POLITECNICO DI TORINO

Master's Degree in COMPUTER ENGINEERING

Master's Degree Thesis

High performance eBPF probe for Alternate Marking performance monitoring

Supervisors:

Prof. Riccardo SISTO

Prof. Guido MARCHETTO

Candidate:

Marino URSO

Academic Year 2019-2020

Table of Contents

Chapter 1

Introduction

In the last years, Service Providers have faced an increment of traffic in their networks due to the widespread use of the Internet and the birth of new services requiring a large amount of packet exchanged such as video streaming or live.
These contents, especially live video and audio but also online gaming and multimedia streaming, need to limit the amount of packet loss and packets delayed. For this reason ISP are obliged to adopt technniques and methodologies to constantly monitor these values (mainly packet loss, delay and jitter) quite accurately, so that they can control User Experience of their customers. These techniques can be useful also to manage the network itself, helping to locate problems and isolating them.

One of these techniques has been proposed by TIM in RFC 8321 [1] and its title is "Alternate-Marking Method for Passive and Hybrid Performance Monitoring". As specified in the name, it is a Passive or Hybrid method (RFC 7799 [2] specifies Passive and Hybrid Methods of Measurement) : it means that it works without altering the packets in transit but only observing them (Passive), or modifying existing field values of the packets (e.g. Differentiated Services Code Point DSCP) than are not used (Hybrid); if the field is dedicated and included in the protocol, the method is considered Passive.
The technique is intended to work with any kind of traffic and the main component is the packet loss measurement, but it can measure delay as well, both one-way

and two-way.

This method is interesting mainly for the following reasons:

- high-precision packet loss measurement: single packet loss can be detected;
- useful for every kind of traffic, packet or frame-based: Ethernet, IP, MPLS, etc., because it does not need modification to existing protocols;
- easy to implement: it can be run exploiting features present on every routing device;
- highly robust: it can manage out-of-order packets and those which follow different routes, i.e. multipoint paths.

To reach the goal of monitoring the network performance, some probes should be installed around the network, especially on the border routers, which must have a probe collecting data about the packets coming from the outside of the network or the ones leaving it.
This means that monitored network must be surrounded by devices able to analyze traffic passively, otherwise not all packets can be intercepted and measurement would be less accurate.
If only border routers are equipped with probes the measurement would be the coarsest possible while, if some data are collected inside the network, finer measurement are possible depending on how many probes and how distributed across the topology.

Other studies after RFC 8321 provided better ways to process data collected by routers. The Big Data approach [3] exploits the power of post processing to analyze fastly the great amount of data collected by backbone routers in a Service Provider backbone.
The basic mechanism that allows this approach is the packet sampling with hashing techniques; through this method, every node in the network can recognize uniquely each packet and compare timestamps in a different place (the big data system) and in a different time (thanks to post-processing).

1.1 Chapters content

The thesis is organized in eight main chapters.
In this section i will describe briefly how chapters are organized and what they contains.

This document can be split in two parts: the first part illustrates the background technologies and methodologies useful to well understand the work done, as it is mainly theoretical; the second part is about the work done, development and organization, together with testing and validation phase. It describes the main parts of the software developed and illustrates results obtained.

Chapters are organized as follows:

- Chapter 2 offers an overview of the thesis work carried out, illustrating the problem this research activity was inspired by and the goal we want to reach.
- Chapter 3: this chapter describes some of the performance monitoring technologies that are the background of this work, with particular attention to RFC 8321 [1] and multipoint monitoring [4].
- Chapter 4 provides a general description of the Big Data method, introduced as support and improvement of the performance monitoring where each device analyzes a great amount of data, consuming CPU resources that can be saved.
- Chapter 5: in this chapter it is illustrated the theory about BPF and the evolution calleld eBPF (extended BPF). The content varies from the internal components of BPF to the overview of the different methods to write BPF code.
- Chapter 6 describes the probe developed during this thesis work. The description starts from an overview of the probe's architecture with the list of all modules, then an illustration of the workflow of the probe is provided and finally an in depth analysis of the code and techniques used to solve some problems is given.

- Chapter 7: in this chapter we face the final phase of the thesis work, that is the development of a testing system and the execution of the test itself. This phase is useful to understand if the code developed meets the project requirements, if it can be enhanced and what are the parts that negatively affect the performance, if there are;
- Chapter 8 illustrates the overall results of the testing phase and give ideas for future works on the developed software. This is an important chapter because it analyzes all the faced problems and all the solutions that have not been implemented due to various issues.

Chapter 2

Problem

2.1 Probe development

Alternate-Marking method is intended to be used within Service Provider backbone networks, where all the customer traffic transits.
By means of the method it is possible to analyze if some users experience problems during the navigation. These problems can vary from difficulties in reaching the destination, to the speed of the service, due to delay in the round-trip-time (RTT) caused by network overload.

With this technique combined with the Big Data approach (described in next chapters), Service Providers have the possibility to monitor the condition of the network, discovering if there is a fault in some nodes and locating it as fast as possible to overcome possible persistent problems.
Backbone links require a large amount of bandwith because every customer packet directed to the internet passes through them. For this reason, routers are required to catch and analyze in a real-time way every single packet and this implies high performance packet collectors installed on them.
The main goals of these collectors are the counting of packets, their sampling based on hash techiques, generation of timestamps for each sampled packets and the main fields storage of sampled packets in a local file with a predefined structure needed

by the Big Data system to process data.

Actually routers have the capacity to catch packet and count them by default, but they can't use more complicated functions such as sampling or generating timestamps for each packet. For this reason the necessity of a dedicated software was born. The idea was to fullfill this requirement writing an eBPF program that acts as a network probe and does all the tasks that we need.

eBPF was chosen because user space programs can not analyze line-speed packets due to the frequent context-switching between kernel and user space, so it would reach low performance. eBPF thanks to its virtual machine can inject code directly in kernel space without having to create a custom version of the Operating System, since BPF virtual machine is included in Linux.

2.2 Goal of the thesis

The main task to face is the development of a software capable of collecting packets information at high speed so that all traffic can be taken in consideration avoiding errors in measurements in case packets are discarded by the probe.
A previous thesis student developed a basic version of the probe [5] that could be a starting point for this work. The cited version was not meant for the monitoring with Big Data approach, which is necessary to integrate it with the new technology developed in "Big data post-processing of multipoint measurements with alternate marking method" thesis [6].
The software is required to be an executable with only one working mode: it should be able to capture all the marked traffic without flow distinction, discarding traffic generated by backbone routers identified by a subnetwork (x.y.z.w/N).

Measurement done by the probe should be of two types:

- Aggregate: referring to the whole traffic, e.g. total number of packets, interface name, number of bits matching the hash, etc.;

- Punctual: measures referring to each single hashed packet; source and destination address, source and destination port, protocol, timestamp, etc.

Software must be configured remotely and for this reason it should be provided with a RESTful interface accessible from the outside. From this interface it is possible to start/stop the probe, send configuration commands and parameters, including:

- Interface to monitor: we need a way to decide which interfaces must capture the traffic, including virtual interfaces because emulated network used by the previous thesis student [6] for the Big Data approach makes use of Mininet, a software that creates virtual devices with virtual interfaces;
- Single period duration: probe must know how long the marked flow lasts to syncronize the change of marking color and the results log;
- Number of test periods: when test ends, software stops running and releases all resources. It should be also possible to have an unlimited number of periods when the aim is not to test the application but the real execution;
- Starting value of marking color is important to know how to recognize traffic as soon as the probe starts its work;
- Reference hash is needed to compare hash value of single packets and check if they match with a given number of bits;
- Number of bits to match: previous point asserts that the packet hash must match the reference one starting from a given number of bits; this is the starting value;
- Type of Layer 4 protocol: UDP or TCP can be chosen in configuration phase. If none is indicated, both are captured.

Once terminated the capture of a single period, the software should write in some log files (one per interface) all punctual measures concerning sampled packets, so that Big Data system is able to understand and process the information.

The last step of the thesis is testing, it means that the correct operation of the software must be validated. This phase is useful both to check if there are bugs in the software and to evaluate the performance reachable when under stressful conditions.
The aim is to recreate a real network portion with at least two Measurement Points (MPs) and generate traffic up to 10Gbps passing through the devices with the software installed on them.

All output data, statistics and information coming from the different configurations of the testing phase will be compared to well understand if software performance meets TIM's requirements.
If not, a whole analysis of results will be carried out trying to realize the motivations that could lead to a slowdown.

Chapter 3

Packet Network Performance Monitoring (PNPM)

In this chapter, Alternate-Marking method will be described to better understand the mechanism used to monitor the performance of the network and to be more familiar with the terms used in this document.

This method is also called Packet Network Performance Monitorng (PNPM) and has been engineered in Telecom Italia (TIM); starting from 2008 TIM has tried to standardize this method with a document published in January 2018 and named RFC 8321 [1], which led to the publication of other drafts improving and refining the method.

3.1 PNPM

PNPM is a technique used to monitor the network and particularly to check if there are packet losses during the customer traffic transit in the network, on the links or inside a device, verify if the delay (one-way or two-way) is under a predefined

threshold and compute some values such as delay variation (jitter), etc.

The solution is adopted mainly to achieve the measurement of the different kind of delay, because the packet loss can be detected also with traditional methods, with the routers that count packets and then comparing the values between subsequent or further devices to know if in the middle some packet loss occurs (the value of the counter is different in the case packet losses are present, it is the same otherwise). The easiest way to retrieve these values for each packet would be exploiting a sequence number to track the route of each packet and saving the timestamp when it arrives in each device that takes part of the measurement. In this way it is enough to compare the timestamps of every packet to obtain the delay and compute the average, but there is another advantage: it is possible to detect which packet is missing in a sequence.

This way, though very simple, is very difficult to implement beacuse it requires the modification of each packet to insert a sequence number inside it and each router should be able to extract it as fast as possible. This is nearly impossible if all kind of traffic is considered; for example in UDP the sequence number is not available and in the case sequence number is inside a higher level protocol, the estraction is more difficult and time consuming, so in cannot be done in real time or the device would consume a lot of resources.

3.1.1 Working principle of Alternate-Marking

Counting packets and comparing values is a simple method as said before, but there should be a way to understand when to read the counter as packet flow is continuous and the mesurement must refer to the same set of packets.
A solution is to split the traffic in groups. But how to signal the start and the end of a block?
Sending a special packet to determine the end of a group and the beginning of another one is an alternative, but it is subject to the reception of out-of-order packets that could led to the detection of a false packet loss or no packet loss

when it is present. Another disadvantage is the generation of additional traffic that requires the devices to be able to understand and process these different packets.

PNPM uses this approach (splitting the traffic in blocks) but it does not use additional packets; it "marks" the packets belonging to the same block with the same "color", while for the packets belonging to a different block it uses another mark. This is the reason why the method is called Alternate-Marking, because it has been demonstrated that only two "colors" (let us refer to them as A and B) are enough to split the traffic as showed in Figure 3.1.

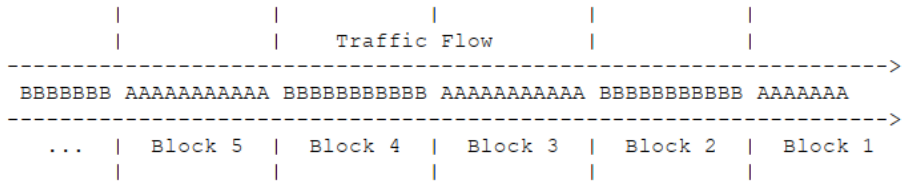


Figure 3.1: Alternate-Marking [1]

3.1.2 Packet loss detection

Each network device counts the number of packets belonging to each block unambiguously split by the Alternate-Marking. Comparing the values recorded by each router inside the network, it is possible to establish if a packet loss occurred and where it was. This method is very flexible because it allows to detect losses in every segment of the path, in a device or on a physical link as illustrated in Figure 3.2. To check if some packets are lost inside a device (e.g. in the device queues) it is enough to compare the counters on the input interfaces and the output ones; if the sum of the input counters is different from the sum of the output then a loss occurred in the device. For what concerns the losses on the links, a similar method is used but considering the interface at the edges of the link instead of two interfaces belonging to the same device.

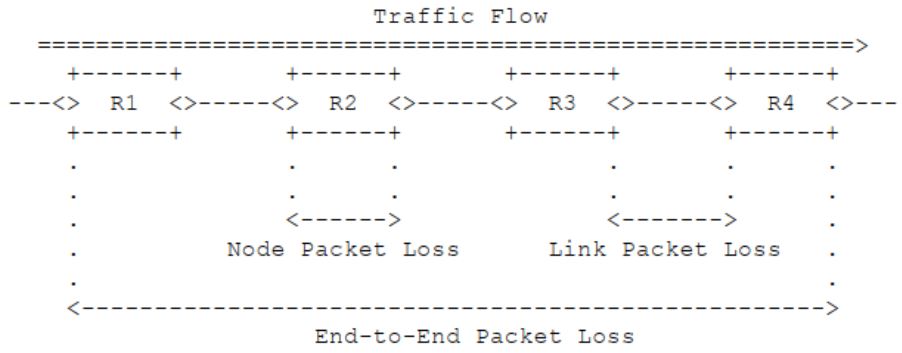


Figure 3.2: Different measurement [1]

There are two criterion to choose the dimension of the blocks (periods):

- based on a fixed number of packets: every counter reads the same number for each block, except when losses occur;
- based on a fixed amount of time: color is switched when a timer expires, so the number of packets in each block is highly variable, depending on the traffic passing the network in a specific moment.

3.1.3 Timing aspects

Using a fixed timer can be better because it allows to choose a certain interval of time large enough to count the packets and to compare the counters of different devices. Moreover it is not recommended to stop the counter when the timer expires, in fact there could be packets out-of-order that arrive during the flow of the following color, so a period of L/2 (where L is the block duration) is waited to read the counter, so avoiding errors due to the reading of a running counter. The choice of the time value should not be too low nor too high (in the last case the measurement cannot be taken very often).
Another way to retrieve a still counter would be reading it with a certain frequency periodically and check if the value does not change over the time (i.e. the counter stops incrementing); this is the signal that the current block has ended. This is not the preferred method because it is not safe and the counter can stop even

if no packets arrive in the probe for a certain amount of time; for this reason is preferable the first one as showed in the Figure 3.3.

Often the network device clocks are not accurate and two different clocks can differ by an error. This error should be less than L/2 so that each counter can be read with the right value and every packet is assigned to the right block by each router. Not only the clock errors should be taken in consideration, but also the distance between network devices that can determine a delay in the propagation and it can produce out of order packets: we can consider an upper bound called D_max and a lower bound D_min to include also the network delay. Let's bound the difference between the clocks in the network by a value A (assumed that the clock accuracy is +/- A/2).

The minimum time range that we have to wait to obtain the right measurement is given by:

$$d = A + D\_max - D\_min$$

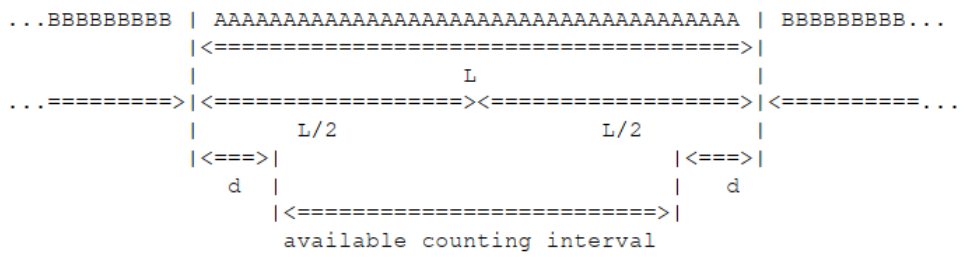


Figure 3.3: When to read the counter [1]

The requirement is that d < L/2, so the block period need to be L > 2d.

3.1.4 Counting packets

Traffic colouring can be done by the edge routers when the packets come into the network. This is the best case considering an ISP network that have to control all the traffic inside the backbone.
Each interface of the routers needs to have at least four counters: one for the

incoming packets (hereinafter they will be referred with the suffix _IN), one for the outgoing packets (suffix _OUT), both duplicate for the A and the B color.

3.1.5 One-way delay

As for the measurement of the packet loss, one-way delay can be computed with the same methodology. There are different alternatives, illustrated in the following paragraphs.

Single-Marking method One method consists in exploiting the synchronization given by the change of colors to calculate the delay. in practice, a router stores the timestamp of the first packet received of the new block; this timestamp, compared to the timestamp of the same packet (the first one received) stored in the following routers, can give the delay of the packet to traverse that portion of network by subtracting them.

For example, router R1 stores the timestamp of the first packet of the block with color A; let's call this timestamp TS(A1)R1. Router R2 does the same with the first packet marked with A: TS(A1)R2. We obtain that $delay = TS(A1)R2 - TS(A1)R1$ where *delay* is the delay associated to that packet. If we want to measure the delay of more packets it is enough to store timestamps of different ones, for example every N packets received and, in the borderline case, a router can store the timestamp of every packet. This method is sensitive to packet loss and to out-of-order reception because the router cannot know if the sampled packets are the same of the previous router, for example if a packet is lost in the path between R1 and R2 it will never be catched by R2.

Mean delay The problem of the sensitivity to packet loss and out-of-order reception can be avoided with a new method of measurement: the mean delay. Routers collect the timestamp of packets and then compute a mean dividing by the number of packets; in this way the error introduced by the loss of a packet or the different reception order is very small; we obtain more robustness to packet loss and we save disk space because the number of timestamp to send to the NMS is greatly

reduced.

This method does not have only advantages; in fact the measurement is only one for the entire block even if the duration of the block is high (e.g. 5 minutes). Moreover the maximum and the minumum delay cannot be computed with just one value.

Double-Marking method The limitation of mean delay is due to the fact that it does not give information about the delay's distribution during the block duration in addition to the missing of maximim and minimum delay computation. A new approach has been introduced to overcome these issues: its name is Double-Marking methodology.
It consists of the same method as the Single-Marking to color the flow, but within a single block it adds another marking to select the packets that will be analyzed by the routers in order to measure delay/jitter. The first marking mantains the role of splitting the traffic in blocks so that the packet loss and mean delay can be computed. The second marking instead is used to select a new set of packets so that routers can store the timestamp of these packets only and compare it to the timestamp of the same packets stored by other devices.

The second marking should not be changed too frequently to avoid out of order problems, but between them there should be a time interval greater than the mean delay computed with the previous method. If one of these packet is lost, the delay cannot be computed for that block because the measurement is corrupted.

3.2 Multipoint Alternate Marking

The Alternate Marking as described so far is only applicable to point-to-point unicast flow, because it assumes that all packets catched by a router pass from a single following node. In real networks this is not correct; a packet can follow different directions due to routing mechanism, Equal Cost Multipath Routing (ECMP), etc.

The main flows we can find observing any network are the following: point-to-point single path, point-to-point multipath, point-to-multipoint, multipoint-to-point and multipoint-to-multipoint as reported in the following figures:

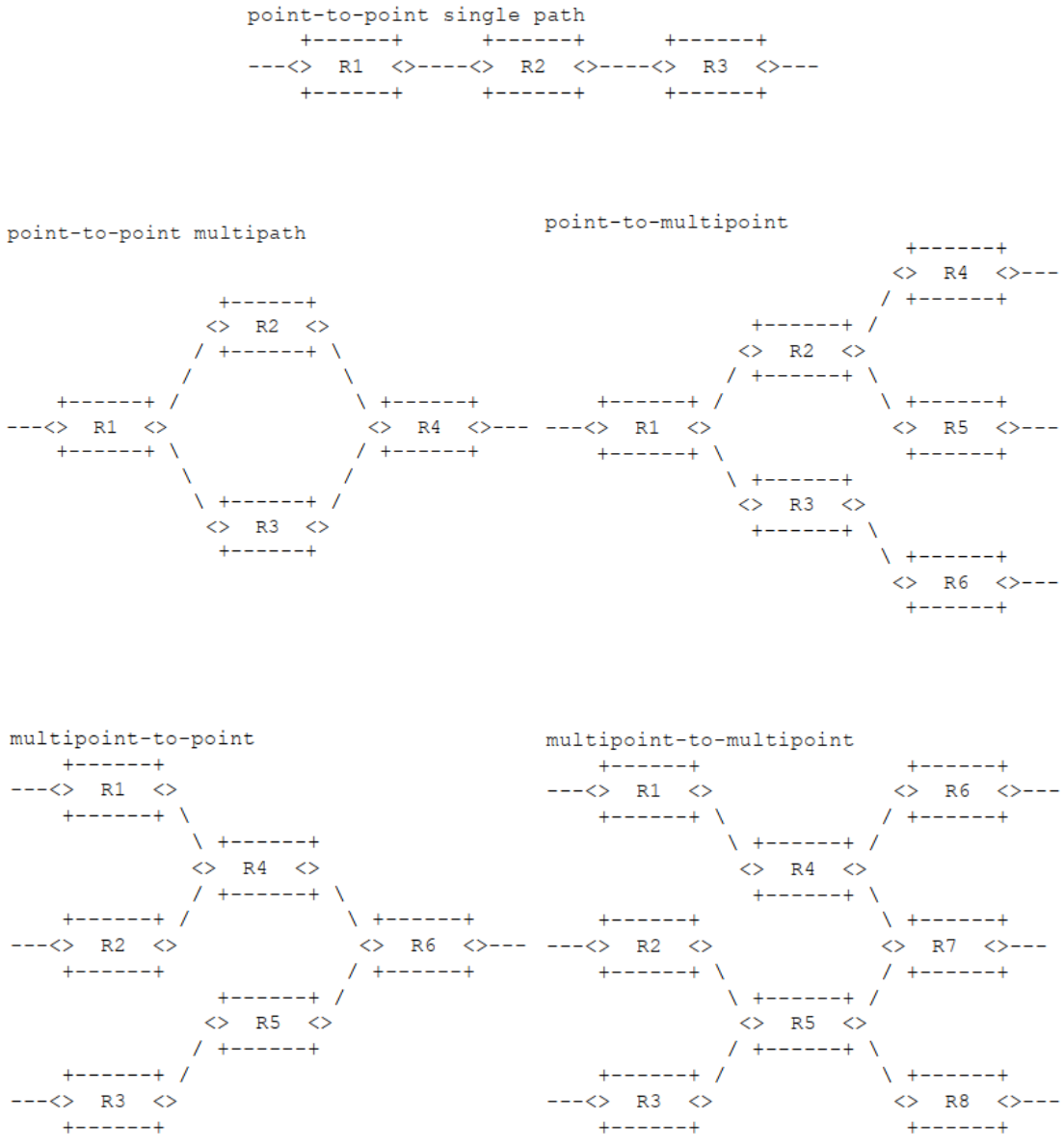


Figure 3.4: Flow classification [4]

The Multipoint Alternate Marking [4] mechanism covers the most general case of Multipoint to Multipoint path, and with the network clustering approach is possible to guarantee a high level of granularity and a flexible implementation of the performance measurement.
This is due mainly to the splitting of the network in clusters that can be treated as autonomuous entities and the measurement can be done inside each of them; differently, the classic Alternate Marking permits to analyze the traffic only inside the entire network or per single flow.

A Cluster is the smallest subnetwork guaranteeing the condition that the number of incoming packets is equal (or greater than, in the case of packet loss) to the number of outgoing packets.

3.2.1 Packet loss detection

The Measurement Points (MPs) inside the clusters should be at least the edge routers, i.e. routers connected with other devices not belonging to the cluster itself. There is an analogy with the Alternate Marking, where the mandatory Measurement Points are the border routers connected with a different network outside the Monitored Network (usually the first routers that handle customer traffic).

Since all packets that leave the cluster have entered the cluster from an input node, the number of packets counted by the input nodes must be always greater or equal than the number of packets counted by the output nodes.
In brief, if a Monitored Network has n input nodes and m output nodes, the Packet Loss is given by:

$$PL = (PI1 + PI2 + ... + PIn) - (PO1 + PO2 + ... + POm)$$

where:

- PL is the Packet Loss;
- PIi is the number of incoming packets in node i;

- POj is the number of outgoing packets from node j.

3.2.2 Clustering

When considering clusters, the entire network is included because it is the biggest cluster in a Monitored Network. But, what is technically a cluster?
A cluster is a subnetwork obtained from the Monitored Network Graph that still satisfies the Packet Loss equation as introduced at the end of the previous section. All the nodes of the graph are represented by the Measurement Points and the arcs are the links that connect the MP to each others.

Algorithm for Cluster partition The simplest algorithm to partiton the network in the smallest clusters possible is defined in multipoint draft [4] and it can be summarized as follows.

It is composed of two steps:

- group all the links that share the same starting node;
- join the groups that share at least one ending node.

Let's consider that the links are unidirectional, the creation of clusters can be done for both direction separately.
The first step, in practice, is executed listing all the links (with the name of the 2 nodes that they connect; e.g. R1-R2) and grouping the links that have the same node as starting point.
For what concerns the second step, it is similar to the first one but it should be checked if two or more groups have one or more ending nodes in common and join them.
Each of the joined groups composes now a cluster.

Here an example of Monitored Network:

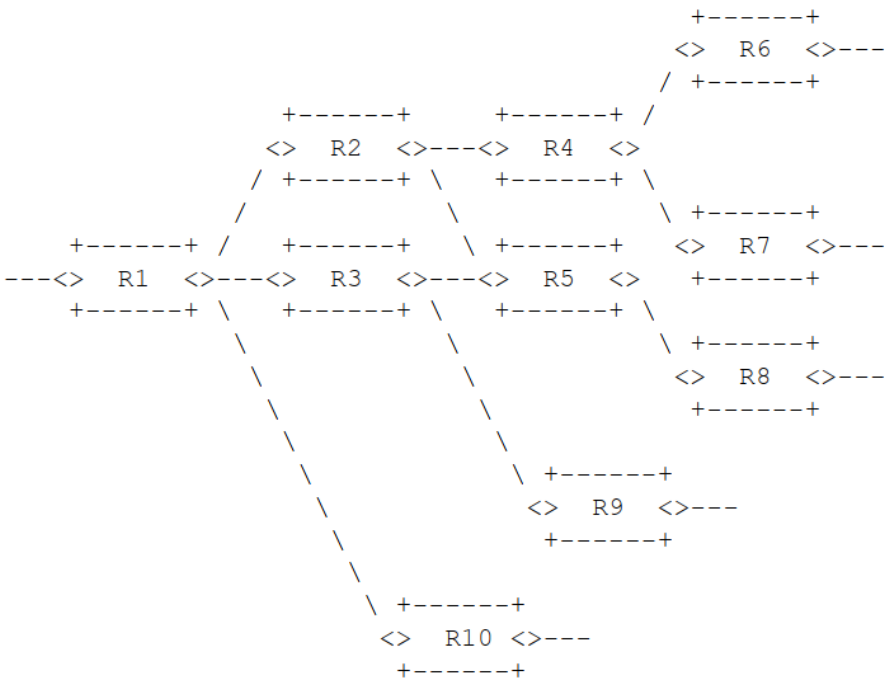


Figure 3.5: Monitored Network graph [4]

If the graph of the Monitored Network looks like Figure 3.5, with the first step we can identify the following groups:

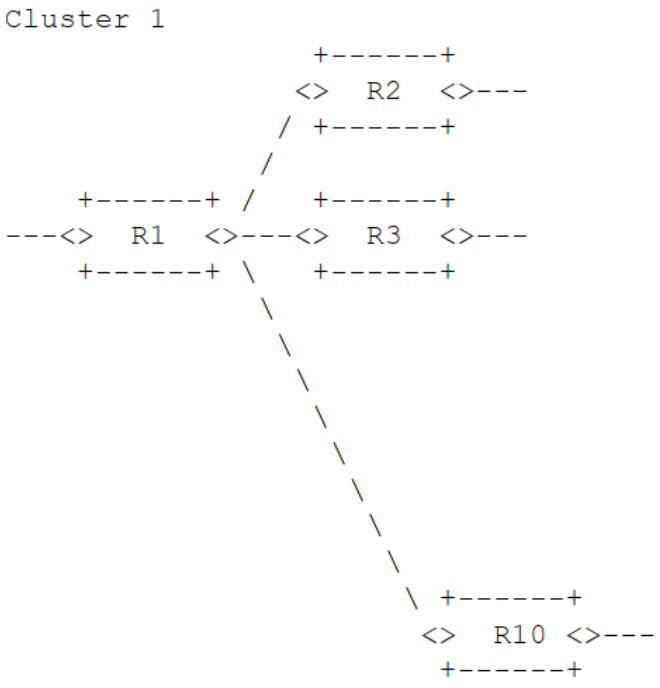
- Group 1: (R1-R2), (R1-R3), (R1-R10)
- Group 2: (R2-R4), (R2-R5)
- Group 3: (R3-R5), (R3-R9)
- Group 4: (R4-R6), (R4-R7)
- Group 5: (R5-R8)

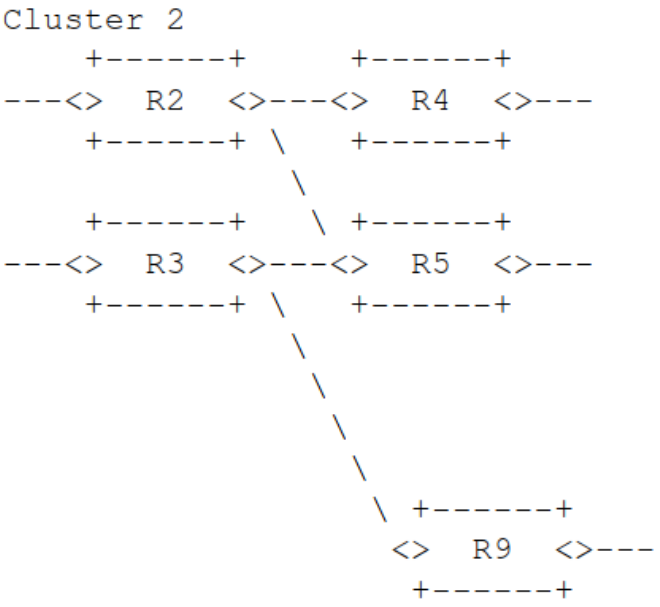
Then, with the second step, let's join the groups with an ending node in common:

- Cluster 1: (R1-R2), (R1-R3), (R1-R10)
- Cluster 2: (R2-R4), (R2-R5), (R3-R5), (R3-R9)

- Cluster 3: (R4-R6), (R4-R7)
- Cluster 4: (R5-R8)

This clusters can be graphically represented by means of the following figures:





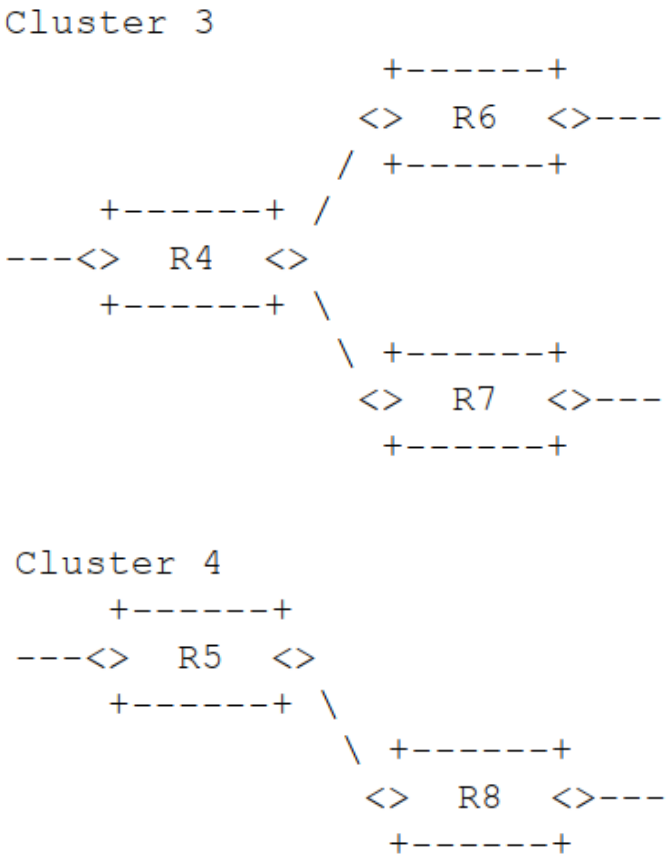


Figure 3.6: Obtained clusters [4]

There is a two-nodes cluster (Cluster 4) where the possible detected losses are on the link connecting the two nodes, while in the multi-nodes clusters we cannot know exactly in which link the losses occurs.
In addition to the analysis on Cluster basis, if two or more clusters are combined in a new subnetwork (called Super Cluster) the equation seen for the packet loss is still true.

3.2.3 Delay measurement

Delay and delay variation (jitter) are important measures in a multipoint scenario; the jitter is calulated using the same packets selected for the delay measurement.

The best way is to sample packets to avoid analyzing all the traffic passing through

the network, as the Double-Marking methodology proposes. But single and double marking are not representative of the whole flow because single marking takes only one measurement for each block and in double marking the packets cannot be recognized in a multipoint-to-multipoint scenario because they follow different paths and reach different network devices.
For this reason the Multipoint Alternate Marking exploits a different way to achieve the same advantages of double marking but in a way where it is possible to recognize packets and link the measures between different nodes. We are talking about the hashing method.
The first documents introducing the Hash method are RFC 5474 [7] and RFC 5475 [8], and "Compact Alternate Marking Methods for Passive and Hybrid Performance Monitoring" [9] explains how to combine it to Alternate Marking method. The coupling between hashing and marking method helps recognizing uniquely the packets because the latter analyzes the packets selected with the hashing technique and this semplifies the matching of the hashed packet all along the network.

There are two types of hashing: simple hashing or dynamic hashing. The simple hashing consists of fixing a certain number of bits of the packet's hash value that must match a reference value. This led to an extremely variable number of packets hashed during a period, depending on the amount of traffic.

The dynamic hashing has been introduced to overcome this problem. The mechanism is the following: a starting number of bits that must match the hash is given (typically a few bits, to sample a high percentage of the traffic), together with the maximum number of packets to be caught in a marking period (NMAX). The length of the hash is dynamically adapted to the amount of traffic: when NMAX packets are sampled, the hash length is increased by 1 bit so that only the half of those packets (more or less NMAX/2) still match the hash and every time the hashed samples reach again NMAX, the process is the same. The dynamic hashing method converges to a number of selected packets that is between NMAX/2 and NMAX, bounding the quantity of stored data.

All the MPs send to the management system the list of packets sampled with the hash value and the timestamp. The hash value is used to couple the packets captured in the input MPs with the output ones and the timestamps are needed to compute the delay.
Differently from the double-marking, this method works also with if packet loss occurs because if some hashed packets are lost, the correspondent hash is not present in the measurement points that didn't receive those packets, but all others are correctly coupled thanks to the hash value.

Chapter 4

Big Data

4.1 The Big Data approach

4.1.1 Introduction

In a real big network, the Management System has a huge amount of data to process. The Big Data approach [3] is meant for the performance measurement based on a posteriori calculation and it is inspired by the Alternate Marking Method [1], Multipoint Alternate Marking Method [4] [10] and Hash Sampling [7] [8]. The method is called Big Data Multipoint Alternate Marking performance measurement.

This method can perform two types of measurement: per cluster and end-to-end. The per cluster approach is needed to obtain a list of values that characterize the performance of each single cluster; the end-to-end approach aims to collect measurement giving information about the entire path.

The results are computed in a non real-time scenario and for a single marking period. The method is based on packet sampling methodology, applied to all the incoming traffic without flow distinction. In fact, the Big Data infrastructure deals with the splitting of the data in flows, after having collected the identifying field of each sampled packet (in addition to timestamp, hash value and cluster identifier).

An Internet Service Provider backbone is surrounded by routers that are required to have a probe running on them because they are the first to handle and collect customers traffic. Packets need to be marked outside the monitored network allowing the routers to manage already marked packets; this can be done by the customers devices or by the edge routers themselves. Both the solution must be synchronized.

4.1.2 Working principles

The method described is divided into different steps:

1. Data collecting;
2. Sending data to NMS;
3. Preprocessing;
4. Results.

The following schema summarizes the methodology:

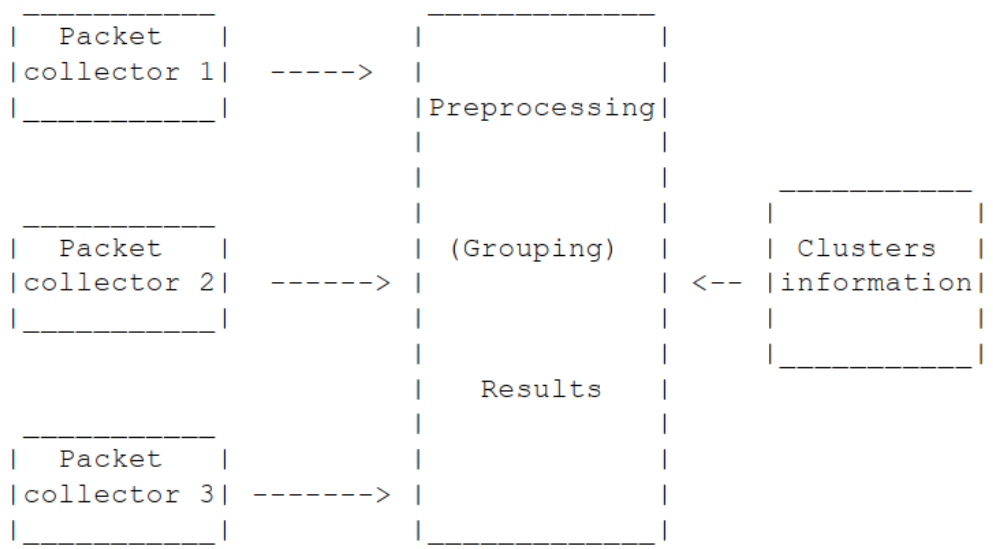


Figure 4.1: Big Data method schema [3]

Packet collectors must be placed in each router we want to monitor and they analyze data passing through interfaces used for monitoring.
In the configuration phase a set of parameters must be passed to the program:

- the set of interfaces that must be monitored;
- the reference hash;
- the maximum number of packets to store;
- the period duration;
- the value of the marking bits.

For what concerns the flow to monitor it is possible to monitor all the flows without distinction. The packet collector only checks if the packet is consistent with the filters and if its hash value match the reference hash, then store it. When the number of packets stored reach the maximum NMAX, the number of bits to match is increased by one and a variable number of packets are discarded (this number ranges from 0 to NMAX but the probability suggests that the mean value is NMAX/2).

The packet collectors send two different types of data to the management system: detailed data of the packets and aggregate data about the period. Detailed data includes the fields that identify the flow, packet hash value, timestamp and period. Aggregate data includes, for each interface, interface ID, total packets counted, total hashed packets, mean timestamp and period.

4.1.3 Preprocessing

The preprocessing is useful to produce, from input records, new data ready to be postprocessed and easier to analyze to obtain performance parameters. The second advantage is the decrease of the total amount of data to be stored, saving space in the disk.

In the preprocessing phase it is possible to aggregate incoming data from all

devices and compute the path followed by each sampled packet; this is done by ordering the records belonging to a given packet by increasing timestamp.
The Network Management System knows the topology of the network and the nodes which compose each cluster, thus it can determine the clusters crossed by packets by comparing the identificator of the interfaces with the nodes belonging to cluster partitions.

4.1.4 Results

Preprocessed record are stored in the NMS database and it can be queried when needed. Results are given by querying the storage system and giving as input parameters the identifiers of the flow we want to analyze plus the identifier of the required time period.
One of the measures that can be obtained is the cluster mean delay D_i (referred to cluster i), computed as the sum of the delays of each record d_j (related to record j), that is the difference between the output timestamp (when the packet left the cluster) and the input timestamp (when the packet came into the cluster). The result is then divided by the number of records that belong to the same cluster. It can be explained mathematically as:

$$D\_i = [d\_0 + d\_1 + ... + d\_(N\_i - 1)]/N\_i$$

where D_i is the delay associated with cluster i, d_j the delay associated with record j and N_i the number of records catched in cluster i.

It is also possible to compute the end-to-end mean delay AD as the sum of all the delays belonging to all the records, divided by the total number of records:

$$AD\_i = [ad\_0 + ad\_1 + ... + ad\_(M - 1)]/M$$

where AD is the end-to-end mean delay, ad_j the delay related to record j and M is the number of all the records.

Other possible values that can be computed are min/max/avg link delay, comparing

all the timestamps relating to all the records.
Furthermore, packet loss details can be obtained. If the total number of packets counted for each output node is less than the sum of packets counted by input nodes, a packet loss occurred.
The packet loss per cluster is:

$$PL\_i = [p\_(i,0)+p\_(i,1)+...+p\_(i,K-1)]-[p\_(o,0)+p\_(o,1)+...+p\_(o,L-1)]$$

These measures can be repeated for end-to-end packet loss considering the input and output nodes of the whole monitored network.

4.2 Implementation

A Big Data emulation environment has been developed by another thesis student [6] who focused on a storage system with the following features:

- Scalable: in a real scenario it should be possible to increase the storage and processing capacity without too much effort because it could be necessary to scale up the system;
- Very large storage capacity: a lot of data could be stored in a big data scenario but it can not be known the amount a priori, so it has to be large enough to forecast all the possibilities;
- Remotely controlled: for the post-processing phase, it is necessary to acces the database remotely because it can be accessed at a later time. In this system it is an important feature.

The choice was *Hadoop Distributed File System (HDFS)*, which meets all these requirements. It is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. [11].

4.2.1 Components

The architecture of the emulation system proposed is explained in the following lines, and it was useful to demonstrate how the Big Data approach can be applied to a real network.
It is composed as follows:

- **Mininet**: tool used to build and simulate a real backbone network where the traffic must be monitored;
- **Probe**: program installed on the monitored interfaces of the routers, needed to count packets, sample them with the hash method, collect information of the sampled packet and save them in a directory where Flume agent periodically checks for new data;
- **Apache Flume**: it is a service for efficiently collecting, aggregating and moving a lot of data, based on data flows. It has been used to automate the process as much as possibile relating to the connection between the monitored system and the storage system. Two instances of Flume are present in the environment, one in the system that emulates routers and the second in the NMS. Data are collected as soon as available on the first one and then automatically pushed in the second one.
- **HDFS**: it is the tool used to obtain big data processing and storage, its main feature. Moreover, it can scale horizontally when needed and manages cluster partitions and distributed storage.
- **Spark**: technology a hundred times faster than MapReduce allowing a parallelization of the execution and fault tolerance. It also allows to store "in memory" data to query the system repeatedly and faster.

4.2.2 Architecture

Components have been split in two different Virtual Machines, not only for performance reasons but also to clearly mark the difference between the network

emulation environment and the Big Data one (including HDFS clusters and processing phases).
The first VM called "source" contains the mininet network emulator, the probe software to be installed on the routers and the Flume agent needed to transfer data to the second VM. The other VM is called "sink" where HDFS is installed together with Spark and the second instance of Flume, where the data coming from the first one are received. The following picture illustrates graphically the overall architecture:

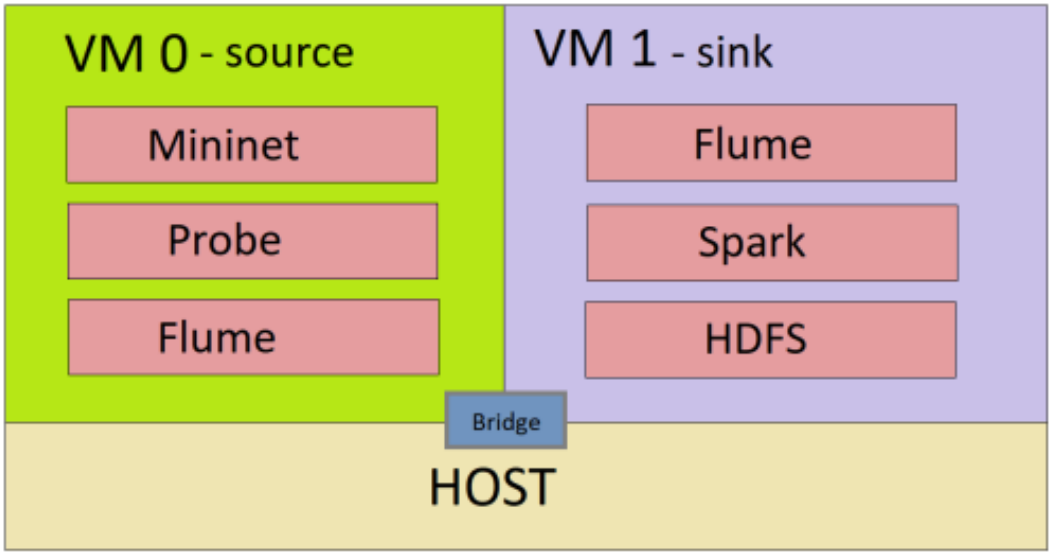


Figure 4.2: Overall architecture [6]

Real implementation

In a real scenario we would have the same distinction between data collecting module and data processing module, with the difference that the collecting one is the real network with routers with a probe installed on, and not a stand-alone machine as the VM0.
Connection between routers and data center can be done in different ways; the simplest one is to send data through a socket as soon as data is available, but this would require an improvement of the solution to guarantee safety and reliability.
Another way could be using Flume as in the emulation environment; it would be

installed on every router guaranteeing all the safety requirements and no manual configurations by the administrators would be required, only a general setup of the software. Another instance of Flume is installed on the NMS where it listens on an IP address and port and receives log files as soon as they are available in each device to be acquired by the Flume instance on the routers.

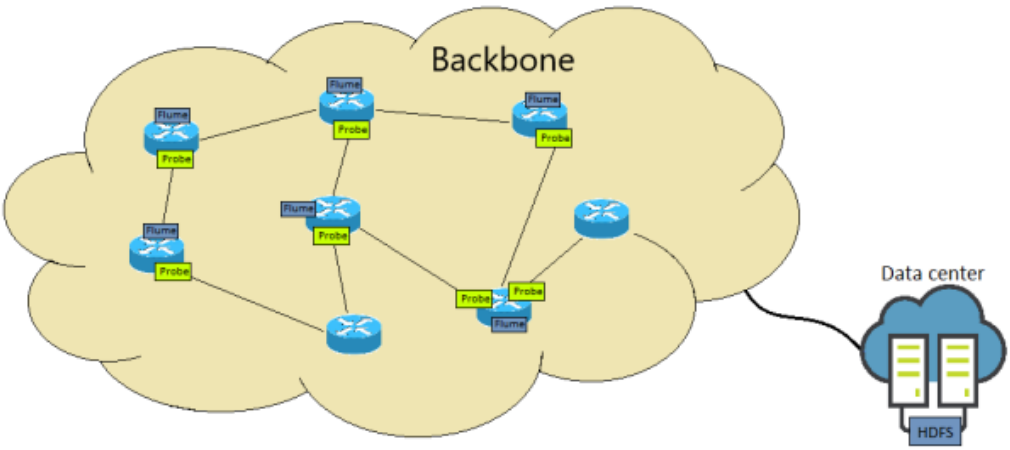


Figure 4.3: Architecture in a real implementation [6]

Chapter 5

BPF

Designing a packet collector has been one of the major goals of this thesis. Among the different alternatives useful for this work, the choice fell on eBPF. In this chapter, its working principle are explained.

5.1 Introduction

BPF (*Berkley Packet Filter*) was developed in the late '80s with the purpose of creating a new fast packet filter. It was specific for BSD, then it was introduces in Linux and it is still present.
It is usually implemented in kernel-space because an early packet filtering is useful and it avoids useless packet copies in user-space, no syscall overhead or kernel/user space context switches. Previous Unix versions provided networking monitoring tools that ran in user space and it was very unoptimized because packets had to be copied in user space and then filtered. So BPF proposed a new pseudo-machine language that permits to filter packets directly in kernel space, avoiding useless packet copies.

Thanks to a Virtual Machine (special purpose virtual CPU, with a dedicated set of instruction specially created for packet filtering purposes), BPF code can be injected in the kernel after being interpreted to emulate a real CPU. Now a lot

of BPF implementations offer a JIT (Just-In-Time) compiler that can translate directly the BPF bytecode in native machine code for the execution; this is a good improvement with respect to the interpreter.
In writing BPF high-level code, some precautions should be taken relating to safety (correct termination of BPF programs avoiding CPU overload or stall). vCPU checks in fact that:

- The bytecode is valid (there is a *Default* branch);
- There is a finite number of instruction;
- There are no loops in the code;
- Destinations of jump/branch are valid;
- No backword jumps are present;
- Read and write are done from/to valid destination addresses.

The main features of the Berkley Packet Filter include the portability (thanks to the VM the bytecode is independent from the hardware and can be executed on every machine) and efficiency (BPF runtime consumes a little amount of resources and, for example, it cannot be used to attack the machine with a "Denial of Service" attack).

5.2 eBPF

In the last years the BPF implementation has been really improved with the introduction of eBPF (*extended BPF*). The original cBPF (classic BPF) was designed as a packet filter module, while eBPF is more powerful and allows, with a programmable interface, to adapt at runtime to user-specific needs.

There are several key features in this new version. The main ones are:

- Possibility to inject bytecode runtime, so that eBPF programs can be created and injected in the kernel in any moment;

- It can react to generic kernel events (i.e. it does not only act as a packet filter). This can be achieved because eBPF code is hooked to a kernel event and it runs as soon as the event happens;
- Shared memory: there is no more raw memory to which a BPF program can access, but some special data structures called Maps are present. They are shared between user/kernel space and also between different BPF programs. This is important because classic BPF (cBPF) was stateless, while with shared memory we can achieve a sort of stateful program, with the state decoupled from the code.
- *Helpers*: a set of precompiled functions available in the linux kernel. These functions allow to use some library functions of the Operating System, overcoming the restrictions given by eBPF. Unfortunately only a subset of the OS functions are present among the helpers, and if someone wish to add some of them there is a complex process for asking the linux community to include the new code in the following kernel versions.

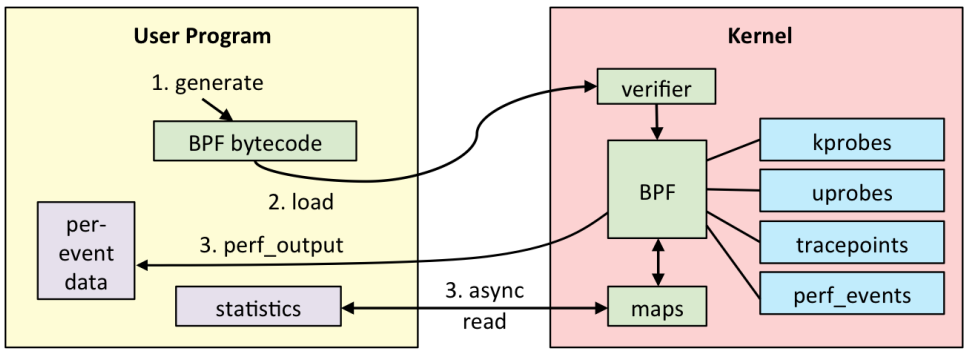


Figure 5.1: eBPF internals [12]

5.2.1 Infrastructure

The language used for eBPF programs is a restricted C, because of the limits imposed by the verifier and the safety of the CPU. Initially, cBPF implemented a 32-bit RISC Instruction Set Architecture (ISA), but with the evolution of the CPUs

that now rely on 64-bit registers, the new BPF extends the classic one by removing 32-bit registers and passing to 64-bit ones. Furthermore, the new architecture evolved increasing the number of instructions and adding a more complex ISA; in general it has been designed to be similar to native 64-bit ISAs.
eBPF code is attached to a special point in the kernel; this point can be located in different places and they are called *Hook Points*; the choice of the hook point depends on what the developer wants to implement.

Helpers

BPF programs cannot call user and kernel functions, but there are some functions that can be used and they are called *Helpers* or *Helper functions*. These functions are implemented in the kernel and act as a proxy between BPF code and kernel space. There are a lot of predefined helpers but only a subset of them can be used depending on BPF program type.
They internally call functions that cannot be called from BPF programs, for this reason the term "proxy" has been used.

Maps

BPF code and user space can't communicate directly because of the safety that has to be guaranteed by the verifier. For this reason a new element was introduced in eBPF: Maps. Maps are data structures that reside in kernel space, and can be accessed by different BPF programs through helper functions but not only: user space can access the Maps too. This is a good solution that permit to share data among BPF programs and between kernel and user space.
These data structures are implemented as key-value structure. The main helpers are `bpf_map_lookup_elem` and `bpf_map_update_elem`, needed respectively to search for a key inside a map and modify a value associated with a certain key.

5.2.2 IOVisor and bcc

The IO Visor Project is an open source project composed by a community of developers to accelerate the innovation, development, and sharing of virtualized

in-kernel IO services for tracing, analytics, monitoring, security and networking functions [13].
bcc (BPF Compiler Collection) is a toolkit for creating efficient kernel tracing and manipulation programs [14]. Developed by the IOVisor project, it makes BPF programs easier to write thanks to a C wrapper around LLVM, and a frontend in Python and lua. It makes the injection of BPF code easier from these languages and provides useful functions to access BPF from user space.

The aim of IOVisor was making the process that lead to the creation of a tool easier. As we know the process of writing, validating and compiling a BPF program is very hard and time consuming; with bcc, everything is hidden to the programmer who has only to care about the implementation of the code.
Another important aspect of bcc is a further level of abstraction: it compiles C code in bytecode through "clang/llvm" and then it injects the bytecode in the kernel with the system call `bpf()`. With Python, the main language used for the frontend, it is possible to interact with the program through the maps.

Chapter 6

Probe architecture

6.1 Overall architecture

In this chapter I will focus on a high level description of the the probe components, all indispensable for our goal of building a packet collector with the functionalities described in the previous chapters.
The probe is developed using the bcc tool that allows the developer to write eBPF programs without worrying about the writing, compilation and validation of the eBPF program. In fact, thanks to bcc, eBPF programs can be written in a restricted C and translated in bytecode automatically thanks to this tool.

6.1.1 User space modules

The frontend is written in python and can be accessed from a REST interface for configuration reasons or to give commands (i.e. start/stop) and it runs in userspace.
It is composed of two parts: a PNPM manager and a BPF manager.

- PNPM Manager interacts with the NMS through some RESTful APIs. In this way the NMS can configure the probe as needed. Moreover, it interacts with the BPF manager passing the configuration parameters received by the NMS;
- BPF manager is connected with the PNPM manager from where it receives the

parameters to enable the BPF program to run; on the other end it interacts with the eBPF program in the kernel space pushing the bytecode and attaching it to the wanted hook point.

6.1.2 Kernel space modules

eBPF program is executed kernel side beacuse this is the main advantage: the processing speed of the packets is really high with respect to user space programs thanks to the absence of context switches that make the execution slower. Moreover, there are some other components that act in kernel space:

- eBPF program is the code attached to a hook point, more precisely to a kernel function that is called when an event happens. In this case, functions are called when a packet arrives in the kernel network stack and when a packet leaves the stack. The program analyzes the packet fields to check if they match the filters and then it computes the hash, discarding or saving the needed values if the hash matches the reference one. Thus it obtains the timestamp and stores all the information inside the maps.

- Maps: in-memory data structures, introduced by eBPF, needed to interact from kernel to user space and vice-versa. In this context, the probe writes the fields of the sampled packets in the maps and when the period ends, PNPM Manager reads them and collect the values in log files.

- Hook point is a kernel event that calls a function when it is raised and the point where eBPF function is hooked. The probe focuses on two hook points that catch a packet in the input and in the output way through the linux networking stack, allowing analyzing the packets.

- NIC: Network Interface Card, the physical component that handle the sending and receiving of packets and the immission of the packets inside the linux networking stack.

In the following picture it is represented the whole architecture with all the components and it shows how they are linked.

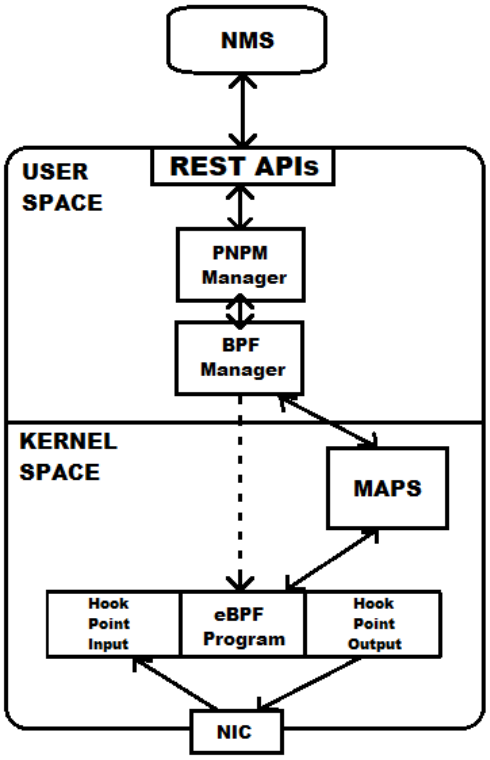


Figure 6.1: Overall probe architecture

6.2 Workflow

In this section I will analyze the different phases that lead to the correct use and configuration of the probe.

1. NMS sends a configuration file in json format to the PNPM manager though a REST interface. This file contains all the parameters regarding Alternate Marking method, such as period duration, duration of the test, starting value and second value of the marking bit, together with the parameters for the dynamic hashing technique (it has been chosen the dynamic one and not the static for the issues reported in Section 3.2.3) such as reference hash, starting number of bits to match and max number of packets to store NMAX.

2. PNPM manager receives the configuration file and interacts with BPF manager.

The latter initializes some constant definitions in the bpf code, written in C, that correspond to the parameter values received by the management system. If no problems are faced, the program is correctly configured and it can be launched.

3. NMS sends a "start" command to the PNPM manager that forwards the task to the BPF manager; it injects the eBPF program in the kernel in a new thread that deals with the management of the probe.

4. When a new packet crosses the linux networking stack at the height of the hook point, the eBPF program is executed and receives as an argument a struct named `sk_buff` that contains the main fields of packets. This struct is allocated by the kernel when a new packet is available.

5. The eBPF program takes the packet information and first of all it checks if the packet came from a monitored interface, otherwise the packet is discarded (the list of monitored interfaces is provided in the configuration file). Then it checks if the packet belongs to the traffic generated by backbone routers and in that case it is discarded too; the marking bits are checked to assign a period to the packet and the hash is computed; if the hash matches the reference one for the least significant number of bit N, the packet is sampled and inserted in the right map. The aggregate measures are updated.

6. After L/2 seconds from the period end, the maps related to that period are examined by a user space function, which discards packets that don't match anymore the reference hash and collects all the aggregate measures in a data structure, while the info of the sampled packets is saved in a log file for each period.

6.3 PNPM manager implementation

In this section I will describe in practice how PNPM manager was developed.
First of all, two ways of marking method have been proposed:

- Two bits, alternately 0 or 1, if traffic is marked externally (but not all flows) so that routers can understand which packets should be analyzed. The choice of the bits has been made taking in cosideration the least used bits of the IP header, i.e. the least significant of the DSCP (Differentiated Services Code Point).

- Some service providers uses the DSCP field to guarantee a Quality of Service to their customers. In this case it is better to avoid using that field. There is a bit in the IP header that is useless, it is called *unused bit* or *evil bit* and it corresponds to the bit 0 of *flags* field. Using one bit to mark the traffic is a complex choice because the marked traffic can be confused with unmarked traffic, so the principle of this method is that all the traffic coming into the network should be marked and the probes only contain a filter to discard the traffic generated by internal routers (routing protocols, configuration protocols,...). This filter is based on the IP subnetwork of the backbone.

+	Bits 0–3	4–7	8–15			16–18		19–31
0	Version	Internet Header length	Type of Service (adesso DiffServ e ECN)		X	X		Total Length
32	Identification					Flags	Y	Fragment Offset
64	Time to Live		Protocol			Header Checksum		
96	Source Address							
128	Destination Address							
160	Options (facoltativo)							
160 o 192+	Data							

Figure 6.2: Fields used for packet marking (X for the first case, Y for the second) [15]

PNPM manager determines if the configuration file contains a subnetwork from which to discard packets. If so, the second method is used, otherwise the first one. In both cases PNPM manager prepares the BPF manager to run in one of the previous working modes.
In practice, it is an intermediate component between the REST interface and the BPF manager, guaranteeing a better subdivision of tasks.

6.4 BPF manager implementation

This is the component that deals with the eBPF program itself, no logic is provided to the BPF manager but it carries out the following functions:

- it sets the parameters inside the C code configuring the tracing program;
- it manages the injection of the code in the kernel and attaching/detaching it from the hook point;
- it deals with the reading of the maps to be accessed from the PNPM manager.

The C code is loaded into a python variable with the following lines of code:

Listing 6.1: Loading BPF C code in a python variable

```
with open (bpf_c_file) as bpf_file:
    bpf_text = bpf_file.read()
```

C code contains some strings surrounded by two hashtag symbols (##STRING##) that are useful to replace the text in that place with some code specified runtime; for example, to make eBPF program know which protocol has been chosen by the configuration file, the string ##PROTO## is replaced with a #define:

Listing 6.2: Example of dynamic configuration

```
if config.proto is not None:
    if config.proto == "udp":
        p = '#define PROTO %d' % IPPROTO_UDP
    else config.proto == "tcp":
        p = '#define PROTO %d' % IPPROTO_TCP
else:
    p = '#define PROTO %d' % 0
bpf_text = bpf_text.replace('##PROTO##', p)
```

where `config` is the data structure that contains the configuration parameters. In this way we will have a dynamic code that changes based on the configuration given. For what concerns the code written above, in case the protocol is undefined, the code assigns to the constant PROTO the value 0 and it means that both

protocols are captured.
Then a BPF object is created, enabling the Just In Time compiler previously:

Listing 6.3: Enabling JIT and creating BPF object

```
import os

os.system('sysctl net.core.bpf_jit_enable=1')
bpf = BPF(text = bpf_text)
```

When the *start* command is received through the REST interface, the BPF manager deals with the attachment of the eBPF functions to the kernel events corresponding to the desired hook points:

Listing 6.4: Attaching C functions to kernel events

```
bpf.attach_kprobe(event="__netif_receive_skb_core",
                  fn_name="receive_packet")
bpf.attach_kprobe(event="__dev_queue_xmit", fn_name="send_packet")
```

that are respectively the ingress and the egress hook points where our functions need to be activated. After the attachment, BPF manager creates a new thread where a function deals with the management of the periods and the test duration. At the end of each period, a function is called to log all the packets contained in the maps.

6.5 eBPF code implementation

In this section it is illustrated the implementation of the probe's core component: the eBPF program. It consists of one main function, called by `receive_packet` and `send_packet`, functions introduced in the code reported in Listing 6.4.
These two functions call the main one with a flag as an argument, to let it know if the packet is incoming or outgoing, together with the arguments passed by the kernel event (i.e. *pt_regs* and *sk_buff* structures).
sk_buff is the main structure because it contains all the fields we need for our task; the most useful for this work are:

- Pointers to the packet fields;

- Timestamp when the packet arrived;
- The pointer to the *net_device* struct which contains the information about the network interface where the packet comes from or is directed towards.

Here a summary scheme describing how *sk_buff* is composed:

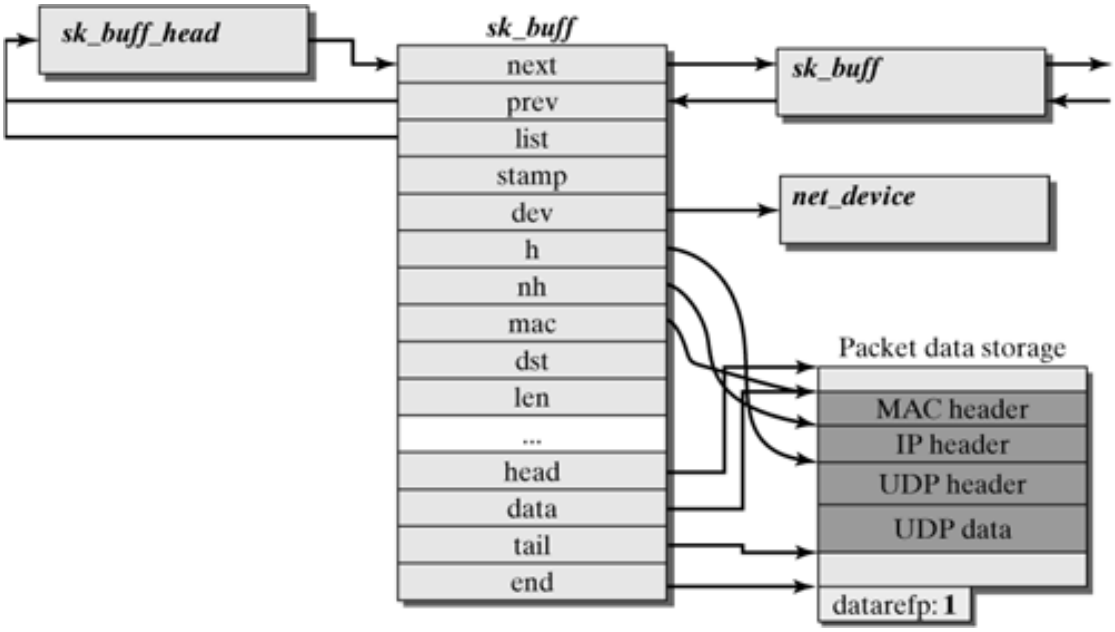


Figure 6.3: Scheme of *sk_buff* [16]

The program receives this structure every time it is executed, it means for every packet traversing the networking stack of the interface we want to monitor.
The executed tasks are various and in the following paragraphs I will explain how they are implemented.

6.5.1 Filter on Network Interface

Each physical interface is split in two logical interfaces (one receiving packets, with the suffix _IN, and one sending packets, suffix _OUT).
The filtering action is based on the interface name, but this is the physical one, so without suffix. To overcome this problem a map called “netif_index” has been

created by BPF manager where the key is the physical name of the interface and the value is:

- 1: if the filter is applied only to incoming packets (logical interface *_IN);
- 2: if only outgoing packets should be collected (logical interface *_OUT);
- 3: if both interfaces are allowed (*_IN and *_OUT).

The eBPF program does a lookup in the map to see if the packet should be discarded.
The code is the following:

Listing 6.5: Filter on the Network Interface

```
struct netif_name interface = {};
bpf_probe_read(&interface.netif, sizeof(interface.netif),
                skb->dev->name);
int *res;
res = netif_index.lookup(&interface);
if (res == NULL || (*res == 1 && recv == false) ||
        (*res == 2 && recv == true)) {
   return 0;
}
```

It means that if the packets are only allowed in input (`*res == 1`) and the current packet is outgoing (`recv == false`), or vice versa (`*res == 2 && recv == true`), or the current interface is not present in the map (`res == NULL`), the packet is discarded and the function is returned.
This method has been introduced also because in the restricted C is not possible to compare strings with the common user-side functions.
The check could be done using the number of the interfaces to avoid all the previous work with strings, but in our case, with the emulation network created in *Mininet*, the logical interfaces don't have an interface number in the host system files and it cannot be retrieved.

6.5.2 Marking fields check

There are two different types of marking as showed in figure 6.2. From the configuration file the program receives the starting mark and the following mark values, whether there is a single marking bit or there are two.
In the first case a value of 1 or 0 should be provided in the configuration file while in the second case 1 or 2 (that in binary correspond to 01 and 10). With this assumption the following code has been developed:

Listing 6.6: Filter on the marking bits

```
struct iphdr* iph = (void*) skb->data;
//if configuration file contains the backbone subnetwork filter
if(discarded != NULL){
    *mark = (bpf_ntohl(iph->frag_off) & 0x2000) >> 13;
    u32 ip_shift = bpf_ntohl(iph->saddr) >> (32 - msk);
    if(ip_shift == ip) //ip is backbone IP shifted right by (32 -
    mask_size)
        return false;
}else{
    *mark = iph->tos & 3;
}
if(*mark != MARK0 && *mark != MARK1)
    return false;
```

As you can see, *flags* field is included in `frag_off` field of the IP header struct and it is together with Fragment Offset field as showed in Figure 6.2. The wanted bit is isolated with a bitwise masking and shifted to be compared with the values defined in the program (MARK0 and MARK1). Then, in case of one marking bit, a check is made on the source IP to be sure it is not a packet generated by backbone routers.
One bit is assumed to be used when there is a subnet to discard `discarded!=NULL` among the configuration parameters, else two bits are used for the marking. In the latter case, the value of DSCP (or Type of Service ToS) is masked using a bitwise AND with 0x3 value (00000011 in binary) to extrapolate the two-bit value and compare it with the reference marks.

6.5.3 Aggregated measures

The probe stores aggregated measures (measurement about the period, i.e. total number of packets catched, timestamp of the first and the last packet passed in the period, length of the hash value, ...) and measures regarding the single packets. To do the first computation, a data structure has been used:

Listing 6.7: Structure for aggregate measures

```
1  struct marked_flow_general_info
2  {
3      u32 count_all_pkts;
4      u16 count_pkts_measure;
5      u64 first_tstamp;
6      u64 last_tstamp;
7      u8 length;
8      u64 and_val;
9      u64 boot_time;
10 };
```

In this structure there are all the fields needed to reach the goal but also other tasks I will explain in the next sections.

- *count_all_pkts* contains the number of packets catched by the probe in a given period. This value is increased each time the function reach the true branch of the conditional structure where it is checked if the packet belongs to the monitored flow, it means for each packet that needs to be monitored.
- *count_pkts_measure* is a counter needed to count how many packets match the hash reference value; when this number reaches NMAX, the hash length (field *length*) is increased by one bit and the value is reduced to NMAX/2.
- *first_tstamp* is initialized to 0 and updated the first time a packet is catched, while *last_tstamp* is updated every time a packet passes.
- *and_val* has been introduced for convenience, as it contains the value of the current bits to match and it is used to do a bitwise AND. This value is updated each time the hash length is increased.

- *boot_time* is needed to compute the timestamp, but further information is provided in the next sections.

Four maps have been used to store these results.
It has been necessary to create two different maps to differentiate input and output packets because the physical name of the interfaces did not include the suffix; the other two maps are needed for the distinction between the two mark values (or colors). The following is the way of creating new maps in C:

Listing 6.8: Maps declaration for aggregate measures

```
1 BPF_HASH(flow_measures_color0_IN, struct marked_flow, struct
      marked_flow_general_info);
2 BPF_HASH(flow_measures_color1_IN, struct marked_flow, struct
      marked_flow_general_info);
3 BPF_HASH(flow_measures_color0_OUT, struct marked_flow, struct
      marked_flow_general_info);
4 BPF_HASH(flow_measures_color1_OUT, struct marked_flow, struct
      marked_flow_general_info);
```

where "struct marked_flow" contains the name of the interface and it is used as a key (second parameter). Third parameter is the map's value type and the first one is the name used to refer to the structure.

6.5.4 Punctual measures

Together with the aggregate measures we need also the punctual ones, that is the set of values stored for each packet, useful to the Big Data system's elaborations. Among these values we have the 5-tuple that identifies a flow (source and destination IP addresses, ports and protocol type), timestamp, hash value, etc.
To reach this goal, a set of four maps has been used as in the case of aggregate measures.
The map's content is the information of the packet, as resumed in its data structure:

Listing 6.9: Structure packet info

```
1 struct pkt_info
2 {
```

```
3       u64 tstamp;
4       u64 hash;
5       u32 sip;
6       u32 dip;
7       u8 proto;
8       u16 sport;
9       u16 dport;
10      u8 hash_len;
11      u8 dscp;
12  };
```

This structure is filled with the values extracted from the packet and inserted into the correct map, depending on the interface direction (input or output) and the color of the marking bit. After the insertion in the map, the aggregate measures of the flow to which the packet belongs should be updated, such as the number of matching packets, last timestamp and total number of packets.

6.5.5 Timestamp computation

A lot of hard work and research have been done to find an efficient way to store the timestamps. eBPF does not allow to call kernel functions, but only helpers that are such a "proxy" or wrapper function that contains a call to the kernel functions; only these wrappers can be used inside an eBPF program.

Among the avalilable helpers, only one is related to a timestamp; it returns the value of a clock at the instant it is called.
The real problem is that the clock is not the one we need, i.e. CLOCK_REAL_TIME but the CLOCK_MONOTONIC. The helper is called `bpf_ktime_get_ns()` and the returned value is the time elapsed since the system boot, in nanoseconds.
This value could be useful only if all the devices boot in the same instant; but it is not the truth, so we need another way to get the timestamp.

New helpers can be implemented in linux kernel to meet the needs of developers. An idea could be creating a new helper that returns the timestamp of the

CLOCK_REAL_TIME instead of the monotonic one but this represents a big problem, beacuse a new kernel patch should be developed and it must follow a very long iter to become part of the official version.
The patch have to be proposed to the linux community and it should be accepted and applied, but it is not that easy; if developer is a well known person among the linux kernel developers there is a higher possibility that the patch is accepted, but if a novice programmer like me tries to propose a patch he must describe why the patch should be accepted, how the new functions could help linux users and if they are really necessary.
Although the description contains valid motivations, there is a high probability that the patch is refused or that the application would require months or years.
Another option is the use of a personalized linux version, with the insertion of the needed helper and the kernel self-compiled for our machine. This solution, although it works, is a complex scenario because in a real network the probe is installed on a lot of devices; each one of them has to run the customized kernel version to correctly execute the probe with this solution and it is not feasable to install it on every network device.
Moreover, updates would be a problem: as it is an unofficial version, to apply an update it is important to recompile the updated kernel with our modifications. Thus, this idea was abandoned.

There is the possibility to let the system generate the timestamp once the packet crosses the NIC and the sk_buff structure is filled.
This is the method I tried to use initially, because it seems very convenient since it automatically generates the timestamps directly from the system without having to worry about further details.
Some problems are anyway present:

- The generation of the timestamp is a complex operation that could influence negatively the performance and speed of our software;
- It's not possible to choose to which packet apply the timestamp but it is an operation done for each packet. We need the timestamp only for those packets

that match the reference hash and in this way the performance is decreased because it is generated uselessly for those packets which are discarded.

- The most important problem for the correct functioning of this technique is the following: the system generates the timestamp in the place between the Nertwork Interface Card and the Hook Point we are using; this implies that it works for the incoming packets but not for the outgoing ones.

For what concerns the third point, different solutions have been studied. The problem reported during the work is an absence of timestamp in the sk_buff while catching outgoing packets (0 value).
This is probably due to the Hook Point position, it means that when eBPF program receives the sk_buff structure relating to a packet about to leave the device, the system has not generated the timestamp value yet.

Retrieving Boot Timestamp from userspace

The first idea to overcome this problem was to obtain the boot time in nanoseconds, i.e. the time elapsed from 1st January 1970 to the moment when the system boot. There are two values needed for the computation of the boot time:

- the time elapsed from boot to the moment when the computation is done (Monotonic Time);
- the second is the value of the clock_real.

Both values can be accessed from userspace.
These two values can be useful to compute the boot time, store it and use it to retrieve the correct timestamp of the packets

$$BT_{OS} = RT_{OS} - MT_{OS}$$

where BT_{OS} is the boot time of the Operating System, RT_{OS} the real time and MT_{OS} the monotonic time retrieved by the OS. Packets are stored in the maps together with their Monotonic Time because it is the only value available through helpers. When the period ends and packets have to be saved in the log file, the MT

value of the packet is added to the Boot Time (BT), so it is possible to reconstruct the Real Timestamp (RT).
After a series of tests and comparisons between the value generated by the system in the input hook point and the calculated value in the output hook point, it has been noticed that the value differed by tens of milliseconds.

This is greatly uncommon as packets are very fast (order of a few microseconds) when they are forwarded from a port to another one and the maximum error we want for our measurement is 100 microseconds.
In conclusion, this problem is due to the context switch delay when retrieving RT_{OS} and MT_{OS} from user space and this is not the solution we are looking for.

Retrieving Boot Timestamp from kernel space

We need to find a different way to compute the timestamp for the outbound traffic so that is comparable to the inbound one automatically applied by the system.
As we saw, the inbound packets have a Real Timestamp from where the Boot Time can be computed; in a nutshell, if we subtract the Monotonic Timestamp to the Real one, we obtain the Boot Timestamp that can act as a reference to compute the RT of the outboud packets.

$$BT_{NIC} = RT_{NIC} - MT_{HELPER}$$

where BT_{NIC} is the value computed by subtracting MT_{HELPER} (Monotonic Time returned by helper *bpf_ktime_get_ns()*) to the value contained in `skb->tstamp` (RT_{NIC}) of the packet catched.
This operation can be done for a random packet but it has to be completed in the same eBPF call.

For sake of simplicity this is done for the first inbound packet catched by the probe. In this way the value of BT_{NIC} is saved together with the values contained in the map for aggregate measures.
A problem rises if the probe is configured to catch only traffic passing through output interfaces. In this case, no inbound packets are catched and the RT_{NIC}

value can't be retrieved.

A solution to have at least one packet with RT_{NIC} stored in *sk_buff* is to catch an inbound packet.
But how to do it if Input interfaces are filtered out by the probe?
The best way that allows to keep the filters enabled and catch at least one incoming packet without knowing the name of the physical interfaces (so it can work on every device) is to send one (or more) test packets to the loopback interface.
This interface is present on every device, be it a server, pc, smartphone or a router.
When the probe is attached to the hook points, BPF manager deals with the generation of a packet with random payload towards the loopback interface (127.0.0.1).

Listing 6.10: Sending first packet

```
s = socket (AF_INET, SOCK_DGRAM)
s.setsockopt (SOL_SOCKET, SO_TIMESTAMP, 1)
byte_message = bytes("Random string")
s.sendto(byte_message, ("127.0.0.1", 5005))
```

First, a socket is created with the desired options applied. Then a message is created with a random text and the last step is sending the message to the IP address associated with the loopback.
BPF program contains a piece of code that intercepts this packet and isolate it from the other traffic.
Specifically it checks if the destination address is "127.0.0.1" with the following code:

Listing 6.11: Loopback packet detection

```
if ((bpf_ntohl(iph->daddr)^0x7F000001) == 0 && recv == true)
                              //127.0.0.1
   flag=true;
```

By exploiting a bitwise XOR between destination address and exadecimal value 0x7F000001 (127.0.0.1) we can obtain 0 if all the bits are the same and a value different from 0 if destination address is different from loopback.

In the case it is the packet we are looking for, loopback interface is added as an entry to the flow map and the Boot Time $BT_{NIC} = RT_{NIC} - MT_{HELPER}$ can be retrieved in user space because RT and MT values are stored.

BPF Manager deals with the computation of BT and the storage in a global variable that will be used to compute RT of every outbound packet.

Listing 6.12: Saving Boot Time

```
for key, leaf in measures_map.items():
    netif = cast(key.netif, c_char_p)
    if netif=="lo_IN":
        if (boot_time is None):
            boot_time = leaf.real_time - leaf.monotonic_time
        continue
    ...
    ...
```

When the entry is inserted in the map, the name of the interface is modified to "lo_IN" to recognize it.
To be sure about the correctness of the measures, the saving of the two values (RT and MT) is done in the BPF program not only for the loopback packet but also for the first packet of every input interface.
In this way we are sure that it works because:

- When a period ends, BPF manager iterates first on the receiving interface, so the BT can be stored before reading the packets on outgoing interfaces (the ones that need BT value);
- We don't know what position the loopback interface occupies inside the map, so the first interface analyzed is useful to store BT.

When the maps relating to outgoing packets are examined, the value contained in the timestamp field is the monotonic time retrieved from `bpf_ktime_get_ns()` helper, so this is added to the previously calculated BT to obtain the Real Timestamp.

Timestamp conclusions:

the comparison between the Real Time calculated in kernel space for the outbound packets and the RT generated automatically in the sk_buff of inbound packets demonstrates that the measures differ very little (few microseconds).
This is an acceptable error that does not influence the computation of delay.

Chapter 7

Testing and validation

After the implementation phase we needed to test the correct functioning and the speed performance reached by this solution.
In this chapter I will analyze the testing environment, i.e. how the emulation network is built to get closer to a real scenario.
After that, it will be illustrated how test operations have been organized to fulfill all the possibilities that can occur in a real backbone network, starting from the bandwith to the size of the packets and the flows (intended as layer 3 combination of IP 5-tuples).
Finally, the results obtained are described in order to understand the ability of the software to do its work, in terms of packet captured and maximu throughput, delay measurement, etc.

7.1 Scenarios

First of all we need to emulate the behaviour of a programmable router with N ports where each one of them is bidirectional.
The routers should be programmable because the probe software need to be installed on them to correctly catch and analyze transiting packets.

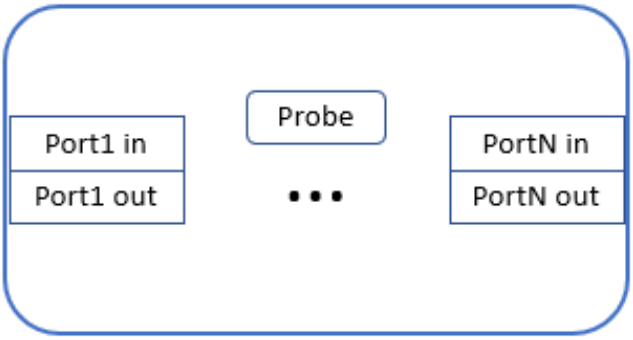


Figure 7.1: Programmable router

In this scenario we have provided two working modes:

- packets go through the probe and they are forwarded as in their natural path (original traffic is analyzed); they cross both input and output port and the probe catch twice each packet;
  If a packet is lost during the probe analysis (e.g. in the device queue), it will never reach the destination.

- packets are duplicated to be analyzed separately from the original ones. This case would be useful if we don't need the internal device delay and we can report only one timestamp related to both input and output port referring to each packet.

In the first solution we can notice that the probe is executed twice for each packet, increasing the CPU usage of the device and potentially discarding packets if the network speed is too high.
The second solution would consume lower resources because the packet is analyzed once, only when it enters the device.

As the goal is to measure the software performance, it has been decided not to use programmable routers but some servers that emulates the router behaviour.

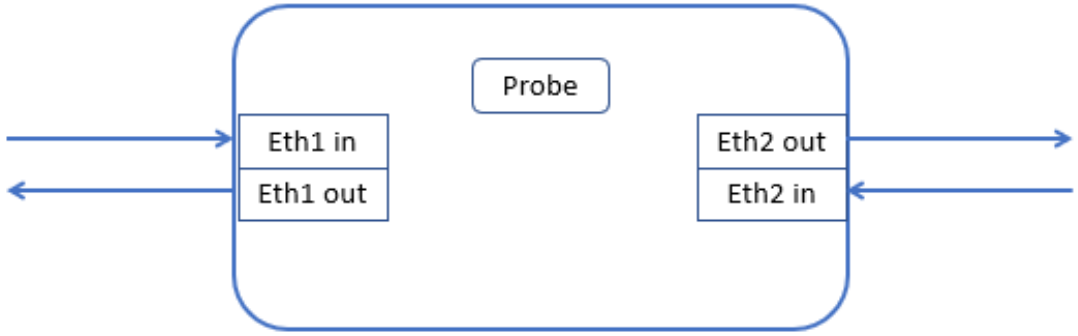


Figure 7.2: Server emulating a programmable router

This server is supposed to act as a two-port router, so it can be installed on a physical link, intercepting all traffic passing through it. The traffic entering the interface Eth1 is forwarded to Eth2 through static routing configured in the server. For the reverse path (Eth2 -> Eth1) the same configuration should be provided, adding a static route into the routing table.

Also here we have two working modes:

- server can be used as an isolated probe, where traffic is duplicated and switched towards the server, and there it ends (traffic is terminated into the server). It acts as a passive probe because if packets are lost in the probe, original traffic is not affected;

- the server can act as a transit probe, where original packets cross it from one interface to the other an then they continue on their path. In this case a malfunctioning of the probe can affect the traffic.

The problem of traffic duplication in a router (or server) depends on how it is done: if the device can perform an hardware duplication there is no overhead problem, but if this functionality is not provided the only way to duplicate packets is via software.

This can represent a problem because a lot of CPU resources are used for this purpose.

7.1.1 Emulation network

To emulate at least a two-device network, two servers have been provided where a probe has been installed.
Both servers run Ubuntu 18.04, while one has a 56 core CPU *Intel(R) Xeon(R) CPU E5-2680 v4 @2.40GHz* and the second holds a 48 core CPU *Intel(R) Xeon(R) CPU E5-2690 v3 @2.60Ghz*.
The idea is to generate traffic toward the first server (let's call it Server1) which is crossed by the traffic and connected to Server2, which is connected to the other end of the traffic generator.

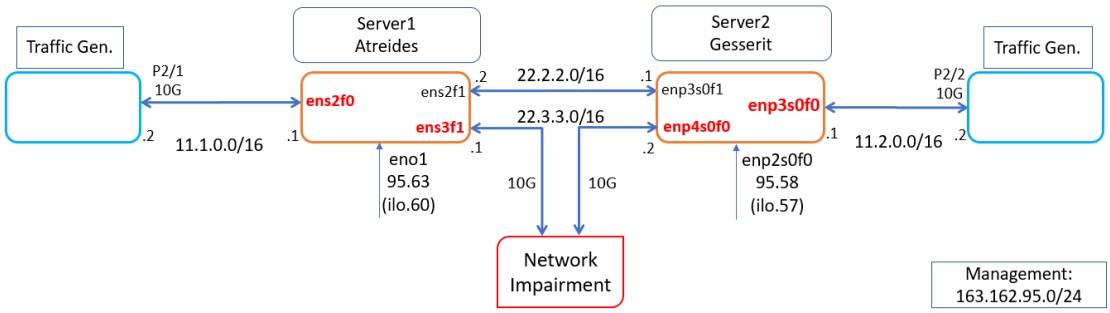


Figure 7.3: Emulation topology

As illustrated in the picture, there are two traffic generators to emulate a bidirectional path. Actually, I used Spirent TestCenter, an end-to-end test solution delivering high performance with deterministic answers. Service providers and Enterprises use it to test, measure, and validate their networks and deploy services with confidence [17].
This tool has different ports that can be configured in a great variety of ways.

In Figure 7.3 the configuration is illustrated and it shows that port 1 of blade number 2 (P2/1) is physically connected to Server1 through a fiber cable because this port can reach 10Gbps, so a UTP cable was not sufficient.
Server1 is directly connected to server2 through two links, one 1Gbps with a UTP cable and one capable of reaching 10Gbps whit a fiber cable.
Lastly, Server2 is connected to port 2 of blade 2 (P2/2) where the traffic is received

and generated in order to obtain a bidirectional flow.

To start the testing phase, the two generators had to communicate between them, so a manual configuration was important to reach this goal.
The configuration was organized as follows: three private subnetworks associated to each link (let's leave aside the 1Gbps link), starting from left to right 11.1.1.0/16, 22.2.2.0/24 and 11.2.0.0/16. Contiguous devices should be able to communicate directly because they know the link subnetwork but those which are not directly connected are not aware of the intermediate addresses.

For example, Server1 cannot reach addresses belonging to 11.2.0.0/16 because it is not a directly connected subnetwork. For the same reason Server2 is unable to reach 11.1.1.0/16. We need to add static routes that allow to communicate in both directions.

Server1 routing table			
Destination	Netmask	Gateway	Interface
0.0.0.0	0.0.0.0	Default gateway	ens0
11.1.1.0	255.255.0.0	0.0.0.0	ens2f0
22.2.2.0	255.255.0.0	0.0.0.0	ens2f1
11.2.0.0	255.255.0.0	22.2.2.1	ens2f1

Server2 routing table			
Destination	Netmask	Gateway	Interface
0.0.0.0	0.0.0.0	Default gateway	ens2s0f0
11.2.0.0	255.255.0.0	0.0.0.0	ens3s0f0
22.2.2.0	255.255.0.0	0.0.0.0	ens3s0f1
11.1.1.0	255.255.0.0	22.2.2.2	ens3s0f1

The highlighted rows are the static routes added manually to each server, while the others are the connected routes and the default ones.

After configuring static routes, I noticed that both Server1 and Server2 dropped packets directed to non-directly connected subnetworks. First of all I tried to set sysctl variables **net.ipv4.conf.{*interface*|all}.rp_filter** to 0, since default value 1 prevent the system to receive packets coming from IP addresses that are not part of the device interfaces. This was not the problem because also the ipv4 forwarding was disabled by default, so i had to set **net.ipv4.ip_forward** to 1. After these modifications, both ends of Traffic Generator were able to ping themselves.

TestCenter is configured creating a device (or more) on each port that will act as source/destination of our data flows. On port P2/1 an host is created with IP address 11.1.1.2/16 and gateway 11.1.1.1 (Server1 interface).
On the other end the address is 11.2.0.2/24 with gateway on 1.2.0.1 needed to forward all the generated traffic to Servers that will manage the routing through static entries provided. if we set a number of device per-port greater than one, the software will create N devices with IP address incremented by a given value, this to allow the generation of many layer 3 flows (different source and destination IP addresses).

7.2 Test

The testing phase aims to push the software to the limit, under stressful conditions, to discover which is the maximum throughput reachable by the system without packet losses. The desired results consist in a processing capacity at line speed, which in the backbone networks is near to 10Gbps.
The probe should be able to analyze packets of different sizes without discarding them; this is important because the measurement of packet loss is one of the key aspects of Alternate Marking method and if probes don't collect every packet we cannot retrieve a correct value for packet loss in the network.

7.2.1 Single-core performance

The first test I executed was the generation of a single-flow stream to see how far a single core can go. As our NIC supports multiple receive and transmit queues, one for each core, it applies a filter to each packet to assign it to a given CPU, based on the flow to which it belongs. This mechanism is also called *Receive Side Scaling (RSS)*. The filter applied by RSS is usually an hash function over the network/transport layer, for example over source/destination IP and ports.
This means that if we generate only one kind of traffic with the same layer 3/4 fields, RSS will steer the packets over a single core, thus allowing to measure the performance of a single core.
The first stream was composed of big packets (1500 Bytes, Maximum Transmission Unit MTU) that permit the lowest packet rate possible for each bandwith; in this way the CPU have to process the minimum number of packets and we can measure the maximum performance possible.
The test showed that the probe is able to process packets at the speed of 3.1Gbps without packet loss as shown in the following figure:

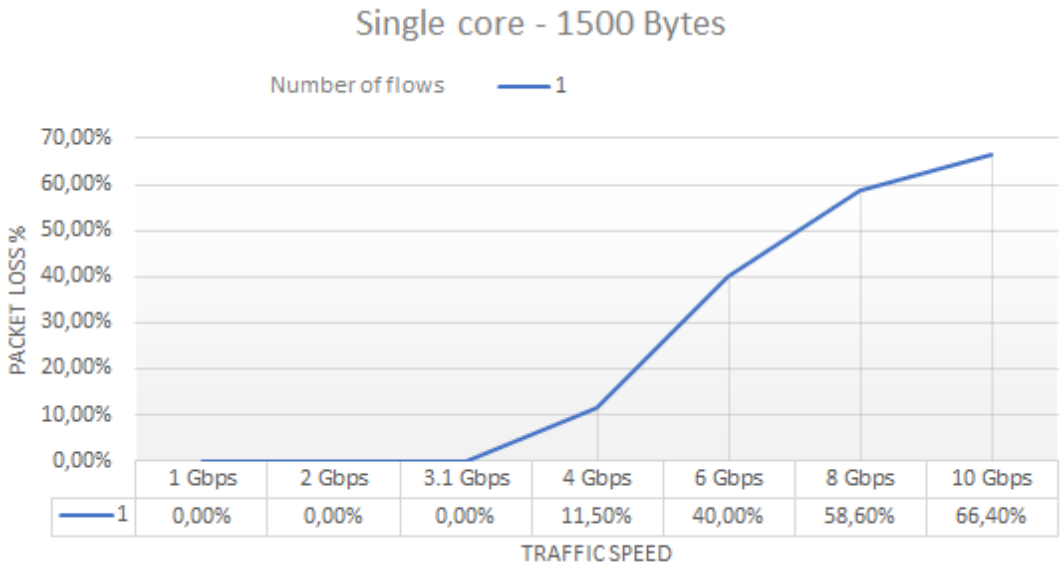


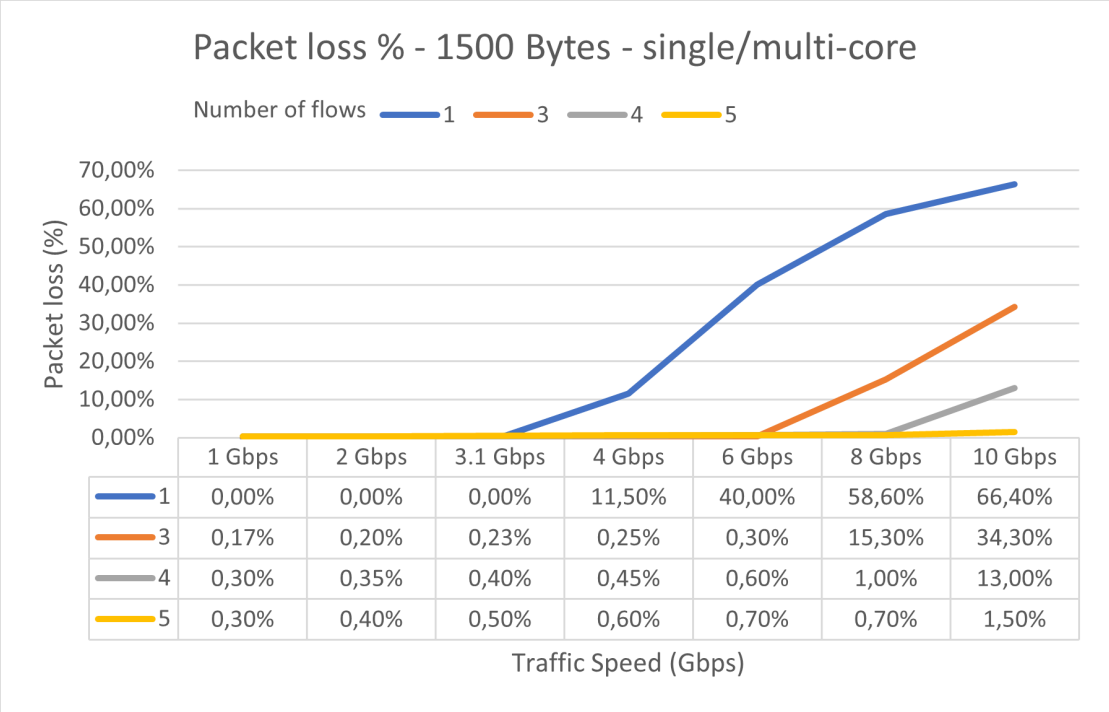
Figure 7.4: Packet loss (%) with 1500 Bytes packets, single core

As the figure shows, the loss is 0% until the bandwith reach the maximum at which the probe starts to loose packets. When this threshold is exceeded, the software starts to drop a higher percentage of packets that grows with increasing speed.
The tests are repeated with different packet sizes and, as expected, the threshold decreases for smaller packets; the results of the measurement are resumed in Figure 7.7 and compared with multicore results.

7.2.2 Multi-core performance

Our scenario includes higher bandwith with respect to the one reached by the single CPU. For this reason we need to exploit all the possibilities given by the devices; the most useful in this case is the possibility to split the packet processing among different cores, improving speed and results reliability.
This can be done increasing the number of flows generated by the TestCenter: to achieve the result, a number of devices greater than one have been created on the ports and with increasing IP address (e.g. 11.1.1.2 - 11.1.1.11 when creating 10 devices on port P2/1, corresponding to 10 flows). There was only one limitation as I configured /24 networks on the links and in the routing tables, when the emulation network was set up, so a maximum number of devices equal to 254 could be set; after the initial tests, the necessity of trying with a bigger amount of flows has born and the configuration was changed to support /16 networks.

Problem in the code - First attempt failed The first try did not work as expected, because when the flows from one became more (even 2), a strange thing happened: the probe started to lose packets even at low speed and moreover, the counters on the two interfaces of the servers were not coherent and differed by varying amounts. This was suspicious because the expected behaviour had to be better than the single core one.
In the following chart I illustrated a comparison between the single-core and the multi-core:

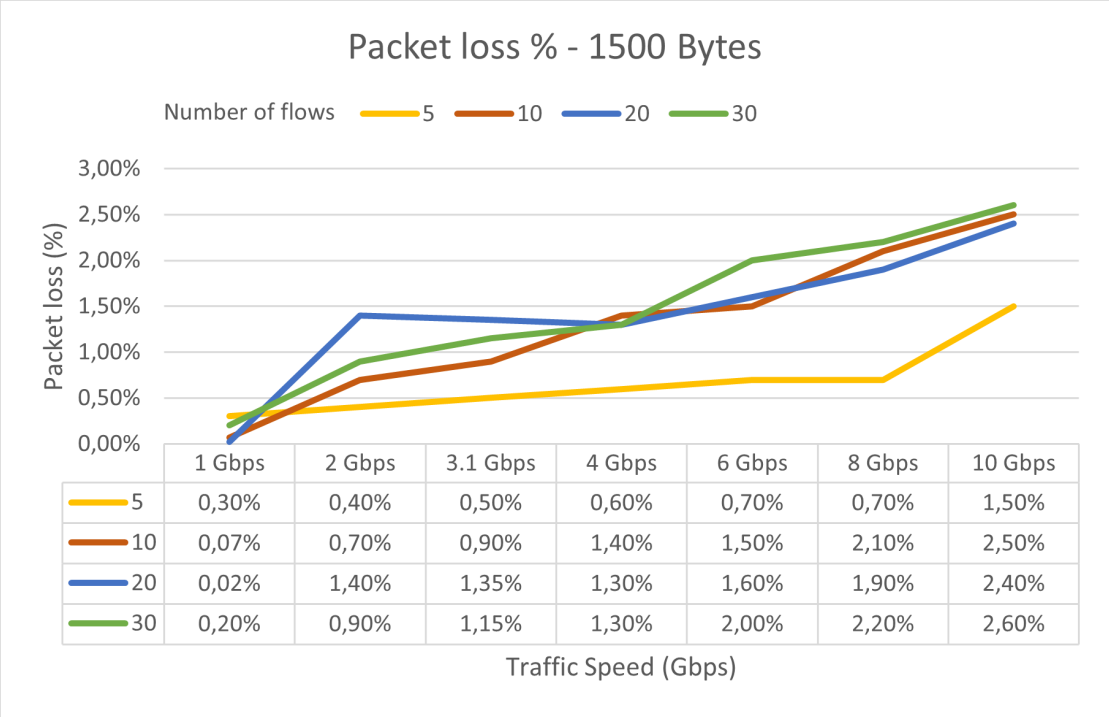


	1 Gbps	2 Gbps	3.1 Gbps	4 Gbps	6 Gbps	8 Gbps	10 Gbps
1	0,00%	0,00%	0,00%	11,50%	40,00%	58,60%	66,40%
3	0,17%	0,20%	0,23%	0,25%	0,30%	15,30%	34,30%
4	0,30%	0,35%	0,40%	0,45%	0,60%	1,00%	13,00%
5	0,30%	0,40%	0,50%	0,60%	0,70%	0,70%	1,50%

Figure 7.5: Comparison between single-core and multi-core

As showed in the picture, the overall packet loss decreases for high speed traffic, but when more than one core is used a small percentage of packets are not registered by the inner counters of the software, while one single core starts losing packets only when the threshold is exceeded.

The following picture is a zoomed chart showing packet loss percentage of 5-10-20-30 flows for different bandwith. The percentage value is not always fixed but it varies by a little amount every time a test is executed; values present in the table are the average on all the tests I took.
It can be noticed that the percentage of packet losses is highly variable, but in the overall the higher packet loss occurs with the increase of the number of flows.



	1 Gbps	2 Gbps	3.1 Gbps	4 Gbps	6 Gbps	8 Gbps	10 Gbps
5	0,30%	0,40%	0,50%	0,60%	0,70%	0,70%	1,50%
10	0,07%	0,70%	0,90%	1,40%	1,50%	2,10%	2,50%
20	0,02%	1,40%	1,35%	1,30%	1,60%	1,90%	2,40%
30	0,20%	0,90%	1,15%	1,30%	2,00%	2,20%	2,60%

Figure 7.6: Maximum throughput per packet size

From this chart I noticed that when the number of flows increases (more cores are used), the percentage of packets not counted increases too. It is an uncommon situation as the multiqueue solution is useful for the opposite case: more cores should be capable to guarantee a smaller number of packet lost.
Together with this situation, I analized statistics created by the packet generator to understand where the problem was. I noticed that all packets sent by one port arrived correctly to the second port and viceversa, so packets were not dropped by the servers.
Another thing that helped me to locate the problem was the fact that counters on input interfaces differed from the ones on the output interfaces: sometimes C_IN > C_OUT and other times the reverse, without a logical sense.
I started thinking to a synchronization problem of the counters, it means that when two or more cores try to increase the value of the counter related to an interface, they don't access atomically to the memory where the variable is located, so there could be erroneous update (race conditions).

Counter variables in the probe are located in Maps, corresponding to the entry belonging to the correct interface. eBPF maps are protected by the kernel Read-Copy-Update mechanism that makes the access thread safe, but this does not exclude possible race conditions given the multithread functioning of eBPF programs, as RCU guarantees only a correct read of the values.
In this case the operation that increments the counter was not atomic, because the pointer to the memory space was retrieved before, and then an increment (counter++) was executed. The latter consists of more than one operation (read value, increment and store it) and this can lead to race conditions. Usually the value is incremented by one when two or more threads modify it, because they read the same value and store it, and when they increase it the result stored in the map is the same in both cases.
This problem was hard to solve because eBPF forbid the use of locks, but fortunately an helper is provided by bcc that is the one calling the atomic function *__sync_fetch_and_add*; this helper is called *lock_xadd*.
Thanks to this helper the probe started to count packets correctly and the real test could start.

Problem solved - Second attempt successful After solving the problem that locked my work for some days, finally the real performance measurement can take place.
First of all the measures relating to the single-core mode were not affected by the problem as one core does not cause race conditions. Then, the system has been pushed to the limit to record the maximum speed reachable without packet loss. All the tests have been executed with different packet size.
Data collected are resumed in the following histogram, where the threshold reachable without losing packets is compared between single-core and multi-core. As expected the use of different cores where the traffic is split by RSS gives a higher performance improvement with respect to the single-core.

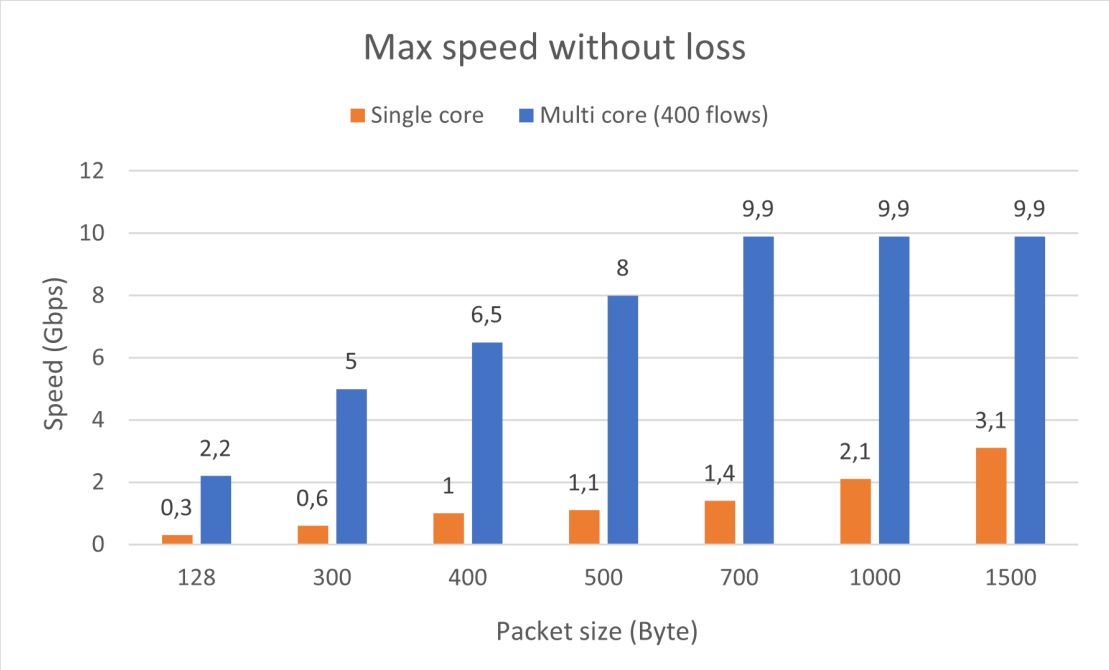


Figure 7.7: Maximum throughput per packet size

To well understand the limit reached is important to compare these results with the limit bandwith bearable by the servers NIC without the use of the probes:

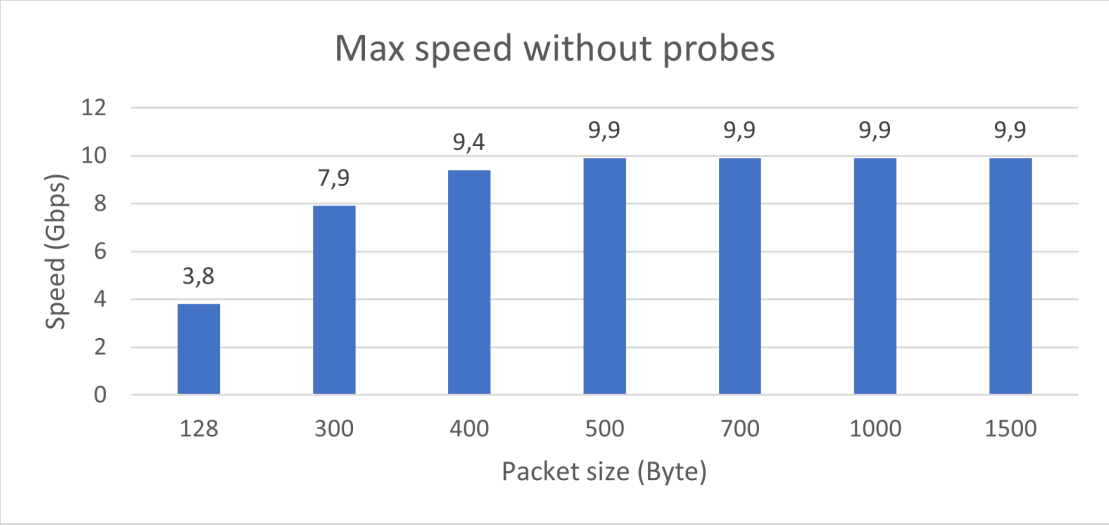
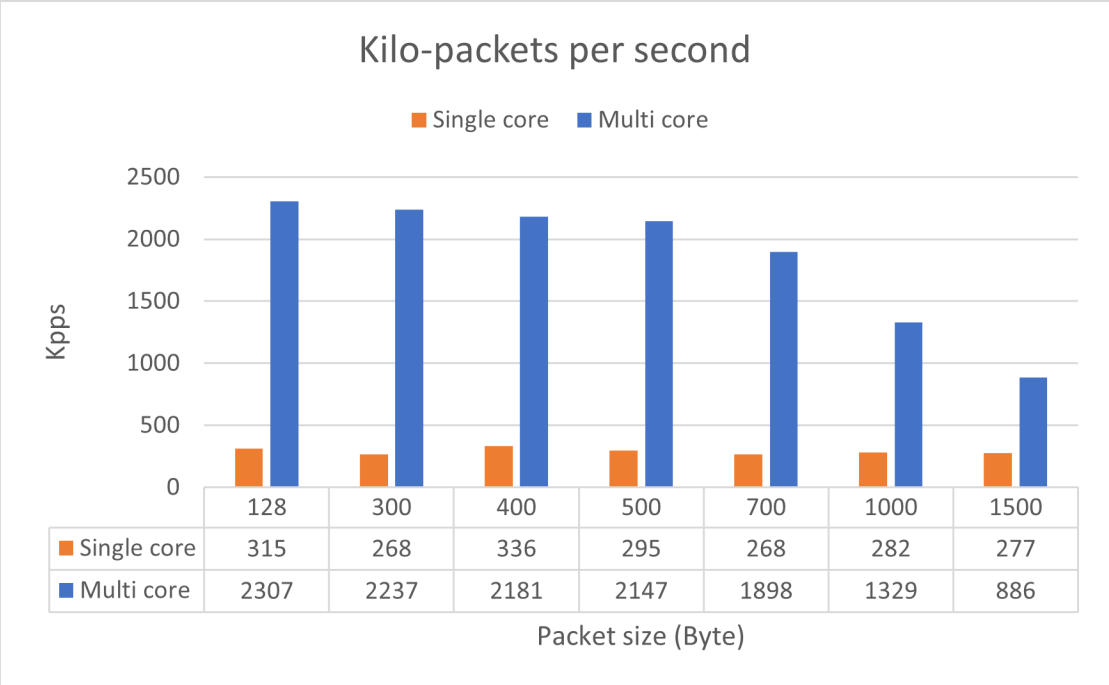


Figure 7.8: Maximum throughput manageable by the hardware

The limit bandwith used in the tests is 9.9 Gbps because 10 Gbps is the physical limit of the NICs and they could discard packets if the generator reaches peaks that exceed this value.
As shown in the image, the software is able to handle the highest speed with packets down to 700 Bytes or less and then it starts to decrease the speed at which there are no packet losses.
This is due mainly to the fact that with big packets the queues empty rapidly as soon as one packet is analyzed, while with smaller packets, queues take more time to make space for new packets once they are full. To better understand let's analyze the chart showing packets per second processable.



	128	300	400	500	700	1000	1500
Single core	315	268	336	295	268	282	277
Multi core	2307	2237	2181	2147	1898	1329	886

Figure 7.9: Maximum number of packets processable per seconds

The reported charts have been built with 400 flows after a series of tests. Performance was slightly worse with a lower number of flows and it improved with increasing number of them. The peak was reached with 400 flows, and then performance became worse again.

7.2.3 Delay introduced by probe

An important aspect not to be understimated is the delay of the traffic. The goal was not to introduce a lot of delay but to make the system work as if the probes are not present.
For this reason I measured the average time it takes for packets to go from one edge to the other. From this data we can understand that the contribution of the probes on the delay is minimal and it is usually less than 10 µs, a very little value.

In the following table I decided to put the average delay took by packets of different dimensions, at the limit speed at which they are not lost, with the software not running and compare it with the average delay of the same packets at the same speed but with the software enabled to see the difference.

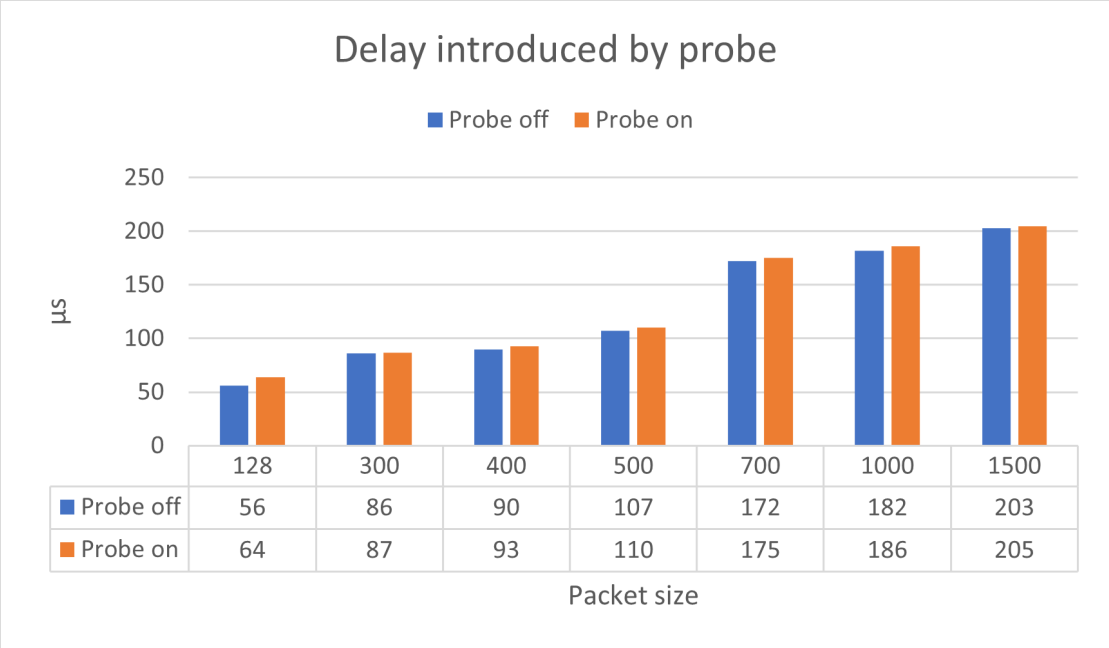


Figure 7.10: Average delay with and without probes

As the picture shows, the average delay differs from few microseconds, an irrelevant amount. This is a good result because it means that this software can be used in a real network without affecting the traffic original delay.

7.3 Timestamp precision

An important aspect of the probe to be considered is the precision of the timestamp, useful to have a good reliability in calculating delay and jitter. To achieve this target, a Network Impairment has been used to generate delay in the traffic flow. The device has been put in the middle of the Servers, to locate the delay in the link connecting the two interfaces of the different machines.

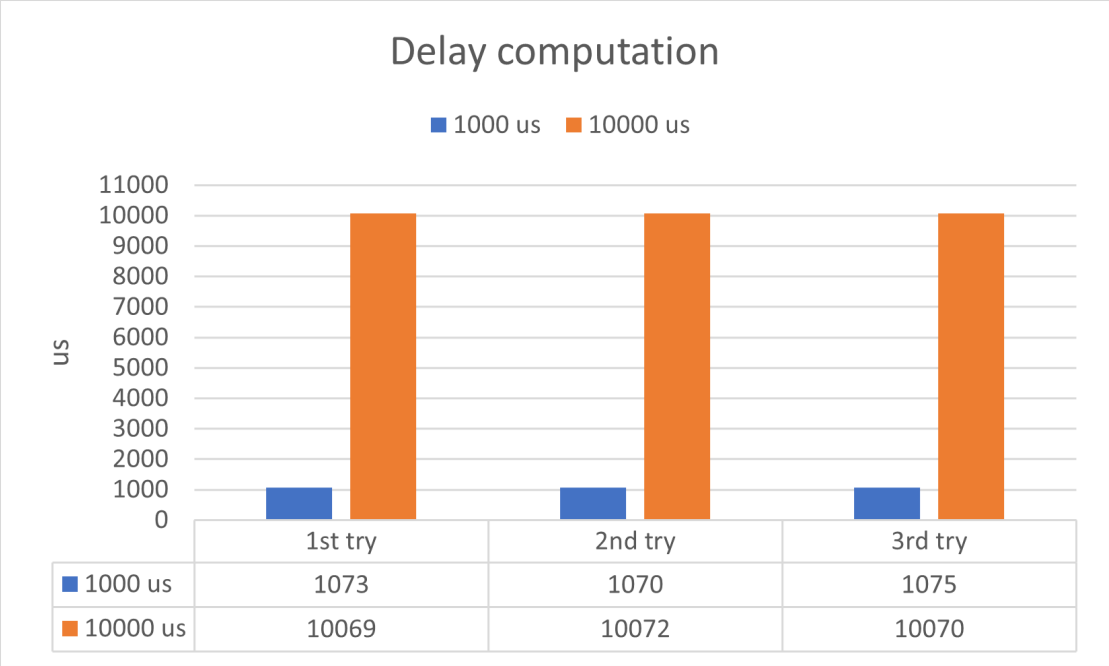


Figure 7.11: Delay computed with the probe timestamp

The figure shows that timestamp is correctly computed, as the difference between the two interfaces reports the delay value set in the Impairment Generator. Exceeding microseconds (about 70) are the delay due to the physical medium to be crossed.

7.4 Results analysis

After having collected data from the simulations taken, the focus moved to the analysis of the results obtained. For what concerns the single core performance,

the maximum throughput of the software is given by the CPU performance that limits the speed at which packets are analyzed.
When packet loss starts to happen it is due to the overload of the core managing the packets; if the flow is bidirectional there are two overloaded cores because the load balancer assigns each of the two flows to a different CPU.
With the linux command `htop` we can see the usage of each core.

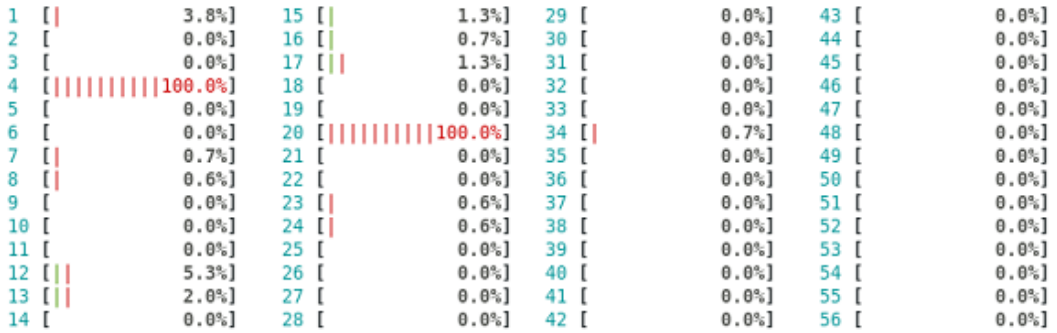


Figure 7.12: Cores usage

In this case core 4 and core 20 have been chosen by RSS and as the traffc was too fast for their computation capabilities, the resources consumed are 100% and some packets are lost during that overload.

As imagined before starting testing, when the number of flows increase, computation is splitted among several cores, thus allowing to improve performance and the number of packets elaborated per seconds exploiting the CPU parallelism.
It has been demonstrated that starting from packets with size 700 Bytes to 1500 Bytes the probe is able to collect all the packets without discarding any; this is a good result as the average packet size in the backbone networks is included in this range.

It is important also to evaluate the effects that the probe could cause to the traffic delay; the tests show that the delay introduced by the start of probes is minimum and it does not exceed 10 µs.
This is a good result because it demonstrates that probes can be installed in a

backbone network without affecting the normal flow of the traffic.
Tests were taken with only one probe turned on in the simulation network and after with both probes on; the results obtained say that the delay introduced by the probes does not vary depending on the number of collecting points.

For what concerns generation of timestamps, tests show that the accuracy is of a few microseconds, thus the delay can be correctly computed by the Big Data system when it collectss all the packets information and process it.

Chapter 8

Conclusions

Thanks to this thesis work it has been possible to deploy a new component needed for the Big Data Alternate Marking monitoring method to work correctly in a real scenario and specially in a real network.

Now, after testing its performance, it can be possible to integrate the software with the Big Data environment implemented in the previous thesis [6].
Line speed packet analysis has been reached and this was the first goal of the thesis, thus the installation of probes in a bigger emulation network or in a real one is the next step to see how it behaves given the prolonged execution and the different probability distribution of packet sizes with respect to the fixed size used in these tests.
Thanks to eBPF it has been possible to develop a high performance software with code that is injected directly in kernel; this lead to the improvement of processing speed as every time a new packet comes in, the kernel immediately execute the code without performing context switching between user space and kernel space, as for the programs executed in user side.
The bcc tools developed in the project IOVisor [13] helped me to implement a powerful eBPF program without worrying about the writing, compilation and injection of the real BPF code and thus simplifying and speeding up the work.

This thesis path took a lot of effort and many problems during the writing of the code, as I had never written an eBPF program before this experience and it requires a lot of knowledge about the linux kernel and networking stack.
In the end, I consider it a completely positive experience because it helped me to carry out a difficult work during this pandemic period due to COVID-19 virus, in which i was not allowed to work in Politecnico or TIM laboratories in the company of other Computer Science students or specialists to whom I could have asked for information about the new topics I didn't know.
Working from home was a different way but the goal of completing the thesis was reached succesfully with a little more effort.

8.1 Possible future works

For what concerns the generation of timestamp, a solution has been adapted to achieve the goal of the program, but it is not the best way because the time computation is not symmetrical for input and output side: for the input packets the timestamp is generated automatically in the networking stack and inserted into the sk_buff structure, while for the output packets there was not the possibility as the hook point is located before the generation of timestamp in the linux networking stack, so it has been used the helper that returns the monotonic clock and then added to the boot timestamp.
This is a temporary solution, as there is not yet an helper that can return the real clock value. Some possible future work can be creating this helper and proposing it to the linux community to be approved.
Another way to retrieve the timestamp can be done by hardware, exploiting the NIC potential; this could improve significantly the performance as the software timestamping requires a non negligible overhead.

During the development of the software a different way to organize the probe was thought: instead of attaching the code to the TC hook points it can be done in XDP mode where the packets are catched before entering the networking stack and they can be discarded earlier if they don't match the hash and the filters. The

hypotesis is that in this way performance can get better, but it has not been tried because XDP hook point is only present in receiving side and here the timestamp value is not available because sk_buff is not built yet. This try can be made when an helper that returns the real clock is included in the linux kernel and if in the future an XDP hook point will be available for the outbound packets.

The current version is a good working one and it can remain as it is, but if there is the need to further improve its performance, the previous attempts can be done to see if they really lead to the wanted gains.

Bibliography

[1] G. Fioccola, A. Capello, M. Cociglio, L. Castaldelli, M. Chen, L. Zheng, G. Mirsky, and T. Mizrah. «Alternate-Marking Method for Passive and Hybrid Performance Monitoring». In: *IETF: RFC 8321*. 2018. URL: https://www.rfc-editor.org/info/rfc8321.

[2] A. Morton. «Active and Passive Metrics and Methods (with Hybrid Types In-Between)». In: *IETF: RFC 7799*. Vol. DOI 10.17487/RFC7799. May 2016. URL: https://www.rfc-editor.org/info/rfc7799.

[3] M. Cociglio, C. Corbo, G. Fioccola, M. Nilo, and R. Sisto. «The Big Data Approach for Multipoint Alternate Marking method». In: *IETF draft*. Vol. draft-c2f-ippm-big-data-alt-mark-00. March 2020. URL: https://tools.ietf.org/html/draft-c2f-ippm-big-data-alt-mark-00.

[4] G. Fioccola, M. Cociglio, A. Sapio, and R. Sisto. «Multipoint Alternate Marking method for passive and hybrid performance monitoring». In: *IETF draft*. Vol. draft-ietf-ippm-multipoint-alt-mark-09. March 2020. URL: https://tools.ietf.org/html/draft-ietf-ippm-multipoint-alt-mark-09.

[5] F. Salvini. «Monitoraggio delle prestazioni di reti a pacchetto con IOVisor». MA thesis. Politecnico di Torino, 2018.

[6] C. Corbo. «Big data post-processing of multipoint measurements with alternate marking method». MA thesis. Politecnico di Torino, 2020.

[7] N. Duffield, D. Chiou, B. Claise, A. Greenberg, E. Grossglauser, and J. Rexford. «A Framework for Packet Selection and Reporting». In: *IETF: RFC*

5474. Vol. DOI 10.17487/RFC5474. March 2009. URL: https://www.rfc-editor.org/info/rfc5474.

[8] T. Zseby, M. Molina, N. Duffield, S. Niccolini, and F. Raspall. «Sampling and Filtering Techniques for IP Packet Selection». In: *IETF: RFC 5475*. Vol. DOI 10.17487/RFC5475. March 2009. URL: https://www.rfc-editor.org/info/rfc5475.

[9] T. Mizrahi, C. Arad, G. Fioccola, M. Cociglio, M. Chen, L. Zheng, and G. Mirsky. «Compact Alternate Marking Methods for Passive and Hybrid Performance Monitoring». In: *IETF draft*. Vol. draft-mizrahi-ippm-compact-alternate-marking-05. July 2019. URL: https://tools.ietf.org/pdf/draft-mizrahi-ippm-compact-alternate-marking-05.

[10] M. Cociglio, G. Fioccola, G. Marchetto, A. Sapio, and R. Sisto. «Multipoint Passive Monitoring in Packet Networks». In: *IEEE/ACM Transactions on Networking*. Vol. 27. no. 6, pp. 2377-2390. December 2019, DOI: 10.1109/TNET.2019.2950157.

[11] URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[12] URL: http://www.brendangregg.com/ebpf.html.

[13] URL: https://www.iovisor.org/.

[14] URL: https://github.com/iovisor/bcc.

[15] URL: https://it.wikipedia.org/wiki/IPv4.

[16] URL: https://flylib.com/books/en/3.475.1.30/1/.

[17] URL: https://www.spirent.com/products/testcenter-ethernet-ip-cloud-test.