POLITECNICO DI TORINO

Master's Degree
Computer Engineering - Networks

Master's Degree Thesis

"Big data post-processing of multipoint measurements with alternate marking method."

POLITECNICO
DI TORINO

Supervisor:

Prof. Riccardo Sisto

Co-supervisor:

Prof. Guido Marchetto

Candidate:

Calogero Corbo

March 2020

Contents

Chapter 1

Introduction

Nowadays, Service Providers offer more and more services very sensitive to network impairments, such as packet loss, delay and jitter. The applications based on voice and video streaming are a straightforward example of services that need a small delay and a low packet loss to work properly.

For this reason, a lot of work has been done towards fault detection and connectivity verification by IETF, Internet Engineering Task Force, but only a few works concerning performance monitoring.

In this scenario, service providers are trying to develop a performance monitoring system capable of providing network measurements as simply and effectively as possible. This would allow them to have an overview of their network parameters and to be able to troubleshoot breakdowns. Their aim, however, is not only to detect faults but also to locate where these occurred and for what "type" of traffic.

Because of this, several monitoring methodologies have recently been proposed. One of these is RFC 8321 [1], which will be better described later, which introduces the concept of alternate marking, the core of the whole implementation of the proposed methodology.

Basically, it is a passive performance monitoring technique, potentially applicable to any kind of packet-based traffic, including Ethernet, IP, and MPLS, both unicast and multicast. The method addresses primarily packet loss measurement, but it can be easily extended to one-way or two-way delay and delay variation measurements as well.

The proposed method performs passive monitoring, that is, observes the network and collects data, without making any changes.

Some of the advantages of these methods are:

- easy to implement: it can be implemented by using features already available on network nodes;
- accuracy: you can choose between different levels of monitoring granularity.
- robustness: this method has been designed to bear out-of-order packets.

RFC8321 [1] was proposed by TIM and subsequently a series of modifications has been introduced to deal with the various problems that emerged over time.

At the same time, TIM started, with the collaboration of Politecnico di Torino, a design phase of a new methodology for passive traffic monitoring, which is not limited to RFC 8321, but integrates other studies that have as their final aim to allow large-scale and extended monitoring of multipoint network (described below).

This project is known as the Packet Network Performance Monitoring PNPM [9][13]. In this context, a probe has already been developed which widely uses the extended version of BPF, eBPF (Enhanced Berkley Packet Filter) [2].

The Berkeley Packet Filter (BPF [3]) is a technology implemented in the Linux kernel that initially aims to analyze network traffic, but also provides some other functionalities. It provides a raw interface to NIC (Network Interface Card), permitting raw packets to be sent and received.

The biggest advantage of BPF is the possibility to inject ASM code in real-time. It allows to a userspace process to supply a filter program, written in a C-like language, that specifies which are the features that a packet must have to be used from some other applications. Instantly an injected ASM code instructs NIC, which is in kernel level, to discard not interesting packets, after fast processing. The result is that the unwanted packets don’t cross the kernel level, avoiding the gap introduced by context switching, greatly improving performance.

1.1 Goal of the thesis

This thesis aims to present a system architecture capable of meeting TIM's requests to take out performance measures starting from the raw data collected by the

network devices, through the probes. The main challenge is to find a solution that fits with the multipoint networks and that can scale up to involves large amount of flows. RFC8321 already defines by which method it is possible to perform measurements in a point-to-point network, but does not add anything on how to do with multipoint networks. The multipoint draft [13] care about the generalization of this methodology to the multipoint networks and, in doing that, provides some interesting notions about network topology partition and delay computation method.

Nevertheless, to achieve performance monitoring split by flows (a flow is a set of packets sharing some parameters, e.g. the same IP source header field in IP packets) both RFC8321 and multipoint draft, use a filter-based approach, namely a filter is installed on network device for each flow we want to monitor. However, this approach raises two issues:

1. The number of filters we can set up is limited and the total number of flows can go up to make this approach unfeasible.
2. Filter setup and configuration requires a considerable effort. To increase the monitored flows would require new filters and new effort and this would make this approach not scalable.

In this scenario this thesis arises. The objective is to keep the multipoint networks compatibility and to think up a different approach to achieve flow-based results overcoming previous issues.

The idea is to consider a single filter instance per network node that capture all passing traffic and get flow-based performance details a posteriori, by means of a big data system. This makes the measurement effort independent from the total number of flows. Furthermore, the possibility to split results by “group of nodes” (i.e. clusters) contextually with flow-based approach, raises the powerful of the methodology.

In this scenario, TIM wants to develop this method with some constraints. It should be as much as possible easy to implement, scalable and accurate. To achieve that I will describe hereinafter a passive/hybrid method of measurement that involves a lot of technologies, such as eBPF to build the probe, Mininet [5] to easily configure

and deploy a complex network on which to test, Apache Flume [6] to harvest data and send them to a storage point in an efficient way, Hadoop Distributed File System (HDFS) [7] to store data in a scalable way and some other technologies that I will describe better in the next chapters.

The architecture designed in this thesis is not final and is mainly used as a means to implement the methodology. The architecture, however, provides a reference model and may be different in a real situation, since in this case the problems to be addressed will be bounded to simulations cases.

Next, some simulations will demonstrate the potential and limitations of this implementation. They will not be accurate and will have the aim of clarifying the effectiveness and weaknesses of all the work done in this thesis.

1.2 Chapters content

The thesis is split in two main parts: the theoretical part and the emulation part. The first part illustrates the methods used to get measures, while the second one emulates a network that implements the architecture outlined in the first part and show results.

The following chapters covers different aspects, they are organized as follow:

- Chapter 2 presents an overview about performance monitoring with a special focus to RFC8321 and multipoint draft, that constitute the background of the thesis.
- Chapter 3 provides a general view about some of the technology involved in this thesis to develop various components.
- Chapter 4 exposes the principles of the new methodology, without examining in depth implementation detail. It is sort of manifest of the big data approach in performance monitoring.
- Chapter 5 describes the overall architecture deployed for simulation environment. It aims to describe how each component has been setup, showing technical details.

- Chapter 6 illustrates the workflows of the simulation environment: how each component interacts with the other one.
- Chapter 7 exposes the results, taken by some simulations. It aims to present the methodology powerful, without analyzing accuracy or validity of the measurements.
- Chapter 8 presents some ideas about how the work could go on and the main challenges I addressed in carrying out this thesis.

Chapter 2

Performance monitoring

Actually, most of service providers are focusing on what is the best way to gather details about carried traffic. What they need to know about it are details that emphasize traffic quality over time. Why this is a key point for the service providers is easy to understand: a good way to monitor traffic means easy maintenance and less effort to understand what and where do improvement, since it could potentially detect faults and weak points of their network infrastructure.

A lot of works has been done in this direction by Standards Developing Organizations [4], in particular about faults detection and troubleshooting, while a minor effort has been dedicated to network performance monitoring. Essentially what is meant when I say network performance monitoring is a way to record and show some network crucial parameters such as packet loss, delay, jitter.

When I say passive/hybrid method I refer to RFC 7799 [8] to define passive and hybrid terms. There is a difference between passive, active and hybrid methods. The active method needs to pump in special packets or to modify network traffic to get measures (such as PING utility that sends ICMP packets and wait to receive them to outline some statics), the passive method doesn't need any kind of additional packet but base its capability on network properties and packets analysis. Finally, the hybrid method combines both active and passive methods to achieve final results.

2.1 Alternate Marking Method

The technique that I will describe here has been developed by TIM and then standardized by IETF (RFC 8321). It aims to monitor a network with some precise features:

- Easiness: the method must be easy to implement.
- Lightweight: the additional workload must be minimized.
- Accuracy: It must be possible to determine how many packets have been lost, where (with a different granularity), and what is the delay of the received packet.
- Robustness: the method is designed to tolerate out-of-order packets and it's not based on special packet whose loss could compromise results validity.

  The proposed approach is based on counting how many packets a node sends, how many packets the next node receives and comparing these values. Because the measure makes sense, the counters must refer to the same set of packets, so we need a way to batch packets correctly.

  The solution proposed by RFC8321 to solve the problem is to mark packets with different "colors", so that each color belongs to a different batch. The number of markers needed to split packets and recognize to what blocks each one belongs to doubtless is two. Since we use an appropriate time to switch from one color to another, we could be able to distinguish precisely the packets of a block from those of the next. In fact, even if two colors might seem insufficient, what we want to achieve is to differentiate one block to another and not to record all blocks produced by the source of the packets (network node). In this way, packets belonging to the same batch will have the same color, whilst consecutive blocks will have different colors.

  In figure 2.1 an example of how traffic coloring works.

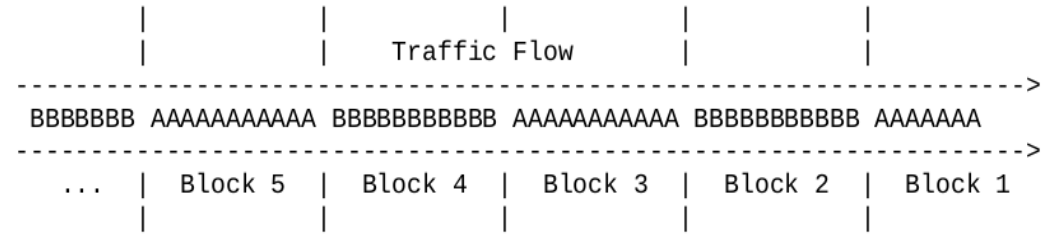


Figure 2.1 – Traffic flow coloring [1]

2.1.1 Marking the packets

The marking operation is the core of the entire methods, to generate blocks correctly. I will discuss in this paragraph where to mark packets and how to do it.

To understand what's the best way to mark packets I need to clarify what's the different strategies we can follow to implements method:

- flow-based: the flow-based strategy is used when we are interested to monitor a portion of the traffic. Only a subset of the flow that cross a node needs to be marked. A flow is defined by a set of packet's common features (e.g. the same header fields).
- link-based: when we want to monitor all the traffic passing from a link. We don't need to distinct packets and all the traffic might be potentially marked.

In case of flow-based measurements, the set of packets to monitor can be defined by selection rules (e.g. header fields). Marked traffic does not cross potentially any nodes available in the network, but a subset of them. However, each node of the network must be able to recognize marked packets, because we can't predict its path. In general, it's possible to have multiple coloring nodes or single coloring node, that is easier to manage and set up. However, coloring in multiple nodes can be done if the jobs are performed in sync, so we need a synchronization mechanism that ensures, with reasonable approximation, color switching is done without raise any conflict. In general, the most advantageous solution is to mark packets as close as possible to the source. In a backbone network a service provider may wish to monitor the entire traffic, so it could be clever to enable marking job in the edge nodes, so that all the traffic generated by customers can be marked and, therefore, monitored.

For the link-based measurements, all the traffic needs to be colored before to spread it across the link. In this case, each node must mark the traffic even if packets already are because of the color must be consistent on the link. The marking operations don't need to be in sync between devices.

Traffic coloring can be done by modifying some unused bit of packet header and changing it periodically. In this thesis, I will use the DSCP field, in particular some unused bit of it.

2.1.2 Switching colors

The RFC 8321 evaluates two possible approaches: one is based on a fixed number of packets and another on a fixed number of times.

In the first approach, the color switch after a counter detects a fixed number of packets. Even if this method seems to be easy to implement, it could be more difficult to manage because we can't predict when the marker will switch color. This approach could lead to a synchronization issue that has been left out in this topic.

The method based on a fixed number is more deterministic and for this reason, it is preferable. This is the way that Alternate Marking Method suggests adopting. In the next examples, I will use only this approach.

2.1.3 Packet loss measurement

After splitting traffic into blocks, each of these blocks can be potentially measured by all network devices along the path. The method is simple: each node counts the packet of a block and comparing these values with values recorded by next node we can detect if a loss occurs in any block between any two points (the numbers are not the same). This is true in a point-to-point path.

Suppose to have a network as depicted below:

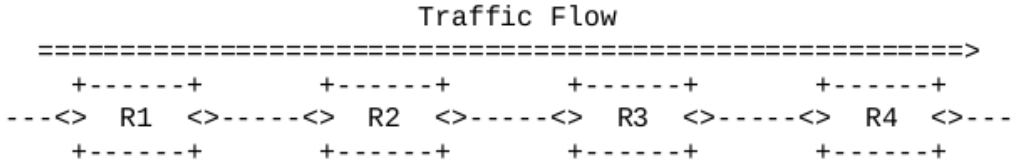


Figure 2.2 – Network example [1]

According to the method the traffic is marked with two different colors: A and B. Let's assume we want to monitor packet loss between R1 and R2. To do that, we need two counters per each router, in particular, router R1 need counter C(A)R1 that count packets marked with A color and counter C(B)R1 that count those marked with B.

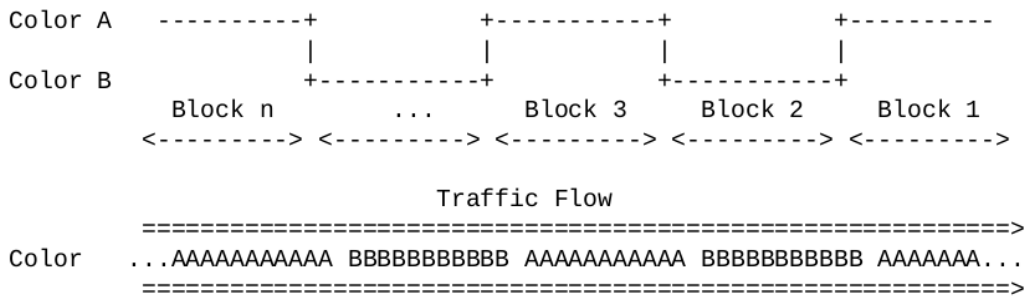


Figure 2.3 – Packet loss computation [1]

When marked traffic cross R1, C(A)R1 count how many packets are there, while C(B)R1 is stopped. Then, after color switching, C(B)R1 is counting, while C(A)R1 is stopped to the previous value.

The same job is done by next hop, R2. After that, the job starts again in a loop.

However, this simple job requires a synchronization issue. Both R1 and R2 must select the better instant when to read the counter value. In fact, if it is stored too early the risk of out-of-order packet may lead to wrong result. Since network devices perform this task simultaneously, a timing consideration are needed to preserve effectiveness. In addition to clock misalignment, R1 and R2 are affected by the delay between the moment the packet arrives at R1 and the moment it arrives at R2. A safe choice to overcome this issue is to wait L/2 time units, since L is the period duration (properly selected to avoid timing issues), after switching color, to read the counter of the previous marker.

The loss can be computed by comparing counters of R1 and R2 referring to the same block and color. Obviously, C(A)R1 must be greater than C(A)R2, so that total loss is equal to their difference.

Block	C(A)R1	C(B)R1	C(A)R2	C(B)R2	Loss
1	375	0	375	0	0
2	0	388	0	388	0
3	382	0	381	0	1
4	0	377	0	374	3
...	...	...	...	...	...
2n	0	387	0	387	0
2n+1	379	0	377	0	2

Figure 2.4 – Packet loss counters [1]

2.1.4 Counting packets

An important issue that is linked to the previous chapter is when to read the counter. To do that we must be sure that the counter has finished its task and has counted all packets of the same block.

What we want to do is to identify a time interval in which we are sure that if all nodes read in that interval, we can guarantee:

- The read value refers to the same block along with the entire network;
- Value is not affected by the out-of-order packet and relative counter is stationary;

These issues introduce to two main timing considerations: clock error, network delay.

If we fix a time in which all nodes must read counter, they will not do it exactly to the same instant, because of clock error between nodes R1 and R2. What we need is that they are synchronized to the same clock reference, with an accuracy of +- L/2 (L is block time duration).

Network delay introduces another factor to take into account because a packet can arrive at a different time to the routers. These may expose us to out-of-order packets so that the counter could be unstable.

Here a representation of previous issues, where L is the duration of the block, d the contribution introduced by delay.

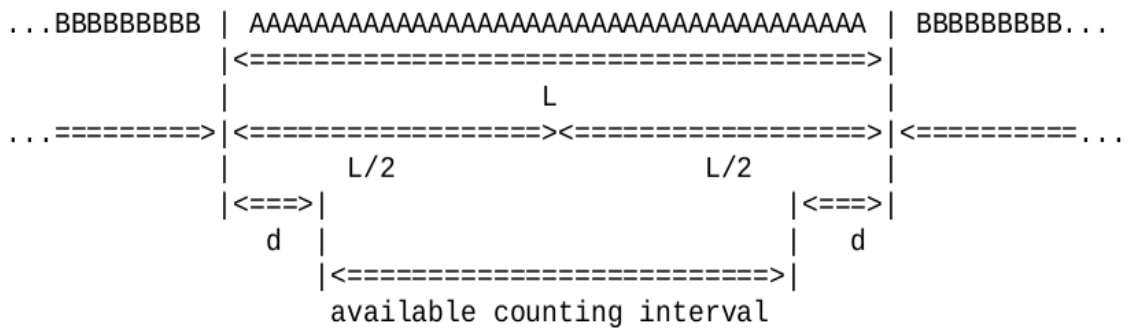


Figure 2.5 – Timing considerations [1]

In general, the most advantageous solution is to read counter at L/2, even if other timing aspects would affect the methodology. However, these considerations let's perceive that there is a minimum boundary for L length and that must be evaluated in every implementation. Choosing L long enough increases our chances to get the right value.

2.1.5 Delay measurement

The RFC 8321 can be applied to compute network delay, but it requires some consideration. From a theoretical point of view, the delay measurement can be achieved by comparing the timestamps recorded by two different nodes that refer to the same packet. For example, we can compare the timestamp of the first packet of R1 with color A (TS(A)R1), with the timestamp of the first packet of R2 with color A (TS(A)R2). If we postulate that the first packet to which TS(A)R1 refers is the same as what TS(A)R2 refers to, we can compute the delay of a block. We can iterate the job each N packets of the same block and do the comparison between timestamps. In practical this approach is not achievable, since packets

may not arrive with the same order with which they were sent. Possible packet loss could compromise our measurement.

A better methodology consists in calculating mean delay. Instead of comparing single timestamp measurements, we can compare mean delays between two routers. Each node records more timestamps and divides the sum of them for the number of measurements. This method is robust to out-of-order and loss packets, however, it only gives one measure per block, and we lost minimum, maximum and median delay values, which could be necessary to get the statistic distribution of delay measurements.

Another approach proposed in Alternate Marking Method is based on a double marking methodology. Some packets are marked one time and are used to compute mean delay, while double marked packets to obtain other measures that give us a static distribution of delay measurements.

2.2 Multipoint Marking Method

In principle, RFC 8321, contains all the theoretical elements to fit with any kind of unicast flow, which definition is inspired by [10]. However, the guidelines to implement method lead to a solution that works only in a point-to-point scenario, because it assumes that all the packets detected by one node are detected again by a single second node.

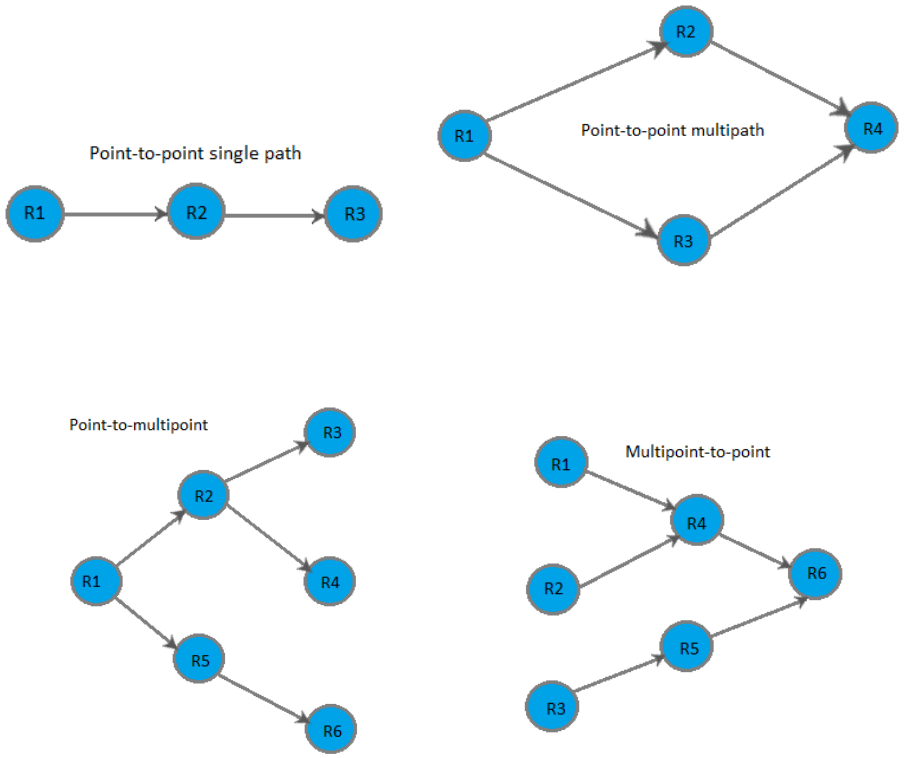
To apply the current methodology to a real network, we must consider a more complex scenario, where the flow can follow a point-to-multipoint path, multipoint-to-multipoint path and so on.

The Multipoint Marking Method [9] aims to generalize the Alternate Marking Method to all possible unicast flow, including the multipoint-to-multipoint path. What has been done in this approach is to introduce the concept of cluster. With this intuition, we can perform network monitoring with a different degree of detail. We can analyze a large portion of the network and if a packet loss occurs, we can zoom monitoring and detect where the problem happened, with an in-depth analysis.

2.2.1 Flow classification

A unicast destination means that the packet is sent to one and only one destination. A unicast flow is a set of packets having a set of common characteristics. As an example, a set of packets that share the same IP source and IP destination belong to the same flow; their path may not be a point-to-point path, depending on the network topology and on the source and destination location.

Here (figure 2.6) I depict all the possible paths that can we encounter in a unicast flow and that is treated in the Multipoint Marking Method.



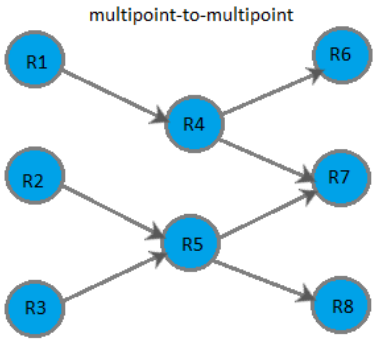


Figure 2.6 – Possible paths

Note that the current method covers also anycast flow because it does not introduce any packet replication.

2.2.2 Packet loss measurement

The network nodes can be of three types: input nodes, output nodes, intermediate nodes. Considering that, probably, all these nodes belong to a backbone network, input nodes do not correspond to devices that generate traffic, but the first nodes through which traffic flows. These nodes can be designated markers so that their job is to color each input packets according to Alternate Marking Method. In the same way, output nodes are the last nodes through which traffic flows and comes out to the monitored network. All the intermediate nodes are exploited to understand what happens in the middle with different degrees of granularity, thanks to the clustering approach, that I will describe in the next paragraph.

It is easy to understand that each input and output nodes can reverse their role so that input becomes output and vice versa, it depends on which generates traffic and towards to. In general, they belong to the edge category nodes.

To perform packet loss measurement, we can consider a principle: the number of packets counted by the input nodes is always greater or equal than the number of packets counted by all the output nodes.

We define the network packet loss for 1 flow, for 1 period, as the difference between the number of packets counted by all the input nodes and the number of packets counted by all the output nodes, in 1 period and per 1 flow.

We can formalize the Monitored Network Packet Loss with the following formula:

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

where:

- n is the number of input nodes
- m is the number of output nodes
- PL is the Network Packet Loss (number of lost packets)
- PIi is the Number of packets flowed through the i-th Input node in this period
- POj is the Number of packets flowed through the j-th Output node in this period

The equation is applied on a per-time-interval basis.

2.2.3 Network clustering

Whatever subnetwork that keeps the same property is called cluster.

It's important to highlight that both measurements and clustering refers to a monitoring network, that is deduced from the real network. The monitoring network is a graph whose nodes are represented by all network devices that are Measurement Points (MPs) and arcs are all the links that connect directly or indirectly (if passing through a network device that is not MP), each MPs to another. In a completely monitored network (where each network device is an MP), the monitoring network corresponds to a real network.

Exists multiple algorithms that can perform cluster partition, as reported in multipoint draft [13], but I will describe that one I think is the simplest.

It has two steps:

(1) Group the arcs sharing the same starting node;

(2) Merge each group with any other that have the same ending node.

Suppose to have the following monitoring network:

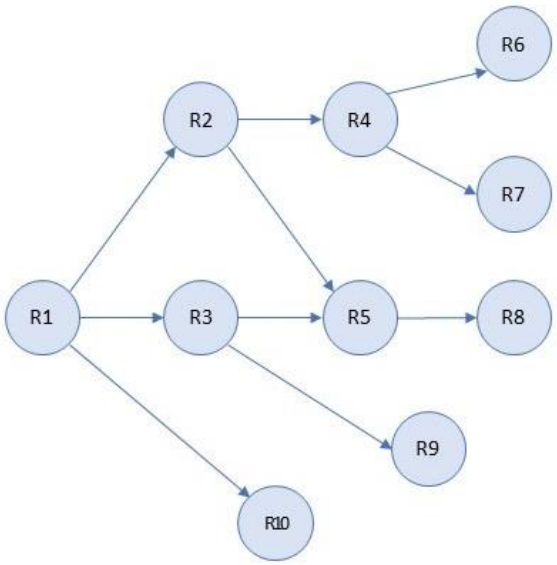


Figure 2.7 – Monitored network

After step (1) we will obtain:

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

Then step (2):

1. Cluster 1: (R1-R2), (R1-R3), (R1-R10)
2. Cluster 2: (R2-R4), (R2-R5), (R3-R5), (R3-R9)
3. Cluster 3: (R4-R6), (R4-R7)

4. Cluster 4: (R5-R8)

At the end this is the result:

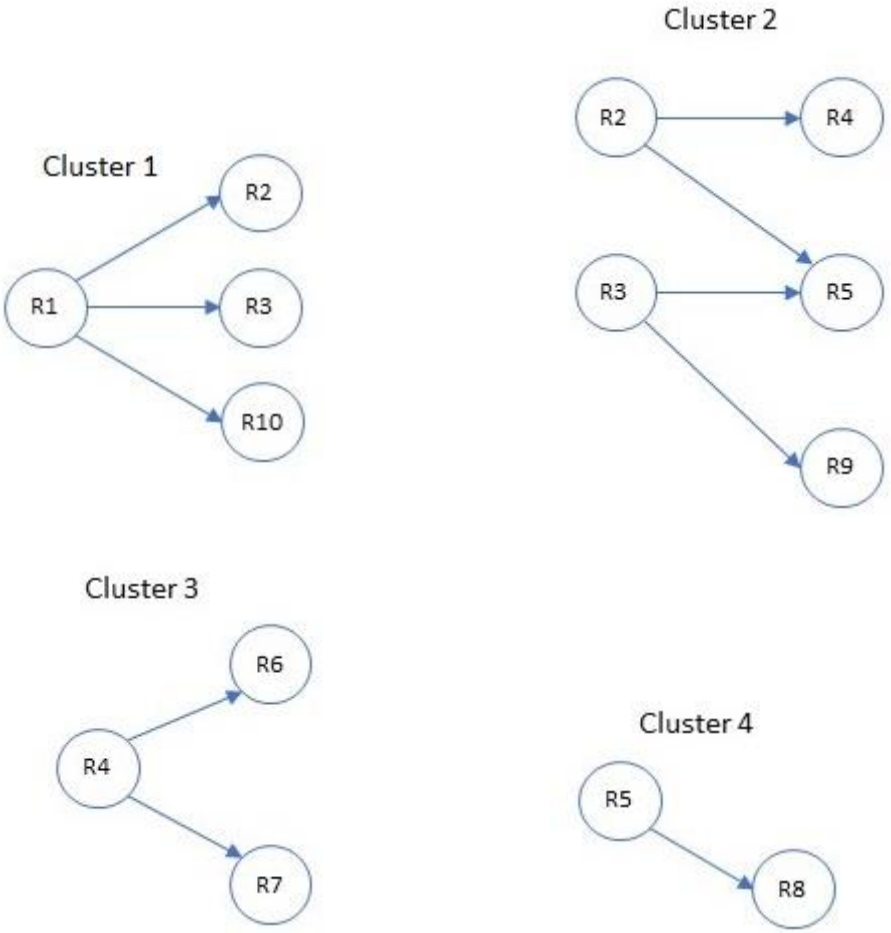


Figure 2.8 – Clustering of monitored network in figure 2.7

It's worth mentioning that all these clusters converge to the biggest cluster, which is the entire monitoring network.

In general, this algorithm, splits the monitoring network as much as possible, finding the smallest clusters. Two clusters can join to converge in a wider cluster if the output nodes of one overlap (or are a superset) with the input nodes of one other.

For example, cluster 1 and cluster 2 can join because of outputs of CL1 are (R2, R3, R10) and inputs of CL2 are (R2, R3), which is a subset of the first. To verify the validity of this assumption it's enough to apply the cluster property mentioned above.

2.2.4 Delay measurement

Delay measurements include mean delay and mean delay variation.

For this purpose, it's useful to mention the hashing method [11][12], that will be widely used in my implementation.

What we want to achieve is to get the same advantages of the double marking method in a single point-to-point path and extend them to a multipoint path.

The goal is to capture a set of packets, to record their timestamps and use them to perform delay measurements related to the period that the Alternate Marking technique introduces.

There are two possible methods that we can use to achieve the goal, both based on hashing. One is the basic hash and the other one dynamic hash method.

Both methods exploit the Alternate Marking Method to get the blocks and then create a set of hash samples by applying some hash function to the packets. The main difference is that in dynamic hash we can select the maximum number of sampled packets per period, while we can't in the basic one.

The algorithm of dynamic hash sampling needs a random number of which we want to match a variable number of bits. We select an NMAX value, that means how many packets, at maximum (more or less), we want to sample. For each input packets, we apply a hash function to the entire packet and start trying to match 0 bit between hash value and a random value. If it's true, we store the hashed packet and add 1 to a counter that reminds us how many samples we have currently. When the counter reaches the NMAX value, we increase by 1 the number of bits to match, so that, now, in a probabilistic way, half of the previous samples will be discarded (we reach NMAX/2 total samples) and the new hashed packet must match 1 bit with the

random value, that means probably half of the new packets will do it and the other ones will be discarded. The algorithm starts again increasing the number of bits to match and loops until the current period ends. When a new period occurs, it restarts from 0 bit to match.

Note that, if M is the number of bits to match for each hashed packet, the probability that it happens is 2^-M. Furthermore, total captured packets are between NMAX/2 and NMAX, but they can slightly exceed NMAX or be less than NMAX/2.

The dynamic approach scales the number of measurements per period. This aspect makes dynamic hash preferable than the basic one.

2.3 An in-depth clustering analysis

This section summarizes the method presented by [13]. As reported in paragraph 2.2.3, to determine the clustering partition we start from the monitored network.

Let's define a monitored network as a directed graph G = (N, A) with a set N of n nodes and a set A of m directed arcs. Considering that each network interface i can receive or send traffic, it's modeled with two nodes $i_o \in N$ and $i_i \in N$, one as input point, the other one as output. Since an input node is a node that can receive packets from other network devices, it has one incoming arc from outside and one outgoing internal arc per each router interface, also considering itself (this assumption considers the case a packet gets in and gets out from the same interface), directed to all the output nodes of the same router.

At the end of this analysis, known the number of total links p, total network device q, each one with r_i interfaces, with $i \in \{1, \dots, q\}$, we can calculate the total number of nodes n and arcs m of graph G:

$$n = 2 \cdot \sum_{i=1}^{q} r_i \qquad m = 2 \cdot p + \sum_{i=1}^{q} r_i^2$$

Once we selected the interfaces to monitor in the real network, we can determine the network monitored topology, namely the new graph $\bar{G} = (\bar{N}, \bar{A})$, with $\bar{N} \subseteq N$ that include only the nodes corresponding to the monitored one, and with a set of arcs $\bar{A}$

where each arc (i, j) is a possible directed path in G between two nodes $i \in \overline{N}$ and $j \in \overline{N}$, not crossing any other node in $\overline{N}$.

In figure 1 an example of how to get clusters from a real network.

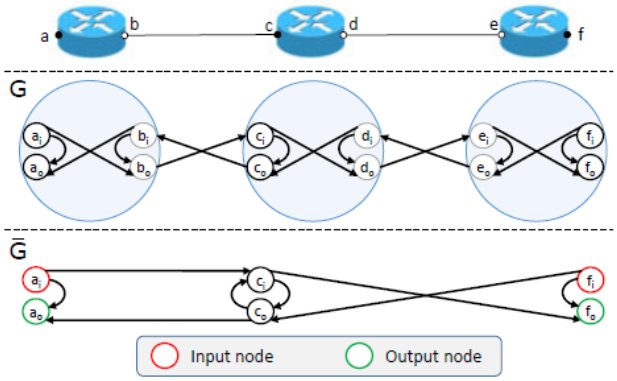


Figure 2.9 - Real network, intermediate network (G) and monitored network ($\bar{G}$). [13]

To find all the arcs in $\overline{A}$, we can proceed by applying Dijkstra's algorithm to check the existence of a single path.

As has been discussed before, a network $\overline{G}$ has three types of nodes: input, output, intermediate. This network has a crucial property: the number of packets entering by input nodes P_{in} is equal to the number of packets exiting from the output nodes P_{out}, if it isn't, their difference is the total network packet loss *L*.

$$L = P_{in} - P_{out}$$

A cluster $\tilde{G} = (\tilde{N}, \tilde{A})$ is a minimal subnetwork of $\overline{G}$ that satisfies the equation above.

Since we have the monitored network (i.e. $\overline{G}$), we can use the algorithm discussed in paragraph 2.2.3 to find all the clusters.

(a) Cluster C_1. (b) Cluster C_2

Figure 1.10 - Clusters deriving from the network depicted in figure 2.9. [13]

Chapter 3

Technologies

Developing this thesis involves a lot of different technologies, some in common use, others developed specifically.

A software called IOVisor-PNPM (IOVisor Packet Network Performance Monitoring) [14] has been developed for the same scope by another thesis student and it represents the core of the probe system. It allows capturing packets in three different ways (one of these is the dynamic hash method), to select what kind of packets we care and to do an initial analysis about gained data. However, the IOVisor-PNPM, that constitutes the probe, has raised some compatibility and functionality issues, that has been faced successfully and that are reported in paragraph 3.2.5.

A storage system is required. It should have some features:

- Scalable: in a real implementation the system could evolve quickly. So, a solution that could scale and increase its capacity with a minimum effort could make a difference.
- Large capacity: we potentially will store a large amount of data. It's not possible to forecast how many they are, but we can expect at least hundreds of gigabytes in a real deployment.
- Remotely managed: to examine data afterward, it's recommended a management system that can be accessible remotely. It's a key feature in our context because it makes the system easy to use.

Hadoop Distributed File System (HDFS) has been chosen in this implementation to satisfy all these requirements.

It is a distributed file system that can be easily deployed in a commodity hardware with minimum effort and that can be moved to other platforms by changing configuration parameters, besides the devices it works with. Furthermore, it has been designed to be highly fault-tolerant, a feature that makes it preferable to other distributed file systems.

Related to HDFS, Apache Flume has been used to automate the system as much as possible. Its functionalities have been exploited to connect the monitored system with the storage system. The data are pushed in the storage system automatically after the probe builds them.

Furthermore, Mininet has been widely used to emulate a complex network and test the system with different topologies. Leveraging its features, it has been possible to deploy many routers to carry on packets and generate several traffic flows, that are useful to test the proposed methodology.

In this chapter, I will briefly describe the main aspects of these frameworks.

3.1 Hadoop Distributed File System (HDFS)

A distributed file system is a file system that allows storing data in distributed devices across the network instead of a centralized system. Data lay to more than one network nodes, though the system is accessible in a way that makes the network transparent and allows to read/write data as if there were just a single node.

HDFS is one of the core components developed in the Hadoop Project. It has been thought out comply with the following assumptions:

- *Failure resistance*: hardware failures are common in a data center and can be potentially very dangerous. The more servers there are, the more likely it is to fail.
- *Batch processing*: this system has been optimized for streaming data access, that requires batch data rather than interactive use by users.

- *Large data*: on HDFS a typical file is a gigabyte to a terabyte in size. The system works also with small data, but the large latency introduced to increase the throughput makes this choice disadvantageous in terms of the total amount of time required to perform operations.
- *write-once-read-many*: once a file has been put in HDFS, it does not need to be changed. This principle simplifies the data coherency model and enables a high throughput of data access. When we perform computation on stored data, the programming paradigm deal with this requirement: they typically do not modify existing data.
- *Move computation, not data*: the computation is performed near data. Rather than moving large amounts of data over the network, executable files are displaced as closest as possible to data, whose size is exiguous. This aims to increase the overall throughput and network congestion.

3.1.1 HDFS Architecture

The HDFS has a typical master/slave architecture. Its main component is the cluster, a set of servers that manage data according to HDFS assumptions.

A cluster has a Namenode, the master server that manages the file system namespace and allows the client to access files and folders, and a set of Datanodes, which manage storage on servers that they run on.

The Namenode takes care of executing file system operations, like opening, reading, etc. Its metadata allows the client to access data unaware of the distributed infrastructure; the client can perform operations as if there was a single centralized system.

Datanodes store files split into more blocks, each one of the same size, except the last. Also, they perform instructions coming by Namenode (master), related to blocks management.

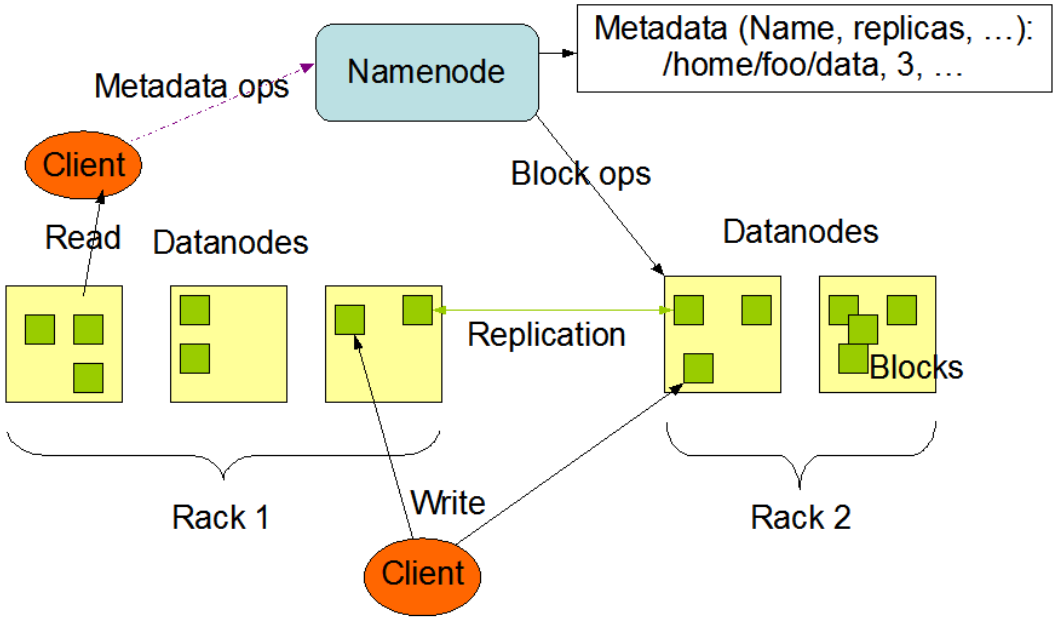


Figure 3.1 – HDFS architecture [7]

In figure 3.1 the overall architecture of the HDFS. The Namenode manages the system and store metadata; Datanodes perform the requests coming from clients and execute Namenode's instructions. Datanodes are grouped in Rack (a set of Datanodes) and per each block a configurable quantity of replicas are distributed over the servers (take care that usually there is one Datanode per server).

3.1.2 Blocks replicas

Block replication is the key point to reliably store data. Each file is split into more blocks and more copies per block are created and stored in some Datanode. The replicas allow fault tolerance. The block size is equal for each block, except the last, and is set by the client. The replication factor can be also set by the client at file creation time. These parameters are stored in Namenode.

The replica placement is one of the points that distinguish HDFS from other distributed file systems. The assumptions since HDFS starts its implementation are that network bandwidth inside single Rack is highest than across different Racks and that a rack failure is less frequent than node failure.

HDFS exploits the Hadoop Rack Awareness process to assign a rack id to each Datanode. Since that, after the replication factor has been chosen, the pattern is to place at least one block in a rack, another to a different rack and (if there is) a third to the same rack of the second block. This policy improves write performance without compromising data reliability and read performance. Eventually, the other replicas are placed across other racks.

After that, when a read request is coming, HDFS tries to select the block replica closest to the reader, optimizing the global bandwidth consumption.

3.1.3 Robustness

The robustness is the main aspect of HDFS and for this reason, it deserves a separate paragraph.

Essentially, the three common types of failures, in an HDFS architecture, are Namenode failure, Datanode failure, and network partitions.

Under normal operating conditions, each Datanode sends a Heartbeat message to the Namenode periodically. When a network partition occurs, some Datanode can't communicate with Namenode. The Namenode can understand that it happens because it does not receive the Heartbeat message and exploiting its metadata it understands what Datanodes have been lost. From this moment on Namenode marks those Datanodes as unreachable: if the replication factor of some lost block falls below the specified threshold, it tries to replicate them by picking up from where they are just alive (i.e. from Datanode to another rack).

Starting the replication process can be due to several factors: a block has been corrupted, a Datanode becomes unavailable, a hard disk on Datanode fails.

Also, a checksum value is computed since a block is built and stored. This value is stored in a hidden place across the HDFS structure and when an I/O operation is requesting a block, the checksum is re-computed and then compared to the previous checksum, if they match the system proceed with the operation, otherwise, the block is marked as corrupted and try to fetch another replica of the same block. This is a

smart method because it verifies data integrity and if it detects trouble, try to solve the issue when possible.

Namenode, unfortunately, is a single point of failure. For this reason, the HDFS keep more copies of Namenode's metadata, because they are the system core. When one copy is updated, then all the other copies are updated in sync. If the Namenode crash, then manual troubleshooting is needed, actually are not available better solutions.

3.2 Probe

It is clear, from the Alternate Marking Method, that a packet collector system is essential. It is possible to write a simple script to capture and count packets passing through a router. However, it presents at least two issues:

- Throughput: the total packets per time unit that a user space script can compute is limited by the continuous context switching between user space (where the script is executed) and the kernel space (where the network interface receives packets), an expensive machine job that may degrade overall performance if occurs too many times.
- Timestamp: not all devices can provide packet's timestamp.

The IOVisor – PNPM (Packet Network Performance Monitoring) has been designed to overcome these issues and maximize performance. It exploits the eBPF technology and related virtual machine on the kernel side to perform packets collecting.

This IOVisor – PNPM has been developed in a previous thesis and then updated to fit with new requirements and technologies.

The enhanced Berkley Packet Filter (eBPF) is a tool that has been introduced to perform high throughput traffic analysis. In the last few years, it has become a versatile tool that can execute generic code, leveraging the eBPF virtual machine. The injected code is subject to restriction, due to security reasons, such as limited code size, no loop, etc.

IOVisor is an open source project that focuses on networking. Its community aims to share, develop and spread virtualized in-kernel IO services. BCC is a toolkit that has been developed by IOVisor and aims to simplify the writing, validating and compiling phases of coding in eBPF. They have been widely used during IOVisor – PNPM development.

3.2.1 Probe architecture

The IOVisor – PNPM can communicate with the external environment through the NIC (Network Interface Controller) as input, and some REST API that can be used to configure parameters, start collecting, read the output.

The system is split into two sections, one in user space that can be managed through API, the other one in kernel space that executes the eBPF programs.

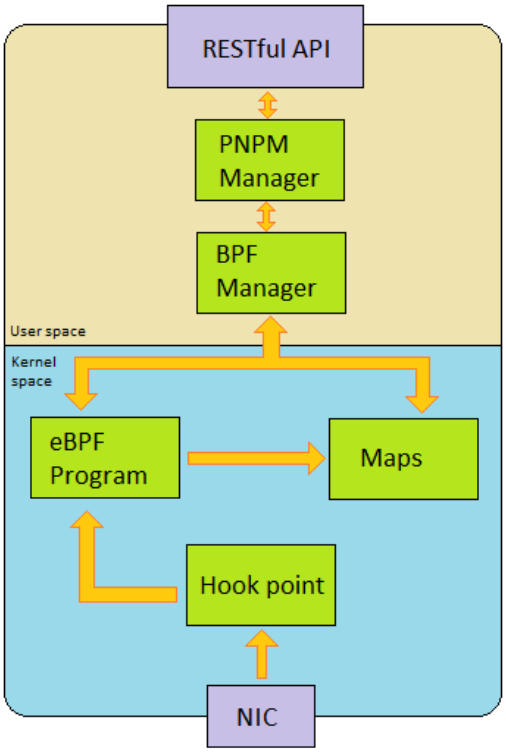


Figure 3.2 – Probe architecture

Figure 3.2 illustrates the IO – Visor PNPM architecture. Each one of these components has a specific task and in the next lines, I will briefly describe them.

- NIC: the Network Interface Card is the input point of the system. It's the hardware component that physically intercepts incoming packets and then sends them to the next hop.
- Hook point: when the system receives a packet, a specific kernel function is called. The hook point can "wakes up" the eBPF program when that kernel function is raised. It basically hooks a kernel function to an eBPF program. Consider that this hook point wakes up a program only when a packet is received, not when sent. To allow to capture packet in output a trick has been used in Mininet, rather than change a lot of eBPF code.
- eBPF Program: the program can access the packet. The information obtained will be manipulated and then stored in maps.
- Maps: they are the tools that eBPF designed to link kernel space and user space. Each one of them can read/write maps.
- BPF Manager: this is the first component in user space. It manages eBPF programs and determines which configuration to run. Its methods concern the creation and configuration of the program, the initialization of maps and data reading.
- PNPM Manager: interacts with BPF Manager and client, through the REST APIs. It implements the monitoring logic and coordinates the operations between the other components.
- REST interface: it constitutes the entry point of configuration parameters. A client can manage the system exploiting the exposed APIs.

3.2.2 Workflow

In this section, I will describe the IOVisor – PNPM workflow and how its components work together to start packets tracing session.

1. An external client interacts with the system through REST APIs and provides a configuration file in which specifies general info, such as what kind of eBPF program to start, what filter to apply and so on.
   The PNPM Manager initializes the correct program and related struct.

2. The BPF Manager interacts with maps to initialize filter correctly. If all these steps are successful, the IOVisor – PNPM ends configuration steps and it's ready to start packets tracing.
3. The program is ready, but it begins its job only when the client communicates, through proper API, to start the session. After that, IOVisor – PNPM injects the eBPF program in a virtual machine and start a parallel thread to manage the program.
4. From now on, when the NIC receives a packet, after the attached kernel function is reached (thanks to hook point work), the eBPF program starts its job and do it for each incoming packet.
5. The eBPF program starts. It verifies that incoming packet matches with a specified filter if it's false the packet will be discarded without further processing. If true, the program assigns values to structs and update maps.
6. The user space program read periodically maps, stores them in user space structs and makes it available for the client.

3.2.3 Working modes

The IOVisor – PNPM provides four working modes, the simplest, zero prog, is used just only to debug.

- Prog one: it analyzes all the packets that match with filter and compute measures related to block and not to single packets. It provides the total amount of packets detected, mean timestamp, first timestamp, last timestamp.
- Prog two: the fixed hash method. In this case, the hash sampling technique is applied to select a variable number of packets. After the hash is computed, the program checks if a fixed number of bits of that hash match with the hash reference. If it happens, other measures are derived from packets, related to block yet, otherwise, they are discarded. The final result is the same as the prog one, but it refers only to sampled packets and not to all incoming packets.

- Prog three: the dynamic hash method. In this configuration, we specify, through JSON format, a filter, a hash reference and the expected total packets to sample.

  After the packet hash is computed, the program verifies that a dynamic number of bits match with the hash reference. If true, a counter records how many sampled packets we have and if this value exceeds the number of total packets we need, the total number of bits to match increase by one. If it happens, previously sampled packets may be invalid, should be rechecked. It does not happen in kernel space, due to eBPF limitation, but it's performed by the program running in user space. The total number of sampled packets is between N/2 and N, if N is the total packets we want to achieve.

  The available measures refer to single sampled packets. The prog three is the one used in this thesis with modality "catch-all", to track all traffic and get info.

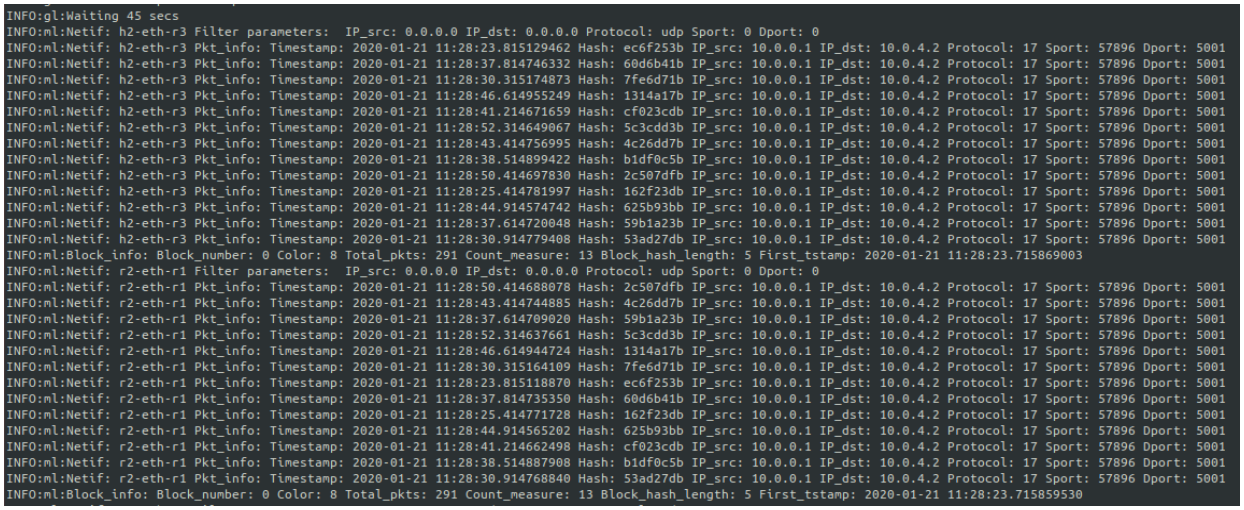


Figure 3.3 – Probe output

In figure 3.3, an example of measures coming from prog three. We can see the network interface name, sampling timestamp, hash, and other data.

3.2.4 Performance

A series of tests were carried out to verify the performance limit in single core and multi core environments, changing the size of the analyzed packets.

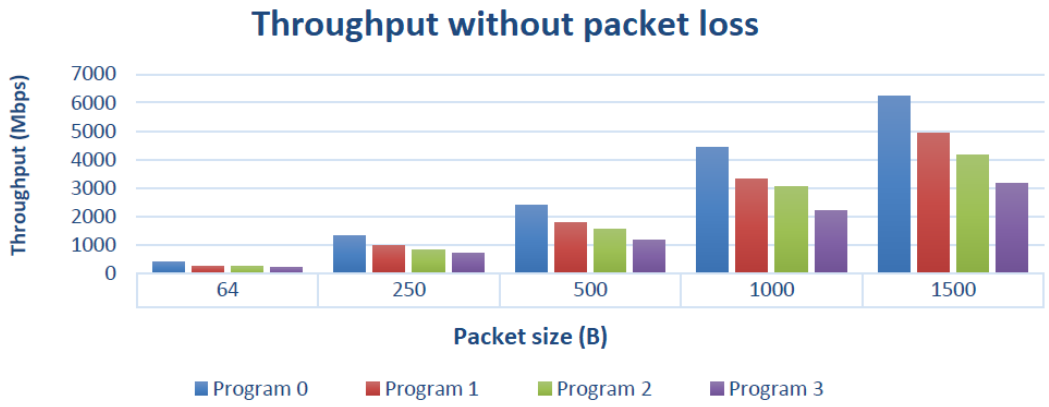


Figure 3.4 – Probe performance single core [14]

In figure 3.4 the results in a single core environment. As we can see the program 3, which is the most complex because of the dynamic hash method, is the slowest, even if its throughput is towards 3 Gbps. Also increasing packets size, performance worsens linearly.

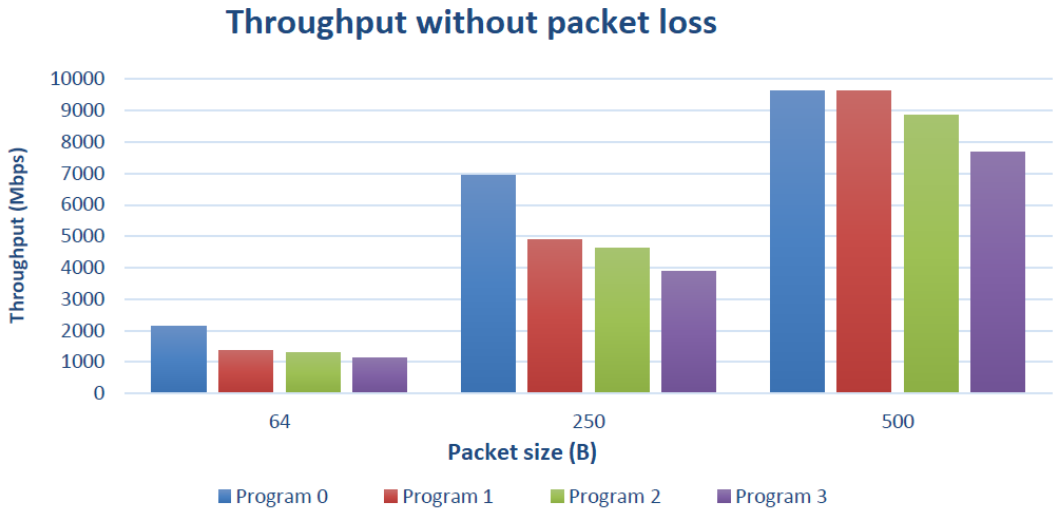


Figure 3.5 – Probe performance multi core [14]

However, figure 3.5 shows that in a multi core environment, performance improves significantly. That means the probe can scale with a multicore architecture.

3.2.5 Limitation

The probe has been designed in 2017 and this makes it old. It was thought out to be used differently from now.

While initially it was thought to select by flows, now the probe captures all traffic (simply one flow that includes all packets), without distinction. Furthermore, updates of the Linux kernel raised incompatibility issues, which were solved after a review of the probe.

In particular, the changes related to compatibility issues are two:

- Since linux kernel v. 4.9 sk_buff.tstamp.tv64 has been changed into "sk_buff.tstamp". The "struct sk_buff" has been modified and the tv64 field has been merged into tstamp. This changes are available inside "dyn_hash_noloop.c" file. To clarify:

  From

  struct sk_buff myskb;
  update_block_measure(&mf, fgi, mark, hash_low, hash_high , myskb.tstamp.tv64, &pf);

  To

  struct sk_buff myskb;
  update_block_measure(&mf, fgi, mark, hash_low, hash_high , myskb.tstamp, &pf);

- Another one is related to eBPF side. After several attempts we discover that the double in-line addressing by pointer is not allowed in some eBPF calls like "bpf_probe_read(…)". So, we modify the previous implementation simply by splitting the double in-line addressing into two single addressing. However, we didn't find the reason it happened, and it could be an issue bounded only to my device. To explain better I report an example of before and after modification:

From

```
u16 ifindex = skb->dev->ifindex;
bpf_probe_read(&nn.netif, sizeof(nn.netif), skb->dev->name);
```

To

```
struct net_device *dev;
dev = skb->dev;
bpf_probe_read(&mf.netif, sizeof(mf.netif), dev->name);
```

But the main problem, in this case, concerns network interface filter. Since initially it has been designed to do that, it has been tested on real device interface, but actually, with Mininet emulated interfaces it seems to not be able to detect the latter. This imply that probe raises the eBPF attached program whatever interface is involved, even if it has been specified in the configuration parameters not to consider it.

This drawback would require changing the eBPF side probe to recognize network interfaces properly (i.e. recognize Mininet emulated interface as they were real hardware interface), discard unnecessary ones and works only on the specified ones. To speed up the work of the thesis, it was preferred to make the changes on the python side, much simpler to manage and modify. So, when a packet reaches whatever interface, the attached eBPF program performs computations, then results are sent to the user space program, and here they are distinguished between useful or unwanted. If they are classified as useful, they are stored into PNPM structures, otherwise discarded. PNPM manager has been modified to fix that.

The reason why this job is simpler in python-side than in eBPF-side is related to the different way of recognizing network interfaces. On python-side it is possible to distinguish by name and validate an interface if it matches with one among those specified as input, in eBPF-side the recognition of the interfaces is based, at least in this implementation, on the search in a specific folder of all the interfaces available in the system. The problem is that the files in this folder show only the real interfaces and not the virtual ones.

The result is that the probe now processes data on all packets on all interfaces in eBPF, but subsequently discards unnecessary data on the python side. This makes the probe much less performing than it was initially. However, this is still an

acceptable solution, because performance optimization of the solution is out of the scope of this thesis.

Here the code part I added to “pnpm.py” after I defined the nodes list properly, based on results “clusters.json”, that is the output of “CompleteIterativeClustering_v5.py” file:

Line 367 - if netif in nodes:

The next problem was packets detecting. The probe is able to detect and then start its job, only incoming packets, while to make the system working we need to detect both incoming and outgoing packets. To overcome this trouble, I changed edge topology by adding two switches and one router per edge, that can only receive packets and that outcome from one of the two routers that are start and end of the edge. A detailed description is available in chapter 5.

Chapter 4

Scenario and methodology

In this chapter, I will describe the scenario and the methodology that can be used to get performance details from the monitored network.

The following description is inspired by the concepts illustrated in the previous chapters, in particular RFC8321, Multipoint Alternate Marking Method ([I-D.ietf-ippm-multipoint-alt-mark]), and Hash Sampling (RFC 5474 [RFC5474] and RFC 5475 [RFC5475]). I will describe in which case the system is applicable and what are the prerequisites, then how it works and a proceeding to process input data and achieve performance details.

Before to get into further details, is useful to improve the readability of this chapter, to provide a definition of flow.

A **flow** is a set of packets sharing some common characteristics. Its definition is related to the working context; for example, in an IP network, a flow is defined as a set of packets sharing some header fields (e.g. the same IP source and IP dest). In TCP packets a flow could be a set of packets having the same port destination and the same IP source, or the same port, and so on.

A flow can be split in smallest flows. Let's just consider only IP networks. Consider a flow as a set of packets that have IP source equals to 100.100.0.0/16, the same flow includes further flows, for example one composed by packets having IP source equals to 100.100.128.0/17 and one that includes the packets sharing 100.100.0.0/17 as IP source address.

Nevertheless, a flow can be extended to include all packets. If we consider a flow as a set of packets sharing the same IP source, that is 0.0.0.0/0, we are including all traffic.

However, the smallest flow in an IP network is identified by the quintuple header fields IP source, IP destination, Port source, Port destination, protocol.

The measurements refer to a flow that we specify when asking the system to perform them. They can be split into two main categories: per cluster and end-to-end. The first is all the details that refer to every single cluster and provides a list of parameters that characterize it (packet loss, mean delay). The second category provides more general info about the entire path (packet loss, mean delay).

The results are provided on-demand, in a not real-time processing environment, and each one of that refers to a single monitoring period, even if it is possible to broaden your search to more periods, and a single flow, until to select a flow that includes all the packets if necessary.

Packet sampling is performed on all incoming traffic, without any flow distinction. Nevertheless, thanks to data postprocessing, results are split by flow afterward, since the storage system memorizes packet's fields that identify its flow. Final queries require as input parameters the flow identification fields, as well as the time to which they refer.

The use of sampling improves packet tracking performance and thus overall performance. It allows you to track the path followed by each packet without further efforts by the Network Management System.

A partial system simulation will follow in the next chapter.

4.1 Background

I assume that a service provider's network is made up of a main backbone network surrounded by routers that handle customers' traffic input and output.

The proposed methodology requires that the traffic is marked before getting in the backbone network, following the Alternate Marking technique. The marking process can be made by the edge routers (as depicted in figure 4.1) or by the customers themselves, keeping in mind that it requires the markers are synchronized.

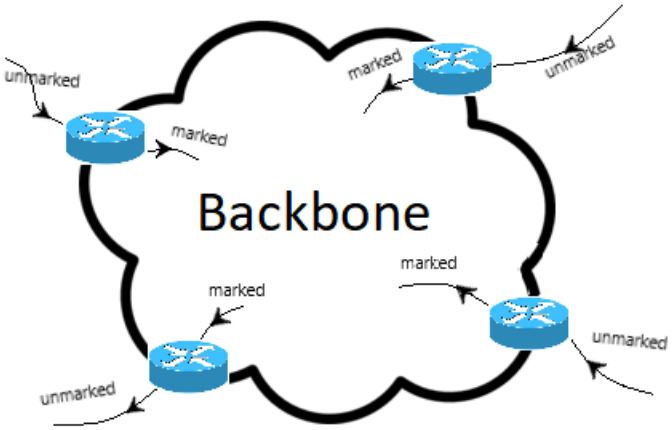


Figure 4.1 – Backbone marked network

Only marked traffic can be monitored. It is possible to mark traffic partially; the results will not be affected by unmarked packets and will refer only to marked ones.

To do that a time reference period and a marking method must be fixed at the beginning. The time reference period must consider the misalignment between the marking source routers, clock error between network devices and the interval we need to wait to avoid packets being out of order because of network delay.

A possible marking method could be to use two bits of the header and set them to 0x01, to identify a period, and to 0x10 to identify the next one. This allows to distinguish marked traffic with unmarked traffic, instead of using just only bit that can generate misunderstanding between the unmarked traffic (that has the special bit set to 0 by default) and the marked traffic (that alternate between 0x0 and 0x1, with 0x0 as marker value and not as default).

Otherwise, is possible to use just only one bit to mark, and use a filter based on IP address, which distinguishes the flows to be monitored from the others.

To enable monitoring, the system requires also the probe, the software installed on the network device that collects measurements. The portion of the network to be monitored must be surrounded by routers with a probe installed on. The rest of the network can't be monitored even if the traffic is marked.

So, the size of the monitored network depends on the network devices' placement. However, the size of the network surrounded by probes must be less than or equal to the size of the network with marked traffic.

4.2 Steps

The method described here consists of the following steps:

1. **Data collecting**: the probe analyze data passing through a network interface. A probe needs to be placed into each router interface we want to monitor.
   The software is configured by setting the reference hash, period duration, the two values that identify the marked flow, the maximum number of packets to store, interface to monitor and flow (identified by header IP fields) to monitor. Note that the flow can be set to includes all packets.
   Each incoming packet is hashed, compared with the reference hash and recorded if the number of bits that are matching is the same that the packet collector requires. When the number of matched packets exceed the maximum number we request in configuration, the number of bits to match increase by one. At this point, all the previously stored packets could be potentially discarded and must be rechecked.
   Until here data are stored temporally and are subject to changes, discards, and additions. After the period ends, previous data are still subject to change, but after a guard band (reasonably L/2 if L is the period duration) data are stored permanently and ready to be sent.
   Please notice that the packet collector (carried out with probe) isn't aware of what flow a packet belongs to, but it works with every incoming packet, without distinction.
   This setup greatly increases the probe configuration easiness. Otherwise, the probe should save all possible flows (potentially too many), which would be too expensive for the device, and need to be reconfigured if a new flow is available for performance monitoring.
   Stored data includes two kinds of details: one refers to every single packet and the other are aggregate measures. The first data includes: fields that

identify the flow (IP header fields), packet hash, the timestamp when the packets come in, period to which data refers.

The second kind of measures report network interface identification, total counted packets, total hashed packets, mean timestamp based on all the timestamp of all packets that passed through the interface, period.

2. **Sending data**: this part of the chain is separated from the previous one. Once the data has been stored and collected as logs by the network device following the provisions of the theoretical model, the sending system has only the task of carrying data safely and reliably.

   It is possible to use a synchronous mechanism, in which the sending system periodically checks the availability of new data, or an asynchronous mechanism. In the last case when a new batch of data is ready, an alert wakes up the sending system that carries them to the destination. In the next implementation, has been used the synchronous approach.

3. **Preprocessing**: this phase has two main goals: aggregate input data to produce a new record that is ready to be postprocessed and that makes it easier to obtain performance parameters; decrease the total amount of data size.

   Although this step is not mandatory, it is recommended to speed up subsequent operations and to give a better shape to the stored data to fit well with the last queries.

   Preprocessing can be done after data has been stored into NMS in a loop that parses that periodically or before to be sent to NMS, through a consolidator, that collects data that comes from all network devices, parse them and then send them to the NMS. In this implementation data are preprocessed after they reach the NMS.

   However, in this phase, it is just possible to join incoming data from all devices and determine the path followed by each sampled packet.

   To do that we notice that if we group data by hash and order them by timestamp, we outline the path. After providing the data management system with the topology of the monitored network, it can also track the crossed cluster for each couple of sorted data, by analyzing the interface ID

available in the stored record and comparing them with the edge that characterizes the clusters available in the monitored network.

Follow example:

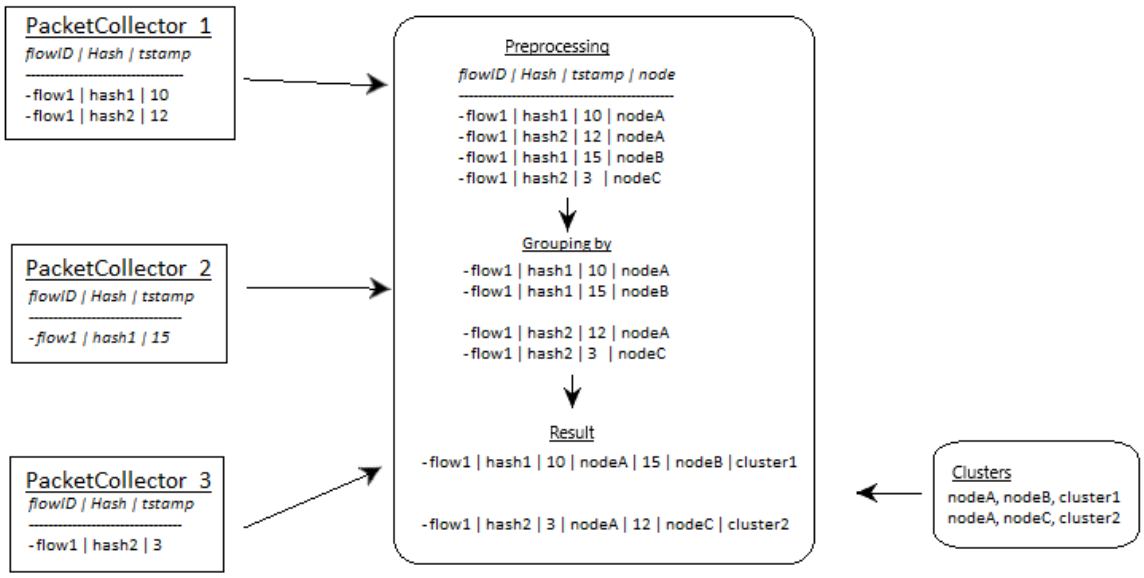


Figure 4.2 – Preprocessing

4. **Results**: the preprocessed records (the results in figure 4.2) lay into the database. When necessary the storage system can be queried, in deferred time. The records are organized to fit well with the queries that care about timing and loss aspects.

   Results are achieved by querying the storage system properly. Certainly, input parameters that identify which flow we are addressing are required. Additionally, the time reference is needed to select only the packets of interest. The big data system is aware of flow identification fields and performs packet flow grouping on the fly.

   The results described below can refer to different flows, it depends on which parameters have been specified for the query.

   We can deduce the cluster mean delay D_i (mean delay referred to cluster i), by analyzing each record, computing delay d_j (delay referred to record j) as the difference between the two available timestamps, that corresponds to the input timestamp (when the packet has gone into the cluster) and the output timestamp (when the packet has gone out of that cluster), and

summing it with all other delays; then the result is divided by all the records that refer to the same cluster:

$$D_i = \frac{d_0 + d_1 + \cdots + d_{N-1}}{N}$$

Where D_i is the mean delay related to cluster i, d_j the delay related to record j, N the total number of records belongs to cluster *i*.

We can also compute the end-to-end mean delay E as the sum of all delays available in our database, and dividing it by all the records:

$$E = \frac{e_0 + e_1 + \cdots + e_{M-1}}{M}$$

Where E is the end-to-end mean delay, e_j the delay related to record j, and M total number of records.

If necessary, after observing an unusual cluster delay, could be possible to compute also max/avg/min link delay, by analyzing records again, and exploiting the difference between the two timestamps.

In addition to that, also details about loss are available. Since we know total packets counted by each node, we know that the sum of the input packets must be equal to the sum of the output packets inside each cluster. If their difference is greater than 0, then a loss occurs, and the result is the total loss.

The total packet loss per cluster:

$$PL_i = \left(p_{i,0} + p_{i,1} + \cdots p_{i,N-1}\right) - (p_{o,0} + p_{o,1} + \cdots p_{o,M-1})$$

considering cluster *i*, with N input nodes and M output nodes.

In the same way is possible to get the entire packet loss, as the sum of all the packet loss per cluster.

The same measure can be obtained by using only the hashed packets, but in this case, we get an approximate measurement that might reflect or not the real one.

4.3 Considerations

Towards input and output nodes in a cluster. Each cluster, necessarily, has at least one input node and at least one output node. We define an input node as a node that

has only incoming edges and an output node as a node with only outcoming edges, considering only edges inside one cluster. An output node could be an input node for the next cluster and vice versa.

Each input and output nodes can belong to at most two clusters, not more.

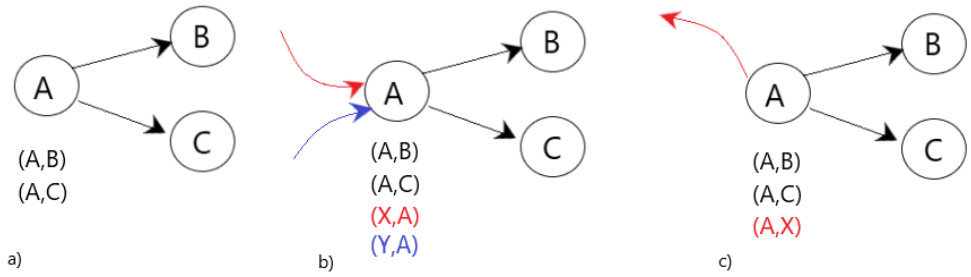


Figure 4.3 – Input/output node consideration

As we can see in figure 4.3, in a) a simple cluster with three nodes, defined by the edges described under the figure ((A,B), (A,C)). In this case, A is an input node and B and C are output nodes, they belong to cluster 1.

Keeping in mind that, to demonstrate that A can belong to at most two clusters we consider that the set of clusters to which A belongs can be conditioned by incoming or outgoing edges.

In figure 4.3 b) A has two incoming edges that belong to two different clusters, the red edge belongs to cluster 2 and the blue edge to cluster 3. Considering, for a moment, only the red edge, A is the input node of the cluster 1 and probably an output node of cluster 2. The same statement can be repeated for the blue edge, and A seems to belong to clusters 1, 2 and 3. This is not true, in fact, blue and red edges must belong to the same cluster, because, following the clustering method, if two clusters have in common an ending node they must be grouped to the same set. The same goes for all possible incoming edges in A and any other input nodes.

In figure c) the issue is simpler than previously. Each link with the same starting node belongs to the same cluster. So, in this case, it means that cluster 1 includes at least the edge starting from A and ending to X (the red edge).

Similar reasoning can be applied for output nodes.

Towards intermediate nodes in a cluster. An intermediate node is a node that belongs to a cluster and that is neither input nor output node. When it occurs, it doesn't increase performance parameters quality in terms of packet loss and delay, in fact, usually, clusters don't have it.

It's worth remembering that an intermediate node cannot belong to more than one cluster.

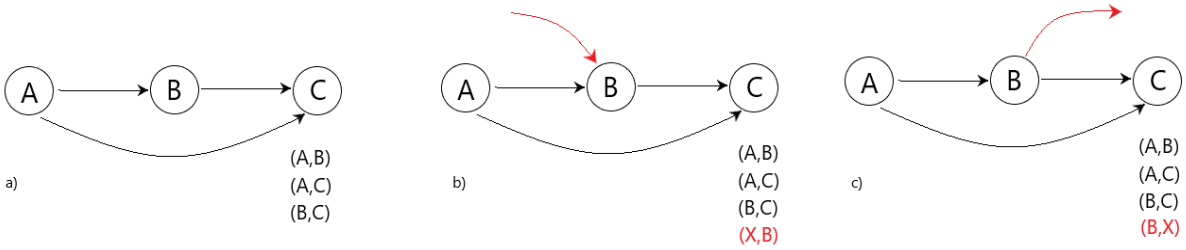


Figure 4.4 – Intermediate node considerations

In figure X a) an example of a cluster with the internode. With the same procedure used previously I try to understand if there is a possibility to deny the statement that I want to prove.

With example b) an incoming edge end in B. In this case, the node from where the red edge is starting will belong to the same cluster of B.

Also, if an edge is starting from B, as depicted in case c), the ending node of that edge will belong to the same cluster of B.

So, b) and c) are the only possibilities that could affect node B in terms of clusters. In fact, multiple cases with both incoming and outgoing edges from/to B could be treated as separate cases like b) and c).

Chapter 5

Architecture

In this chapter, I will describe the architecture of a simulation system that aims to replicate a real network as much as possible. The simulation helps me to show a proof-of-concept of the Alternate Marking Method applied to multipoint measurements with Big Data post-processing.

To achieve this goal the system is composed by the following programs:

- **Mininet**: has been used to build routers and to link them to work as a real network.
- **Probe:** it has been used to collect data, process and send them into a directory, pending for Flume.
- **Flume**: Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data. It has a simple and flexible architecture based on streaming data flows. It is robust and faults tolerant with tunable reliability mechanisms and many failovers and recovery mechanisms. It uses a simple extensible data model that allows for online analytic application. [6]

  It runs into two instances, one per VMs. In the source VM it waits for new files into a spooling directory (the same in which probe store its files), in the sink VM, flume intercepts incoming batch file and store them directly in HDFS.
- **HDFS**: the Hadoop Distributed File System is the framework that is used to enable big data processing. It has been designed to store and process a large amount of data, that is our goal. Furthermore, it can scale horizontally by adding new clusters or new nodes into the same clusters.
- **Spark**: the framework that provides a faster alternative to the MapReduce paradigm to code into clusters environment. It is not only up to 100 times

faster than MapReduce, but includes some key features such as data parallelism and fault tolerance. It was originally developed at the University of California, Berkeley's AMPLab. Then, the Spark code was taken up by the Apache Software Foundation, which maintain it currently [15].

5.1 General aspects

The entire system has been deployed by using two Virtual Machines, in order to separate two main aspects of the thesis: network emulation, data processing.

The purpose of this choice is dual: on the one hand, it aims to underline the difference between the emulation, which is an important part for implementation, but irrelevant to theoretical purposes, and the processing, the core of this project, that involves the most recent IETF discussions on performance monitoring and try to find a congenial solution towards multipoint paths measurements. Furthermore, the VM containing Mininet and other tools is a simulation platform useful for the next tests that require a working simulation. This choice goes to a modular approach direction: to build separate modules optimized in working together but able to work stand alone and fit with different contexts.

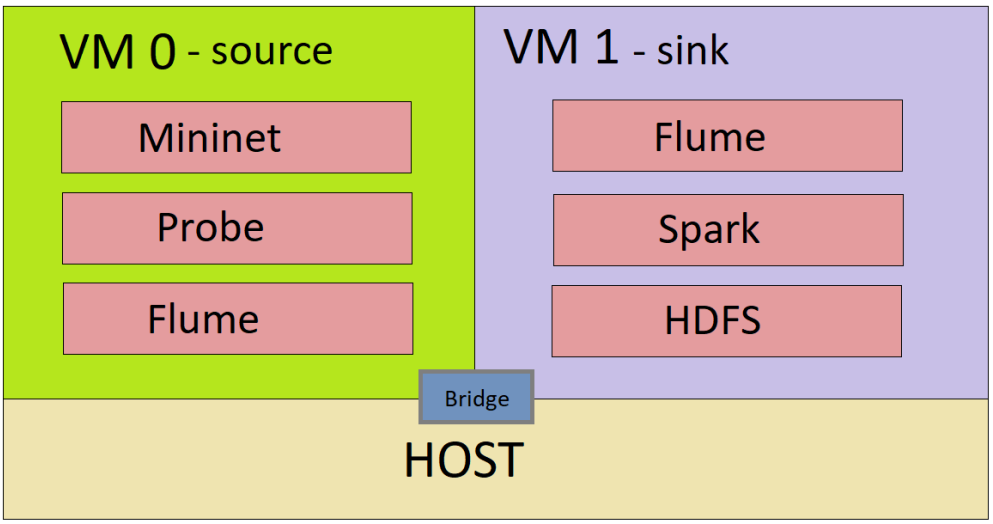


Figure 5.1 – Overall system architecture

Figure 5.1 shows the general architecture of the system. Both VM0 and VM1 are running Ubuntu 18.04 LTS. VM0 is running the network emulation with Mininet, data collecting with the probe and data pumping with Flume, properly configured. It can be defined as source VM, due to its capability to generate data to forward to the other VM.

Meanwhile, VM 1 is hosting the storage system (HDFS) by using Docker to run one Hadoop cluster with three DataNodes. In addition to that, the Spark system lay on cluster NameNode and it starts working when a new job to process data is required. Flume, in this case, is configured to receive data and store them directly on HDFS, without passing through the Ubuntu File System. This VM is called Sink (by following the Flume terminology) because it is the data destination.

Both VM0 and VM1 are linked with a bridge, just to easily connect them. This is a setting available in Oracle VM VirtualBox. Connecting them in some way is a necessary condition to run the system entirely.

5.2 Architecture in a real deployment

The challenge is to collect data produced by the probe and send them to an NMS for data post-processing. There are several solutions available; the simplest could be to open a socket that connects the network device to the NMS. However, this is a manual solution that can easily run into implementation bugs and would require thinking and developing a way to ensure safety and reliability.

On the contrary, Apache Flume allows implementing a similar solution simply by setting the configuration of some parameters that Flume itself makes available to the client. The architecture, in this case, is composed by a flume agent inside each device and another inside NMS, that is listening on an IP address and port. Let's install an agent inside a network device that would watch for newly created logs, acquire these log data, and publish them into a log transport network. We could set up similar agents inside on hundreds of devices.

This is a typical client-server architecture, widely used for server log accumulation. This solution is safe, reliable and scalable as guaranteed by Flume.

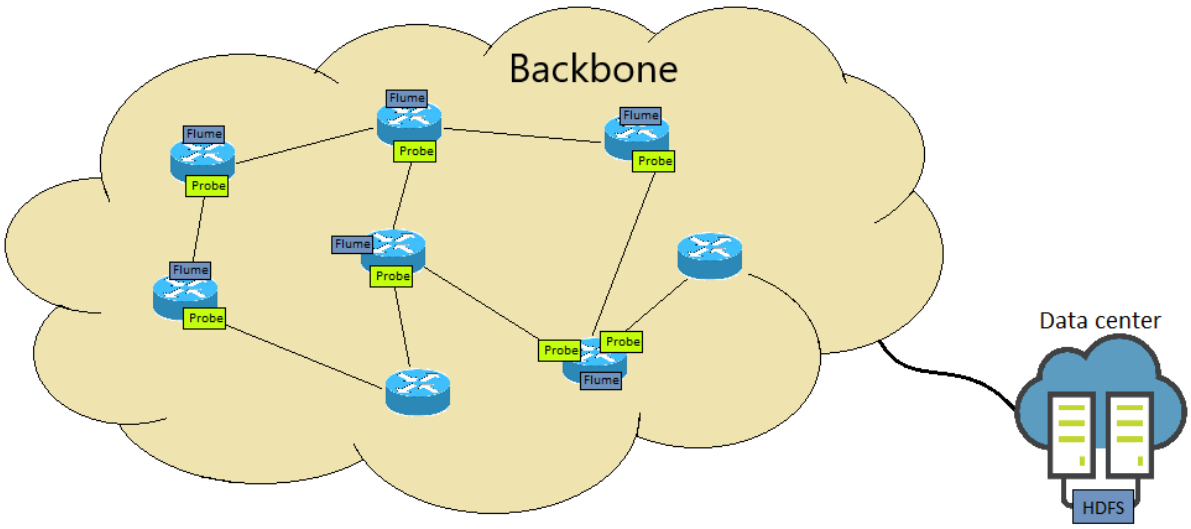


Figure 5.2 – Possible real architecture

In figure 5.2 what could be the real architecture. As we can see there is a probe instance for each network interface we want to monitor and a flume agent for each router that runs probe.

The architecture is thought out to allow performance monitoring by laying out fewer probes than all available network interfaces, which is the sense of cluster partitions.

Data produced by probes do not have a very large size, even if this parameter can be set by specifying how many packets per period we need. However, once data has been stored inside the data center (managed by NMS), they can be kept for a long time (a month for example). Furthermore, the number of network devices pumping data is variable, for this reason, the total amount of data size could be potentially up to some terabytes.

5.3 Architecture in the emulation environment

Once outlined a possible architecture deployment, we need a way to test it, at least the data production and data processing parts.

The emulation of this system is a very difficult task, because it must deal with the reduced availability of devices, the reduced computing power of them and the difficulty of reproducing a real working network.

Nonetheless, thanks to Mininet, VirtualBox and other tools, I succeeded to outline an architecture that reflects the real one as much as possible.

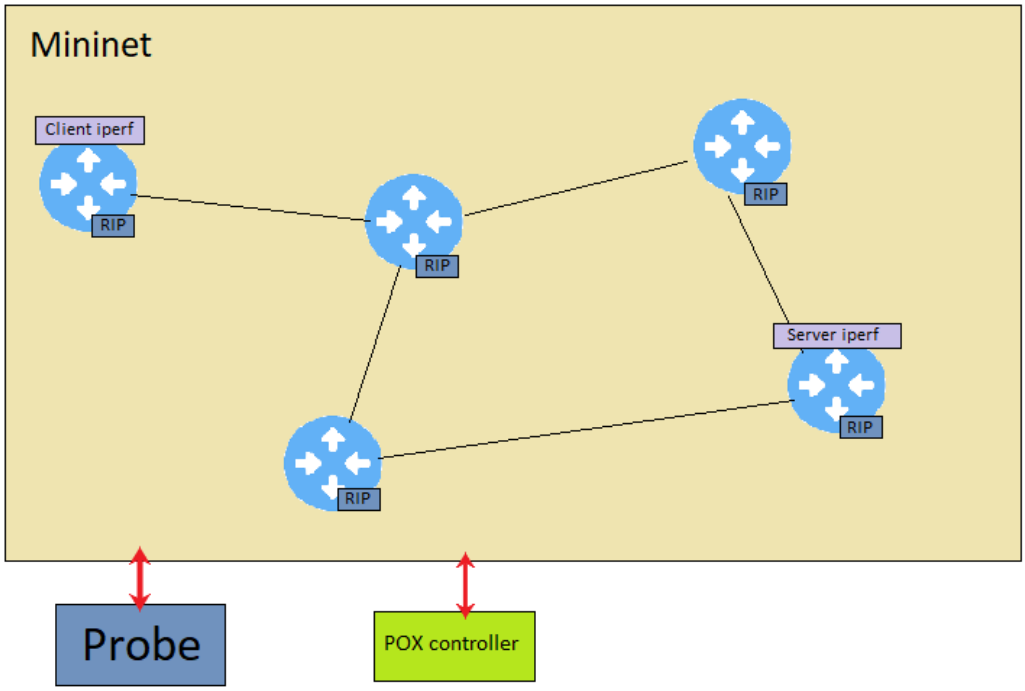


Figure 5.3 – Inside Mininet

Here (figure 5.3), what has been done specifically with Mininet to emulate the real network.

The first problem addressed was to build routers, link them, and generate traffic that could cross the network properly. This wasn't immediate. In fact, after building the network with many routers, some loops occur. Mininet manages packets routing with Open Flow controller, but I should have to set manually it and specify to what port to forward incoming packets. It could be possible only in a network with a few nodes.

The solution was to apply a routing protocol to each router, such as in a real network. The simplest routing protocol [17] to apply in this case is RIP [20] (even if other protocols like OSPF [19], IS-IS [18] are preferable because of no hop limits and less time convergence), and I found a script that implements it in [16]. This script has been coded in python and an instance of it runs inside each Mininet hosts (router). However, RIP has a significant limitation: can't manage more than 15 hops. This limitation bounds to maximum network diameter and for this reason, I choose to do some tests with a network having 40 routers. I tried to apply this solution to a bigger network, but often there were couples of routers that were unreachable.

This is the simplest RIP version and does not implement all features. So, split horizon, route poisoning and hold-down timer are disabled. RIP protocol code is available in rip.py file and its explanation in [16].

Once routing has been managed, the next step is to generate flows. Even if traffic can be generated with a lot of shell commands, we need marked traffic, namely packets that have one bit of TOS field set to 1 for a period and the same bit set to 0 for the next period, in an alternate way. The iperf tools is right for us. In fact, among various options, it allows us to generate traffic with a specific TOS value, to select between TCP or UPD traffic, time duration and so on. A script that runs it has been written and then deployed inside the router source. How the client and server iperf has been chosen and how they run them, is detailed in the Mininet paragraph.

As we can see in figure 5.3, differently from figure 5.1, there is a single instance of a probe. It's because since the simulation runs on a single device, a single instance of the probe is able to intercept all packets from all interfaces. The probe monitors only interfaces that have been selected by configuration.

Finally, the POX controller [21] is an open flow controller that has been developed by using the POX framework. It interacts with Mininet to manage packets forwarding with switches placed between two routers.

5.4 Implementation

Next paragraphs will be dedicated to explaining how each component has been developed, what has been created from scratch, what has been reused or modified.

The most used languages are python and java, while additional bash scripts have been used to simplify deployment.

5.4.1 Mininet

During the development phase, as already mentioned, I encountered a problem with the probe. In fact, it manages to capture only the incoming traffic of an interface, while both incoming and outgoing packets are needed. To solve this problem, I introduce a new router, the intermediate node, placed between two routers linked with an edge. Its function is to intercept packets that are outcoming from a router and record them.

To do that, I exploit the fact that all outgoing packets from an interface are equal to all incoming packets to the next interface, if no loss occurs. If, for example, r1 is linked to r2, an intermediate node R_1_2 has been placed between them. When r1 sends a packet to r2, it is sent both to the r2 interface linked with r1 and to R_1_2 interface linked with r1. The latter interface is reporting details I want to achieve: outgoing packets from r1. The same issue is true for traffic going from r2 to r1.

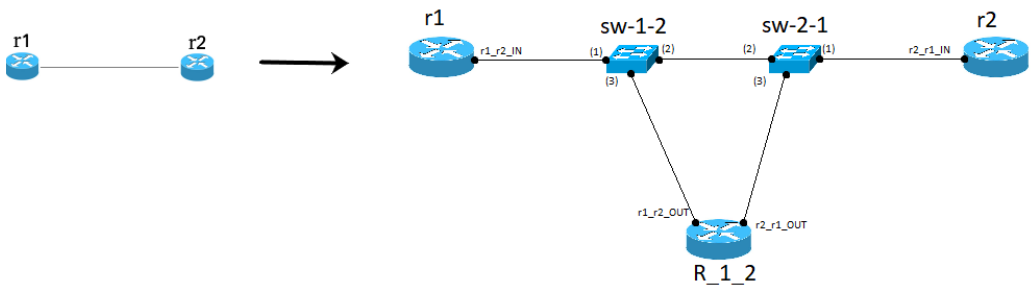


Figure 5.4 – Link customization

As depicted in figure 5.4 each link r1-r2 has been modeled by adding two switches and one router.

Each router interface has a name that fully identified it. The name is composed by the name of the source router, the name of the destination and the direction of the edge, separated from an underscore. For instance, the network interface placed in r1 in figure 5.4.x is “r1_r2_IN”, the other one in r2 is “r2_r1_IN”. While to intercept outgoing traffic another two interfaces have been placed in R_1_2. The interface name, in this case, is “r1_r2_OUT” (because of the link direction) for the link starting from sw-1-2 and “r2_r1_OUT” for the link starting from sw-2-1.

Also, switches have a name that identifies them: sw-1-2 means that a switch is placed between r1 and r2 and is the nearest to r1. By following the same criterion, it is sw-2-1 the other switch name.

The intermediate node name follows the same criterion, with a difference. There is a single instance of it per link and the first number of its name is always the smallest number between the two routers making the edge.

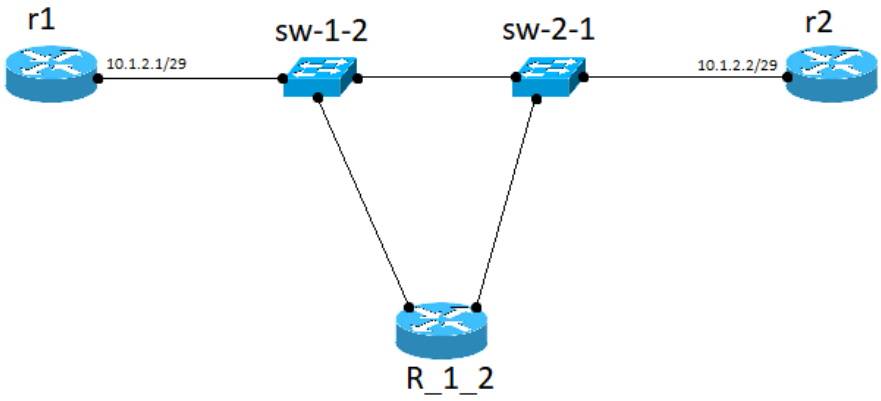


Figure 5.5 – IP link customization

The next step was to set the IP address. The adopted solution is simple and fits well with our emulation system. Considering the link that connects r1 to r2, the IP

address of the r1 interface is taken by adding the number of the smallest router, then the number of the biggest and finally a reference number, that can be 1 or 2, that says to what router has placed the interfaces. A link that connects two routers, for example, r11 and r7, has the IP address 10.7.11.0/29. The interface in r7 has the IP 10.7.11.1, while the interface in r11 has the IP 10.7.11.2. Another example is depicted in figure 5.5. Intermediate node has no IP addresses because it is reached differently from the other routers (by using switches) and doesn't send any packets.

Now the challenge is to forward packets towards the correct interfaces. Since we want to record all traffic, distinguishing between incoming and outgoing, the solution is that when a packet starts from r1 to r2, it ends on the correct interface of R_1_2 and r2.

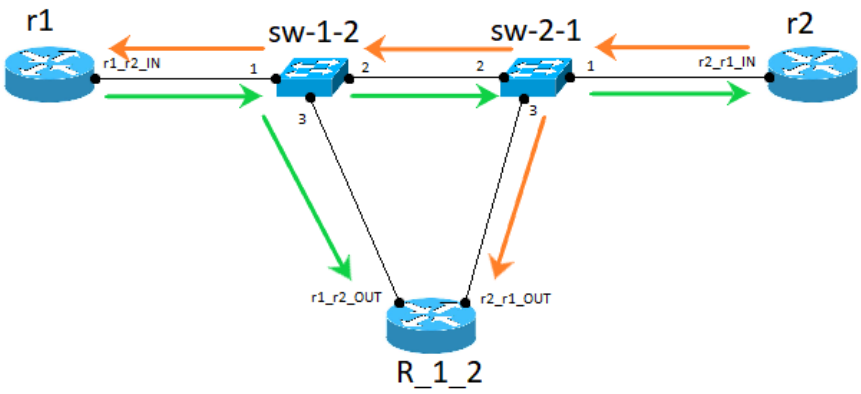


Figure 5.6 – Packet's path

Figure 5.6 shows how the packets should be forwarded to be recognized properly by the system. Different colors depict two different paths: green stands for path packets going from r1 to r2 should go through; orange the same things when a packet goes from r2 to r1.

It is worth remembering that the probe is only aware of incoming and not outcoming packets. So that, if a packet goes from r1 to r2, it crosses r1_r2_OUT, with the meaning that it left r1 to go to r2, and r2_r1_IN, with the meaning that it reaches r2 from r1.

While r2 and r1 known path packet must follow because of RIP protocol, switches didn't. As we can immediately see, considering the port number, there is a general criterion that allows it to follow the correct path: when a packet reaches any switch from port 1, they must be to forward to port 2 and 3, when it reaches the switch from port 2 it must be sent only to port 1. That's the rule.

To do that, Mininet makes available Open Flow controller, whose programming is supported by the POX framework. This is the meaning of the POX controller in 5.3.

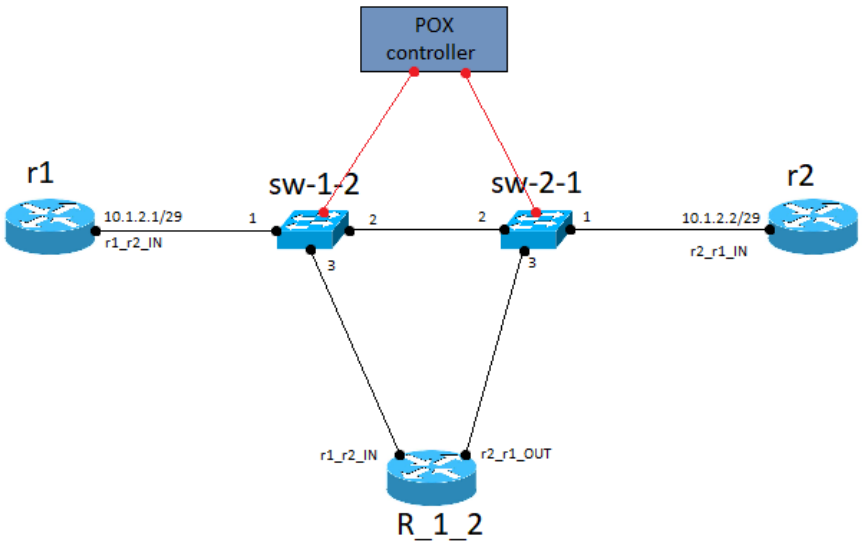


Figure 5.7 – Link customization complete

In 5.7 the last configuration. Switches are linked to the POX controller waiting for instructions.

Its code is in the python script "./pox/ext/controller2switches.py".

POX programming details are available at [21].

```
74  class MyController (object):
75    """
76    Waits for OpenFlow switches to connect and makes them custom switches.
77    """
78    def __init__ (self):
79      core.openflow.addListeners(self)
80
81    def _handle_ConnectionUp (self, event):
82      print("Connection %s" % (event.connection,))
83      CustomSwitch(event.connection)
84
85
86  def launch ():
87    core.registerNew(MyController)
```

Launch() function is the method that is raised when the script starts. *core.registerNew(MyController*) is registering a controller to listen events.

The _handle_ConnectionUp function means that when an event of type *ConnectionUp* is raised the attached function handles it. In this case, a CustomSwitch object is created. The connectionUp event is raised when a switch connects to the controller.

```
47  class CustomSwitch (object):
48    def __init__ (self, connection):
49      self.connection = connection
50      connection.addListeners(self)
51
52    def _handle_PacketIn (self, event):
53      print("Received packet from port: %d" %event.port)
54
55      msg = of.ofp_flow_mod()
56
57      # forward to
58      if event.port == 1:
59        msg.match = of.ofp_match(in_port = 1)
60        msg.actions.append(of.ofp_action_output(port = 2))
61        msg.actions.append(of.ofp_action_output(port = 3))
62
63      # forward to
64      elif event.port == 2:
65        msg.match = of.ofp_match(in_port = 2)
66        msg.actions.append(of.ofp_action_output(port = 1))
67
68      # do nothing
69      else:
70        msg.match = of.ofp_match(in_port = event.port)
71
72      self.connection.send(msg)
```

This is the CustomSwitch details. It handles the packetIn event, raised when a switch receives a packet and there aren't flow tables matching that. If it happens a new rule is prepared to be installed on. The rule is the same show before. After that, a message containing rule is sent to the same switch raised the event and from this moment on it will forward packets without no querying again POX controller.

The Controller is the preliminary part to allow the network works properly. The code that takes care of generating the network is available in net2switch.py.

The entry point is this:

```
180  if __name__ == "__main__":
181
182      if (len(sys.argv) != 2):
183          print "Usage: " + sys.argv[0] + " topology.graphml num_flows"
184          exit(0)
185
186      topo_file = sys.argv[1]
187
188      myNetwork(topo_file)
```

After running the script, that needs one parameter, this is the part that will be executed, like every python script. The specified parameter is the network topology in graphml format. In fact, the network topology is not random but is taken by the XML network topology that Internet Topology Zoo makes available for the scientific

```
def myNetwork(topo_file, num_flows):

    #number of routers in the topology
    n_routers = 0
    #number of links between routers in the topology
    n_links = 0

    G = nx.read_graphml(topo_file, str)
    links = {}

    n_routers = nx.number_of_nodes(G)
    n_links = nx.number_of_edges(G)



    mytopo = MyTopo(G=G, links=links)
    info("*** Links: \n")
    print(links)
    net = Mininet(topo = mytopo, link=TCLink, autoSetMacs = True, ipBase="10.0.0.0/8", controller=None)
    c0 = net.addController(
                name="c0",
                controller=RemoteController,
                ip="127.0.0.1",
                protocol="tcp",
                port=6633)
    net.start()

    print("Starting pox controller...")
    os.system("./pox/pox.py controller2switch &")

    info('*** Router configuration\n')
    for controller in net.controllers:
        controller.start()
    for node in net:
        if(node[0] == 'r'):
            for next_id in links[node]:
                net[node].cmd('ifconfig '+node+'_r'+str(next_id)+'_IN '+ip2(node[1:], next_id, 24))
            rp_disable(net[node])
            net[node].cmd('/usr/sbin/sshd')
            net[node].cmd('python3 rip.py &')
        elif(node[0] == 's'):
            net[node].start([c0])

    info("\n\n\n****** Test in progress ******\n")

    os.system("sudo rm -rf /home/miniubuntu/Desktop/test/*")

    CLI(net)
    net.stop()
```

community at [25]. In my tests, I used the Geant2012.graphml network, that has a good number of nodes (40).

Then, myNetwork function is called.

myNetwork function starts parsing graphml network file, then execute MyTopo() function that instantiates hosts and switches. So, Mininet(…) start the network and then listen for a Controller to localhost with port 6633.

Pox controller is run by os.system(..) command in the background. So that now the network is ready but without IP addresses. The second loop is giving IP to any interfaces and is starting RIP protocol inside each host.

```
        for node in G.nodes():
            node_id = "r" + str(int(node) + 1)
            self.addHost(node_id, cls=LinuxRouter, cpu=.7/number_of_nodes)

        for edge in G.edges():
            id_0 = int(edge[0]) + 1
            id_1 = int(edge[1]) + 1

            internode = "R"
            if id_0 > id_1:
                tmp = id_0
                id_0 = id_1
                id_1 = tmp
            edge0_id = "r" + str(id_0)
            edge1_id = "r" + str(id_1)
            internode += "_"+str(id_0)+"_"+str(id_1)

            self.addHost(internode, cls=LinuxRouter)

            switch_id_0 = "sw-"+edge0_id+"-"+edge1_id
            switch_id_1 = "sw-"+edge1_id+"-"+edge0_id
            sw0 = self.addSwitch(switch_id_0, cls=OVSKernelSwitch)
            sw1 = self.addSwitch(switch_id_1, cls=OVSKernelSwitch)

            self.addLink(edge0_id, switch_id_0, port1=1, port2=1, intfName1=edge0_id+"_"+edge1_id+"_IN", /
                    intfName2="sw-"+edge0_id+"-"+edge1_id+"-1", bw=100) # r_ab_IN -- sw_ab
            self.addLink(edge1_id, switch_id_1, port1=1, port2=1, intfName1=edge1_id+"_"+edge0_id+"_IN", /
                    intfName2="sw-"+edge1_id+"-"+edge0_id+"-1", bw=100) # r_ba_IN -- sw_ba
            if edge1_id == "r16" or edge0_id == "r16" :
                self.addLink(switch_id_0, switch_id_1, port1=2, port2=2, intfName1="sw-"+edge0_id+"-"+edge1_id+"-2", /
                        intfName2="sw-"+edge1_id+"-"+edge0_id+"-2", bw=100, delay='10ms', loss=30)
            else:
                self.addLink(switch_id_0, switch_id_1, port1=2, port2=2, intfName1="sw-"+edge0_id+"-"+edge1_id+"-2", /
                        intfName2="sw-"+edge1_id+"-"+edge0_id+"-2", bw=100, delay='10ms') # sw_ab -- sw_ba
            self.addLink(internode, switch_id_0, port1=3, port2=3, intfName1=edge0_id+"_"+edge1_id+"_OUT", /
                    intfName2="sw-"+edge0_id+"-"+edge1_id+"-3", bw=100) # sw_ab -- i_ab
            self.addLink(internode, switch_id_1, port1=3, port2=3, intfName1=edge1_id+"_"+edge0_id+"_OUT", /
                    intfName2="sw-"+edge1_id+"-"+edge0_id+"-3", bw=100) # sw_ba -- i_ab

            add_link_to_router(links, edge0_id, id_1)
            add_link_to_router(links, edge1_id, id_0)
```

MyTopo class contains, among other lines code, this part that is the core of the simulation. The first loop creates one host for each node in the Graph generated from reading the graphml file, the second loop builds the links. As we can see, the first part of the second loop is dedicated to set up the intermediate node (i.e. R_1_2), after that switches, then the links.

Links are crucial in this case. They include, in some cases, delay. Why? If it is omitted, I have observed that the probe processes a lot of data simultaneously. This implies that, if traffic on the network is too high, data are not processed in chronological order, but computation request is such that it is not possible to process all data as they come and therefore they are enqueued to be analyzed afterward with a policy that is not first come first served. The results were that timestamps were no longer consistent.

However, the delay must be placed in the right link, to avoid unexpected behavior. If it is put in the link that connects the router to the switch, the result will be that the packets will first reach the destination interface and then the source one.

It has been introduced in link connecting the two switches, as shown in the figure 5.4.(x+4).

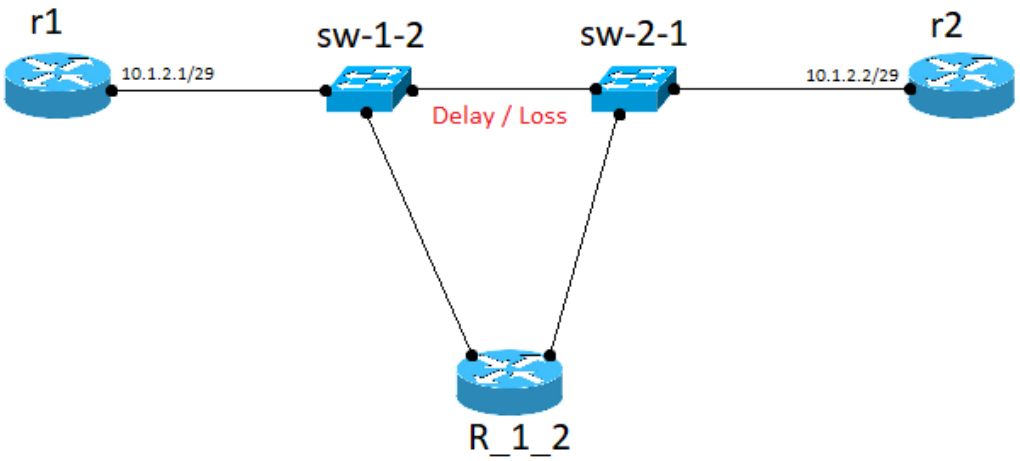


Figure 5.8 – Link customization delay/loss

As can be seen from the figure 5.8, the loss has been added in the same link. Loss is required to test the cluster partition and recognition effectiveness.

Lines 76-81 are an example of how to introduce a loss in a specific link. In this case, the loss has been introduced in all links that end in r16.

The last step concerns packets generation. As I said I used iperf command to do that. A script has been developed to easily set up that and start testing. First, I run “iperf -s -u -B [ip interface]” to start server iperf on a router, then I start the script from another one to generate one flow towards the server. Code can be found in “iperf_test.sh” file.

```
16  if [ "$#" -ne 6 ]; then
17      echo "Parameters number is "$#
18      echo "Wrong number of parameters"
19      exit
20  fi
21
22  PERIOD=$(($4-1))
23  ITER=0
24  TOTAL=$3
25  echo "############################"
26  echo "############################"
27
28  while [ $ITER -lt $TOTAL ]
29      do
30      echo "Executing command: iperf -u -b $1pps -c $2 -t $PERIOD --tos $5"
31      iperf -u -b $1pps -c $2 -t $PERIOD --tos $5
32      sleep 1
33      echo "Executing command: iperf -u -b $1pps -c $2 -t $PERIOD --tos $6"
34      iperf -u -b $1pps -c $2 -t $PERIOD --tos $6
35      sleep 1
36      echo "end sleep..."
37      ITER=$((ITER+2))
38  done
```

Generation is totally driven by parameters. This is a running example: "./iperf_test.sh 4 10.0.4.2 40 30 8 16 ". This means to execute client iperf with 4 packets per second speed, towards a server iperf in 10.0.4.2, 40 times, with each period of 30 seconds, alternating TOS value between 8 and 16 (0x01000 and 0x10000).

5.4.2 Flume

A flume agent per VMs is running. From its configuration depends how to take data to send, where to send them and how they will be stored. All details can be found at [6].

"Apache Flume is a distributed, reliable, and available system for efficiently collecting, aggregating and moving large amounts of log data from many different sources to a centralized data store." Its architecture inspired building the current system in the way it has been presented.

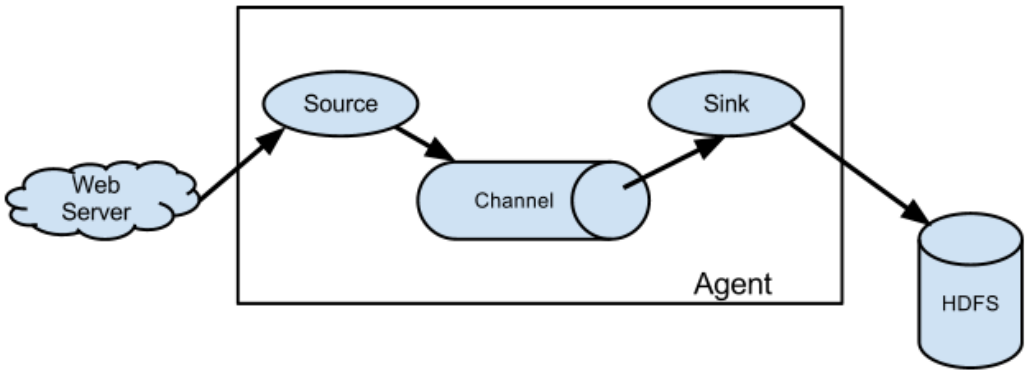


Figure 5.9 – Apache flume agent architecture [6]

Figure 5.9 depicts the flume architecture example.

In our case, there are two Agents, one running in VM source that looks for new files to send to the other side of the link and another one running in VM sinks that is listening to events.

To easily deploy flume I create two folders, one to execute in VM source, called "apache-flume-1.9.0-bin_source" and the other one to run in VM sink with the name "apache-flume-1.9.0-bin_sink".

To run each one of them just executes "./run.sh" command inside the folder.

Their configuration is handled by "*.conf" files inside the conf folder. The source agent conf file is "dir-source-avro-sink.conf". This agent has been configured to periodically look inside ~/Desktop/test folder for new files that are not ending with ".tmp" suffix. When it found that, it sends them to the destination.

```
# Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# Describe/configure the source
a1.sources.r1.type = spooldir
a1.sources.r1.spoolDir = /home/miniubuntu/Desktop/test
a1.sources.r1.channels = c1
a1.sources.r1.basenameHeader = true
a1.sources.r1.pollDelay = 30000
a1.sources.r1.ignorePattern = .*\.tmp$

# Describe the sink
a1.sinks.k1.type = avro
a1.sinks.k1.channel = c1
a1.sinks.k1.hostname = 192.168.43.181
a1.sinks.k1.port = 44444

# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 8192
a1.channels.c1.transactionCapacity = 128
```

In this case, agent “a1” has one source, one sink, and one channel.

Each one of them has different purposes: source takes care of how to catch data from the device, the channel is about what kind of tool to use to send data; finally, the sink takes care of where to send data and how to parse event.

In this case, the source is “spooldir”, namely it looks for new files inside directory “/home/miniubuntu/Desktop/test”, every 30 seconds. It is ignoring files ending with “.tmp”. This tip is very important. In fact, if we leave that and set a pollDelay too small, flume may start sending files before the probe has finished to write it, by sending an inconsistent file. To overcome this drawback, I set the probe to generate new files with suffix .tmp and when ends to write it, renames it by leaving .tmp. This avoids this kind of issue.

The sink is configured to transport data as Avro events to IP 192.168.43.181, port 44444. The IP parameter is variable and depends on where we bind the flume sink (the flume is running in VM sink). Avro is a data serialization system developed by Apache and is used to transport data across the network in a safely and reliable way. It relies on schemas. Further details available at [22].

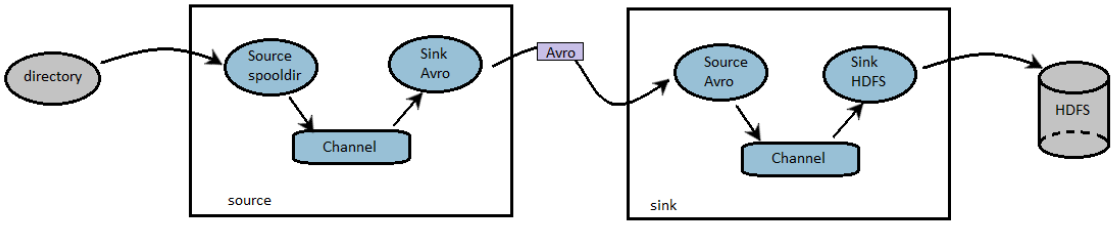


Figure 5.10 – Flume design over the system

Figure 5.10 is showing overall Flume architecture.

The Flume configuration in VM sink is shown below and is available at apache-"flume-1.9.0-bin_sink\conf\myconf\avro-source-hdfs-sink.conf".

```
# Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# Describe/configure the source
a1.sources.r1.type = avro
a1.sources.r1.channels = c1
a1.sources.r1.bind = 192.168.43.181
a1.sources.r1.port = 44444

# Describe the sink
a1.sinks.k1.type = hdfs
a1.sinks.k1.channel = c1
a1.sinks.k1.hdfs.path = hdfs://localhost:9000/user/root/input/%y-%m-%d/%{basename}
a1.sinks.k1.hdfs.useLocalTimeStamp = true
a1.sinks.k1.hdfs.rollCount = 100000
a1.sinks.k1.hdfs.rollSize = 0
a1.sinks.k1.hdfs.fileType = DataStream
a1.sinks.k1.hdfs.minBlockReplicas = 1

# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 10000
a1.channels.c1.transactionCapacity = 10000
a1.channels.c1.byteCapacityBufferPercentage = 20
a1.channels.c1.byteCapacity = 800000
```

Source expects to receive data with Avro at IP 192.168.43.181 (be careful this is the IP address of the network interface of VM sink that is coming from the bridge, properly configured with Oracle VM VirtualBox), port 44444; the channel is memory. Sink configuration contains some noteworthy features. After defining HDFS as sink type with channel c1 (the only available here), we need to define the HDFS path where to store data. In this case is "hdfs:// localhost:9000/ user/root/

input/%y-%m-%d/%{basename}", this means that HDFS is available at localhost, port 9000, and the file will be stored into a folder that begins with /user/root/input/ and is followed by year – month – day, that are set by the HDFS itself, then the basename is the name with which files have been saved by the probe (it will come in handy later) and will be replicated as foldername inside HDFS.

5.4.3 HDFS

HDFS is the storage system developed for Big Data processing. It has a distributed file system that can be reached by typing its IP address and port to which it is bound. As described in chapter 3, it runs into a cluster, that is typically composed of multiple nodes. To run a single cluster we need at least one Namenode, one resourcemanager, one historyserver, one nodemanager and one Datanode (where data are physically stored). Obviously, in a real context, it requires multiple server machines, that I hadn't for the thesis purposes. Therefore, the possible solution are:

- Run all server machines by using Virtual Machines. This solution would require at least 3 or 4 VMs dedicated only to HDFS. Since a VM needs at least one CPU core, this solution is not feasible with my pc. The setup would be more complicated if we wanted to run a cluster with more than one datanode.
- Run all server machines by using containers. This approach relies on available resources by sharing them between more containers. It is feasible since not too high performance is needed to run containers.
  I select Docker as a container virtualization environment, and I run an HDFS cluster with 3 dataNodes. A docker solution for cluster deployment is available at [23]. It is enough to execute docker-compose up command to run the entire cluster.

Docker is running in the VM sink and is communicating with Flume, waiting for incoming data.

After typing “docker-compose up” will be set up 7 containers: Namenode, resourcemanager, historyserver, nodemanager, 3 datanodes. I’ll briefly describe each one of them.

- Namenode: is the main node. It contains all the useful metadata to manage the system, such as where replicas are located and how many copies of each one are available in the cluster. It is available at localhost:9870. From this webpage is possible to examine the Hadoop file system (files and folders).
- Historyserver: manages some logs information about finished applications. Its logs are accessible via REST APIs.
- Resourcemanager: it cares about tracking the resources in a cluster, and scheduling applications.
- Nodemanager: The NodeManager is responsible for launching and managing containers on a node. Containers execute tasks as specified by the AppMaster [24]. It cares about health checking running nodes.
- Datanode: where data are physically stored. It communicates with other components to provide data availability.

5.4.4 Probe

Probe code has been detailed before, in chapter 3. It has been developed by Fabio Salvini in 2017, in his thesis [14], and reviewed by me. After changing some lines code because of the new structure of the updated kernel version and some other compatibility issues due to eBPF instability, I added also the part about data saving in “/test” folder. Follow lines 429 and 430 of “pnpm.py” file.

a_file.write(str(pkt_info.sip)+'_'+str(pkt_info.dip)+'_'+str(pkt_info.sport)+'_'+str(pkt_info.dport)+'_'+str(pkt_info.proto)+','+hash_hex+','+str(l.tstamp)+','+str(clusters_per_node[netif][0])+','+str(clusters_per_node[netif][1])+','+str(period)+'\n')

Probe creates one file for each interface – period pair and save data in a file named period_interface.tmp (i.e. 1801_r5_r12_OUT.tmp). When the writing phase ends, it will be renamed by leaving .tmp suffix. The filename will be reused by HDFS to store that into the correct folder, that will have the same name.

5.4.5 Spark Code

Spark is a useful framework that allows making code to be run into HDFS.

The Hadoop File System has been built, initially, to run Map – Reduce program, but, as has been widely proven, it could generate network congestion inside the data center for data intensive jobs. Spark changes the patterns and offers a new paradigm: move code instead of data. This principle exploits the fact that is less expensive to move executable file inside each node containing data rather than to move data towards running code. This approach solves the network congestion issue and speeds up the job's execution up to 100 times than the map-reduce paradigm.

However, Spark is the framework used to develop code that follows the guidelines listed in chapter 4. I prepared two different programs, of which executables have been made in jar format to be run on HDFS. One of them is about data preprocessing and is ready to run in a loop. It cares about the first and the last period that is stored and work with three periods per time if the last considered period is not the last one.

The jar file is available at /Script_hadoop/PreProcess/ folder. To execute them and another executable is necessary to follow these steps:

1. Start HDFS with docker-compose up
2. Create three folders: spark, preprocess, postprocess.
3. Copy spark-3.0.0-preview-bin-hadoop3.2/* content inside spark folder
4. Copy PreProcess/* folder content inside preprocess docker folder
5. Copy PostProcess/* folder content inside postprocess docker folder
6. For preprocess: run ./periodic.sh [date] by adding date with format yy-mm-dd
7. For postprocess, you must change run.sh contents. How it should be made will be illustrated afterward.

While the jar file is in the PreProcess folder, code is in Preprocessing/ src/ …/ SparkDriver.java file. To understand that is essential to understand how data are stored inside HDFS. Thanks to HDFS file system data are stored inside folders that

have a path through which to reach them. Incoming files are stored inside /root/input/period_interface/* folder. Thank to this name is possible to process data one period at a time.

The first part of the code is dedicated to iterating over the file system, starting from the input folder, looking for folders that start with the correct period. It is possible thanks to folders' names, of which the first part is the period they refer to. In this phase, the program is also looking for a file that names “edge_cluster”. Each line of this file has this format: “interface1,interface2,clusterNumber”. These lines are stored in a Java Map structure.

After that, data processing follows the same criteria listed in chapter 4. Data are grouped by hash and then sorted by timestamp. To works properly the system requires the edge_cluster described before. The final result is showed in figure 4.2.1.

While preprocessing is done in a loop, postprocessing is performed on demand by the data owner. Queries need to know the flow, start and end time (by providing period), date.

To run that is necessary to follow previous steps, until point 7. I show run.sh content:

```
9  /spark/bin/spark-submit \
10   --class it.polito.bigdata.spark.exercise31.SparkDriver \
11   --master yarn \
12   --deploy-mode cluster \
13   ./prog.jar /user/root/output/ final/ 10.0.0.0 255.0.0.0 20-02-20 1384 1387
```

Line 13 contains parameters that define the output directory, flow, date, and start/end period. These are query parameters that must be changed if we want to perform different jobs.

In particular, flow is defined as IP subnetwork to match with IP source parameter from stored data. Even if the flow is identified, in these examples, by 5 parameters, I have introduced this simplification to improve code readability, trying to focus on the purpose of the thesis. So, flow is identified, in this case, by “10.0.0.0” and “255.0.0.0” (means 10.0.0.0/8 flow), refers to date 20 February 2020, with start period 1384 and end period 1387. To change query parameters is necessary to change them and rerun.

Postprocessing code is available at PostProcessing\src\... \SparkDriver.java file. It takes as input parameters IP source to match, date, start period and end period. The code pattern is the same as the preprocessing one: the first code part identifies all data between the start and end period and the second part processes them and then store results inside HDFS.

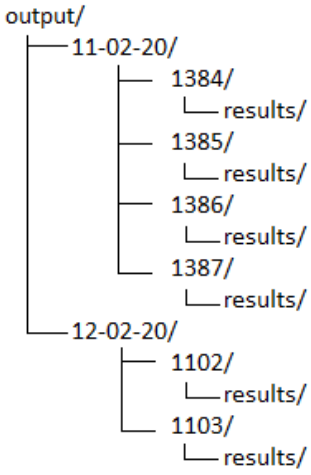


Figure 5.11 – Results' path inside HDFS

Figure 5.11 is showing what is data path inside HDFS, after preprocessing.

In the data processing part, initially, packets are filtered by selecting the ones belongs to selected flow, then they are grouped by hash. Before proceeding with further processing, hashes are filtered by considering loss (looking for records that end with "-2").

After that, delay per cluster and e-to-e delay are computed. Finally, results are split into three folders: loss_per_custer, mean_delay_per_cluster, stat.

The first contains how many losses per cluster occurs and a reference rate. The second one contains average, minimum, maximum values of delay, referred to clusters. While stats contain overall info: end-to-end mean delay, detected loss, loss rate. Those measures are based only on hashed packets.

When results are performed, data are stored with the same format of preprocessed files. A folder named final/ is created; the internal format (date /period /loss_per_cluster etc) is the same as preprocessed ones. Results can be examined by listing one of that folder contents (typical command is “hdfs dfs -cat /user/root/final/10-01-20/1091/loss_per_cluster/*”).

Chapter 6

Workflow

After describing the stakeholders and how each one of them is working, the last step is to present the workflow and to describe their interactions.

Picture 6.1 is showing the overall architecture and components interaction.

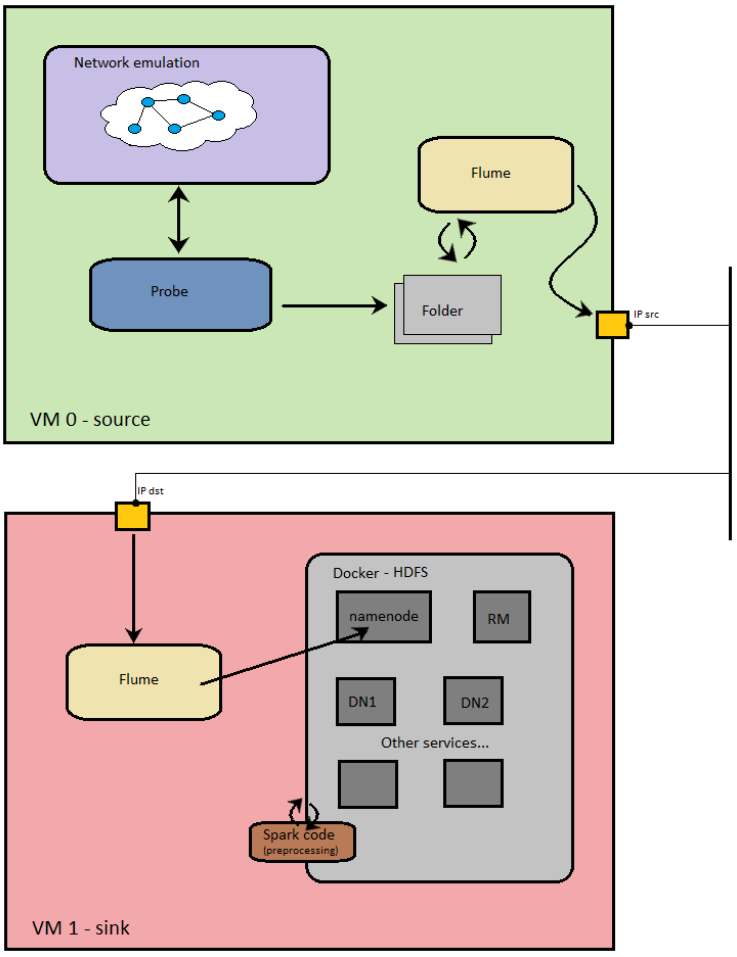


Figure 6.1 – Overall system architecture

The workflow can be represented as a sequence of steps, that I'm going to list here and that are depicted in picture 6.1.

1) Network emulation is going on with the Mininet framework. Multiple flows are simulated by starting servers iperf and clients iperf that are communicating among them. Meanwhile, the probe is waiting that the system call to which eBPF is attached, is raised. It's worth remembering that system call to which probe refers is raised only when a packet is getting on the network interface, both real or emulated.
2) Probe, when enabled, starts packets detecting. It computes first raw measurements, based on timestamp, hash, identification fields and when it ends to parse data coming from a specific interface, store them inside a folder (i.e. /test). Notice that probe can detect only marked packets, if they are not it will not perform any computations.
3) The folder is a container to which data are stored and picked up periodically. It doesn't play any significative role, except that of linking point between the probe and the Flume agent.
4) Flume agent on VM 0 is continuously looking for new files inside the folder. It looks at every 30 seconds. When new data are available, it picks up and sends them to the next hop, after wrapping them into the Avro serialization system. I suggest setting time to wait between two consecutives read to the same value of the marking period. Since the flume agent starts reading a few seconds after probe store data into folder, the flume agent will never interfere with probe while writing data.
5) Flume agent on VM 1 (the sink) is waiting for new incoming data. When it happens, the agent immediately stores them inside HDFS, at the path /users/root/date/input/… .
6) HDFS, that has been deployed with Docker since the beginning, is ready to receive data and run the job on that. Note that, to make jobs working properly, the system requires a file containing all the edges and to what cluster they belong to.
7) A Spark job for data preprocessing is continuously running in an infinite loop. It takes care of looking for new data, what is the last period and if it can

perform computation, otherwise it sleeps before rechecking the last stored period.

8) Results are available on demand. To do that is necessary to run the jar file for postprocessing, by configuring run.sh file properly. Then they are observable by looking inside HDFS "/final" folder contents by typing command: hdfs dfs -cat /path/*.

   Notice that the previous command must be typed inside Namenode container, otherwise, it will fail. So, it requires that cmd for Namenode is run by typing "docker exec -it namenode bash".

Chapter 7

Results

This chapters shows results based on two simulations. Each one of them has three flows and has been launched with the Geant2012 network. This topology belongs to the Internet Topology Zoo dataset [25], which provides an XML file containing nodes and edges, and a picture that depicts the topology map.

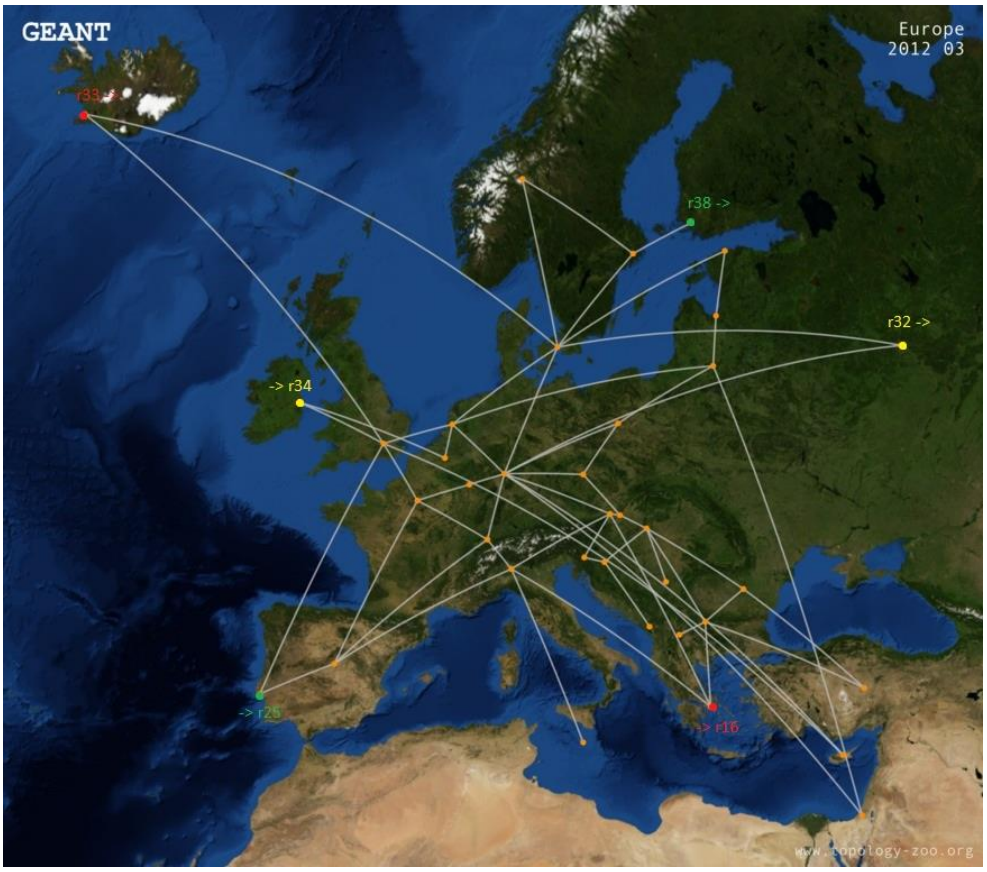


Figure 7.1 – GEANT2012 topology with flow [25]

In figure 7.1 the topology map of Geant2012. Each one of the orange spots is a node, while the yellow, green and red spots symbolize the three flows that have been used for current simulations. As I said before, a client and a server iperf have been deployed for each flow to build it. In this case, routers that report the arrow before the name means that they are servers (i.e. "-> r16"), otherwise, they are clients (i.e. "r33 ->"). In these simulations, there are three clients and three servers iperf.

The steps that have been followed to perform the simulations are:

1) Select nodes that constitute flows.
2) Select nodes to monitor, that must include flow nodes, edge nodes, and at least one backbone node.
3) Build a monitored network by running "CompleteIterativeClustering_v5.py" script, available at [26] by selecting which nodes we want to monitor.
4) Run "cluster.py" script (inside probe folders), that output the "edge_cluster" file, containing as many lines as edges, with the related cluster, and take as input the file created after running step 3)
5) Start from step 1) of the workflow mentioned in chapter 6.

In these cases, I select to monitor all the edge nodes (including flow nodes) and a few backbone nodes, in order to have a fair nodes' distribution per cluster.

Results are illustrated employing charts and they are split in the overall measures (based on all observed packets) and by flow measures (related to each one of the three flows). They are furthermore split into end-to-end and per cluster measures and concern both delay and loss. So, they can be summarized in:

- **Overall**: related to all available flows.
    - **Delay**:
        - End-to-end
        - Per-cluster
    - **Loss**:
        - End-to-end
        - Per-cluster
- **Per-flow**: related to a specific flow.
    - **Delay**:

        - End-to-end
        - Per-cluster
    - **Loss**:
        - End-to-end
        - Per-cluster

These are some examples of what kind of measures we can achieve. Further performance details can be obtained by improving the big data running code and increasing details collected by the probe.

However, I want to clarify that these measures are not accurate (i.e. packet loss is not exactly the real one affect the node) and that they aim to underline the fact that with the big-data approach in a multipoint network is possible to outline performance measures split by flow and cluster, without increasing network devices effort due to filters' deployment and configuration.

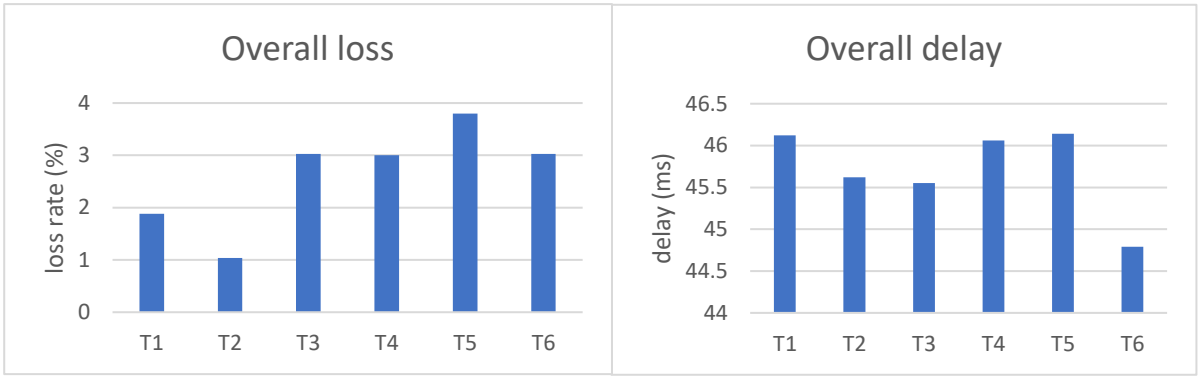
The accuracy is out of the scope of this thesis and may constitute a good starting point for the next works towards these arguments.

7.1 1st simulation

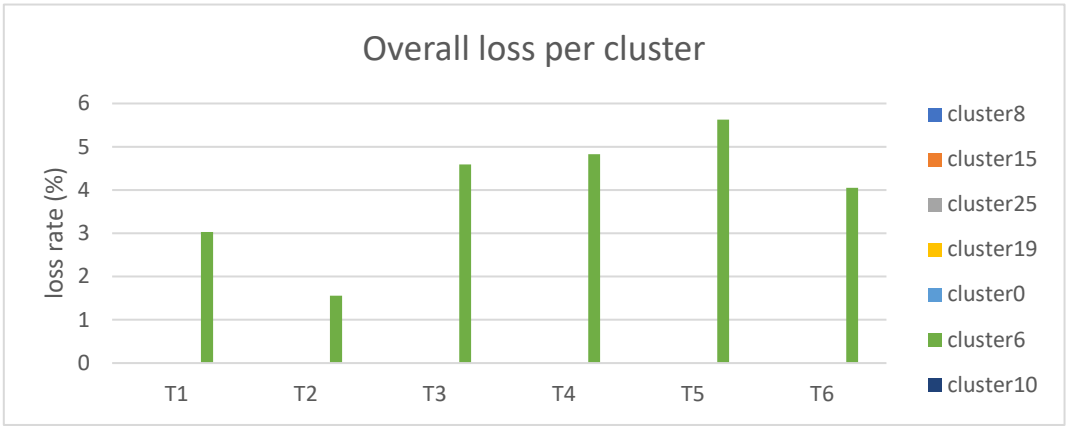
The first simulation has been raised with a 10 ms delay in each link and a loss on all links ending in r16. Only one of the involved flows will be affected by a loss, the other ones do not cross links where the loss occurs.

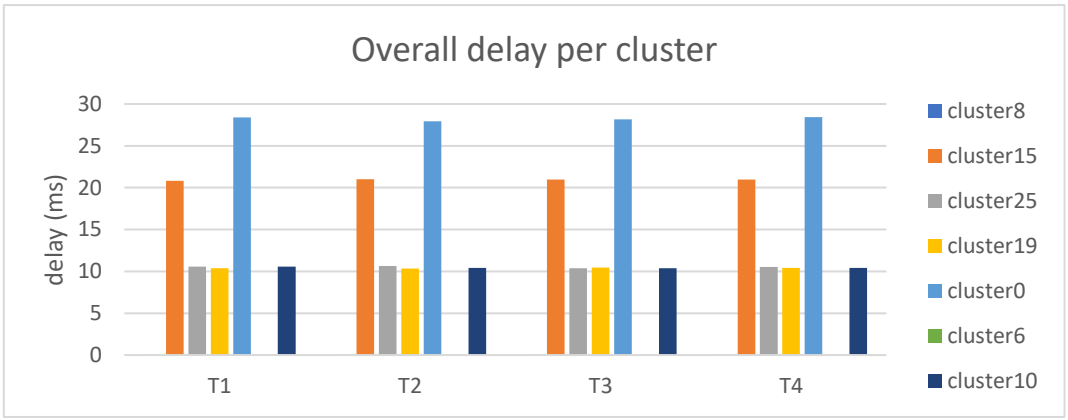
So, we expect that the overall results detect loss, but they cannot give us further details about what flows have been affected and what clusters have been involved. The per-flow results should detect loss in one flow and zero loss in the other ones.

Furthermore, delay measures will give us an overview of the time taken by the packets to reach their destination. Certainly, the results split by clusters and flows, if needed, give us a more detailed view.



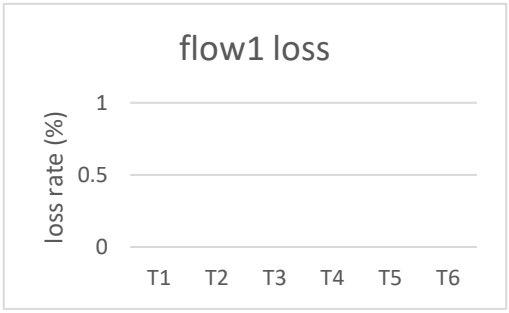
These charts are showing that loss occurs in every period and that delay is towards 45 ms and is almost constant.

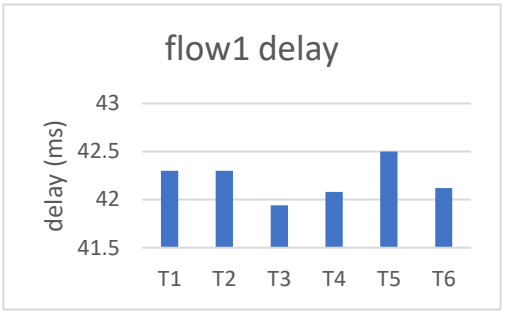


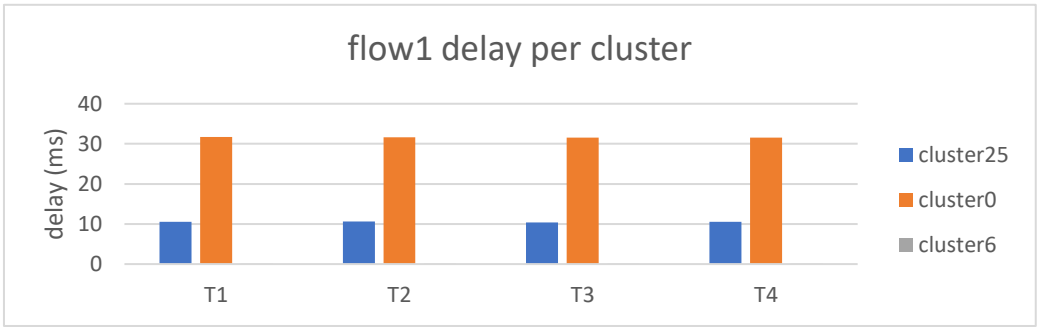


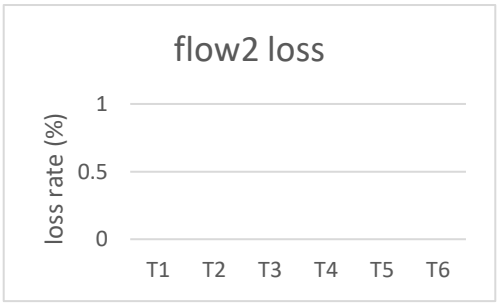
After further research, we note that loss is bounded to only one cluster (i.e. cluster 6). A per-flow investigation will show which flow has been affected by that loss. The per-cluster delay show delays split by clusters. Some of them are towards zero value, this is since one network interface can become a cluster with the IN part as

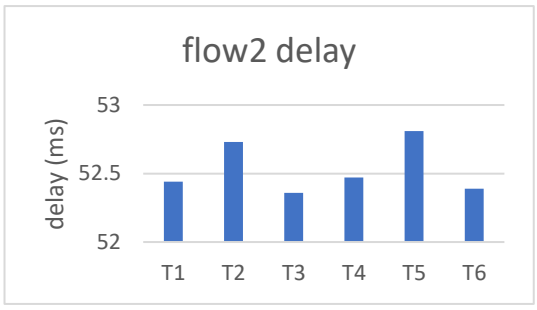
starting node and the OUT one as ending node. The time spent to cross that cluster is due to the internal router computation, which is approximately zero.

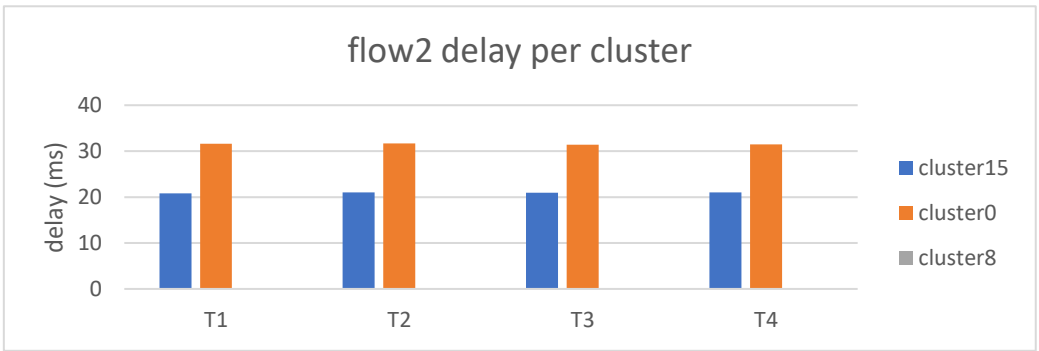


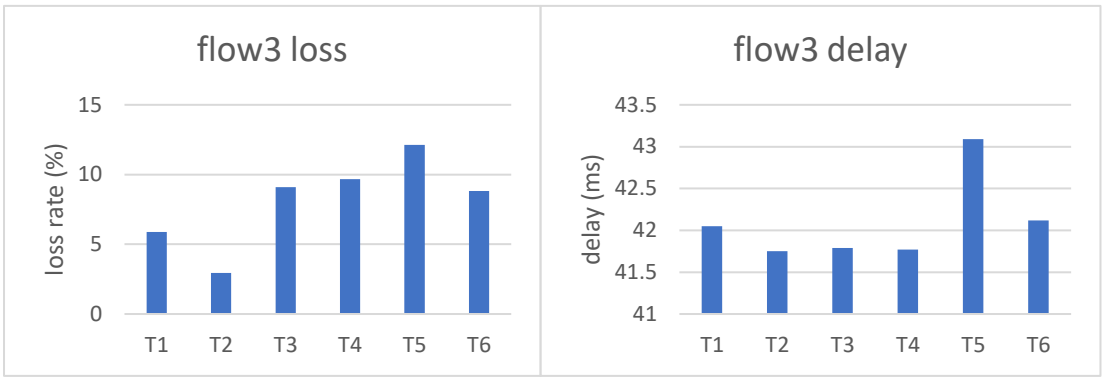


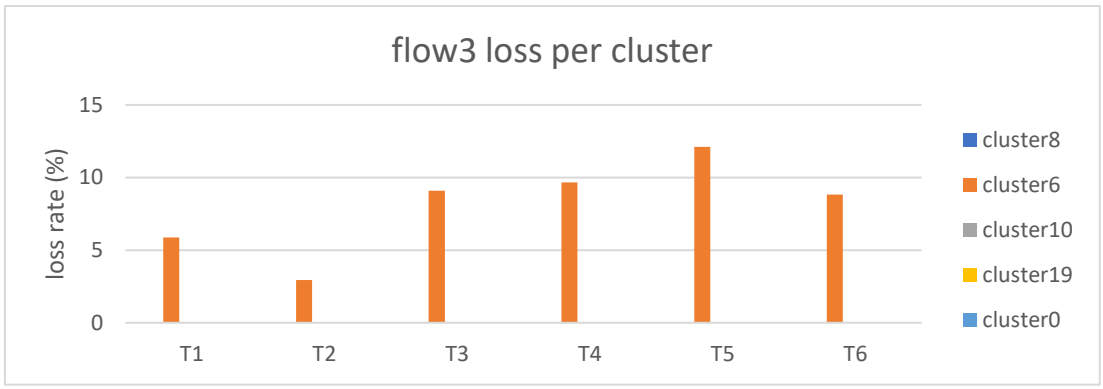


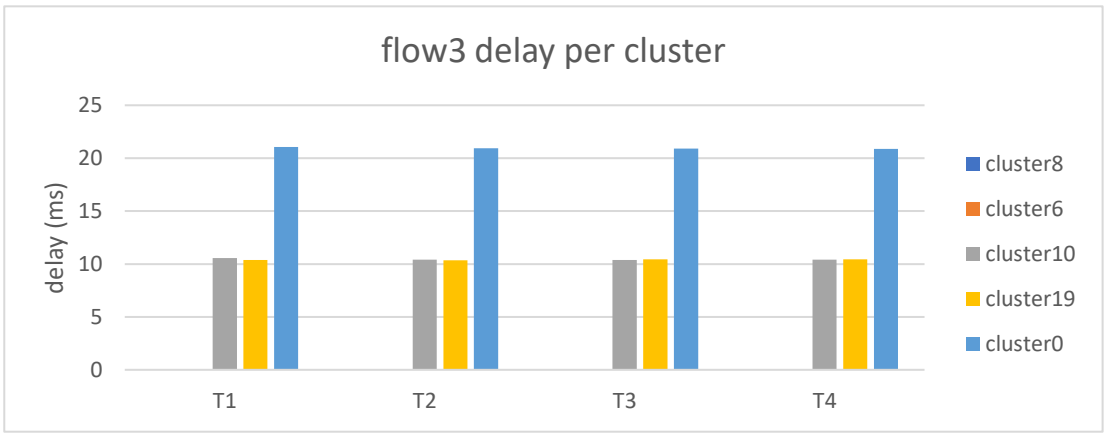










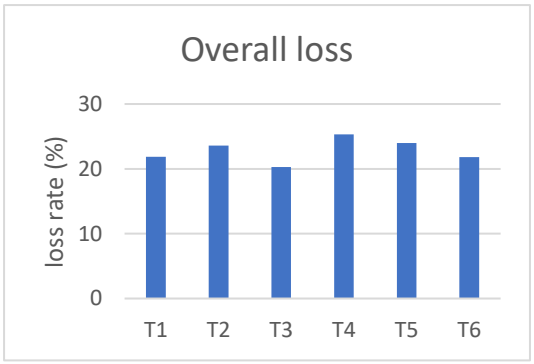


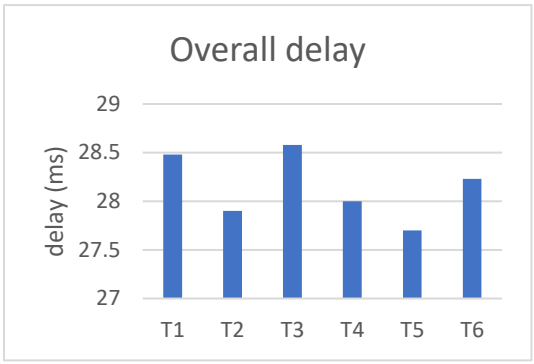
These last results showed that the flow affected by loss is only the flow3.

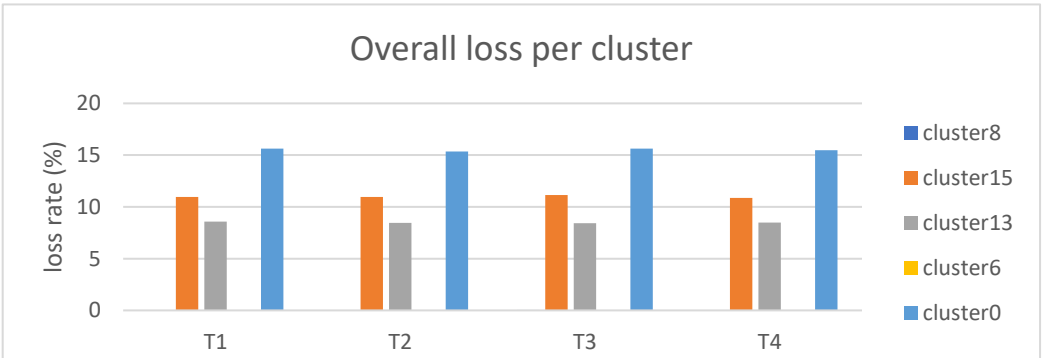
Finally, I want to highlight that charts are showing what we were expecting: losses are bounded only to one flow (as I set when running the simulation) and only to a cluster. This is an example of how the zooming approach can succeed in identifying the trouble.

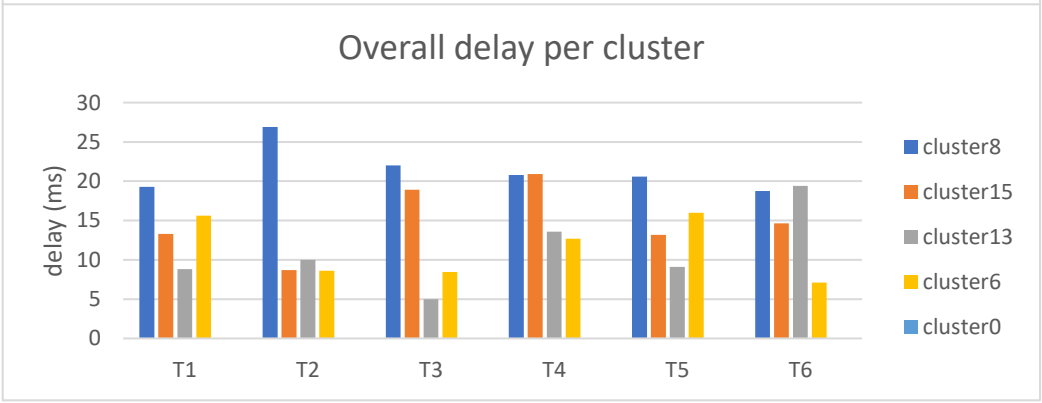
7.2 2nd simulation

The second simulation aims to provide an overview about the big-data approach. To do that I launched a simulation with a random loss and delay for each link. Follows charts.









These charts report what is possible to monitor in a network with the as general query as possible (an overall research). Starting with these charts we can decide to proceed in a fine-grained research that involves each single flow, or to stop here since we believe that further researches are not relevant. In this case a per-flow investigation would not be relevant; since the losses were inserted homogeneously over the entire network, it would not help us to find a specific point where the loss

occurs or a specific flow that has been affected by it. Moreover, this kind of investigation is out-of-the-scope of this paragraph, since the previous one provides an example of an effective way to identify the trouble over the network.

Nevertheless, this first overview about the network behavior is providing some relevant details about where losses happening. As we can see, not all the network is affected at the same way; in cluster6 and cluster0 has not been detected any troubles. This kind of information could lead next deeper research. In fact, a possible further query could be related to only flows that cross at least one of the clusters 8, 15 and 13, excluding the other ones.

Finally, in this case, charts are showing again what we expect. Loss has been detected over the network, as I configured before running the simulation, with some exceptions and it could be true, because loss, even if random, could be zero.

Chapter 8

Conclusions & future works

Carrying out this thesis was not easy. Many difficulties have been faced and efforts have been made to try to get results that live up to expectations.

The main issue was making the probe compatible with the version of ubuntu 18.04 LTS, which runs on virtual machines. It has not been easy and has required research work on some changes made to the Linux kernel versions in recent years. The work on the probe has continued involving Fabio Salvini (the author of the probe), whom I thank for his support.

It was, however, a fun challenge that increased my knowledge both on networks in general and on all the different software involved during development.

The methodology described in chapter 4 was taken up and rewritten in an IETF-draft-compatible format and then submitted as a draft [27].

However, the developed system still has some limitations and is a good starting point for subsequent research works. First of all, the probe implementation. I want to list which are the weaknesses of the probe that should be improved according to me:

- Actually, the probe (coded in eBPF and python) doesn't discard packets related to network interfaces that have not been set in the configuration. It performs all the processing on the eBPF side and then discards unnecessary results in the python side. A very expensive operation, which, however, concerns the compatibility of the probe with the Mininet framework, as described in paragraph 3.2.5.
- Concerning more general aspects, the probe could be redesigned to carry out all the measurements in a single mode. Let me explain. It works in three different modes, each of them makes different measurements, mode 1 deals

with aggregate measurements on the timestamps, while the 3 (which is like the 2 with some improvements) makes single measurements on the hashed packets. To get as many details as possible, it may be useful to create a single operating model that provides for all these measures together, both aggregate and single.

- More importantly, and which constitutes the bottleneck of the probe, is the need to avoid loops. When the number of bits of the hash to be matched increases by 1, the packets that had previously been marked as valid are no longer rechecked on the eBPF side but are all sent user space and rechecked on the python side. This slows down performance. Thinking about a method that minimizes loops would make the probe more efficient.

Even the architecture presented here could be modified to ensure maximum performance in a real case. The thesis, in fact, outlines a possible solution, suitable for simulation work and less for a real application. However, the decoupling of the modules involved in the creation of the system, makes, in my opinion, subsequent work of adjustment of the overall architecture and of the individual components themselves easier.

Other issues that I encountered during the thesis were related to clusters size. The available software is very useful for test configurations. It is possible to configure it by giving a network input and a percentage of the nodes to be monitored and the output returns the monitored network. However, what often happens is that few very large and very small clusters are generated. In my opinion, it would be interesting to have a software that suggests how to arrange the nodes to be monitored to have a more homogeneous distribution of nodes per cluster.

Next works could be related to develop a user interface to manage the system and make them easier to setup and configure.

However, a test over a real network could be the next step to face up. It would provide a straightforward view about what are the more relevant issues to address before to reach a final implementation.

As I said, this thesis is a starting point for subsequent works. In addition to the ones I have listed, there will surely be others. I am sure that the Politecnico di Torino, together with TIM, will try to explore these fields and continue these works.

References

[1] G. Fioccola, A. Capello, M. Cociglio, L. Castaldelli, M. Chen, L. Zheng, G. Mirsky, T. Mizrahi. RFC8321, Alternate-Marking Method for Passive and Hybrid Performance Monitoring. IETF, 2018.

[2] http://www.brendangregg.com/ebpf.html

[3] https://www.kernel.org/doc/html/latest/bpf/index.html

[4] T. Mizrahi, N. Sprecher, E. Bellagamba, Y. Weingarten. RFC7276, An Overview of Operations, Administration, and Maintenance (OAM) Tool. IETF, 2014.

[5] http://mininet.org/

[6] https://flume.apache.org/

[7] https://hadoop.apache.org/

[8] A. Morton. RFC7799, Active and Passive Metrics and Methods (with Hybrid Types In-Between). IETF, 2016.

[9] G. Fioccola, M. Cociglio, A. Sapio, R. Sisto. Multipoint Alternate Marking method for passive and hybrid performance monitoring draft-ietf-ippm-multipoint-alt-mark-02. IETF draft, 2019.

[10] B. Claise, B. Trammell, P. Aitken. RFC7011 Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. IETF, 2013.

[11] N. Duffield, D. Chiou, B. Claise, A. Greenberg, E. Grossglauser, J. Rexford. RFC 5474, A Framework for Packet Selection and Reporting. IETF, 2009.

[12] T. Zseby, M. Molina, N. Duffield, S. Niccolini, F. Raspall. RFC5475, Sampling and Filtering Techniques for IP Packet Selection. IETF, 2009.

[13] 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, Dec. 2019.

[14] F. Salvini. Monitoraggio delle prestazioni di reti a pacchetto con IOVisor. Politecnico di Torino, thesis, 2018.

[15] https://en.wikipedia.org/wiki/Apache_Spark

[16] http://intronetworks.cs.luc.edu/current/html/mininet.html#ip-routers-with-simple-distance-vector-implementation

[17] https://en.wikipedia.org/wiki/Routing_protocol

[18] https://en.wikipedia.org/wiki/IS-IS

[19] https://it.wikipedia.org/wiki/Open_Shortest_Path_First

[20] https://en.wikipedia.org/wiki/Routing_Information_Protocol

[21] https://noxrepo.github.io/pox-doc/html/

[22] https://avro.apache.org/docs/1.9.2/

[23] https://github.com/big-data-europe/docker-hadoop

[24] https://hadoop.apache.org/docs/r2.8.0/hadoop-yarn/hadoop-yarn-site/NodeManager.html

[25] http://www.topology-zoo.org/dataset.html

[26] https://github.com/netgroup-polito/Multipoint-monitoring

[27] M. Cociglio, C. Corbo, G. Fioccola, M. Nilo, R. Sisto. draft-c2f-ippm-big-data-alt-mark-00. The Big Data Approach for Multipoint Alternate Marking method. IETF, 2020.