselves.

More specifically, it consists of BPP headers addition on the standard IP packet between the IP header and the IP payload field. Here we select the metadata field to store our metrics modified by switches of the emulated network for header integration and packet management reasons.
The last step is to integrate the metrics in an existing RL-based TCP protocol, Owl, that helps select the correct value of the congestion window and can react to different network conditions.
We finally experienced how this solution can improve the performance of TCP congestion control, impacting only limitedly on the network traffic overhead.

Table of Contents

List of Figures

Listings

Chapter 1

Introduction

At the end of the 1990s, thanks to the huge evolution of network infrastructures from a physical and electronic point of view, it was common to think that there was no longer much room for improvements and innovation. Moreover, standard protocols like IPv4, UDP and TCP were widely used and, accordingly to the need of the time, there was no necessity to improve their functionalities.
Since the beginning of the early 2000s, there was no way to get direct access to the forwarding plane. The absence of such interfaces, resulted on networks composed by monolithic devices, mainframe-like. Then a different approach has been used.
The idea was to exploit programming languages capabilities in order to overcome management limitations due to the protocols' rules and implementation. For example, one of the biggest problem of IP, was that the protocol does not provide a way to manipulate traffic through the network. It is impossible to change the path of the packet based on network conditions, but it's necessary to follow the one computed.
In this perspective, the development of SDN (Software-Defined Networking), which provides a way to decouple "data plane" (forwarding process of packets) from "control plane" (routing process), has been fundamental to enabling the network control to become directly programmable and the underlying infrastructure to be abstracted from applications and network services.[1]
Regarding SDN, one more step towards this direction has been made by "OpenFlow", which con be considered the first standard communications interface acting between control plane and data plane. This technology allows to get direct access to the switches which populate networks and manage traffic according to the logic implemented on the code injected.
In this scenario, the focus of this thesis is the transport layer of the OSI model. In particular, it aims to understand how it is possible to give support to TCP protocol decisions and improve them by means of data collected on networks and processed by RL algorithms.

The main steps of the implementation proposed can be summarized as follows:

- Implementation of a topology with P4 switches capable to have complete control of the traffic through the network;
- Introduction of additional actions in order to test the behavior of the P4 switches and a correct network management;
- Testing of the network with two different implementations, introducing some metrics inside the IP Options in the first test and inside the BPP Metadata in the second one, in order to measure and compare the performances;
- Selection of the "best solution" and collection of the metrics on server side in order to process them and integrate inside the recently developed "Owl" congestion control protocol.

The final goal of this thesis, as introduced in the last point, is to make an additional step to overcome standard protocol limitations, trying to add intelligence to the network devices by means of Machine Learning algorithms. The focal point of Machine Learning is that requires a good amount of data to be processed in order to perform well. Our focus, indeed, is to feed the Reinforcement Learning model, in which Owl is based on, with the metric collection, improving its capabilities to understand different network situations and react properly.

1.1 Thesis structure

The second chapter of this dissertation deals with an exhaustive description of the background concepts needed to understand how P4 language works, its advantages and disadvantages using it and a brief overview about Big Packet Protocol and TCP congestion control challenges.
In the following chapter, a couple of concrete application examples are reported in order to demonstrate how In-band Network Telemetry (INT) concept, used in this work, is widely adopted to resolve many network issues. Then, the state of the art of the BPP protocol integrated in IP packets is analyzed.
The fourth chapter is reserved to the description of the System implemented, all the generic feature are described with the goal to understand the components involved, the technologies used the idea of interaction presented.
This chapter is the preparation to the implementation, it is necessary to underline that the system thought can be realized in several ways.
It is independent from the software version and tools used during the implementation and test phases, but, with shrewdness, it can be replicated anywhere.
During the implementation phase, all the choices are discussed and motivated. Here, the focus is on a detailed explanation of the implementation of the system described in the previous chapter.
The most important pieces of code realized are detailed, highlighting the general meaning, the reasons behind the solution proposed instead of any different one.
The two projects studied are inspected separately looking at the difference between them and the common part respect to the starting implementation.
The P4 code is deeply described trying to clarify the most crucial point of the implementation, focusing on the key point of the functionalities. The same is performed for the Python scripts used, with less level of details because they are more related to the test part. Only the core part are extracted.
After this, an accurate section regarding the code and the actions performed for the Owl integration is made. After giving the generic overview in the System chapter, in implementation the actual files and environment touched are defined.
Due to its complexity, the Machine Learning algorithm behind Owl functionality is hided. The choice is done accordingly to the purpose of this thesis which is not the inspection and the improvement of the algorithm itself but just the way it works. For this reasons, the RL model is treated like a black box and the test performed looks at the obtained output after the input received.
The last step of the implementation phase is the testing one, so here the description of how the implemented solutions are tested and some details of the script written are reported.

In addition, some setting aspects are clarified in order to understand and justify the results obtained. Then, several graphs and table are exposed in order to show the data obtained. After the discussion of results part, a comparison between the different approaches and the overall situation can be found.
In the last few steps of this work, the final evidence of the successful integration with Owl, the target congestion control protocol, is provided.
The conclusion just summarizes all the results achieved in chapter 6, with a look at the future work scenarios, giving a direction for the best solutions for the system improvement.

Chapter 2

Related Work

2.1 In-band Network Telemetry

One of the starting point of this work is the In-band Network Telemetry (INT) concept. It is a mechanism to inspect and query the internal status of network devices such as switches and routers. The main idea behind this technique, is to exploit the well know structures of packets in order to collect technical network information and parameters like load of the links, queuing latency, queue size, processing time and store them into the packet itself. In order to provide this kind of behavior, P4 is widely used because it allows to operate regardless the packet format and the protocol adopted for the communication.
Network telemetry is a term which simply refers to techniques for network data collection and consumption. Telemetry, instead, is a process for gathering and processing information from network automatically. Network telemetry could be considered an ideal mean to obtain sufficient network visibility with better accuracy, scalability, and performance with respect to traditional network measurement technologies. In-band network telemetry is an emerging evolution of network telemetry, which is widely receiving attention all over the world.
Different from the traditional network and software-defined network measurements, in-band network telemetry is a combination of data packet forwarding process and collection and analyzing process of data and metrics. The way it performes the telemetry process, is basically collecting the network status by inserting metadata into packet by switching nodes.
Originally, in-band network telemetry technology was adopted for network performance measurement and has then been extended to network data plane verification, congestion control, fault location and traffic engineering.
It is possible to consider two different macro-groups of applications of network telemetry: performance applications and functional ones.

Common network performance telemetry applications are the delay and packet loss management or for example available bandwidth and QoS estimations. For delay measurement, Kim et al. [2] implemented the HTTP request one-way delay computation based on INT of the programmable data plane.

2.1.1 HTTP Communication

An example of usage of INT is the possibility to study and debug network issues analyzing the data stored into the packets. In this scenario, it is possible to set up an HTTP communication between two different hosts monitoring the latency though the telemetry recording process.
The following topology is realized with Mininet [3] environment. Each switch executes a given P4 program and implement a controller which is capable to modify entries in the switch tables. This operation can be performed by means of HTTP requests that allows the controller to insert, delete, and modify entries in the tables. These APIs are generated automatically by the P4 compiler and provide hooks to perform run-time tasks, such as inserting and removing routes. [4]

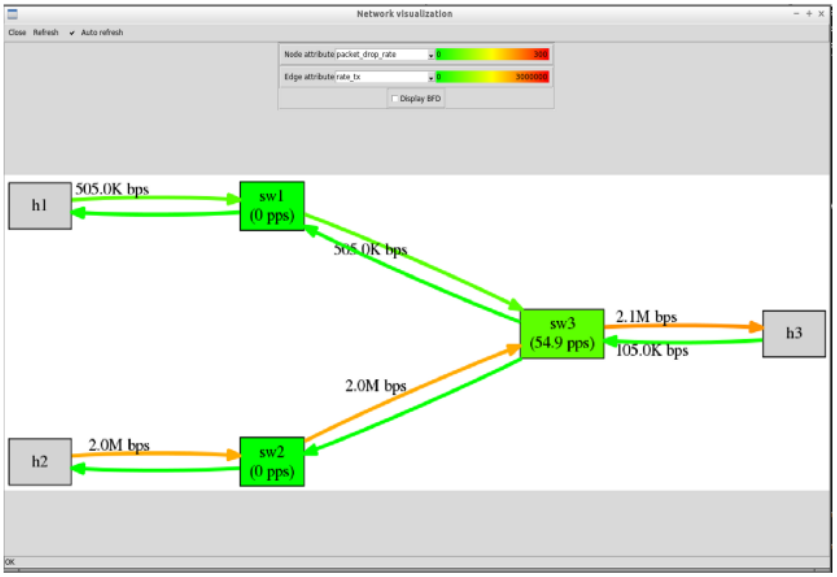

Figure 2.1: Mininet topology

In this topology, two different communication are depicted.
First, H3 sends a periodic web request to H1 and starts a timer. When the latter returns the requested page, H3 stops the timer and computes how much time is taken in the network. Meanwhile, an UDP communication is opened by H2 to H3 in order to create a congestion and slow down the first communication.

In this simple scenario, the INT switches push telemetry packets which contains information like switch ID and time spent on their queue into the TCP Options field. In this way, when the data arrive on the final host, it is possible to determine the switch with the biggest queue size value and take mitigation actions accordingly.

2.1.2 Network Security Applications

The increasing spread of cybersecurity threats together with the different approaches used to perform the attacks, are making necessary the implementation of more defense mechanisms.
Software defined network (SDN) switches with fixed-function data plane are being gradually replaced by switches with programmable data planes that allow to realize more network protocols.
P4 (Programming Protocol-independent Packet Processors) is proposed in the article: "A Review of P4 Programmable Data Planes for Network Security" as a good way to implement user's applications, such as for example data center networks and security. [5] Thanks to the capabilities of P4, packet filtering and inspection can be done. Some of the functions that can be realized by programmable switches are for example access control for traffic, encryption to protect network flow and the possibility to perform countermeasures to the attacks.
Figure below shows some researches in network security based on P4 data plane:

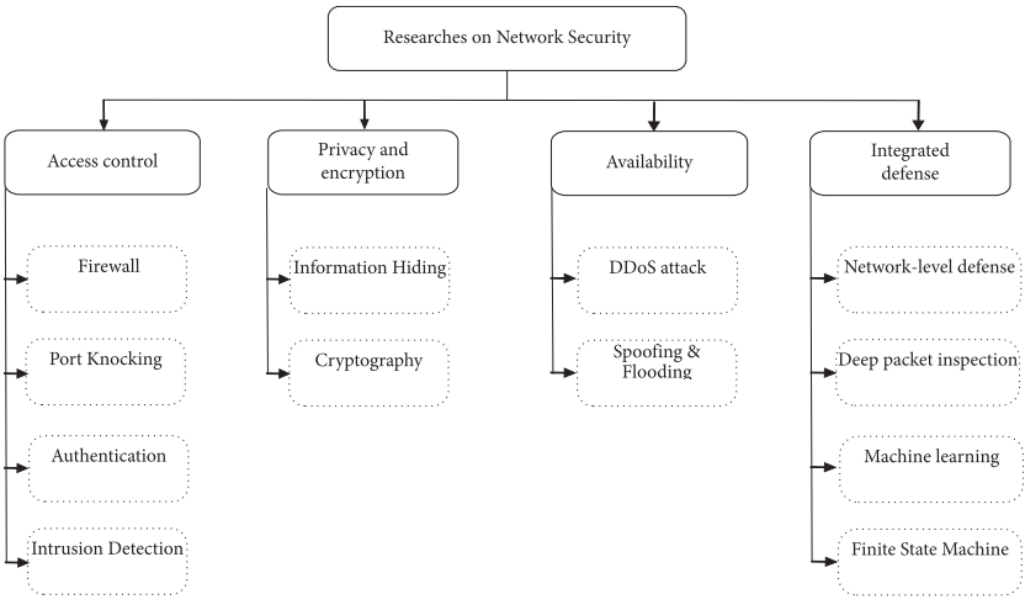

Figure 2.2: Network security research based on P4

Access Control

Programmable switches can be able to delegate authentication and authorization to the data plane
Access control is the restriction of access to resources or digital objects. Untrusted traffic can be filtered in advance in order to prevent the server from receiving and processing intrusion information.

Privacy and Encryption

Identity and information about the users inside the network can be stolen from the sent packet. Scrambling and encryption are usually adopted in order to hide information or make cracking more complex.
Most of the common solutions have a lot of limitations because some of them need to modify the Internet architecture or are based on software so that cannot be used with traffic at high-speed.
The new researches about P4 make possible to realize anonymity or encryption of the communication in a simpler way. Moghaddam et al. [6] propose a network solution, called "PANEL" (Practical Anonymity at the NEtwork Level), to solve the problems of anonymity without making changes on the forwarding protocol or routing rules. PANEL offers a solution that, rewriting the source address, or making randomization of TTL, IP identification and TCP sequence, makes possible to reach anonymity. The routing response is preserved thanks to P4-based switches that store the original information and maintain the forwarding table fwd_panel and the reverse table rev_panel. The switches check that the destination IP address of the packet is stored locally and enter the reverse table to search for an address replacement. If no match is found, it means that the packet does not belong to the expected communication, so the normal behaviour of searching for the sending port through the forwarding table, is applied. Figure below shows the P4 program of the data plane ingress processing flow.

```
control ingress {
    //Check if packet is a valid IPv4 packet
    if (valid(ipv4)) {
        //Check if TTL is larger than 0
        if (ipv4.ttl > 0) {
            //Check if dest IPv4 address belongs to us
            apply(match_panel_ip)
            {
                hit {
                //Apply reverse tables
                    if(valid(tcp)) {
                        apply(rev_panel_tcp);
                    }
                    if(valid(udp)) {
                        apply(rev_panel_udp);
                    }
                }
            }
            //If packet's IPv4 address does not belong to
            //us, apply forward tables
            if(panel_metadata.match_panel == 0) {
                //Assume all sessions are matched
                if(valid(tcp)) {
                    apply(fwd_panel_tcp);
                }
                if(valid(udp)) {
                    apply(fwd_panel_udp);
                }
            }
        ...
}
```

Figure 2.3: P4 control ingress for PANEL

Integrated Defense

Data plane computing can increase throughput by at least one order of magnitude with respect to applications implemented in software. This performance improvement can allow to implement some functions like deep packet inspection, distributed computing or machine learning acceleration, that were performed initially by the control plane. This make possible to deploy more accurate solutions to reduce problems in the control plane.

2.2 BPP over P4

In [7] François et al. realize a first attempt to develop a Proof-of-Concept implementation of NewIP/BPP, using P4. NewIP/BPP adds some novel requirements that imply P4 limitations very challenging to overcome.
The latest version (P4_16) of this language is here considered. P4 offers high flexibility, and is capable to realize an hardware-agnostic switch abstraction in order to express complex network processing.
P4 programs can also be transparently compiled with respect to specific platforms such as FPGA (Field Programmable Gate Arrays), using a target-independent representation. The v1model included in the standard architecture of P4 is here used to build a L3 switch named BMv2 (Behavioral Model v2).
It is made up of the following parts:

- parser: P4 packet parser that is like a state-machine
- verify checksum: Verification of Packet checksum
- ingress control: Actions on ingress packets
- egress control: Actions on egress packets
- compute checksum: Checksum computation
- deparser: Deparsing and forwarding.

An important point is here the necessity to specify the structure of packets handled by the P4 switch. This consists in defining the different packet header fields and their size. All these information are contained in the parser and deparser.
Ingress and egress control determine how the packets have to be processed together with match-action tables (for example, rewriting the IP address). In order to do this, certain switch metadata can be accessed:

- intrinsic metadata: given by the device platform,
- queue metadata: contain the switch queue,
- user-defined metadata: defined by the user.

Regarding the BPP properties, there are two particular challenges: variable substructures and variable number of processing steps required to deal with Collateral. BPP Collateral can have inside different substructures with variable length and different components. The command block, for example can have inside a certain number of commands. Each command can also contain various conditions and

actions with inside a varying number of parameters.
The metadata field, can be also of variable length. P4 simply does not support multiple fields with varying lengths. In theory, the entire Collateral should be put inside a unique varbit field but this goes against P4's purposes because packet handling will be done manually. In order to manage this limitation, the implementation has been limited accordingly by the authors. A fixed Collateral with a fixed number of commands, parameters and actions is here used. Another problem regards the fact that parameters could need to be taken from different sources depending on the parameter category:

- Value: The parameter field contains the value of the parameter itself.
- Meta: The parameter field has a reference to packet metadata.
- Data item: The parameter field is referred to data item.

Some examples of data items are the packet's length, egress, queue depth, or memory utilization. The most difficult operations to manage in the P4's pipeline, are the parsing and deparsing of packets. This is due to the complexity of BPP headers with many variable-length structures. In order to simplify the nested structure of the BPP packet, the following chain is built, thanks to a "next field" value, which creates an horizontal schema (Fig. 2.4 (a)). For instance, a BPP packet could be composed by:

1. BPP Header
2. BPP Command
3. BPP Condition Set
4. BPP Condition
5. BPP Parameter
6. BPP Parameter
7. BPP Condition
8. BPP Parameter
9. BPP Parameter
10. BPP Action Set
11. BPP Action
12. BPP Parameter

13. BPP Metadata

The parsing process follows also the steps reported above.
Even if the problem of the nested structure is resolved, deparsing becomes very difficult with this kind of structure. It is not possible in P4 define a method to pull the headers in the correct order. For this reason, the static implementation of conditions and actions is preferred even if it decreases the flexibility of the BPP solution with fixed amount and lengths of parameters. Adopting this affordable strategy, the chaining mechanism is no longer required (Fig. 2.4 (b)).
The two main proposed solution are reported in the following picture

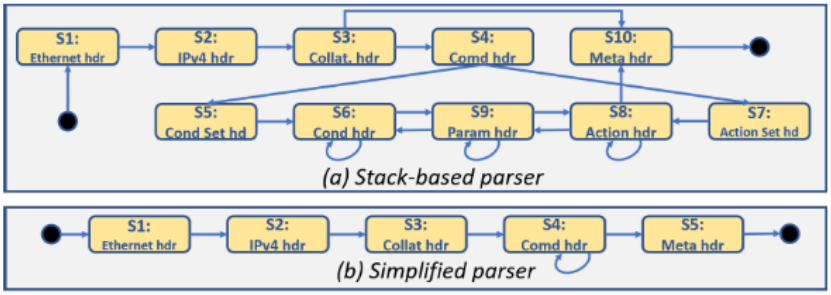

Figure 2.4: P4 parsers for BPP

The last BPP implementation has been then integrated with a L3 switch and validated using p4app that uses Mininet and bmv2 model. The topology is composed by three switches and three hosts as figured below:

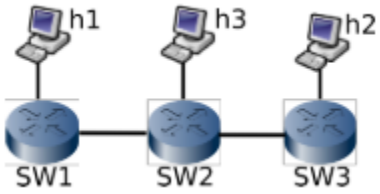

Figure 2.5: BPP - P4 Topology

The shown topology is simple but able to test and verify the BPP implementation correctness with ICMP traffic.

Chapter 3

Background

3.1 P4

3.1.1 P4 Overview

The P4 (Programming Protocol-Indipendent Packet Processors) project was released between 2015 and 2017. Its declared goal is to provide a flexible way to program network devices in order to optimize network traffic and improve the flow management giving the full control directly to the programmer.
This programming language works in conjunction with SDN (Software-defined Network) model and it is based on the same idea of the OpenFlow protocol.
The latter specifies a set of rules and specifics which do not fit for critical use cases, so the implementation of a new programming language, capable of managing, in a more pr precise manner, the forwarding logic, was required.
P4 allows programmers to manage packets by adding, removing or modifying headers. It is easy to get access to network layers, extrapolate information and parsing them in a very granular and low level way. It can be considered like an high level programming language with C-like capabilities to control data organizing them in row structure and controlling them in terms of bytes allocated.
It is also possible to create "actions", which can be considered like functions or methods of other programming languages, and tables, which are basically the rules applied on the devices. In this scenario, it is important to highlight that P4 is capable of control traffic independently by the protocol adopted and that rules can be injected on devices not only when they receive a packet but also when they forward them.

More specifically, a P4 program should be composed by:

- Headers
- Parsers
- Actions
- Tables
- Flow Control
- Match-action units
- Extern object
- User-defined metadata
- Intrinsic metadata

Starting from the beginning, headers are defined as a continuous set of structures composed by bits fields. The custom position of each field and its size setting is responsibility of the programmer.
The following piece of code shows the way the header is built and the possibility to encapsulate different structures into it.

Listing 3.1: Header example

```
header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<6>    dscp;
    bit<2>    ecn;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}
```

Regarding Parsers, they are basically the components in charge of the syntax and syntactic analysis of a sequence of symbols.
They are fundamental in order to manage packets in the correct way and let each network component deal with data as they expect them.

Listing 3.2: Parser example

```
state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition select(hdr.ipv4.protocol){
        6 : parse_tcp;
        IPV4_BPP_PROTOCOL: parse_bpp;
        default: accept;
    }
}
```

Actions represents the interactions between network components. They are basically functions which describe what they should do when a packet is received or before sending one. They can be simple or complex, and they are strongly related to tables.
The following code is an example of how it is possible to create a function in P4 which elaborates the ECMP (Equal-cost multi-path routing) group mechanism thanks to an hash function which takes as arguments some specific bits taken from the packet header.

Listing 3.3: Actions example

```
action ecmp_group(bit<14> ecmp_group_id, bit<16> num_nhops){
    hash(meta.ecmp_hash,
    HashAlgorithm.crc16,
    (bit<1>)0,
    { hdr.ipv4.srcAddr,
      hdr.ipv4.dstAddr,
          hdr.tcp.srcPort,
          hdr.tcp.dstPort,
          hdr.ipv4.protocol,
          meta.flowlet_id},
    num_nhops);

    meta.ecmp_group_id = ecmp_group_id;
}
```

Regarding tables, they take the actions and execute them when certain conditions are verified. Actions and tables, together, can be used to write device commands, which can be for example a ".txt" file, where all the arguments eventually passed to the actions are specified.

Listing 3.4: Table example

```
table ecmp_group_to_nhop {
    key = {
        meta.ecmp_group_id:    exact;
        meta.ecmp_hash: exact;
    }
    actions = {
```

```
            drop;
            set_nhop;
        }
        size = 1024;
    }
```

The Flow Control is the mechanism which evaluates the order execution. Considering all the pairs actions-tables, it specifies what pair should be performed first based on the program logic. Making a comparison with the other programming languages, it is like a main function launched when the program starts. The control program determines the order of match-action tables that are applied to a packet. A simple imperative program describe the flow of control between match-action tables.[8]
The Match-action units perform the three different operations:
First, they build a lookup keys from packet fields or computed metadata, then they make a table lookup using the previous obtained key, choosing an action (including the associated data) to execute, and at the end, execute the selected action. It is basically the way different actions are managed.
Extern objects are architecture-specific constructs that can be manipulated by P4 programs through well-defined APIs, but whose internal behavior is hard-wired (e.g., checksum units) and hence not programmable using P4.
User-defined metadata: user-defined data structures associated with each packet.
Intrinsic metadata: metadata provided by the architecture associated with each packet—e.g., the input port where a packet has been received. [9]

P4 is a language used to express how packets are processed by programmable forwarding elements such as a hardware or software switches, network interface cards, routers, or network appliance. Initially, P4 was designed only for programming switches and then its objective has become to be used also for other kind of devices. Some devices are capable of implementing both a control plane and a data plane but P4 can be used only to implement the data plane functionality. P4 is also designed to define the kind of interface between the control plane and the data plane, but it is not capable to describe the control plane functionality of the specific device. In the figure below it is possible to observe the differences between a traditional switch and a P4-programmable one:

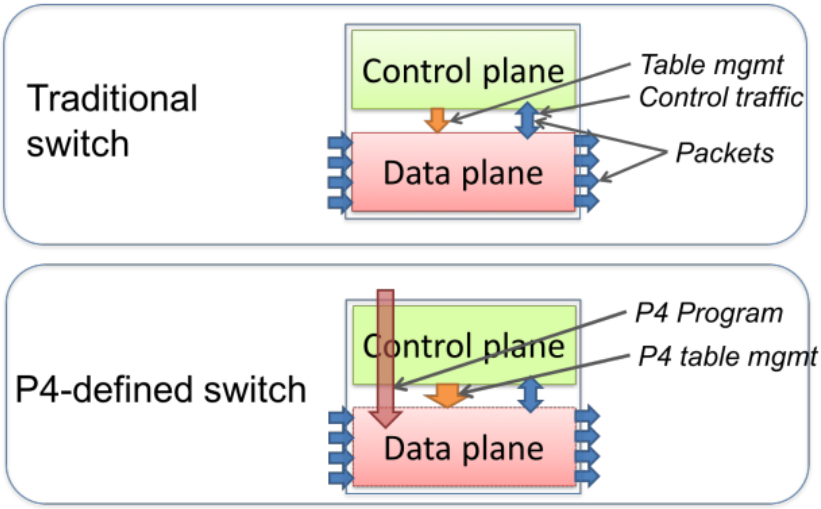

Figure 3.1: P4 switch vs Traditional switch

The two main differences from a traditional switch reside in the fact that the data plane functionality is not established in advance and the set of tables and other objects in the data plane are no longer frozen, since the P4 program itself is capable to define them. In a simple switch the manufacturer defines the routing tables and the configuration of the meters in order to establish the interactions between the control plane and the data plane.
P4 makes possible to configure the data plane to implement the features described in the P4 program itself and has no knowledge of the network protocols: it generates the APIs that the control plane uses to communicate with the data plane.
Even if P4 could be considered protocol independent, it allows to express different types of data planes and protocols.
Compiling a set of P4 programs has two direct consequences:

- A data plane configuration is implemented in order to set up the forwarding logic described in the input program;
- an API for managing the state of the data plane objects from the control plane is provided.

P4 is a domain-specific language that is designed to be implementable on a large variety of targets including programmable network interface cards, FPGAs, software switches, and hardware ASICs. Due to this assumptions, the language is restricted to constructs and structures that can be optimally implemented on all of these hardware devices and platforms.
Assuming that, for table lookup operations and interactions with external objects have a fixed cost, all P4 programs execute a constant number of operations for each

byte of an ingress packet received and processed. Knowing that the parsers may contain loops and segments of the header are extracted on each cycle, the packet size itself provides a bound on the total execution time of the parser. In other words, under these assumptions, the computational complexity of a P4 program is linear in the total size of all headers, and never depends on the size of the state accumulated while processing data (e.g., the number of flows, or the total number of packets processed). These guarantees are necessary (but not sufficient) for enabling fast packet processing across a variety of targets.

3.1.2 P4 Advantages and disadvantages

Despite is specific application, P4 is quite used today for networking management. However it is not a general purpose programming language so it presents some disadvantages

Pros

Starting from the advantages, the following list explains the focal points which encourage its usage:

- More flexibility: Differently from the traditional switch static management, P4 switches allows to inject lots of forwarding policies;
- Resource management: P4 allows to manage low level structure like bits variables. Moreover, during the compilation phase, all the resources defined by the user are mapped to hardware available resources;
- More expressiveness: P4 has the possibility to interact with packets regardless the protocol encapsulated. This allows its usage independently from the application encouraging the portability;
- Heterogeneous libraries: P4 allows to import several external libraries. Also the proprietary ones are admitted in order to allow different manufacturers to import their hardware specification into P4 structures;
- Software engineering: It is possible to exploit this programming language in order to perform traffic engineering, perform some checks on a specific type, hide some information or reuse the software;
- Testing and debug: Manufacturers can provide software models of an architecture to aid in the development and debugging of P4 programs.[10]
- Decoupling hardware and software evolution: Target manufacturers may use abstract architectures to further decouple the evolution of low-level architectural details from high-level processing;

Cons

Here, instead, are reported a couple of limitations due to the intrinsic applications of the language. The following are the staring points for the future improvement:

- Even if the some external libraries can be imported, there is no a standard set of methods implemented already implemented;

- There is no possibility to create loop constructs like for/while cycles. Only the parser logic provide this behaviour;
- Broadcast behaviour is not supported natively, it must be imported externally;
- Pointer and reference operator is not supported;
- Dynamic allocation is not supported;
- Packet trailer is not supported;
- Limitation in actions performed on packet payload;
- State management is not supported;
- New packet generation is not supported;

3.2 BPP (Big Packet Protocol)

3.2.1 BPP Overview

The emergence of SDN (Software-Defined Networks) has allowed programmers to make custom network behavior and to implement new service features in order to build controller software to program the network's control plane. The importance acquired by P4 in this context together with new networking applications have increased the need for high-precision communications and the warranty for "stringent service-level objectives" (such as end-to-end latency). In order to meet these new objectives a New IP / Big Packet Protocol (NewIP/BPP, or simply BPP), has been proposed in [7]: it is a protocol and frame that makes possible to guide the packet flow from the edge of the network using information encoded in the packets themselves [11]. The route can rely on dynamic conditions, for example the value of the queue depth.
In this way the flow is completely guided by information added to the packets without the need of programming specific controllers. This is the key point that brings BPP over traditional network technologies. The need to be supported on network devices and the definition of a new protocol makes P4 a good candidate to implement BPP. Some BPP features like for example the ability to support parametric commands lead to complex parsing of each field and this represents an aspect that for sure presses P4. The gap between the capacities of P4 and BPP is indeed very large.

3.2.2 BPP Background

The starting point is the introduction of a piece of BPP containing commands and metadata between the packet header and payload. These commands and metadata contain information about how to route the packets inside different network devices and how to process them. The commands are able to instruct a switch, for example, to change a metadata in certain conditions. In this case commands cannot be used to allow interactivity with payload. The structure of a BPP packet is represented below:

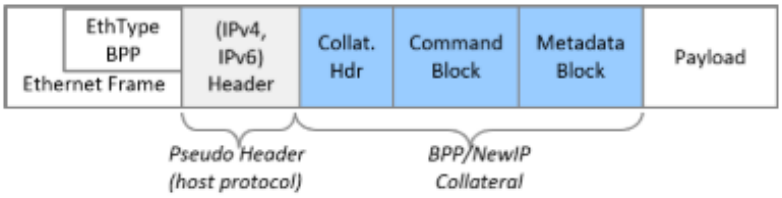

Figure 3.2: BPP Packet structure

The BPP Collateral is made up of a Collateral header, a command field, and a metadata field. Due to the fact that the Collateral should be different from the payload of the packet, the BPP protocol type is indicated updating the IPv4 header.
The Collateral header, instead, contains:

- NewIP/BPP version;
- Collateral length (16, 32, 128 or multiple of 32);
- Flags for error handling;
- A reserved flag for future applications;

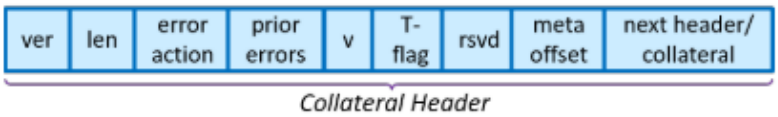

Figure 3.3: Collateral header

The command field contains a sequence of commands structured as described below:

- The command header contains the length of the command and flags;
- A condition set, containing its length and one or two conditions;
- An action set, performed if the condition set evaluates to true: each action contains a definition of the action to be performed and parameters.

According to the parameter's category, the value should be differently interpreted. It could be a static value to be used, a pointer to a next field specifying the correct offset or even an identifier of a data already defined like stadard_metadata structure.

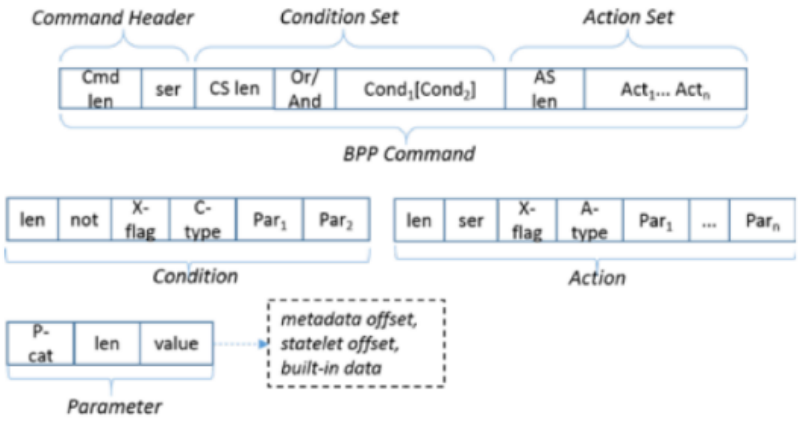

Figure 3.4: BPP Commands

3.3 TCP Congestion

3.3.1 TCP: Congestion Control Algorithms

The TCP protocol provides four different algorithms in order to manage with the congestion window problem. Slow Start, Congestion Avoidance, Fast Recovery, Fast Retransmit.

Slow Start and Congestion Avoidance

Slow Start and Congestion Avoidance are used together client side to control the amount of traffic to send through the network. This control is performed by means of three different variables: congestion window (cwnd) client side, receiver window (rwnd), slow start threshold (ssthresh) which is used to decide when the two algorithms should work. While the cwnd represents the maximum amount of data which can be transmitted before the acknowledgment receipt, the rwnd manage the limit amount the server can deal with. In order to create a reliable connection, the minimum between the two variables drives the communication.
When the transmission begins, the network conditions are unknown, so the Slow Start is necessary to test its capacity and avoid congestion. This algorithm is generally used whenever a communication starts or restarts. It is used the first time the client tries to transmit packets to the server but also every time it tries to

restore the communication after a loss or an error occurrence.
At the beginning, the ssthresh is an high value in order to allows the client to set the desired bit rate but every time a congestion is detected, this value is reduced. In details, Slow Start works when the cwnd < ssthresh and it progressively increases the cwnd, Congestion Avoidance starts when cwnd > sshtresh. If cwnd = ssthresh, one of the two is chosen by the client.
During Congestion Avoidance, the cwnd is linearly increased until a congestion condition is reached. When this event happen, the ssthresh is reduced and the cwnd restart growing according to Slow Start algorithm, starting from a lower value. The two described algorithms work cyclically until the communication is closed.

Fast Retransmit and Fast Recovery

As, for Slow Start and Cogestion Avoidance, Fast Retrasmit and Fast Recovery works together. There two algorithms deal with the retransmission management which in TCP is performed thanks to the acknowledgement (ACK) mechanism.
Server side, if an out-of order packet arrives, it sends an immediate duplicate ACK. The goal of this packet is to inform the client that an out-of order packet is received and the sequence number is expected. From the client point of view, there is not a specific way to justify a duplicate ACKs, but this phenomena can be caused by several network problems.
First, the packets can be dropped by one of the intermediate nodes. If this is the case, all packets after the dropped one will forward duplicate ACKs until the loss is repaired, signaling the event to the client. Second, a packet with sequence number greater than another can be received earlier because of the different path taken by the two, but even along the same path this event is not rare. Finally, duplicate ACKs can be caused by replication of ACK or data segments by the network.[12]
The Fast Retransmit algorithm is used by the client to detect and repair loss, based on incoming duplicate ACKs. The number of consecutive packets which triggers the algorithm is 3. This event is considered like the indication of a packet loss. After this, the client starts with the retransmission process of the packet in order to re-establish the correct stream.
If the Fast Retransmit understands which is the packet loss and provides the timely retransmission, the Fast Recovery algorithm drives the transmission of new data until a non-duplicate ACK arrives. The reason for not performing Slow Start is that the receipt of the duplicate ACKs not only means that a packet has been lost, but also that segments are most likely leaving the network. In other words, since the receiver can only generate a duplicate ACK when a segment has arrived, that segment has left the network and is in the receiver's buffer, so we know it is no longer consuming network resources. Furthermore, since the ACK clock is preserved,

the TCP client can continue to transmit new frames (although transmission must continue using a reduced cwnd, since loss is an indication of congestion).

3.3.2 Owl

As mentioned on the previous paragraph, one of the biggest network problems regarding the Transport Layer is the evaluation of the congestion window (cwnd). The TCP protocol, which is widely used for its reliability, could be not the optimal solution for many new applications we have today.
The reason behind this is the variety of network scenarios. Wireless or IoT networks, which are very common today, are subjected to several constraints like limited bandwidth, battery power, processing capabilities, in relation to dynamic conditions.
The standard congestion control algorithms performs optimally with static conditions, wired networks, few packet loss and noise, no bandwidth restrictions. Considering mobile and data center networks, it is easy to notice how requirements and needs can be different or even opposite from one system to the other. It follows that TCP performs differently setting the congestion window according to its standard rules without taking into account all these environment conditions.
Based on this considerations, the community started to develop and implement transport protocol solutions which are driven by Machine Learning algorithms that can provide a training model based on the metrics collected. This is the scenario where "Owl takes place. Several previous solutions (e.g., PCC [13] and Copa [14]) are based on a set of static strategies which result inadequate with the dynamic environment concept.
Owl is a transport transport protocol based on reinforcement learning, whose goal is to select the proper congestion window learning from end-to-end features and network signals, when available[15].
A congestion control protocol which performs optimally, is fundamental for proper network operation because it ensures reliability, telecommunication stability, fairness in computer network resource utilization, high throughput, and a low switch queuing delay.
TCP is a deterministic protocol which does not fit well with network changes. Focusing on mobile networks it is clear how mobility, which implies multiple recalculations of the best routing path and nodes involved, can produce several retransmissions. As result, the congestion window is often reduced according to the algorithm causing low values of throughput.
In the last few years, many different transport protocols based on Machine Learning support have been implemented. The most recent implementations using reinforcement leaning, however, do not exploit network intelligence fully [15]. One of the main capability brought by Owl is the online training mechanism to decide the

best cwnd value based on the in-band parameters computation.
Owl layer 4 protocol has the capacity to behave as described above not only for single connections but also considering multiple end-to-end communication. The kernel module implemented for Owl is a completely new version able to capture variables of interest to train a reinforcement learning algorithm based on Deep Q-Learning. [16]The main output of Owl model is the value of the next congestion window that is crucial, variable network parameter used to analyze the reliability of a network communication.
The reason why Owl is chosen over any other similar transport protocol implementations based on ML approach, is that, compared to them, Owl has achieved better results in term of delay improvements and bandwidth.
While the other implementations have been tested on specific applications, Owl testing environment tries to be as general as possible, examining its behavior in several Linux distributions or wireless networks.
Owl testing results demonstrate good performances together with less aggressive behavior with respect to others.

Chapter 4

System

4.1 System overview

Studing the In-band Network Telemetry potentialities, the focus of this thesis is moved on the actual implementation of a system which can be useful for the final goal. Since the final scenario is to feed the Machine Learning "Owl" algorithm with metrics collected by switches through the P4 capabilities, an entire network is realized.
The ingredients needed in order to create the latter are the end points and the network. Here, the focus is not on the creation of articulates network topology or powerful hosts, the idea is just to provide a basic schema to be used as a model for any kind of network conditions and development.
The end points mentioned above are responsible of the communication management starting with the creation of the socket client side, ending with the closing the connection server side. On the other hand, the network is the component in charge of the transmission and the one which contains all the information needed.

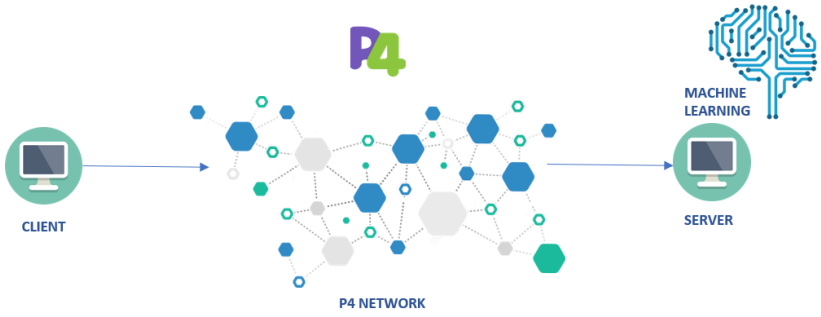

Figure 4.1: System Architecture

The above figure represents the System Architecture of the network proposed:

- Client represents the host of the P4 topology which starts sending packets;
- P4 Network is made up of different P4 switches that forward the incoming packets to the defined destination;
- Server is the final receiver, where Owl is loaded at kernel level.

The idea is to keep as much general as possible the solution in order to be adopted for future works. To do so, two scripts which emulate a client-server service are written by means of general purpose programming languages like Python, C++, Java. Using their features, all the logic for the communication establishment starting from the socket creation, ending to the communication closing, is implemented. While the client is responsible for the creation of the packets, the socket opening and the sending operation with specific bitrate, the server is in charge of the bind, listening and accept functions which drive the main connection. Moreover, the buffer size, which must receive the data arriving from the socket, is managed on server side.
In particular, a TCP Connection is established by means of two scripts. For the standard IP implementation, only simple IP packets are forwarded. Besides, for the BPP application, the BPP integration inside the IP packet is performed.

4.2 P4 Switches

In this section is reported a description of the main component of the P4 network: the P4 switch.
First, some information about the architecture, the main components and the most common interaction are illustrated in the following picture.
The comprehension of these components help to understand P4 programs' flow and which part of it can be modified in order to perform some specific task.

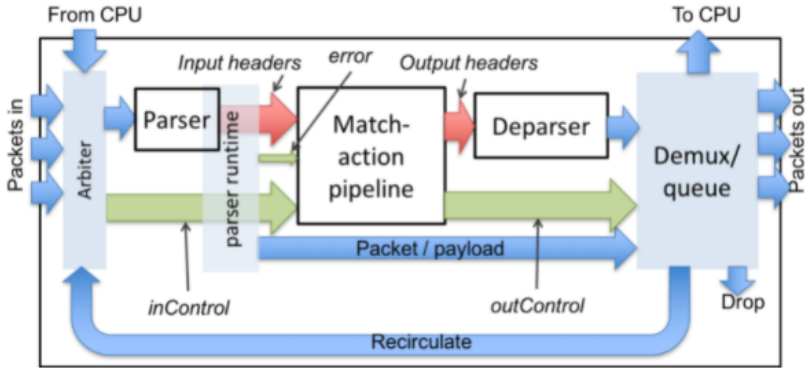

Figure 4.2: P4 switch architecture

There is nothing inherently particular in the diagram, it is just and example to learn the main feature of the switch. The cyan blocks in the architecture are fixed, so they can not be modified internally. Besides, the white ones, are programmable using P4.

The diagram receives packets from different sources: through one of 8 input Ethernet ports, through a recirculation channel, or from a port connected directly to the CPU. [9]
The flow is the following: incoming packets are parsed from the parser component, then they are taken as input into the single match-action pipeline, which forwards them into the deparser, in charge of the recreation of the packet after the parsing operation.
Exiting from the latter, the packets can reach the "drop port", which simply discards the packet, the "CPU port", which sends them directly to the control plane, the "recirculate port" witch is linked to the special input port mentioned above.
If non of this path is taken, the packets are forwarded into one of the 8 output Ethernet ports in output.
The way the white blocks performed is responsibility of the programmer. Each one of them must have a corresponding P4 program which describes the behaviour of

these basic components.
Focusing on the links, different colors means different purpose. While the cyan arrows (and blocks) are fixed functionalities, the green arrows are a way to represent the interaction between the fixed-function blocks and the programmable ones. The red arrows just show the standard path of the user-defined data through the switch.
Moving on the coding specification, each P4 program must contain declarations, type declarations, constants, variables and the declaration of the white blocks. Also the implementation of the latter must be provided by the programmer.

- Parser: it takes packets in input from the dedicated ports. It also reads its input from a dedicated external object defined in the core.p4 called packet_in, in charge of represent in the incoming packets.
  After that, the parser writes its output into the parsedHeaders argument.

- Match-action pipeline: this block receives different inputs. The first one are the parsed data from the parser block. The second one is the parsed error. The third one is the inControl data.
  The output of the block are the outControl and the data ready for the deparsing operation.

- Deparser: this block consumes the date received by the pipeline in order to reassemble the packet previously parserd.

. Once performed the analysis of the white boxes, in order to have a better view of the internal processes of the switch, a description of the non-programmable fixed block is needed.

Arbiter block

This is the first block encountered by the incoming packets. It manages all the traffic incoming from the different sources. According to that it performs the following actions:

- If packets are received from Ethernet port, it computes its trailer, the checksum and verify its integrity. If the integrity check is positive, the checksum is removed from the packet payload and continues its path. In case of negative outcome, the packet is dropped.

- It manages the concurrency of multiple packets incoming. It must run some logic checks or algorithm in order to decide the order and the priority of all the packets red simultaneously.
  According to this, if the component is already working on another packet, it puts the new one in the queue in order to treat it as soon as it finish with the

old one.
In case of network congestion, the queue can be saturated. If this happens, the ports are locked in order to not receive any more traffic and do not compromise the correct switch behaviour. This means that, in this scenario, the packets are simply dropped without any feedback mechanism.

- Before sending the packets to the parser, it sets the soruce port number accordingly to the following strategy: if the source is one of the 8 Ethernet port, the corresponding number from 0 to 7 is set. If the source port is the CPU port, the number set is 14. In case of inputs from recirculation port, the number set is 13.

Parser runtime

As suggested by it name, this component works in conjunction with the programmable block seen before. It is in charge of the runtime error management by setting the related error code, if it occurs, based on the parse actions. It also collects and forwards information about the payload, like the size of the remaining data, taken in input by the Demux/queue component. As soon as the packet parsing program developed by the programmer ends, the match-action pipeline function is called.

Demux/queue

The main purpose of the Demux/queue block is to gather all the header information retrieved from deparser and the ones related to the payload received by the parser. Then, it builds a new packet in order to allow data to be transmitted through the network.
Last step is to send packets to the correct output port. The latter is previously set by the match-action pipeline exploiting the outControl.outputPort value.
The different use cases are summarized like follows:

- When it sends packets to the Ethernet port, the corresponding physical interface number from 0 to 7 is chosen. If the port is busy emitting other packets, the current one is placed on the corresponding waiting buffer, which is basically the queue. When the packet is taken, before transmitting it, the checksum is computes and the trailer containing it is attached.

- If it sends the packet to the drop port, it is simply discarded from the network;

- If it send the packet to the control plane, through the CPU port. If this happens, this packet is the original input packet and not the one received from the deparser, which is discarded;

- if the packet process can not be terminated on a single step, but it requires more computational resources, the packet is sent to the recirculation port, which means that it appears as input one again;
- If the outputPort has an unexpected or wrong value, the packet is discarded;
- If there is no more space on the queue, if it receives more data in input they are simply discarderd, without considering the output port indicated;

The link reported starting from the parser runtime to the S Demux/queue block indicates another additional flows between the two. The latter means that not only the packet itself is parsed but also the offset between the packet where parsing ended, like the start of the payload.

4.3 Host-based TCP Congestion Control

Moving the attention from network side to host side, the goal is to enrich the host capabilities with the newborn TCP congestion control algorithm, Owl.
For this thesis purpose, the algorithm is loaded server side, but this is not a constraint because Owl can run either server side or client side, accordingly to the settings.
Owl Machine Learning Algorithm improves the characteristics of common TCP congestion control mechanisms, adopting a Reinforcement Learning model able to manage different network situation, and collecting specific metrics in order to take the best decision. Owl structure is based on three main components:

- Kernel Module: It is in charge of adapting the value of congestion window at privileged level.
- Environment: It converts the metrics received from the kernel module to a format that fits the RL algorithm needs.
- DeepQ-RL: It is the component able to take decisions inside the network depending on the environment feedback received.

There are two processes running in the user space and kernel space respectively.

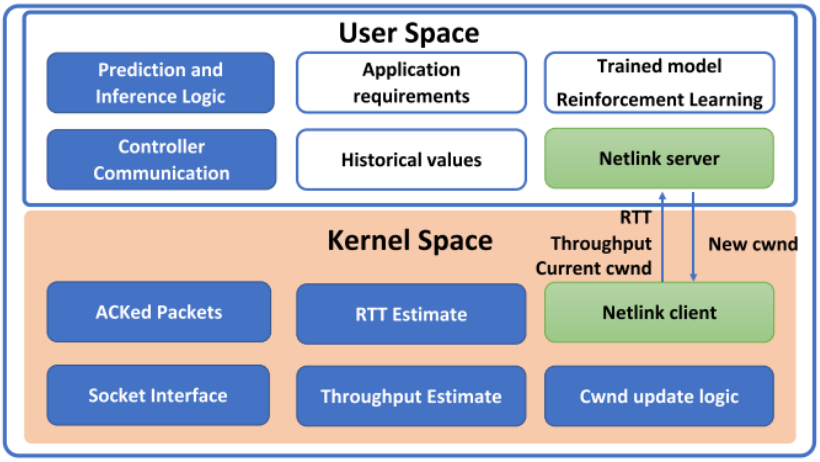

Figure 4.3: Owl Processes

The first one collects all the information which come from the TCP communication established with the client, then passes them to the reinforcement learning algorithm.
The second one has access to the congestion control functions of standard TCP protocol such as computation of throughput or RTT get. The basic implementation of the Owl congestion protocol runs a kernel module which reacts to every feedback received. Every time a feedback arrives, by means of a controller, a secondary thread is awaken to listen to the incoming data, gather them, and give them to the main thread. The reason behind this is that keeping the communication alive with the controller, collecting all the metrics and store them, in order to run an RL algorithm to modify the congestion window, can be an expensive operation in a privileged environment.
The proposed solution aims to integrate directly the metrics through a network socket. Instead of a simple controller, a network TCP server is implemented. Now, every time a packet arrives to the receiver socket, the Server accepts the packet, parses it extracting the interesting metrics for the algorithm and sends them to the main thread.
The usage of a user space thread is beneficial for several motivations. The most important one is the possibility to import external libraries, in particular all the network libraries for the packet management which ca be beneficial for the algorithm necessity. Besides, the context switch overhead must also to be taken into account when working between the two spaces. In fact, error management, which is an important part to ensure the correct functionality without causing any internal problem of the program, is delegated to the user-space.
As mentioned before, while user space is in charge of the network management for the improvement of the congestion window value decision, The Netlink service operates the marshalling of the TCP sockets between kernel and user level. Then, network parameters are collected from acknowledgments. While in the basic solution the controller works at a fixed frequency, the new logic implemented provides a way to react faster to the receipt of the feedback.

Chapter 5

Implementation

The Networked Systems Group (NSG) is a research group in the Department of Information Technology and Electrical Engineering (D-ITET) at ETH Zürich led by Prof. Laurent Vanbever [17].
They have realized the The "P4 learning" [18] which presents the starting scenario for the current work. Here, several exercises and examples are provided in order to explain directly with the code how it is possible to exploit P4 features to implement most of the network functionality used today.
The designed solution is composed by two different Linux Virtual Machines installed on Windows 10 host OS.
Mininet Network topology runs on Ubuntu 16 Virtual Machine in Oracle Virtualbox 6.1, while "Owl" TCP congestion algorithm is executed on a second virtual machine with Ubuntu 18 in VMware Workstation 16 Player.
The final goal is to train the Machine Learning "Owl" algorithm with metrics collected by switches through the Mininet network. Due to version compatibility reasons, it is necessary to run Owl and Mininet in two different environments, so a network communication is established.
The implementation of the system saw in the previous chapter begins thanks to two already implemented public GitHub projects. After a first phase of understanding and learning, the goal is to test their solutions and modify step by step their code.
The project is entirely realized on Linux OS. Due to the different versions, it is possible to follow the guidelines provided for the correct set up of the tools needed, or download directly a Virtual Machine with all the features and libraries already installed.
Using a Windows OS, the choice is the second. It is important to underline that the public repository is constantly updating, so it is fundamental to often pull the commits synchronizing the local and the remote folder.

5.1 Project 1: simple_int

Starting from the "simple_int" project, the goal is to improve the already existing P4 program logic to gather the additional metrics described in "System". To do so, the following action is written:

Listing 5.1: add_int_header action implementation

```
action add_int_header(switch_id_t swid){

    //increase int stack counter by one
    hdr.int_count.num_switches = hdr.int_count.num_switches + 1;

    hdr.int_headers.push_front(1);
    // This was not needed in older specs. Now by default pushed
    // invalid elements are
    hdr.int_headers[0].setValid();
    hdr.int_headers[0].switch_id = (bit<6>)swid;
    hdr.int_headers[0].queue_depth = (bit<13>)standard_metadata.
   deq_qdepth;
    hdr.int_headers[0].time_gap = (bit<13>)(standard_metadata.
   egress_global_timestamp - standard_metadata.
   ingress_global_timestamp);

    //update ip header length
    hdr.ipv4.ihl = hdr.ipv4.ihl + 1;
    hdr.ipv4.totalLen = hdr.ipv4.totalLen + 4;
    hdr.ipv4_option.optionLength = hdr.ipv4_option.optionLength +
   4;
}
```

The metrics chosen for the analysis are the following ones:

1. **"switch_id"** containing the id number of each switched crossed.
2. **"queue_depth"** indicating the number of packets queued in each switch buffer.
3. **"time_gap"** computing the switch processing time as follows:

$$time\_gap = egress\_global\_timestamp - ingress\_global\_timestamp \tag{5.1}$$

This brief piece of code is composed by multiple parts described on "header.p4 and "parser.p4". The metrics specified are chosen to be stored on IP Options field are: switch id, queue_depth an time_gap.
Due to the IP Options size limitation of 32 bit, the number of bits which has to be allocated for each variable, must not exceed this value. Since the analyzed topology

is composed by few switches, 6 bits are sufficient to describe the switch id variable. Queue depth and time gap requires more bits because the can express larger values, so 13 bits for both of them are assigned.
The idea behind this implementation is to represent the int_header, which stands for in-band network telemetry header, as a vector. The reason of this structure is that every time a packet crosses a switch, the metrics are pushed on the vector head.
All these metrics are updated considering the routing path for each packet. Depending on the switch crossed, the correct switch ID is assigned to the switch_id variable. The same logic is applied to the other two metrics.
The queue_depth value is initially always equal to zero because no congestion or critical scenarios are met. Nevertheless, this is an important parameter for network telemetry so, the choice is to maintain it into the packet.
The time_gap value, instead, is computed as the difference between the global egress timestamp and the global ingress one in order to determine the processing time of each individual packet within a given switch.
Once the definition of these technical details are performed, the switches can run the above function through the so called "switch-commands.txt". It follows an example:

Listing 5.2: switch-commands.txt example

```
table_set_default ipv4_lpm drop
table_set_default ecmp_group_to_nhop drop

table_set_default int_table add_int_header 1

table_add ipv4_lpm set_nhop 10.0.1.1/32 =>  00:00:0a:00:01:01 1
table_add ipv4_lpm ecmp_group 10.0.6.2/32 => 1 4

table_add ecmp_group_to_nhop set_nhop 1 0 =>  00:00:00:02:01:00 2
table_add ecmp_group_to_nhop set_nhop 1 1 =>  00:00:00:03:01:00 3
table_add ecmp_group_to_nhop set_nhop 1 2 =>  00:00:00:04:01:00 4
table_add ecmp_group_to_nhop set_nhop 1 3 =>  00:00:00:05:01:00 5
```

Every switch has its related text commands which specifies the actions and tables defined in the P4 program to be run. Moreover, all the parameters taken in input are indicated. For example the add_int_headers action takes the switch ID as first parameter, so 1 is passed. With the same logic, the int_table associated to the action is reported.
Similarly, the ecmp_group_to_nhop and ipv4_lpm tables, set_nhop and ecmp_group actions are added to this switch command. The first table selects the group ID and the hash and chooses the next hop following the implementation of set_nhop. The second table selects the IP destination address according to the long prefix matching rule and computes the hash taking other parameters in input

Once the implementation of the L2 network is completed, the set up of the client-server service, in terms of addresses, interfaces and ports, must be realized in order to test if the switches work properly.
The "send.py" script shall contain the definition of the packet structure to be sent through the network, specifying each layer and their internal parameters like destination and source addresses or input/output ports. According to this, layer 2 must contain source and destination mac address while layer 3 carries all the IP specifications.
Since TCP is the protocol of interest within this project, layer 4 shall always correspond to TCP for the final purpose to integrate the metrics in the Owl congestion protocol, regardless of the session and application upper layers.
Because of the last assumption, the socket setup is performed considering TCP applications. This means that IPPROTO_TCP protocol and SOCK_STREAM type are chosen for the communication.
Receiver side, in this case H2, the receipt is managed by means of the "sniff()"[19] function reported below, which is modified from the starting version:

Listing 5.3: sniff() implementation

```
sniff(filter="tcp and port 1000", iface=iface,
prn = lambda x:handle(x), count=int(1), stop_filter = lambda x:x.
    haslayer(TCP))
```

The parameter passed are the rules related to the "sniff()" function:

- The first parameter is the BPF filter to apply. This means that from all the packets received by the network card, only the ones related to the TCP communication at 1000 port must be sniffed;
- The second parameter specifies the sniffing interface according to the port;
- The prn argument indicates a lambda function which must be execute every time a packet is detected. If it return something, it is shown on the terminal;
- The count parameter just determines the number of packets to capture. If it is set to 0, it means infinity;
- The stop_filter parameter accepts a lambda function to be applied to each packet to determine if the capture must be stopped or not.

The "sniff()" function is basically composed of an asynchronous class called "AsyncSniffer" which returns a list of packets. The "handle()" function is written in order to print the packet sniffed on the terminal by means of the "show()" or "show2()" functions for debugging purpose. The difference between the two is that while the first is meant for a developed view of the packet, the second is the same as "show()"

but on the assembled packet (checksum is calculated, for instance)[19].
In general, the "sniff()" function does not stop working if a mechanism is not implemented in order to be process all the data as soon as something is detected on the interface. For this project's purpose, a specific number of packets must be sniffed, so in order to stop the function, the count parameter is set to 1 and the stop_filter parameter accept a function that recognize if the packet sniffed has a TCP layer or not.

5.2 Project 2: BPPFlowlet

The same flow followed for the simple_int project is exported for the BPPFlowlet example. Here, the metrics to transmit are added to the Metadata field of the BPP headers. In order to leave the format as much intact as possible, it is not foreseen an additional header but free bits of metadata field are exploited.

Listing 5.4: BPP Metadata assignment

```
header bppMetadata_t {
    bit<64>          id;          // SWITCH ID 1
    bit<64>          data1;       // already used
    bit<64>          data2;       // already used
    bit<64>          data3;       // QUEUE DEPTH 1
    bit<64>          data4;       // TIME GAP 1
    bit<64>          data5;       // SWITCH ID 2
    bit<64>          data6;       // QUEUE DEPTH 2
    bit<64>          data7;       // TIME GAP 2
    bit<64>          data8;       // SWITCH ID 3
    bit<64>          data9;       // already used
    bit<64>          data10;      // QUEUE DEPTH 3
    bit<24>          empty;       // TIME GAP 3
    bit<8>           next;
}
```

It is clear that this choice has a space limitation but for testing performances between this solution and the one exploiting the IP Options field, this implementation is reasonable.
More specifically, the number of metadata field existing are used for saving the metrics for only three switches crossed, so every time a device is traversed, a triplet is is added the the header. If the topology presents more switches, or the ID passed does not exist in the network, the metrics are not stored correctly into the packet and the final Server will not receive it.
The solution is similar to the one realized for the previous project, with the only difference that a static structure with the size of bppMetadata_t is used instead of a dynamic structure like the int_headers.

Listing 5.5: add_Metrics action implementation

```
action add_Metrics(bit<64> switchID){

    hdr.bpp_md.setValid();
    if (switchID == 1) {
        hdr.bpp_md.id = switchID;
        hdr.bpp_md.data3 = (bit<64>)standard_metadata.deq_qdepth;
```

```
        hdr.bpp_md.data4 = (bit<64>)(standard_metadata.
    egress_global_timestamp - standard_metadata.
    ingress_global_timestamp);
     }
     if (switchID == 2 || switchID == 3 || switchID == 4 ||
    switchID == 5) {
        hdr.bpp_md.data5 = switchID;
        hdr.bpp_md.data6 = (bit<64>)standard_metadata.deq_qdepth;
        hdr.bpp_md.data7 = (bit<64>)(standard_metadata.
    egress_global_timestamp - standard_metadata.
    ingress_global_timestamp);
     }
     if (switchID == 6) {
        hdr.bpp_md.data8 = switchID;
        hdr.bpp_md.data10 = (bit<64>)standard_metadata.deq_qdepth;
        hdr.bpp_md.empty = (bit<24>)(standard_metadata.
    egress_global_timestamp - standard_metadata.
    ingress_global_timestamp); // egress - ingress
     }
}
```

Once the P4 network environment is updated, the focus is moved on the end point communications. Their scripts must be implemented correctly in order to have evidence of the correct metrics update performed by each switch crossed. In this case, the client and the server Python scripts realized are a little bit different than before.
This time, the construction of the packet to be sent client side is more difficult because of its different structure. Using the "bpp.py" file which defines the "BPP-Header", "BPPCommand" and "BPPMetadata" class, it is possible to insert each segment into the packet in the specific order
. The "clientbpp.py" declares the packet structure composed by one BPP header, two BPP Command and the BPP Metadata. Obviously, the IP and TCP layer are inserted also with the correct parameters. Regarding the IP layes, this time as "proto" parameters there is not the TCP layer directly, but the BPP identifier. In the "bpp.py" file mentioned above, the BPP_TCP constant identifier equal to 0x06 and the IP_BPP one equal to 0xFD are defined.
Either in this project the interest is on the TCP traffic, so the definition of the TCP layer is also respected. What changes here, is the socket management due to parsing problems.
Since the packet build is more complex with the BPP protocol usage, the possibility to access and debug directly the the raw bytes is preferred, so the socket are modified accordingly.
Based on the last consideration, the socket type and protocol become SOCK_RAW and IPPROTO_RAW.
Server side, the receipt is no longer managed by the "sniff()" primitive but the

"recvfrom()" function is used. In this scenario, the "recv()" method is fine too, both of the functions return the binary interpretation of the data received, the only difference between the two is that the "recv()" just returns this, the "recvfrom()" returns a tuple with data and source address, instead.
Now the parsing function has the role to interpret in the correct way the string of bytes received from the socket. The strategy adopted is the following:

Listing 5.6: Parsing BPP packet

```
def parse_bpp_command(bpp_command):
    bppcom = unpack('!HBHH4B2HI2HI4B2HI2HII4B2HI2HI4B2HI2HIB',
   bpp_command)
    '''
    Each command has two actions and two condition each one with
   respective parameters
    '''
    scpy_bpp_comm = BPPCommand(bpp_command)
    scpy_bpp_comm.show()
    return scpy_bpp_comm.next

def parse_bpp_metadata(bpp_metadata):
    scpy_bpp_meta = BPPMetadata(bpp_metadata)
    scpy_bpp_meta.show()
    return scpy_bpp_meta.next

def parse_silent(packet): #find parsing per sock stream
    print("[PARSING] Parsing received packet")

    # The IP header is 20 bytes long
    ip_header = packet[14:34]
    # Interpret the ip_header with the given format.
    iph = unpack('!BBHHHBBH4s4s', ip_header)
    version_ihl = iph[0]
    version = version_ihl >> 4
    ihl = version_ihl & 0xF
    iph_length = ihl * 4
    ttl = iph[5]
    protocol = iph[6]
    # Convert IP addresses to human-readable format
    s_addr = socket.inet_ntoa(iph[8])
    d_addr = socket.inet_ntoa(iph[9])
    print("[PARSING] IP correctly parsed")

    print('Version : ' + str(version) + ' IP Header Length : ' +
   str(ihl) + ' TTL : ' + str(ttl) + ' Protocol : ' + str(
        protocol) + ' Source Address : ' + str(s_addr) + '
   Destination Address : ' + str(d_addr))
```

```
    if protocol == 0xfd:

        # Extract BPP packet IBHB
        bpp_hdr = packet[34:42]
        scpy_bpp_hdr = BPPHeader(bpp_hdr)
        scpy_bpp_hdr.show()
        bp_next = scpy_bpp_hdr.next
        ptr = 42

        # After testing change this if for a while
        while bp_next == BPP_COMMAND:
            bp_next = parse_bpp_command(packet[ptr:ptr + 92])
            ptr = ptr + 92

        while bp_next == BPP_METADATA:
            bp_next = parse_bpp_metadata(packet[ptr:ptr + 92])
            ptr = ptr + 92

    # Parsing TCP layer
    tcp_hdr = packet[ptr:ptr+20]
    tcph = unpack('!HHLLBBHHH', tcp_hdr)
    source_port = tcph[0]
    dest_port = tcph[1]
    sequence = tcph[2]
    ack = tcph[3]
    doff_reserved = tcph[4]
    tcph_length = doff_reserved >> 4
    print ("source port :" + str(source_port) + " dest port :" +
    str(dest_port) + " seq number :" + str(sequence) + " ack :" +
   str(ack) + " tcp h length " + str(tcph_length))
    print ("source port :" + str(source_port) + " dest port :" +
   str(dest_port) + " seq number :" + str(sequence) + " ack :" +
   str(ack) + " tcp h length " + str(tcph_length))
```

It can be noticed that this parser provides a way to understand correctly several Commands and Metadata successively. If the following protocol field is a BPP (0xFD), the two while loop, together with the "ptr" variable, provide this feature increasing the variable by 92 bytes, which is the BPP_Commands' and BPP_Metadata's size.
Moreover, the parsing of the TCP layer is added in order to be sure that the the traffic received from the socket is a TCP flow, even with the SOCK_RAW socket setup.

5.3 Owl integration

In this brief section, the System section "Owl structure" is referred. The goal is to go into more specific details seeing how the metrics are taken from the Server socket and passed to the kernel thread which is responsible of the train model management.
The first operation needed to the correct integration, starting from the basic implementation, is increasing the number of feature accepted by the algorithm. In order to do so the following lines are modified:

Listing 5.7: Changing configuration model file

```
[DEFAULT]
model_name = model_v1
history_name = history_v1
...
[ENV]
...
num_features = 15
...
with_network_feedback = True
...
```

This is a configuration file for the Machine Learning model. In order to add the metrics collected through the client-server system implemented, the number of feature value must increase accordingly with the number of additional metrics. Moreover the feedback functionally must be enabled setting the "with_network_feedback" variable to the Boolean value True.

Listing 5.8: Preparing received metrics

```
class Thread_Request(threading.Thread):
    def __init__(self, url, frequency):
        global sock
        threading.Thread.__init__(self)
        self.url = url
        self.request_frequency = frequency  # 1 second
        # Create a TCP/IP socket
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        # Bind the socket to the port
        server_address = ('192.168.88.79', 10000)
        print ('starting up on %s port %s' % server_address)
        sock.bind(server_address)
        sock.listen(1)

    def run(self):
        self.network_request()

    def network_request(self):
```

```
        global sock
        global val  # made global here

        count = 0

        if len(sys.argv) < 2:
            print("Usage: python <namefile> [packet amount].")
            exit(1)
        else:

        while True:
            try:
                print('waiting for a connection\n')
                connection, client_address = sock.accept()
                print ('connection from', client_address)
                data = connection.recvfrom(65565)
                count += 1
                ...
            finally:
                connection.close()
                if numbers != []:
                    c.acquire()
                    print("Acquire...")
                    val = numbers
                    print ("VAL :", val)
                    c.notify_all()
                    c.release()
```

The above piece of code represents the final Server, with its primitives for the connection management. Another operation to be performed by it is the packet parsing. More specifically, according to the mode, integers must be passed to the algorithm, the metrics value must be extracted by the entire packet. This part is omitted because it is composed by well known Python functions to manage complex strings using also regular expressions.
Once this cleaning operation is completed, a vector filled with all the metrics is build. The latter is assigned to the "val" variable which is the object passed as a feedback for the main thread.
The last operation is performed in mutual exclusion, thanks to the condition variable thread container and its related methods of lock management and notification.

Listing 5.9: Preparing received metrics

```
def _get_network_feedback(self):
        feedback.c.acquire()
        if feedback.val != 0:
            swid_1 = feedback.val[6]
            queue_depth_1 = feedback.val[7]
```

```
            procTime_1 = feedback.val[8]
            swid_2 = feedback.val[3]
            queue_depth_2 = feedback.val[4]
            procTime_2 = feedback.val[5]
            swid_3 = feedback.val[0]
            queue_depth_3 = feedback.val[1]
            procTime_3 = feedback.val[2]
        feedback.c.notify_all()
        feedback.c.release()
        if feedback.val != 0:
            return swid_1, queue_depth_1, procTime_1, swid_2,
    queue_depth_2, procTime_2, swid_3, queue_depth_3, procTime_3
```

The above function is defined in the "KernelEnv" class in which all the interaction with Netlink and the train model are defined. This function returns the collected metrics which are previously extracted from the val vector seen before.

5.4 Additional feature

For implementation reasons, an additional modification from the starting system architecture is applied. As mentioned above, the Mininet topology runs on a Ubuntu 16.04.7 LTS (Xenial Xerus) Virtual Machine, while the final server runs on a Ubuntu 18.04.6 LTS (Bionic Beaver) Virtual Machine. The choice of a different virtual environment is made due to the incompatibility problems found between Owl kernel features and the old Ubuntu 16 version OS. Moreover, several recent updates of the starting projects has been released, so reproducing the Mininet topology developed in the first VM to the Ubuntu18 one, requires several adjustments to the library functions and their management.
Because of these technical constraints, a topology change has been performed. First the two Virtual Machines' network cards have to be configured. Both are set to "Bridged" with "Replicate physical network connection state" flagged, with the aim to guarantee packets exchange between them through Host OS (Windows 10). An important remark to this point must be exposed: every time the Host OS changes its IP address, the IP addresses set up for the client and the server functionality must be set manually, no automatic retrieving methods is implemented.
Then, NAT is added to the first implementation of the Mininet topology:

Listing 5.10: Python topology definition

```
p4utils.mininetlib.network_API import NetworkAPI
from mininet.net import *
from p4utils.mininetlib.node import *

net = NetworkAPI()
natIP = '10.0.6.1'

# Network general options
net1.setLogLevel('info')
net1.enableCli()

# Network definition
net.addP4Switch('s1', cli_input='s1-commands.txt')
net.addP4Switch('s2', cli_input='s2-commands.txt')
net.addP4Switch('s3', cli_input='s3-commands.txt')
net.addP4Switch('s4', cli_input='s4-commands.txt')
net.addP4Switch('s5', cli_input='s5-commands.txt')
net.addP4Switch('s6', cli_input='s6-commands.txt')
net.setP4SourceAll('p4src/simple_int.p4')

net.addHost('h1')
net.addHost('h2')
print ("host added\n")
```

```
net.addLink('h1','s1')
net.addLink('h2','s6')
net.addLink('s1','s2')
net.addLink('s1','s3')
net.addLink('s1','s4')
net.addLink('s1','s5')
net.addLink('s2','s6')
net.addLink('s3','s6')
net.addLink('s4','s6')
net.addLink('s5','s6')
print ("link added\n")

nat0 = net.addNAT("nat0", connect=None, inNamespace=False, ip=
    natIP)
net.addLink(nat0, 's6')

# Assignment strategy
net.mixed()
# Start the network
net.startNetwork()
```

In this way it is possible to overcome the isolation created by the virtualization processes (inNamespace=False) and allow the hosts to communicate with the Bridged network.
For our design, packets are sent by Host-1 inside the Mininet, then switches populate the metrics inside them, and reach Host-2. Now, the latter is in charge of receiving the packets, processing them, and extracting the information of interest. Finally it sends the data to the second Virtual Machine where "Owl" runs, thanks to the NAT addresses translation.
In order to run the Mininet topology with P4-enabled switches, P4-Utils is adopted together with Mininet. P4-Utils adds to Mininet a command-line launcher "p4run" to start the virtual network environment and a way to define its specific configuration using JSON files. The usage of JSON is replaced by Python language which has more flexibility and an higher level of expressiveness.
In this specific case, the new extension of P4-Utils, "NetworkAPI.py", is exploited to create the NAT component, by means of the exposed primitive "addNAT()", originally declared in the Mininet library. This has led to the necessity to import also the NAT class from the "net.py" file in the Mininet library, inside the new extension.
The following lines are extracted by the previous figure:

Listing 5.11: NAT details

```
natIP = '10.0.6.1'
    ...
nat0 = net.addNAT("nat0", connect=None, inNamespace=False, ip=
    natIP)
```

```
net.addLink(nat0, 's6')
```

It is worth mentioning here that the NAT IP address is assigned to the same network of the Mininet topology in order to be reached from the host inside it. In addition, the link between S6 and NAT is added for optimization reasons, because according to our experimental topology this is the nearest switch to H2.
That is why the "s6-commands.txt" shows the fourth additional line:

Listing 5.12: NAT rule on S6-commands-txt

```
table_set_default ecmp_group_to_nhop drop

table_set_default int_table add_int_header 6

table_add ipv4_lpm set_nhop 10.0.6.2/32 =>  00:00:0a:00:06:02 1
table_add ipv4_lpm set_nhop 192.168.88.79/24 =>  00:00:0a:00:06:01
    2

table_add ipv4_lpm ecmp_group 10.0.1.0/24 => 1 4

table_add ecmp_group_to_nhop set_nhop 1 0 =>  00:00:00:02:06:00 3
table_add ecmp_group_to_nhop set_nhop 1 1 =>  00:00:00:03:06:00 4
table_add ecmp_group_to_nhop set_nhop 1 2 =>  00:00:00:04:06:00 5
table_add ecmp_group_to_nhop set_nhop 1 3 =>  00:00:00:05:06:00 6
```

The fourth line specifies that if the IP destination address is outside the Mininet topology, in this case the IP corresponds with the one assigned to the final Server, the destination mac address passed to the set_nhop action is the NAT one though the specific port.

Chapter 6

Evaluation Results

6.1 Evaluation Settings

The Mininet topology taken into account for testing the solution is the the following one:

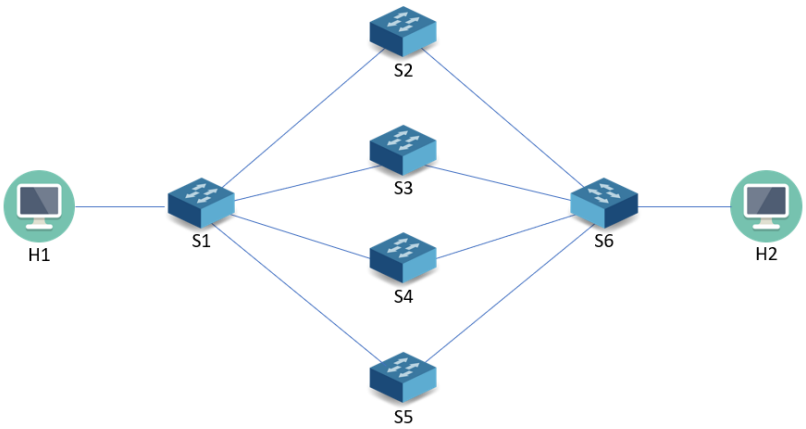

Figure 6.1: Mininet topology

The focus network is made up of two hosts and six switches so that traffic can flow from H1 to H2 and vice-versa through them. According to the rules specified inside the "commands.txt" files (one for each switch), packets can cross S2, S3, S4, S5 based on ECMP (Equal Cost Multi-Path) routing strategy. Every time the packet passes through this network, each switch assigns some specific metrics declared inside "simple_int.p4" and "flowlet_switching.p4" programs.
However, this is not the only possible implementation but different approaches and topology schemes with different network devices can be set up likewise.

In this dissertation two different kinds of packet are passed through the network: standard IP packets and custom IP packets enriched by a BPP (Big Packet Protocol) header. For the first implementation, metrics are stored in the "IP Options" field of the IP packet sent by the client host, while for the second one, metrics are stored in the "BPP_Metadata" field inside the header.

Once the Mininet topology is built, next step is to choose between "IP Options" and "BPP" solutions for metrics forwarding. To do so, Mininet is tested in terms of RTT (Round Trip Time) and FCT (Flow Completion Time) with both implementations. RTT computes the duration from when the client sends a packet to when it receives a response from the server. In this testing part, several packets are sent, so RTT Average is calculated. FCT is strongly related to TCP protocol because it computes the time from when the first packet of a flow is sent (in TCP, this is the SYN packet) until the last packet is received.[2] It is evident that the two parameters are directly related by the following empirical formula:

$$RTTAverage = FCT/N \tag{6.1}$$

with N number of packets.
These measurement are obtained by means of two Python scripts which execute an "Echo Service". This means that when Host-2 receives a packet from Host-1, it transmits the same message received to Host-1, capturing time accordingly to RTT and FCT definitions.
From client side, these metrics are computed thanks to the Python "time" library, while from server side, we only check if data are received correctly, without focus on data parsing, in order to perform a more accurate estimation. The files which compute this service are "eClient.py" and "eServer.py" for both projects. These two scripts are very similar to the ones in charge of the actual communication, the only difference is that the parsing is not performed server side in order to remove the additional overhead of the parser function and achieve a more accurate value.
Unfortunately, in a virtual environment, it is possible that time capturing in two different code executions of the same program may lead to different timestamps because of approximation errors. For this reason, starting from "simple_int" project, we test the solution by sending from 10 up to 500 packets with 0.5 seconds of delay between two consecutive sending.

6.2 Results

In order to evaluate and compare the results obtained from all the test performed, all the time measurement are aligned to seconds. In the next sections, the results are evaluated and the additional overhead due to the non optimal solution is calculated.

6.2.1 "IP Options" results

The following table shows the collected data for "IP Options" implementation:

IP Options				
#Packets	RTT Average (s)	FCT (s)	Network Delay (s)	Delay (s)
10	0,521	5,206	0,206	0,500
20	0,514	10,278	0,278	0,500
30	0,517	15,516	0,516	0,500
40	0,520	20,814	0,814	0,500
50	0,517	25,867	0,867	0,500
60	0,518	31,103	1,103	0,500
70	0,514	35,952	0,952	0,500
80	0,516	41,297	1,297	0,500
90	0,514	46,250	1,250	0,500
100	0,518	51,823	1,823	0,500
200	0,511	102,225	2,225	0,500
300	0,514	154,283	4,283	0,500
400	0,514	205,586	5,586	0,500
500	0,513	256,602	6,602	0,500

Figure 6.2: "IP Options": Data

As we can see, RTT Average is almost the same for each test, this result is expected because with this solution the standard IP packet format is not modified but only "Options" field is exploited to carry our data.

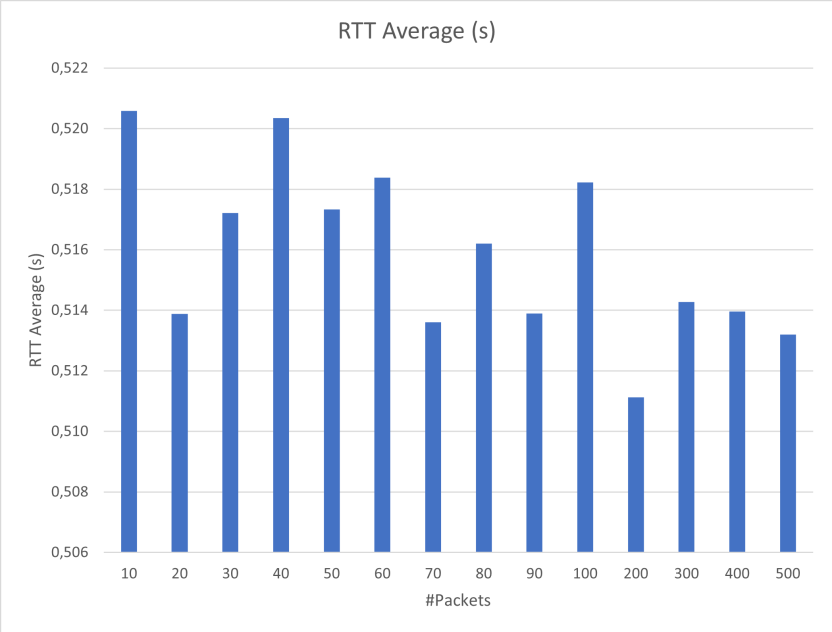

Figure 6.3: "IP Options": RTT Average

Network Delay increases proportionally with the number of packets transmitted except from 30 to 70 packets sent in which the difference is not so meaningful. These measurements are performed with Python time module by means of "time()" method which returns the time in seconds since the epoch as a floating point number. [20]

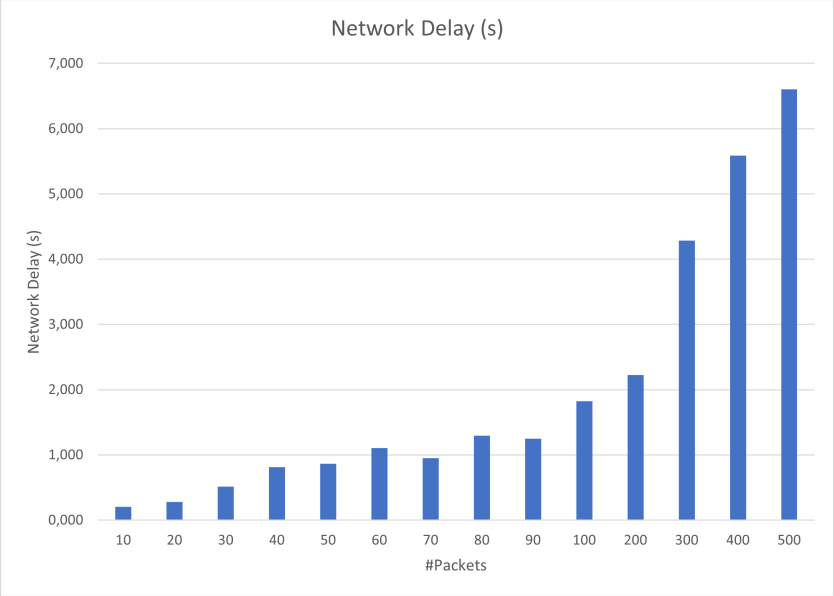

Figure 6.4: "IP Options": Network Delay

FCT increases as well proportionally with the number of packets.
In the next figure it is highlighted how much the network delay affects the overall computation of FCT for each test. This is very relevant for understanding the network status in terms of congestion and reliability. Our use case is very simple, with limited traffic, no congestion is realized, so these results are in line with our expectations.

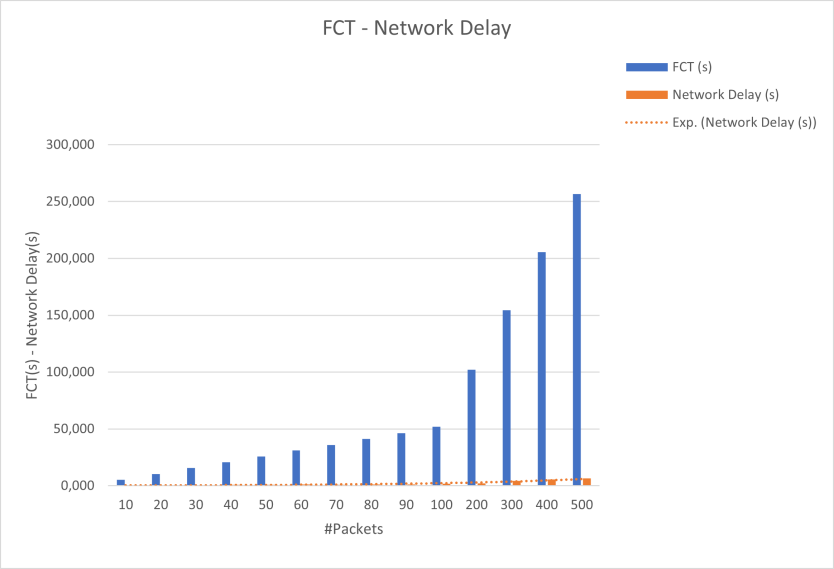

Figure 6.5: "IP Options": FCT - Network Delay

6.2.2 "BPP" results

Using the same metodology of "simple_int" project, the table containing the metrics calculated for "BPP" implementation is here shown:

BPP				
#Packets	RTT Average (s)	FCT (s)	Network Delay (s)	Diff.
10	0,563	5,633	0,633	0,500
20	0,582	11,633	1,633	0,500
30	0,621	18,641	3,641	0,500
40	0,624	24,948	4,948	0,500
50	0,602	30,110	5,110	0,500
60	0,579	34,744	4,744	0,500
70	0,588	41,127	6,127	0,500
80	0,608	48,634	8,634	0,500
90	0,598	53,778	8,778	0,500
100	0,567	56,664	6,664	0,500
200	0,575	115,051	15,051	0,500
300	0,584	175,219	25,219	0,500
400	0,596	238,242	38,242	0,500
500	0,587	293,635	43,635	0,500

Figure 6.6: "BPP": Data

Due to the fact that, in this case, the structure of the packets is more complex, containing an additional BPP header in the standard IP packet format, it is necessary to do different considerations about the results obtained.

RTT Average values are comparable to the ones computed in the previous project, but even little increments of RTT value for each packet can lead to a consistent rise in the total delay of the entire communication.

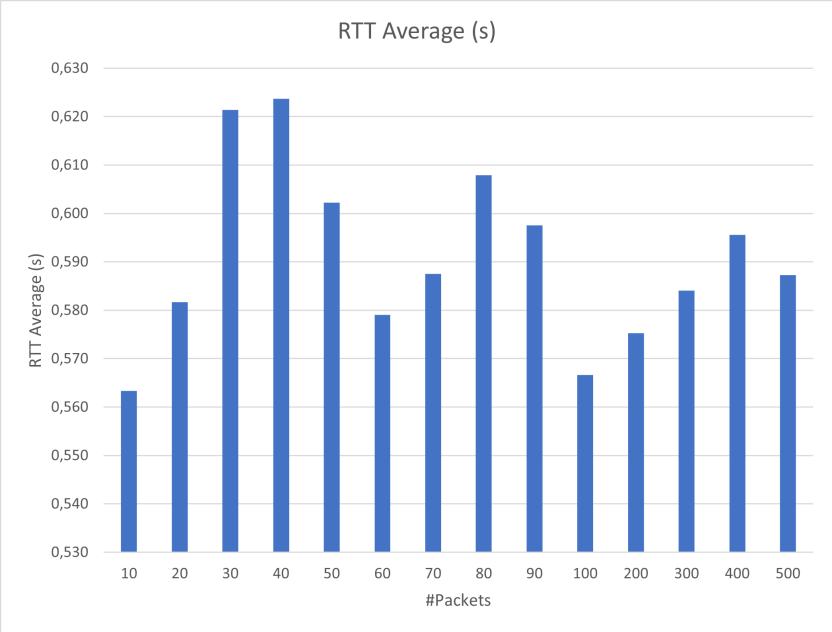

Figure 6.7: "BPP": RTT Average

Regarding the Network Delay, as it is possible to observe in the graphic below, the values are significantly higher than the "IP Options" solution: this depends on the fact that a bigger amount of bytes for each packet passes through the network.

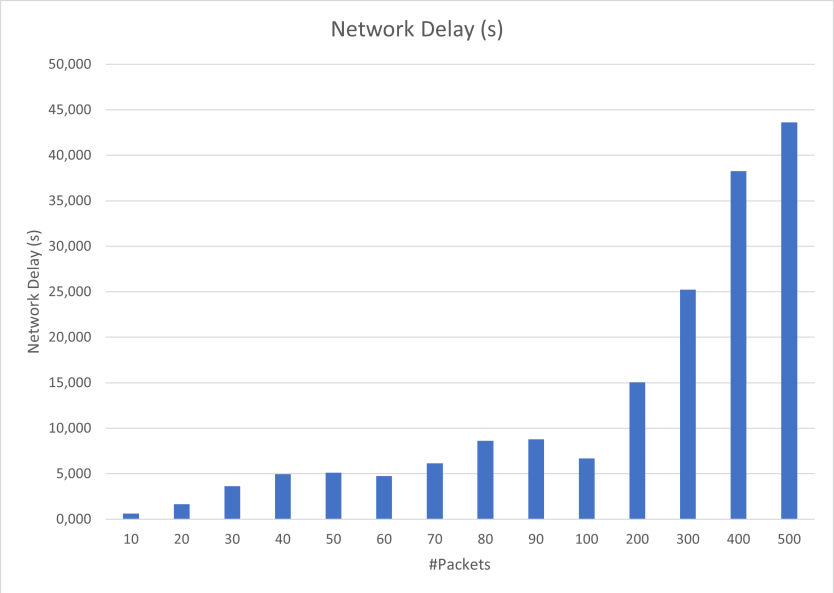

Figure 6.8: "BPP": Network Delay

The FCT values, as a result of RTT and Network Delay increase, become much higher because take into account all the delays and latency propagation inside the topology. It is clear that Network Delay has much more impact on the overall performance of the communication, as noticeable in the following graph:

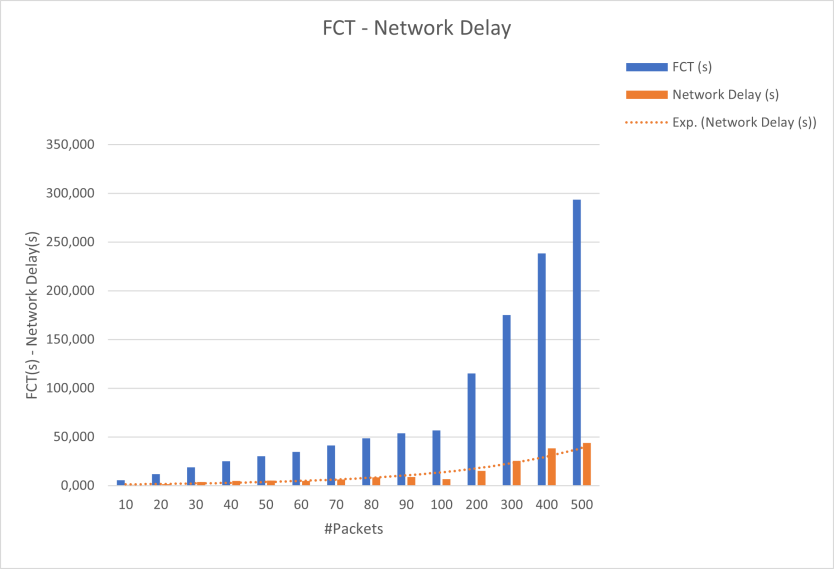

Figure 6.9: "BPP": FCT - Network Delay

6.2.3 Comparison between the results

Thanks to the above results it is possible to plot differences in terms of time obtained for the two solutions. With RTT specific, we consider a more granular measurement because each packet is taken into account, analysing deeply each delay during the transmission from the client to the server. With FCT instead, we focus on the entire communication, from the first packet sent to the last one received through the client socket.
Looking at the following tables, which summarize all the results achived during the test phase, some consideration must be made. For our simple use case it is clear that "IP Options" solution is better than the "BPP" one because of the time differences highlighted. We should consider that data integration on the IP Options is a common way to collect routing information during packets flow.
The reason why this solution is not widely used today can be researched in the effective dimension of the IP Options field. 32 bits are too few for most of the applications to have a significant improvement.
Moreover, network speed and bandwidth are progressively less critical points thanks to huge development of network technologies and infrastructures, so BPP packets can be affordable. These reasons have led to the BPP spread for those application in which is possible to trust on a reliable network communication and an higher amount of data in needed.
Since our final goal is to increase the capabilities of a newborn Machine Learning algorithm, "Owl", only few additional metrics can improve the train model and produce better decision for congested and critical networks. So the "IP Option" implementation is selected for collecting metrics in Mininet and we use them to feed "Owl" for next steps.
From the implementation point of view, once the packets are arrived, a new socket must be created in order to manage the connection between H2 and the final Server where Owl is installed. To do so, a new function called "SendToUbuntu()" is created. Every time a packet is sniffed in the IP Options project, the latter function is called and the packet reaches the destination. This cyclical behaviour is set thanks to the for loop structure, specifying the number of packet to sniff and then forward to Ubuntu 18.04.

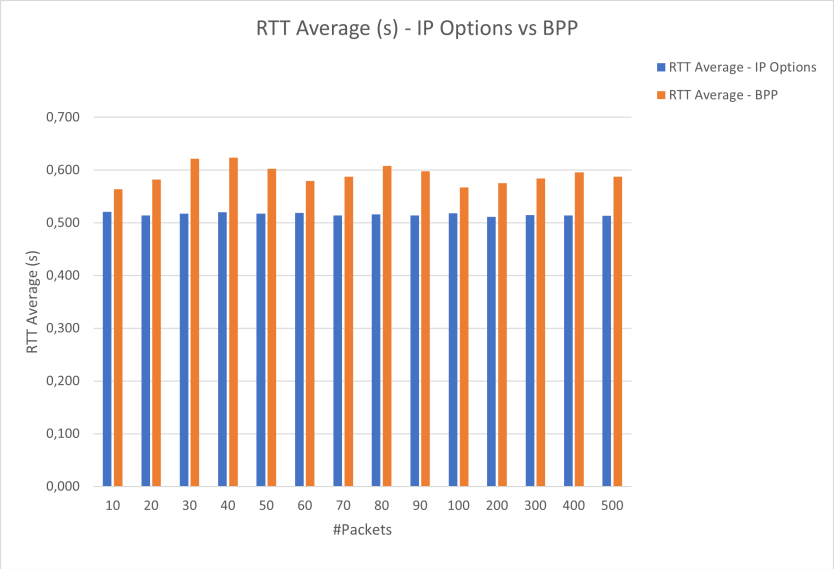

Figure 6.10: RTT Average (s): "IP Options" vs "BPP"

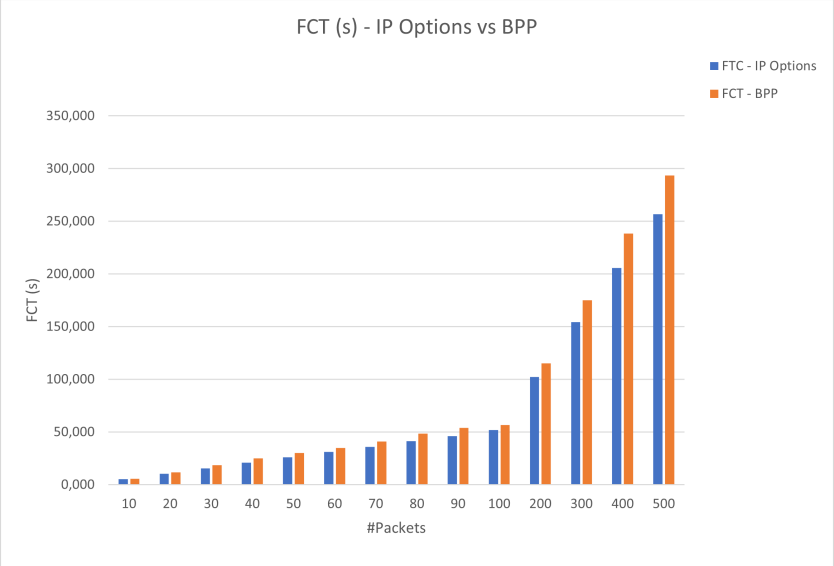

Figure 6.11: FCT (s): IP Options vs BPP

6.2.4 Additional overhead evaluation

Due to the technical constraints described on the System paragraph, the additional overhead due to the communication between H2 and the final Server is computed. This estimation is performed only for "IP Options" implementation because it is already demonstrated that it is the best solution.

The results in the following table shown that Owl will receive packets with the desired metrics with an acceptable delay. About 10 seconds delay for sending 500 packets with 0.5 seconds between two consecutive packets sent is in line with the expectations. It is worth noticing that the corrective action only implies a short additional path for each packet which starts from H2 and arrives on the final Server. So the transmission time for the following path for each packet is the total additional overhead.

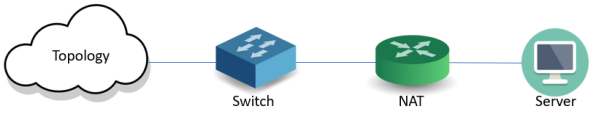

Figure 6.12: Additional path

IP Options		
FCT (s): H1 - H2	FCT (s): H1 - H2 - Server	Additional overhead (s)
5,206	5,208	0,002
10,278	10,514	0,236
15,516	15,716	0,200
20,814	21,028	0,214
25,867	26,967	1,100
31,103	32,278	1,175
35,952	37,105	1,153
41,297	42,041	0,744
46,250	47,712	1,462
51,823	51,981	0,158
102,225	106,954	4,729
154,283	158,207	3,924
205,586	212,783	7,197
256,602	266,528	9,926

Figure 6.13: Additional overhead overview

The following histogram shows that the NAT implementation does not slow down sensibly the communication and that performance remain good. It is noticeable that the solution is better than the BPP implementation without the corrective actions.

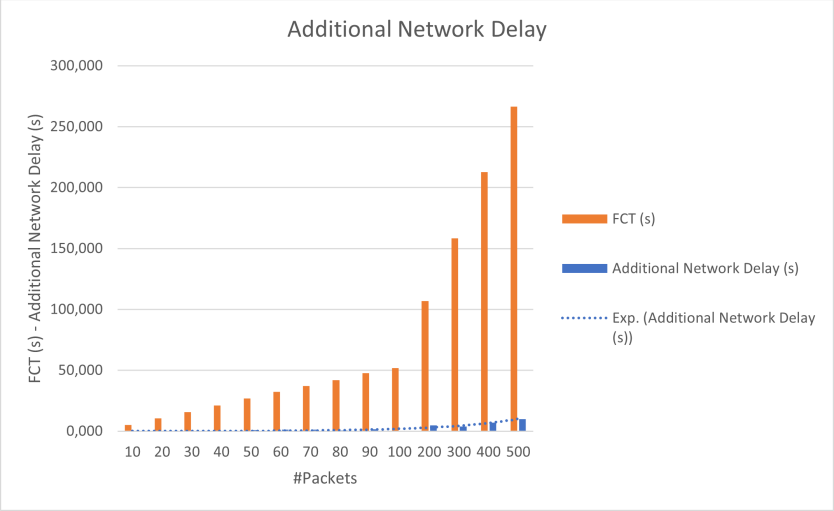

Figure 6.14: Additional Network Delay

6.2.5 Owl integration results

The last test performed is the correct integration of the metrics into Owl. First, the collection of data management is realized with the creation of a file on the already existing kernel function "_get_reward()" which computes several metrics like RTT, cwnd, throughput, packet delivered, packet lost.
Second, some consideration must be done in order to understand the following graphs. Starting from H1, traffic is sent through the system without any "sleep()" function between two consecutive sending, differently from the test of BPP and IP Options performance.
Giving and empirical estimation of the sending frequency, it is just the time spent by the client to read the while loop and perform the "send()" function. The problem with that is the amount of packets loss during the communication due to the complex system designed.
H1 just sends packets through the network, it does not wait for H2 response but it closes the connection as soon as it sends the number of packets specified in the command line. Due to the fact that H2 must receive each packet and forward them to the Server, it does not deal with each packet correctly in time. The same is valid for the Server process which must take every feedback coming from the socket, send

it to the train model and store the data into a file (open, write in append, close). This means that only a portion of the packets sent by H1 is correctly received from the final Server, where Owl runs.
The reason behind this choice is that, due to the fact that packets must cross two different Virtual Machines in the proposed solution, Owl is tested trying to lower end-to-end latency as much as possible. In this step it is not interesting that every packet arrives correctly, but that all the packets elaborated by the Server are correctly integrated in the reinforcement learning model.
Two different test are reported below: the first one consists in a 30 seconds run in which H1 sends multiple packets and Owl calculates all the reward parameters, and the second one, which is a 2 minutes long run. Here are reported only the RTT and throughput trend.

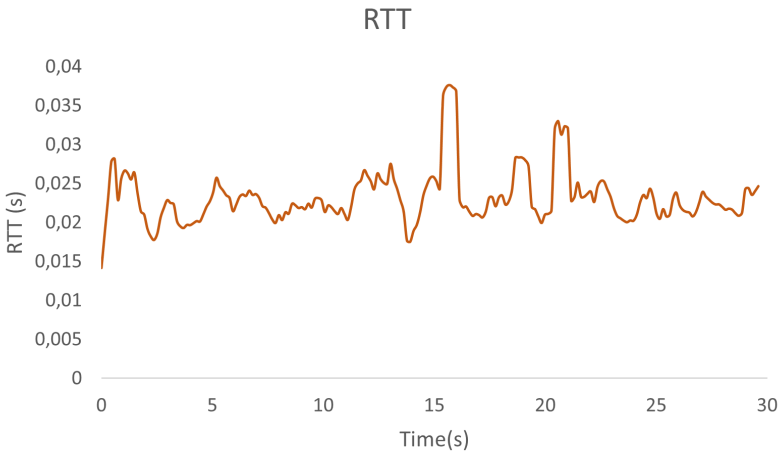

Figure 6.15: 30 seconds run: Owl RTT trend

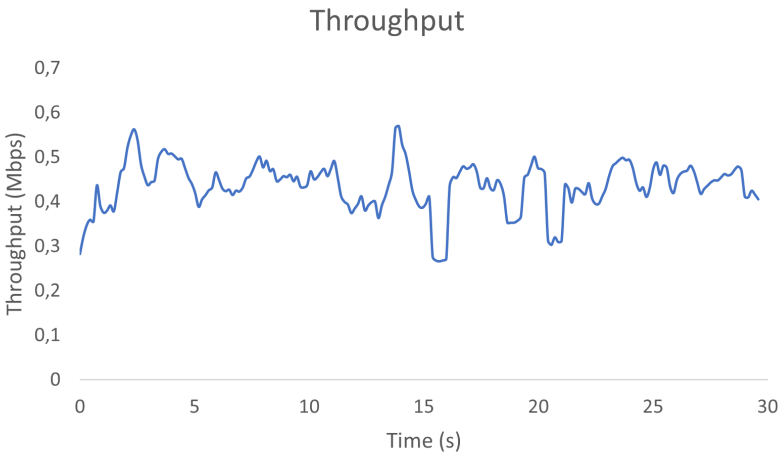

Figure 6.16: 30 seconds run: Owl Throughput trend

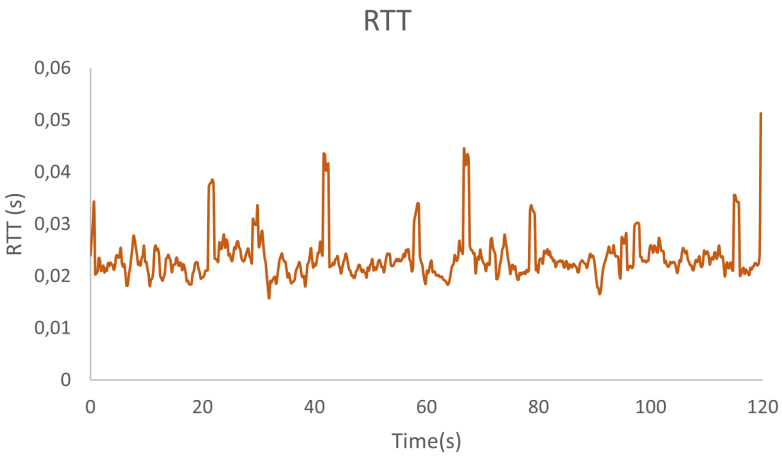

Figure 6.17: 2 minutes run: Owl RTT trend

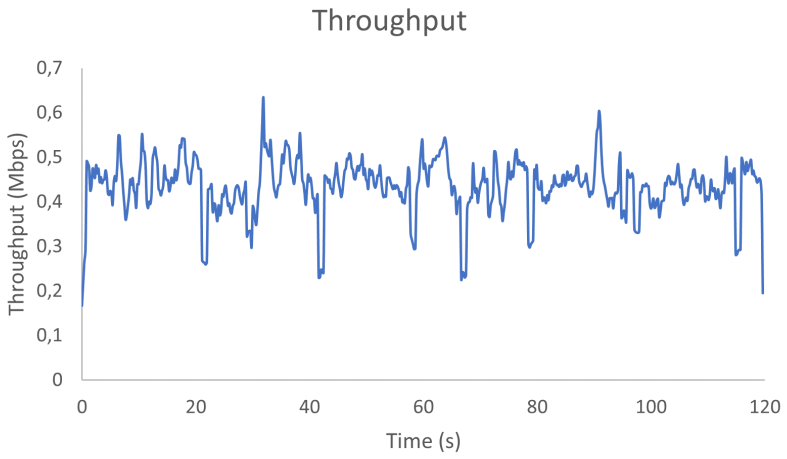

Figure 6.18: 2 minutes run: Owl Throughput trend

The above graphs are build plotting the obtained results wrote on the file created before, using the following formula:

$$throughput = cwnd/(rtt/1000) \tag{6.2}$$

Frome the file mentioned above, only RTT and Throughput are extracted, then, due to the loss of the packets phenomena previously described, not every packet is reported on the graph. In order to create a trend as loyal as possible to the real run, sampling intervals are computed under the assumption that packets receipt follows a continuous uniform distribution. Given the time frame of the simulation, based on the number of packets computed, the sampling time for the 30 seconds test is 0.148 seconds and 0.207 seconds for the 2 minutes test.

Chapter 7

Conclusion

This thesis has dealt with the implementation of a system which collects specific telemetry metrics representative of the network conditions in order to support TCP decisions. In particular, the main purpose of this work has been the final integration to the newborn congestion algorithm Owl, in order to provide additional data to the Machine Learning model.
A P4 topology has been here used in order to select the best way to carry these metrics though the network. The two different approaches analyzed are:

- IP Options field integration, which exploits the free space already present in the standard format of the IP packet;
- BPP protocol integration between the IP and TCP layers;

The first one is preferred for storing few metrics with minimum size because of the 32 bits size limitation of the Options filed.
The second fits better when the transmitted data have a larger size. It allows to carry more accurate measurements at the cost of a major bandwidth usage.
Thanks to the RTT and FCT parameters computation, the tests conducted have reported the supremacy of the first solution in terms of performance, network delay and resources' expenditure. This is in line with the expectation because the metrics involved do not require more space than the Options field's capacity, so there are no complications in the standard IP packet integration.
Such evaluations have been also done in terms of Network Delay calculations that have demonstrated how the first model is in this case more suitable for the final purposes of the work.
On the other hand, BPP results to be a very good choice to customize network behavior carrying commands and metadata and also because of its extensiveness and flexibility.
The modular format of BPP packets is a good compromise between compactness

and semantics expressiveness, so it was worth trying to test this novel approach over P4. This has been useful even if P4 language has demonstrated some lack in supporting dynamic substructures an and variable length fields in related work.
Exploiting P4 Language capabilities, it has been here demonstrated how it is possible to program switches through information contained in the packets themselves.
Having the full control of the network, P4 has been very useful to collect an heterogeneous set of metrics feeding the target algorithm.
After that, the main objective has been the combination of the collected metrics with the Reinforcement Learning model, increasing the number of parameters processed and then improving its decision making ability.
The tests performed in the Owl behavior have confirmed the correct integration of the metrics.
Launching the algorithms for long time intervals has shown the stability of the solution, demonstrating coherence in the values of throughput and RTT observed for each sampling time. This result indicated that Owl has the predisposition to increase the quality of its congestion decision modifying the Machine Learning model.

7.1 Future Work

In the lights of the results achieved, some future steps can be made in order to increase the quality of Owl decision making.
This can be achieved in different ways:

- Performing different tests in order to understand if the metrics collected lead to better decision making. In other words, this means the creation of different network conditions, like congested networks, different topologies, different networks devices involved, and analyze all the parameters used by the algorithm to make the decision.
- Creating more complex metrics by means of different structure combinations. This can be easily performed exploiting P4 potential, not only using standard_metadata variables but also custom constructs which can be optimized in terms of efficiency and memory demand.

Bibliography

[1] URL: https://opennetworking.org/sdn-definition/ (cit. on p. 1).

[2] Nandita Dukkipati. «Why Flow-Completion Time is the Right Metric for Congestion Control». In: () (cit. on pp. 6, 51).

[3] B. Lantz, B. Heller, and N. McKeown. «A Network in a Laptop: Rapid Prototyping for Software-defined Networks». In: () (cit. on p. 6).

[4] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Avait Dixit, and Lawrence J Wobker. «In-band Network Telemetry via Programmable Dataplanes». In: () (cit. on p. 6).

[5] Ya Gao and Zhenling Wang. «A Review of P4 Programmable Data Planes for Network Security». In: () (cit. on p. 7).

[6] H. Mohajeri Moghaddam and A. Mosenia. «Anonymizing masses: practical light-weight anonymity at the network level». In: () (cit. on p. 8).

[7] Jérôme François, Alexander Clemm, Vivien Maintenant, and Sébastien Tabor. «BPP over P4: Exploring Frontiers and Limits in Programmable Packet Processing». In: () (cit. on pp. 10, 21).

[8] Pat Bosshart et al. «P4: Programming Protocol-Independent Packet Processors». In: () (cit. on p. 16).

[9] URL: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html (cit. on pp. 16, 29).

[10] The P4 Language Consortium. «P416 Language Specification version 1.2.1». In: () (cit. on p. 19).

[11] R. Li, A. Clemm, U. Chunduri, L. Dong, and K. Makhijani. «A new framework and protocol for future networking applications». In: () (cit. on p. 21).

[12] M. Allman, V. Paxson, and E. Blanton. «TCP Congestion Control». In: () (cit. on p. 24).

[13] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. «Pcc: Rearchitecting congestion control for consistent high performance». In: () (cit. on p. 25).

[14] V. Arun and H. Balakrishnan. «Copa: Practical delay-based congestion control for the internet». In: () (cit. on p. 25).

[15] Alessio Sacco, Matteo Flocco, Flavio Esposito, and Guido Marchetto. «Owl: Congestion Control with Partially Invisible Networks via Reinforcement Learning». In: () (cit. on p. 25).

[16] V. Mnih, K. Kavukcuoglu, D. Silver, and A. A. Rusu et al. «Human-level control through deep reinforcement learning». In: (cit. on p. 26).

[17] URL: `https://nsg.ee.ethz.ch/home/` (cit. on p. 35).

[18] URL: `https://github.com/nsg-ethz/p4-learning` (cit. on p. 35).

[19] URL: `https://scapy.readthedocs.io/en/latest/usage.html` (cit. on pp. 38, 39).

[20] URL: `https://docs.python.org/3/library/time.html` (cit. on p. 54).