POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering

Master's Degree Thesis

IP networks monitoring with BigData

A passive approach for large-scale networks' performance analysis

Supervisor:
prof. Riccardo Sisto

Co-supervisor:
prof. Guido Marchetto

Candidate:
Roberto Bressani

Company supervisors:
dott. Mauro Cociglio
dott. Massimo Nilo

December 2021

Abstract

Monitoring of traffic in an IP network is a crucial point to prevent big failures of the system and to deliver a good quality of service to end-users. This is a key factor of success for large-scale Internet Service Providers. For this reason, Telecom Italia developed a series of techniques that have been defined as *Alternate Marking Performance Monitoring*. The main requirements of this system are scalability, robustness, and flexibility. This work aims to develop and test a system able to perform passive network monitoring exploiting Alternate Marking techniques in a multipoint environment.

Starting from several works that have been carried out in the past, critical points that have been highlighted has been addressed, proposing a solution for some of them. In order to powerfully monitor performances, it is necessary to split the network into balanced parts, so that the same level of detail can be detected in each of them. To tackle this problem, it has been necessary to formalize it, and then propose a possible solution that exploits Deep Reinforcement Learning.

The importance of performance metrics is strictly related to the time needed for producing them: data that have been generated almost in real-time can help to anticipate problems. In this work, a possible architecture that is robust and deployable in a real environment has been proposed, also providing the possibility to produce data in a real-time manner.

The proposed architecture has been emulated on a large-scale network, obtaining metrics that have been compared with expected performances. Thanks to these results, the viability of the implemented solution has been shown.

Finally, possible future works have been highlighted in the direction followed by this thesis in order to close the gap between a research and development environment towards a real deployed network.

Table of Contents

1 Introduction 1

1.1 Goal of the thesis . . . 3

1.2 Chapters description . . . 4

2 Alternate Marking Performance Monitoring 7

2.1 Metrics under analysis . . . 8

2.2 Alternate Marking . . . 9

2.2.1 Packet loss . . . 10

2.2.2 One-way mean delay . . . 11

2.3 Multipoint Alternate Marking . . . 12

2.3.1 Clustering . . . 12

2.3.2 Packet loss . . . 14

2.3.3 Packet loss probability . . . 14

2.3.4 One-way mean delay . . . 15

2.3.5 Dynamic hashing . . . 15

3 Network modelling for performance monitoring 17

3.1 Extended network . . . 17

3.1.1 Reduced extended network . . . 18

3.1.2 Border interface selection . . . 19

3.2 Monitored network . . . 22

3.3 The dataset . . . 23

3.3.1 Border routers generation . . . 25

3.3.2 Monitored network algorithms comparison . . . 26

4 Clustering 29

4.1 Clustering algorithm . . . 29

4.2 Clustering optimization . . . 31

4.2.1 Problem desiderata . . . 31

4.2.2 Clustering approach . . . 32

4.2.3 Clustering evaluation . . . 32

4.3 Clustering via reinforcement learning . . . 33
4.3.1 Reinforcement learning . . . 33
4.3.2 Deep Q-learning . . . 35
4.3.3 Graph processing through Neural Networks . . . 36
4.3.4 Optimization framework implementation . . . 39
4.4 Optimization results . . . 40
4.4.1 Transfer learning application . . . 42
4.4.2 Final considerations . . . 44

5 Overall system architecture 45
5.1 Network probe . . . 45
5.2 Message queue . . . 46
5.3 Cluster manager . . . 47
5.4 Real time preprocessing . . . 48
5.4.1 Data synchronization . . . 49
5.4.2 Results organization . . . 49
5.5 Per flow post processing . . . 54
5.5.1 Loss computation . . . 54
5.5.2 Time based measurements . . . 55

6 System implementation 57
6.1 Network probe . . . 57
6.1.1 eBPF and BCC . . . 57
6.1.2 Architecture . . . 58
6.2 Message queue . . . 61
6.2.1 Apache Kafka . . . 61
6.2.2 Configuration . . . 62
6.3 Cluster manager . . . 64
6.3.1 REST endpoints . . . 64
6.3.2 Message handling . . . 65
6.4 Real time preprocessing . . . 65
6.4.1 Apache Spark . . . 66
6.4.2 Hadoop Distributed File System and Hive database . . . 66
6.4.3 Architecture and implementation choices . . . 68
6.5 Per flow post processing . . . 70
6.5.1 Queries' and results' format . . . 70

7 Test environment 73
7.1 Network emulation . . . 73
7.1.1 Mininet . . . 73
7.1.2 IPMininet . . . 75

7.1.3 Naming conventions 75
7.2 Traffic generation 76
7.2.1 Iperf network measurement tool 76
7.2.2 Flows emulation 76
7.3 Test hardware setup 77

8 Results 79
8.1 Preprocessing 79
8.1.1 Timing results 79
8.1.2 Loss related measures 80
8.1.3 Delay measures 82
8.1.4 Jitter metrics 85
8.2 Post processing 88
8.2.1 Loss related measures 88
8.2.2 Delay measures 89
8.2.3 Jitter metrics 90
8.3 Overall results considerations 91

9 Conclusions 93
9.1 Possible future works 93

Appendix A Practical simulation instructions 97

Bibliography 103

Chapter 1

Introduction

Internet applications nowadays require more and more demanding requirements to support real time applications, such as video conferencing systems, multimedia streaming and online gaming. This problem is growing day by day, with new technologies introduced and increasing network traffic.
It is crucial for every Internet Service Provider (ISP) to supply to users a quality of service that satisfies them and complies with business contracts' requirements. To ensure that, loss, delay and jitter of network packets belonging to these real time applications should be constantly monitored and kept as low as possible. Another key point is to detect problems in the network as soon as they occur. In order to do so, both a complete overview and a detailed one is necessary to address the root cause of each problem that occurs.

Several tools for performance monitoring exist, but have some limitations, starting from the fact that usually they are proprietary, forcing companies to stick to a single vendor, representing a big business risk. In addition, they are usually protocol specific and not capable to scale up to deal with the amount of traffic that characterizes a national ISP.
Another peculiarity is that performance monitoring is usually carried out as corrective action to respond to a network failure. Typically, probes are activated inside the network to capture flows of traffic that has been affected by a network problem, monitor them and analyse the results to address the source of the problem. This, however, requires manual configurations, thus implying a non-negligible time to solve problems.

In this context, TIM - Telecom Italia has proposed and developed a series of theoretic foundations to provide a different approach to tackle network monitoring problem. The main idea is to constantly monitor the whole traffic across all the network, thus limiting the manual configuration need to monitor single flows of traffic and being able to rapidly address the root causes of problems and even anticipate them. These techniques, which find their basis in `RFC8321` [26], emphasise several aspects:

- ease of implementation: the system should be deployed on a real existing network, without the need to build it from scratch
- flexibility: the system should be able to provide data at different level of detail, that can be configured and rapidly deployed in the network
- multiflow analysis: the system should be able to analyse flows of traffic of a real network, with several sources and destinations, possibly also the whole traffic that crosses the network
- robustness: data generated should not be biased from out-of-order packets and from traffic generated by the network itself (e.g. routing packets end in internal routers, and can be marked as lost packets)
- applicability: network stack should not be relevant, so the system can work with every kind of Internet protocol
- low computational effort: the impact of the system on the overall network performance should be negligible

In addition to this standardization process, several thesis works have been previously done to validate and test this methodology. Among them, the ones that are at the basis of this work are:

- [19] developed a complete system proof of concept, by using a network probe to capture traffic flows from another work [28]. This is key point in the definition of the *BigData* approach. It consist in a framework that allows to analyse the whole traffic flowing across the network, collecting a subset of packets to be stored in a huge database. These information can be processed whenever they are needed.
  The probe that has been used however, has some limitations and was developed in parallel with the standardization phase, when the idea of this technique was not completely defined.
- [23] provides a fresh implementation of the network probe, developing an *ad-hoc* solution to a technique that during the years has been more clearly defined.
- [16] which integrates work done by [19] and [23] and was carried out partially in parallel with this thesis. This is a step towards a deployment in a real system: work by [19] provides a good overall architecture for the *BigData*, but

technologies and assumption made are far from a real and scalable solution. [16] tries to close the development gap by modifying the technology used to send data from the network probe to the processing system, which will be further discussed later on.

1.1 Goal of the thesis

This thesis finds its main objectives on the problems that mainly arose during the previous work [19]. These are mostly focused on two big areas:

1. where to place the measure points so that as meaningful as possible data will be obtained by capturing traffic. This implies primarily two aspects:

    - the network probe [23] has very low performance when capturing traffic that flows out a Network Interface Card (NIC), due to a missing Linux kernel primitive to obtain timestamp for packets on output interfaces. All the analysis and system will be carried out by intercepting only packets coming into the NIC
    - network probes should be inserted in crucial points in the networks, so that monitoring areas, that later on will be defined as clusters, should be in some way balanced and the network should be split in similar subparts

    An optimization technique to give such improvements has been researched during this work, allowing to produce results with higher quality. Since no analytic method has been found to derive an optimal solution for this problem, a technique based on deep reinforcement learning was proposed.

2. the analysis system developed by [19] was just a proof-of-concept to validate multipoint analysis [20] and big data approach [18], but it has not been designed to be deployed in a real system.
    In order to overcome this limitation, a system made up of two different components has been propose. The first one aims to:

    - build an *online* system, able to consume data and produce results as soon as data are collected. To do that, some synchronization mechanism has to be implemented
    - obtain all the metrics that have been theorized in [20] and [18]
    - have the possibility to process data also in case of missing captures from probes, since failures may arise in a real and distributed environment and the system should be robust enough to deal with these situations

The main strength that the implementation of this component aims to provide is to produce and store as much detailed as possible network performance metrics that have been periodically measured in real time. This can allow to analyse and react rapidly to problems that may arise.
The second component, which has been designed to deal with a big amount of data and to scale up, satisfying a huge number of parallel queries, thus optimizing its performance. Additionally, the aim of this component is to produce metrics to be used in reports for end-users about the quality of service of their traffic. Thus, differently from the proposal by [19], there is no need to analyse this kind of traffic in every part of the network, but the main goal of the results is to provide a global vision of this performance.

In addition to those improvements, this work has also designed an architectural component to manage the network of probes, that tries to close the gap between a test environment and a real deployment. It will allow in a real environment, to provide an automatic configuration of probes, to notify their status to other components, thus providing a robust system that may also deal with probe failures. This part, however, is not complete and is just to demonstrate a possible implementation, leaving space for future works.

1.2 Chapters description

In this section, a brief overview of each chapter will be given.

- Chapter 2 will present the theoretic background of performance monitoring techniques developed by Telecom Italia, starting from the *Alternate Marking* and following its evolution

- Chapter 3 will describe how a network has to be modelled in order to apply Alternate Marking Performance Monitoring techniques. The target formalization of the network is a model that represents from a logical point of view the possible links among measurement points. In order to reach this goal, it has been necessary to define an intermediate model that represent the network from a point view of devices' interfaces. In this chapter, these formalizations will be exposed in detail, together with the available algorithms. Additionally, an adaptation to support the goals of this thesis and a brief analysis on algorithms' performance will be presented

- Chapter 4 will define the algorithm to divide a network into clusters, that is to say portion of the network that will be the base units on which to compute metrics. In addition, a possible technique to optimize the network clusterization will be exposed, presenting all the theoretic foundation behind it and the results that can be obtained

- Chapter 5 will describe the overall architecture of a system able to support Alternate Marking Performance Monitoring, highlighting the improvements that have been carried out during this work
- Chapter 6 will deepen the implementation of elements that have been exposed in chapter 5, also describing all the technical aspects that have been used to provide a functioning system
- Chapter 7 will analyse the environment in which tests have been carried out, describing issues that have been faced to have meaningful results, without having the possibility to deploy the system in a real core network
- Chapter 8 will include all the results that have been collected during this work, comparing them with the expected values that can be derived by the simulation environment that has been used
- Finally, chapter 9 will summarize the results that have been collected during the whole work, highlighting the strength and weakness of this implementation, providing also some hints on possible future works that can be carried out during other thesis
- In addition, appendix A has been provided as practical explanation of commands and useful code to carry out a complete simulation of the system that has been implemented during this work, together with the previous ones

Chapter 2

Alternate Marking Performance Monitoring

Collection of data about network performance is a key point for an ISP business, so it is fundamental to collect them as reliably as possible. Several techniques over the last decades have been developed to satisfy this need, that can be classified as *passive* and *active*. According to [32], the former techniques' methodology consists in the observation of existing data traffic to determine the metrics to analyse the network. On the other hand, active techniques use network traffic that has been synthetically generated to measure all the metrics. This solution is easy to implement, it does not require specific network configuration and several well-known tools exist (e.g. `ping`, `traceroute`). However, this method introduces some kind of bias in the measures themselves, since generated traffic influences the network condition. Also some *hybrid* techniques can be implemented, by observing traffic that is already crossing the network, cloning it and applying active techniques to extract performance measures.
Considering these points, in 2008 Telecom Italia started to develop a passive technique defined as *Alternate Marking Performance Monitoring* (AM-PM) that aims to scale up in terms of quantity of data and number of different flows that it is able to analyse, also minimising size of storage that has to be used. During the years, several publications, RFCs and IETF drafts has been submitted. This work is mainly based on some of them:

- [26] and [25] introduce the *Alternate Marking* technique, which will be explained in section 2.2 and is the starting point to measure performance of a single flow of traffic

- [20] and [24] extends the *Alternate Marking* technique to support the analysis of multiple flows of traffic that will be exposed in section 2.3

- [18] formalises what has been done during a previous thesis [19], providing the description of a working architecture that is the starting point for this work and will be described in details in chapter 5

During this chapter an overview of this theoretical foundation will be given, analysing the evolution of the *Alternate Marking* over time.

2.1 Metrics under analysis

Before describing the enabling technologies for this thesis, a brief overview and a clear definition of the metrics to evaluate the performance is given.
Considering a flow of N packets between two hosts `A` and `B`, the crucial metrics for network monitoring in this work are:

- **packet loss**: for a flow is defined as

$$\sum_{i=1}^{N} p_i \tag{2.1}$$

  where p_i is the packet loss for a single packet defined in [31] and has value 0 when the packet reaches host `B`, and is 1 whenever either the packet has not reached the destination after a bounded delay (typically its lifetime) or one of its fragment or the whole packet itself are corrupted.

- **one-way mean delay**: is computed for each flow as

$$\mathbb{E}_i[d_i] \tag{2.2}$$

  where d_i is the delay for each packet and $\mathbb{E}_i$ is the mean operator over a series of data obtained varying the value of i. As defined in [30], it is computed for each non-lost packet as the difference between the timestamp at host `B` when last bit is received and the timestamp at `A` when the first bit is sent and is undefined whenever a packet is lost (a condition exposed at previous point occurs). Considering that, in this work will be computed for every packet that has a match in host `B`. Additionally, it is not possible to apply literally the definition of [30], since only input traffic flowing into interfaces will be captured. As a result, the delay will be the difference among the timestamps captured when the packet is completely received into two different network devices.

- **mean jitter**: is computed for each flow as

$$\mathbb{E}_i[d_{i+1} - d_i] \tag{2.3}$$

where d_i is the packet delay used at previous point. This definition given in [39] represents the variability in packet delay traversing the path from `A` and `B`. Even though this metric has not been directly used during *Alternate Marking* definitions, it will be part of the analysis and implementation of the system. In this work, also the standard deviation of this data distribution will be considered to evaluate its variability.

2.2 Alternate Marking

The *Alternate Marking* has been defined in `RFC8321` [26] and proposes a framework to measure performance of a group of packets. The technique has been designed to work at different levels to monitor different quantities:

- per link values: all the traffic that crosses a link is considered and all the packets have to be taken into account to monitor the status of a link. Data should be collected both on the incoming router and in the out-coming one to monitor the performance
- per flow values: packets belonging to a specific flow, that is to say a group of packets that follows the same path and meets specific constraints is considered. In this case the aim of the analysis is the performance of the flow itself. Depending on the desired level of detail, data should be collected only analysing traffic on some network interfaces in the path traversed by the considered flow

Packet *colouring* is the main foundation of the *Alternate Marking*. It consists of marking groups of consecutive packets with a common label (or colour), that will be changed basing on different criteria and can be obtained by modifying some bits of the packet:

- fixed number of packets: the colour of the group of packets should be changed when the same colour has been applied to N packets
- temporal: a single colour is used for a fixed period of time, then changed. In this way the analysis will be carried out for a varying number of packets, but will be representative for a fixed time period. This is the approach preferred by the authors of [26] and that will be used in this work, since it can be easily extended in case of multiple flows

The number of labels that should be considered, whether the period is great enough, to doubtless distinguish blocks among the others is two (possibly having a label for packet that are not marked, especially for flow based measures). Thus, the traffic observed by a router that monitors looks like figure 2.1.
For every block, some statistics about packets that have been marked can be

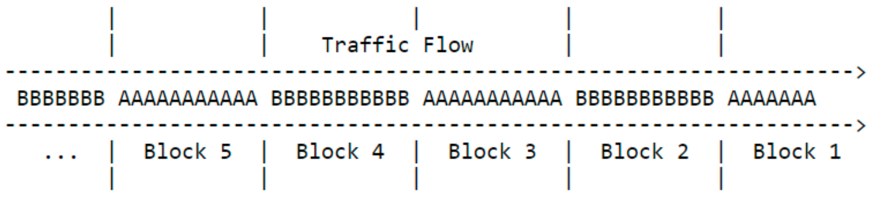

Traffic Flow
BBBBBBB AAAAAAAAAAA BBBBBBBBBBB AAAAAAAAAAA BBBBBBBBBBB AAAAAAA
...
Block 5
Block 4
Block 3
Block 2
Block 1

Figure 2.1: Example of coloured flow [26]

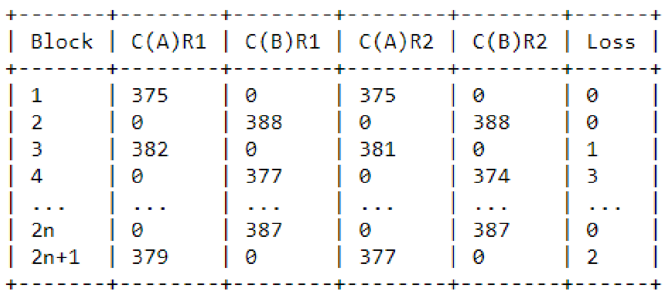
computed by analysing traffic thanks to network probes. In an ideal case, when the period ends these values are stable and can be correctly collected to be aggregated with data coming from the other routers. In a real case, however, this assumption is not valid, since delays may occur and, during periods' transition, interleaved out of order packets may be found and also clocks' skews between different devices may arise. Therefore, the best moment in which statistics about the block can be considered stable is $L/2$ after the end of the period of this block, where L is the duration of the period. Under these conditions, delays $d < L/2$ can be correctly detected and won't affect the measures.

2.2.1 Packet loss

Packet loss is the simplest metric that can be computed with the technique that has been previously exposed. Given a single link starting from `R1` and ending in `R2`, just two counters for each period should be considered to estimate the loss that has affected the link for that period. By defining $C(j)Ri$ the value of the counter for packets of colour j for the i-th device, the packet loss for each colour for each block will be defined as:

$$C(j)R1 - C(j)R2 \tag{2.4}$$

Obviously, the loss will be alternatively computed on the two colours, depending on the colour that has been assigned to a specific block. By considering this, figure 2.2



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

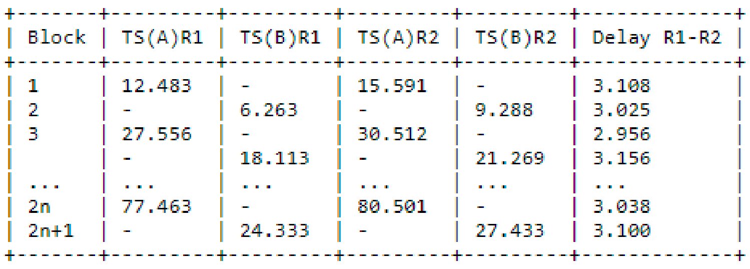
Figure 2.2: Example of packet loss computation between two devices [26]

is showing a possible condition that may arise.
The fact that this method allows to compute data for a single direction of the traffic should be considered. To detect measures in the opposite direction, analogue counters and also data structure that will be exposed in following subsections should be taken into account, by reversing the order of nodes considered and adding other counters.

2.2.2 One-way mean delay

In *Alternate Marking* definition [26], several techniques to compute mean delay are proposed:

- single marking: a packet every N in the block should be captured and its timestamp should be stored on each measure point. Then, by coupling those packets, the delays can be computed and the mean delay obtained by averaging these values. This technique is sensitive to loss, since packets that will be considered may not match each other on different devices. In addition, depending on N value, several packets should be collected, without any upper bound, since it depends on the amount of traffic that flows into the considered area of the network



```
+-------+---------+---------+---------+---------+-------------+
| Block | TS(A)R1 | TS(B)R1 | TS(A)R2 | TS(B)R2 | Delay R1-R2 |
+-------+---------+---------+---------+---------+-------------+
| 1     | 12.483  | -       | 15.591  | -       | 3.108       |
| 2     | -       | 6.263   | -       | 9.288   | 3.025       |
| 3     | 27.556  | -       | 30.512  | -       | 2.956       |
|       | -       | 18.113  | -       | 21.269  | 3.156       |
| ...   | ...     | ...     | ...     | ...     | ...         |
| 2n    | 77.463  | -       | 80.501  | -       | 3.038       |
| 2n+1  | -       | 24.333  | -       | 27.433  | 3.100       |
+-------+---------+---------+---------+---------+-------------+
```

Figure 2.3: Example of mean delay computation with single marking (considering just one packet for each period)[26]

- mean timestamp: equation (2.2) can be rewritten as:

$$\mathbb{E}_i[d_i] = \mathbb{E}_i[t_{OUT_i} - t_{IN_i}] = \mathbb{E}_i[t_{OUT_i}] - \mathbb{E}_i[t_{IN_i}] \tag{2.5}$$

  where t_{OUT_i} and t_{IN_i} are respectively the timestamp of the i-th packet captured at the end and at the beginning of the link (or of the flow). By simply collecting the average timestamp of marked packets on each node, the delay can be computed, even though packet losses may influence it. This is one of the techniques that has been considered during the implementation of this work.

- double marking: by introducing another marking bits in each packet, some packets may be selected at the source node as target for mean delay computation as shown in figure 2.4. Timestamp of these packets are stored similarly to single marking case, but an upper bound for packets that has to be collected can be imposed.

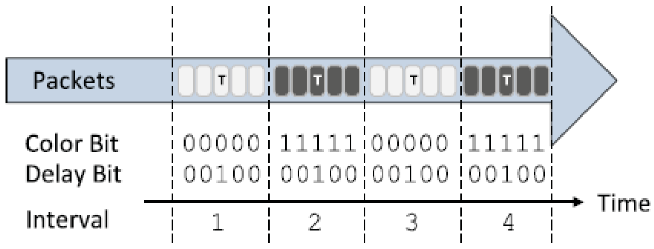


Figure 2.4: Example of double marking for delay computation [25]

Even tough this solution is not directly used in this work, it will be the basis for sampled measures, that will use dynamic hashing, thus removing the need of an additional marking bit

2.3 Multipoint Alternate Marking

Alternate Marking as defined in [26] provides a methodology to monitor a single flow of traffic. In order to apply these techniques on several flows of traffic, a probe, with all relative data structures (packet counter and mean timestamp) has to be set up for each monitored flow. However this solution cannot fit the needs of a large ISP, since number of flows can be unbounded and flows change very rapidly and it is not possible to monitor all of them under these conditions.
To fully satisfy these needs, `RFC8889` [20] has been introduced. It defines a methodology that allows to compute performance metrics also on traffic that does not follow a single path.
This kind of traffic, together with the description of the one of an entire network, has been categorized in several different ways as described in figure 2.5.

2.3.1 Clustering

Before defining how to compute the metrics for performance monitoring when dealing with multipoint flows, it is fundamental to introduce the concept of clustering.

A cluster is a subnetwork that allows to coherently compute performance monitoring. Its fundamental property is that all the traffic that is captured on its input monitoring points must be detected also on the egress ones or possibly it will be

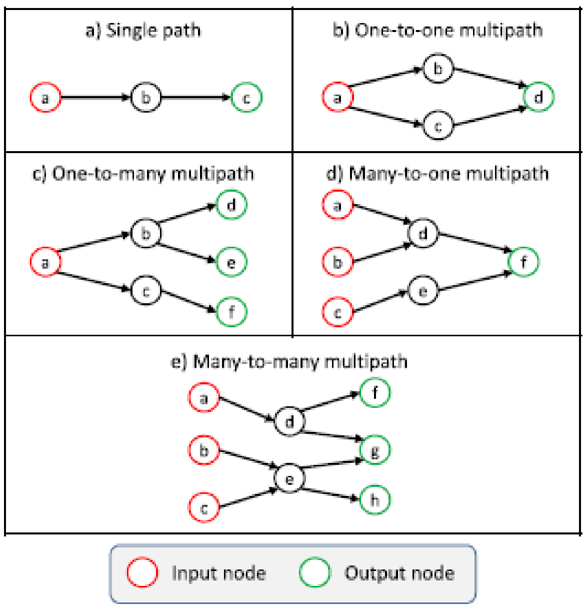


Figure 2.5: Definition of possible multipoint flows[24]

lost inside it.
As also for unicast flow of traffic on a single path, monitoring nodes can be categorized as input, output and intermediate for every cluster, depending on the position they occupy inside the cluster.
The example in figure 2.6 shows a subnetwork (selected via red and green nodes) that does not satisfy the requirements to be classified as cluster: traffic that goes from node `b` to node `d` is only detected when passes through `b`, thus breaking the idea behind clusters. By including node `d` as output node, will result in a clustering that respects the definition.

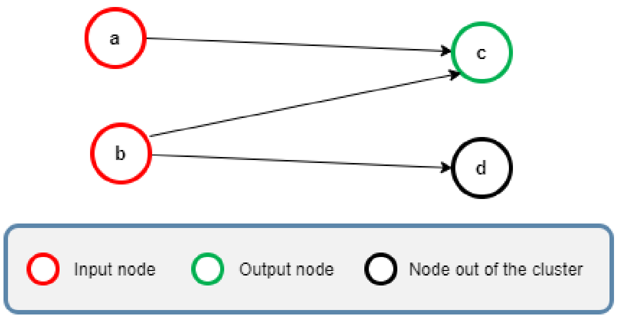


Figure 2.6: Example of wrong clusterizaion

Inside a single network, several possible clusters can be found: clusters can be grouped together to form a bigger one, maintaining the right property previously exposed. Thanks to this, is it possible to monitor a network with various *zooming* levels, without the need to define different algorithms to compute the metrics.
Here, a brief introduction of clustering has been given, but there are other foundations, such as clustering algorithm and how to partition a network into several and balanced clusters, that will be exposed in the following chapters.

2.3.2 Packet loss

Packet loss can be easily generalized to multipoint case. Counters to measure number of packets that flows in and out the network has to be kept, one on each monitoring point in the network. Defining $b_i(p)$ the value of this counter at period p for the *active* colour for the i-th monitoring point and defining two sets of measure points IN and OUT including respectively input and output nodes of the cluster, packet loss can be computed as:

$$\sum_{i \in IN} b_i(p) - \sum_{i \in OUT} b_i(p) \tag{2.6}$$

2.3.3 Packet loss probability

When dealing with a unicast flow of traffic, the root cause of a packet loss can be theoretically found immediately: a loss that occurs in the path from a router `R1` and a router `R2` may be due to a link failure or possibly to a buffer overflow in the destination node.
In a multipoint environment instead, a loss in a cluster may arise in different positions. In [24] the number of packets that have been probabilistically lost around a measure point at the border of the cluster has been proposed. This value is defined differently, depending on the nature of the measure point. For an input node i for a period p, the downstream loss $l_{ds}(i, p)$ value corresponds to the number of packets that traversed this node and will be probably lost afterwards in a given period is computed as

$$b_i(p) \cdot \frac{L}{\sum_{j \in IN} b_j(p)} \tag{2.7}$$

where $b_i(p)$ is the packet counter of active colour and L is the loss of the cluster, as previously defined.
Similarly, for every output measure point in the cluster an upstream loss $l_{us}(i, p)$ value can be computed to represent the packets that are probably lost during the path towards the node i at period p. It can be obtained as:

$$b_i(p) \cdot \frac{L}{\sum_{j \in OUT} b_j(p)} \tag{2.8}$$

Both values, if collected separately for nodes belonging to input and output set of nodes, will sum up to L and consequently represents a possible split of lost packets between different paths in the cluster.

2.3.4 One-way mean delay

A procedure similar to the one used for loss can be followed to extract one way mean delay. Other strategies such as double marking or single marking that have been exposed before should be abandoned, since packets can follow several paths. By having only the information about the timestamp, packets cannot be coupled and consequently delay metrics cannot be computed.
To extract the one-way mean delay, average timestamps t_i should be collected at every measure point, then these values should be further aggregated for the input and the output, weighting them for the number of packets that flows into each node. As a result, the mean delay for a given period p will be computed as:

$$\frac{\sum_{i \in OUT} t_i(p) \cdot b_i(p)}{\sum_{i \in OUT} b_i(p)} - \frac{\sum_{i \in IN} t_i(p) \cdot b_i(p)}{\sum_{i \in IN} b_i(p)} \tag{2.9}$$

This result is meaningful only in the case that no loss occurs. In order to have an accurate and robust metric, it is not possible to rely only on aggregate values, but it is necessary to sample packets that have to be coupled between input nodes and output nodes. To overcome limitation of double marking technique, dynamic packet hashing has been introduced and will be analysed in the following subsection.

2.3.5 Dynamic hashing

As previously exposed and as will be proven later on in this work, loss is the only metric that have been proposed in *Alternate Marking* definitions to be robust enough to work in every condition. Delay values instead give useful results only in case of a network that does not produce any loss, which is quite impossible in a real network.
`RFC8889` [20] also combines the strength of *Alternate Marking* for multipoint flow analysis with packet sampling [38, 37], another powerful technique that has been developed in the last decades to collect random data packets.
Packet hashing can be used to select packets to be used to compute network performance measures, but [37] does not give any advice on how to distinguish packets in blocks and give bounds to the number of collected packages. With *Alternate Marking* and dynamic hashing, these problem has been addressed.

The target of dynamic hashing is to select at most `NMAX` packets for each measure point and every period. In order to do this, a common reference hash has

to be defined for all the nodes. In addition a number of bits n to be matched has to be stored inside every measure point: this value is initialized to 0 at the beginning of the period and possibly incremented later on. As soon as a packet is captured by a node, its hash is computed and whether its first n bits are equal to the corresponding bits of the reference hash, the packet is sampled and relevant information about it are temporary stored.
Once that `NMAX` packets have been collected by a single measure point, the value of n is incremented by 1 at the measure point and all the packets previously collected are re-evaluated with this more restrictive filtering. By considering hash distribution as uniform, most likely half of the packets will survive. By repeating this procedure until the end of the period, a number of packets that with high probability is above `NMAX`/2 and less than `NMAX` will be collected and can be further processed.

This technique will be applied to *Alternate Marking* as improvement of double marking technique for one-way mean delay computation in the implementation of the system.

Chapter 3

Network modelling for performance monitoring

In order to analyse a network, a proper modelling for it is fundamental: it is not sufficient to formalise the network as a graph made of nodes to represent network devices and links to model the physical wire that put two nodes in communication. This is not the most precise level of detail that can be obtain. The target of the monitoring phase are interfaces of network devices. It is so necessary to provide a deeper model to consider that situation. Additionally it is possible to build a model, starting from the previous that allows to represent a logical structure that stands behind monitoring points, thus not considering the interfaces that are not monitored. This model is crucial, since it is the one that allows to build clusters and the one considered by the monitoring system for the analysis.
In this chapter, all the necessary models of the network and algorithms to generate them that have been previously formalized in [24] will be recalled.
Considering the requirement of using only traffic captured in input interfaces to produce results, during this work, some algorithms and models have been adapted to better match this situation.
In conclusion, an overview of the datasets that have been used in this thesis will be given, together with all the needed transformations that have been applied to it. Additionally, tests were carried out to evaluate the time performance of different implementations of the same algorithm so that it can be possible to derive the condition where to apply each implementation.

3.1 Extended network

At a first glance, the network can be simply modelled as an undirected graph, where nodes are network devices (routers, switches or any other, depending on the

protocol that is the target of the monitoring) and edges are the physical links that put them together.
Network devices, however, are not the simplest part of the network that can be monitored. When dealing with the IP stack, for example, there may be no need to capture traffic flowing into routers from all directions and going towards all the others. Here comes the need of having the possibility to select only some network interfaces on which packets will be monitored and statistics will be computed. For this reason, it is necessary to build a model that has a greater level of detail.
It is so possible to define a directed graph, that has been defined as *extended network* in [24], whose nodes are network interfaces. Precisely, each network interface should be split into two nodes, one that will represent its input part and one representing the output. Each input interface is linked to all the output interfaces of the corresponding router and each output interface is linked to the input interface of the router at the other end of the physical link, as it has been reported in figure 3.1.

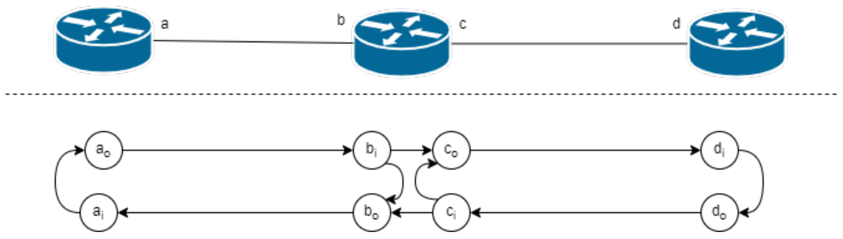


Figure 3.1: Example of extraction of extended network from a simple network topology. Inspired by [24]

By following this definition, every possible sequence of interfaces that packets encounter can be replicated correctly on this model.

3.1.1 Reduced extended network

Considering the requirement to use only input interfaces and the fact that graph processing has high complexity, during this work it has been considered to update the network extension, halving the number of nodes and consequently also edges. This model has been defined as *reduced extended network*.
In this scenario, to generate a model with the same properties of the previous one, but that contains only the input part of each interface, algorithm 1 has been defined.
Following this approach, the example shown in figure 3.1 will be update as shown in figure 3.2. As before, the logic sequences of interfaces that packets can follow can be reproduced also on this model.

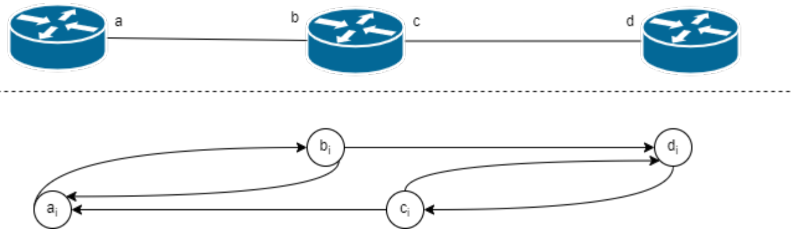


Figure 3.2: Example of extraction of reduced extended network

Algorithm 1 Algorithm proposed in this work to extract the reduced extended network for a network topology G considering only input captures.

```
1: procedure EXTRACT REDUCED($G$)
2:     ▷ Initializing the resulting graph as empty graph
3:     reduced_graph ← DirectedGraph()
4:     ▷ Iterating over all interfaces in the network graph $G$
5:     for all interface ∈ $G$.interfaces do
6:         ▷ Considering the router to which the interface belongs
7:         router ← interface.router
8:         ▷ Iterating over the set of interfaces that are connected to router under analysis
9:         for all interface2 ∈ router.linked_interfaces do
10:            ▷ Adding a new link between interface and interface2 link to resulting graph
11:            reduced_graph.add_link (interface, interface2)
12:        end for
13:    end for
14:    return reduced_graph
15: end procedure
```

3.1.2 Border interface selection

Another fundamental point in performance monitoring is to define the border of the monitored area. This interfaces will be fundamental in the monitoring phase and a measure point must always active on them. During this work formal definitions to these elements have been given, that will previous work were not deepened:

- *ingress* interfaces will be the first one that a packet crosses entering the monitoring area. They must be responsible of marking the incoming traffic
- *egress* interfaces will be the last monitored interface crossed by packets. They possibly should be configured to unmark packets, so that traffic is not seen altered from a point of view that is external to the monitored network

During this work, a case of network made up of a backbone area, and a surrounding access one has been considered. Access devices, such as DSLAM, will represent the edges of the network.
When analysing also traffic on the output of the interface, it is just a matter of selecting the interfaces of the access devices that are linked to the core monitored area, thus enclosing the whole network target of the monitoring. In other words, in case of no loss, all the traffic that flows out from ingress interfaces must be detected flowing in at another egress interface as depicted in figure 3.3.

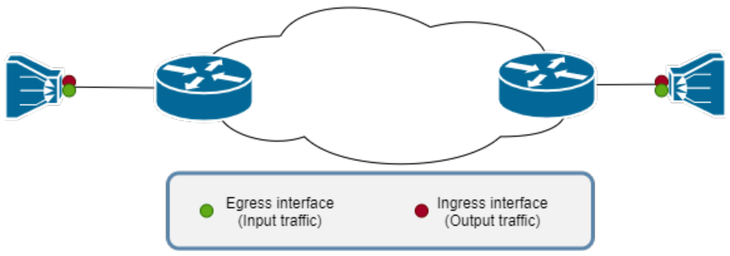

● Egress interface (Input traffic)
● Ingress interface (Output traffic)

Figure 3.3: Example of capture on borders on an access device when it is possible to capture also output traffic

In the case of captures only in input part of each interface, this solution cannot be applied. Capturing both traffic that flows in and out the network in the access devices (as depicted in figure 3.4) has not been considered as a viable solution: access devices can have hundreds of ingress interfaces and configuring a probe on each of them may not be trivial. This is because devices may have limited performance that does not allow to support a huge amount of probe instances or simply, they may also evolve so rapidly that reconfiguration will be needed too often.

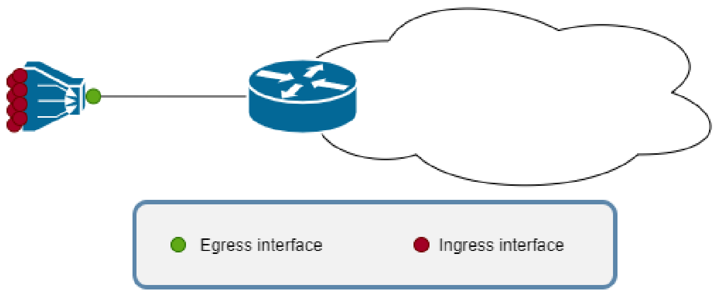


Figure 3.4: Example of capture on borders on an access device

In order to overcome these limitations, the capture of traffic that flows into the network has been moved a step forward in the network. Considering as first monitored interface the one belonging to the first core router a possible workaround can be found. This structure however has a strong limitation: considering a bidirectional flow of traffic, the areas of the network that will be monitored in the two direction will be different. Observing figure 3.5, where an example of this principle is shown, traffic that goes from host `A` towards `B` will be monitored in its path between interfaces `b` and `d`. In the opposite direction, in contrast, the path that will be monitored is from `c` and `a`. This issue has prevented this work from dealing with round-trip measures: values obtained would be biased from the architecture itself and would be meaningless.

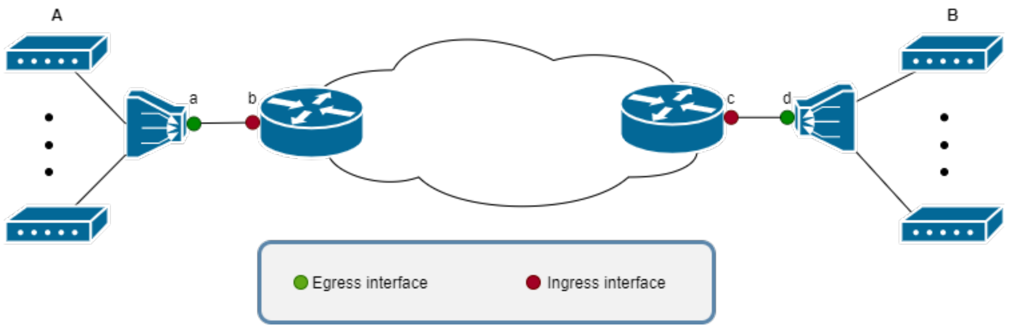


Figure 3.5: Example of capture on borders on core network devices

As a result, a minimal reduced extended network that will be considered as target of the analysis from now on is depicted in figure 3.6.

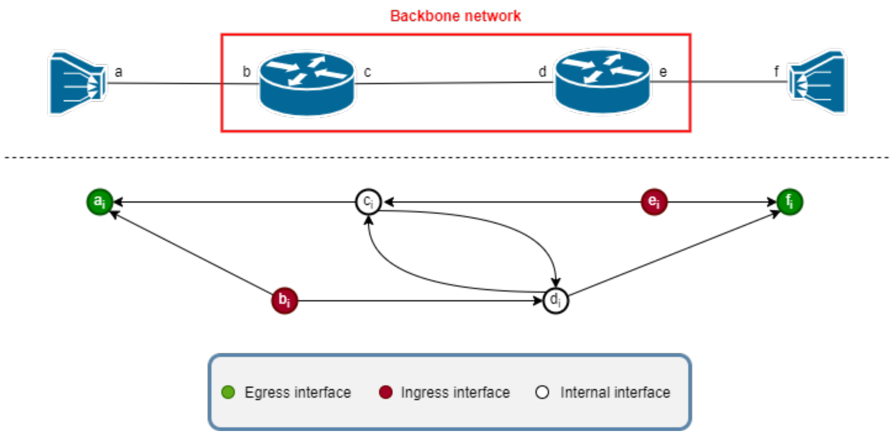


Figure 3.6: Example of capture on borders on core network devices

3.2 Monitored network

Inside the monitoring area, potentially, not all the interfaces will be monitored. This may due to different needs, such not overloading devices, incompatibility of existing devices with the probe software or the need of not producing too much data to be processed. From a logical point of view, it is necessary to introduce another network model, that will include among its nodes only interfaces that are measure points.
In [24] this model has been defined as *monitored network* and it is computed starting from the extended network. In the monitored graph, each monitored interface will have a directed link towards another monitored one whenever there exists a path between them that does not cross any other measure point. From this definition comes the algorithm 2 to extract this model from the extended one and the resulting modelling can be seen in figure 3.7.

Algorithm 2 Algorithm to extract the monitored network from an extended graph $\bar{G}$ and the list of monitored interfaces `monitored_list` [24]

```
procedure EXTRACT MONITORED V1($\bar{G}$, monitored_list)
    ▷ Initializing resulting graph as empty graph
    monitored_graph ← Graph()
    ▷ Iterating over all possible couples of monitored interfaces
    for all interface1 ∈ monitored_list do
        for all interface2 ∈ monitored_list do
            ▷ Verifying that are not the same interface
            if interface1 ≠ interface2 then
                ▷ Searching whether exists a path among them
                path ← $\bar{G}$.path(interface1, interface2)
                if path exists and path ∩ monitored_list = ∅ then
                    ▷ Building a new link whenever a non monitored link exists
                    monitored_graph.add_link(interface1, interface2)
                end if
            end if
        end for
    end for
    return monitored_graph
end procedure
```

Algorithm 2 has a quadratic complexity in the number of monitored interfaces, which has to be multiplied by the path search algorithm one. It may not be convenient when number of monitored interfaces is too high. To overcome this, algorithm 3 was theorized by one of *Alternate Marking* authors while developing a

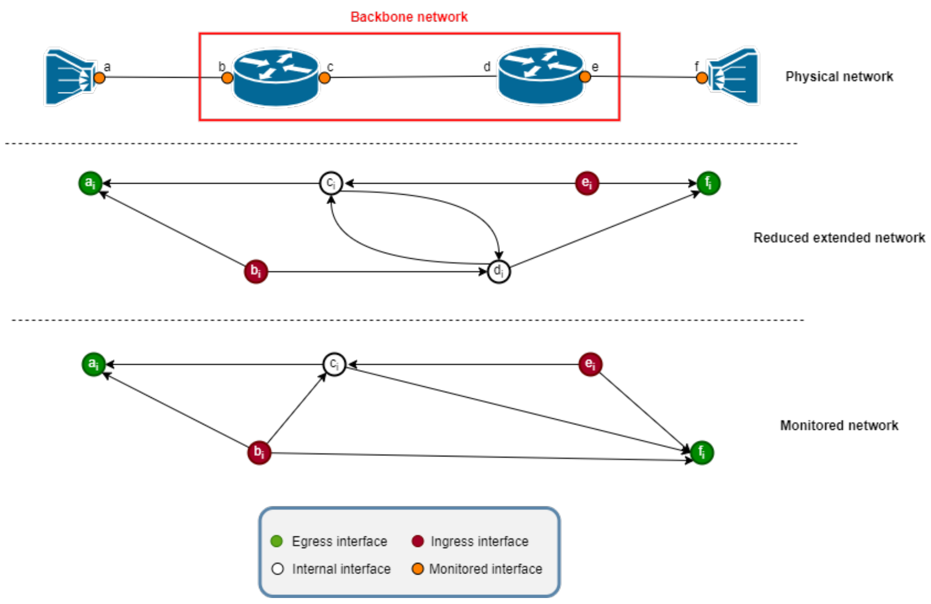


Figure 3.7: Example of monitored graph extraction starting from the previous example of reduced extended

work on a different field [17] but sharing the same principles, basing on the concept of merging links that involve non monitored nodes. It has a complexity that may be represented as $\mathcal{O}(n \cdot max_{in} \cdot max_{out})$, where n is the number non monitored interfaces, max_{in} is the maximum number of input links in every node in the graph and max_{out} is the maximum number of output ones.
It has to be noticed that the extraction of the monitored network can be carried out either using as starting point the extended graph or considering the reduced version. The second approach will just decrease the complexity of the problem, since in case of algorithm 2 the path search will be applied to a simpler graph, while in case of algorithm 3 the number of non monitored interfaces will not include the output ones.

3.3 The dataset

All the analysis that have been carried out during this work are based on network topology that are freely available in `Zoo` dataset [36]. It includes several network topologies represented as **`graphml`** files[1] that have been collected from different sources, representing real networks at various scales spread all over the world.

[1]https://en.wikipedia.org/wiki/GraphML

Algorithm 3 Improved algorithm to extract the monitored network from an extended graph $\bar{G}$ and the list of monitored interfaces `monitored_list` [17]

```
 1: procedure EXTRACT MONITORED V2($\bar{G}$, monitored_list)
 2:     list_1 ← [ ] ▷ Will contain edges that have at least a non monitored node
 3:     list_3 ← [ ]          ▷ Will contain edges that have only monitored nodes
 4:     ▷ Iterating over edges of extended graph to assign them to the correct list
 5:     for all (start, end) ∈ $\bar{G}$.edges do
 6:         if start ∈ monitored_list and end ∈ monitored_list then
 7:             list_3 ← list_3 + (start, end)
 8:         else list_1 ← list_1 + (start, end)
 9:         end if
10:     end for
11:     ▷ Computing nodes that are not monitored by difference between all nodes
    and monitored ones
12:     non_monitored ← $\bar{G}$.nodes ∖ monitored_list
13:     for all node ∈ non_monitored do
14:         ▷ Initializing list with nodes that are directly reachable from current node
15:         start_list ← [ ]
16:         ▷ Initializing list with nodes that can reach directly current node
17:         end_list ← [ ]
18:         for all (start, end) ∈ list_1 do
19:             if start = node then
20:                 start_list ← start_list + end
21:             else if end = node then
22:                 end_list ← end_list + start
23:             end if
24:         end for
25:         list_2 ← [ ]                              ▷ Will contain merged edges
26:         for all node1 ∈ end_list do
27:             for all node2 ∈ start_list do
28:                 ▷ Creating new virtual link to be added to merged links list
29:                 list_2 ← list_2 + (node1, node2)
30:             end for
31:         end for
32:         ▷ Dispatching merged links on correct list, depending on their properties
33:         for all (start, end) ∈ list_2 do
34:             if start ∈ monitored_list and end ∈ monitored_list then
35:                 ▷ Link contains only monitored node
36:                 list_3 ← list_3 + (start, end)
37:             else list_1 ← list_1 + (start, end)
38:             end if
39:         end for
40:     end for
41:     ▷ Building resulting graph from list of edges
42:     return Graph.from_edges(list_3)
43: end procedure
```

The dataset has been processed as exposed in following chapters to match the requirements for this work.

3.3.1 Border routers generation

Firstly, available networks have been analysed, by looking at their geographic organization and by comparing them with the description that have been given by providers on their websites. From this analysis, it has been highlighted that topologies represent just the core network, without the surrounding access part. This lack does not allow to apply network monitoring as defined in the section 3.1.2, so a technique to synthetically expand network topologies has been implemented during this work.

First of all, an access device as been added to each core device. In addition, given N the number of core devices, a set of additional M access devices has been attached to a corresponding number of core devices. This set is randomly sampled, with replacement from the set of core devices, according to a discrete probability distribution with a probability mass function defined as:

$$P(k) = \frac{\frac{1}{l_k}}{\sum_{i=1}^{N} \frac{1}{l_i}} \tag{3.1}$$

where l_i is the number of links to core devices of the i-th device. By following this approach, devices that have many core links are most likely belonging to the

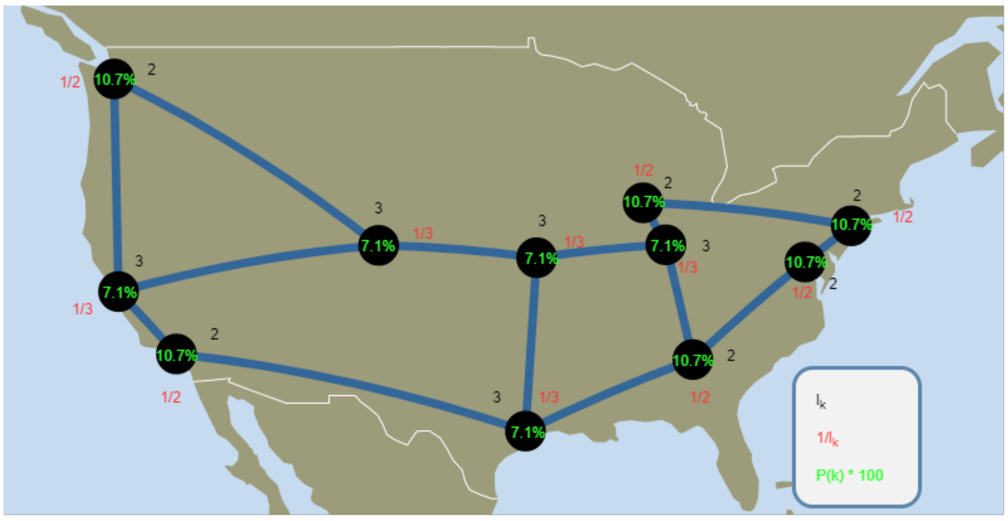


Image derived from http://www.topology-zoo.org/explore.html

Figure 3.8: Example of discrete probability distribution that have been used applied to `Abilene` topology.

central part of the network, thus having less access devices attached to them. Vice versa, devices closer to the periphery of the network will have more access devices. An example of this probability distribution that can be easily representable is shown in figure 3.8.
Furthermore this set of M access devices, will be, with probability p, attached to a neighbour of the core devices they have been linked to. This behaviour has been introduced to emulate dual homing of access devices.
This generation algorithm has been designed to generate network as close as possible to the TIM - Telecom Italia network [21], even though on a smaller scale, to allow faster processing. In particular, during this work equal values of M and N has been considered and p has been used with value 0.7.

3.3.2 Monitored network algorithms comparison

After having defined the complete structure of the network topologies that have been produced, performance of the two previously exposed algorithm to extract monitored graph has been analysed, varying the percentage of network interfaces monitored. The choice of the algorithm to use will be crucial, since it will be run several times during the optimization phase.
Tests have been performed on networks generated starting from different topologies. Monitored interfaces are randomly sampled, incrementing monitored percentage of 5% each step (starting from a network monitored only in border interfaces), and 5 tests have been executed for each percentage level, since random sampling may produce results with relative high variability.

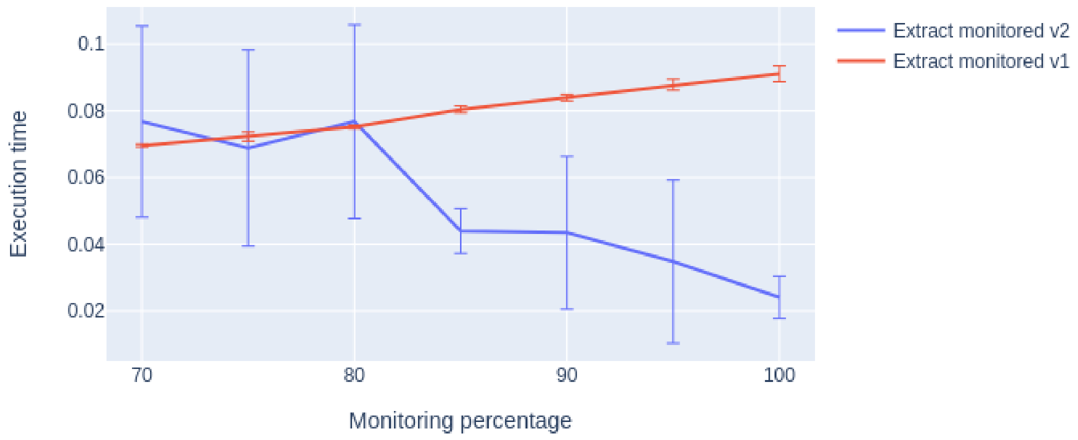


Figure 3.9: Algorithm comparison on `AbileneExtended` topology (execution times are expressed in s)

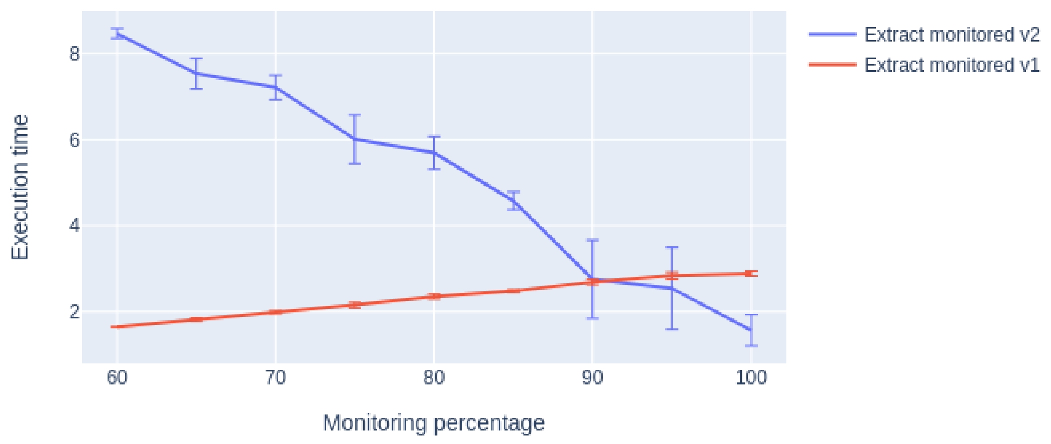


Figure 3.10: Algorithm comparison on `GeantExtended` topology (execution times are expressed in s)

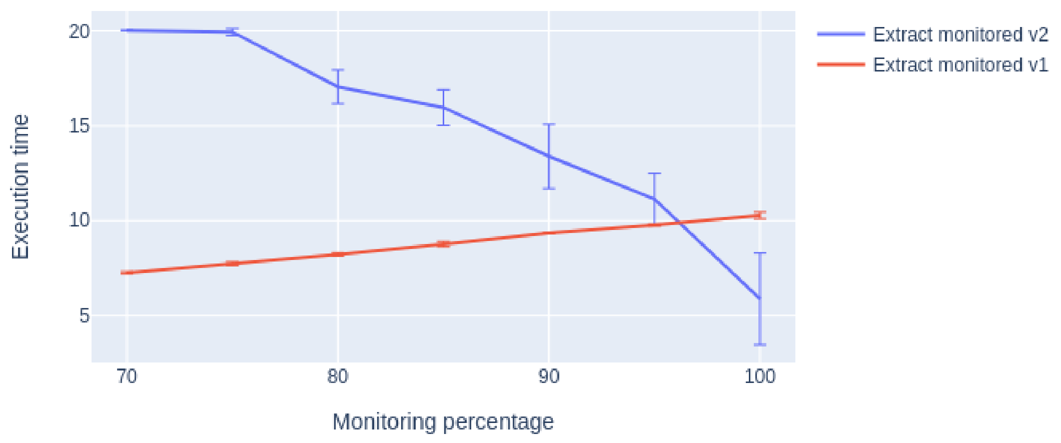


Figure 3.11: Algorithm comparison on `GarrExtended` topology (execution times are expressed in s)

Figures 3.9 to 3.11 show average execution times (with a timeout in the execution of 20 s) for topologies that will be of interest in the following chapter. Trends are always the same (in figure 3.9 the effect is less evident due to the very limited time needed). algorithm 3 shows better performance only when the network is monitored with high percentage of nodes, while the other one (algorithm 2) performs well at lower percentages, but in this case the difference is more relevant.
For every value of monitoring percentage, also the standard deviation of the measurements has been reported as error bar:

- algorithm 3 presents an high variability: this is due to the fact that non monitored nodes may have variable number of in and out links, thus varying the number of merging that is necessary
- algorithm 2 instead, has a very low variability. This has been attributed to the fact that path searches inside the graph have almost the same complexity, thus providing a lower standard deviation

In conclusion, it can be clearly seen that a well defined threshold can be found for every topology that is considered: depending on the monitoring percentage, it would be preferable to use one algorithm or the other.

Chapter 4

Clustering

Clustering is a key point for performance monitoring. The value of data that the system will generate depends on how uniform clusters are.
Considering a network that has just two clusters, cluster 1, which is huge, and cluster 2, which has just few nodes. Surely, in the same network conditions, the loss that will be detected in cluster 1 will be considerably higher with respect to the one detected on cluster 2. Clearly, it is not possible to define a single threshold to signal a problem that has eventually occurred in the network. In other words, a loss that may result normal in cluster 1 can highlight an anomaly in cluster 2. Loss has been taken as an example, but same arguments can be discussed for all the performance metrics.
For all these reasons, clustering problem has been carefully analysed during this work. This chapter is completely devoted to this theme. Firstly, clustering algorithm will be described, then a possible viable optimization will be presented and relative results will be exposed.

4.1 Clustering algorithm

After having defined the monitored network by assigning measure points in the network, it is necessary to divide it into as many clusters as possible, so that detailed metrics can be computed for them.
The algorithm that splits the network in clusters have been developed in [20]. It starts from analysing the edges of the monitored graph and it is divided into two phases:

- group together all edges that have a common starting node
- merge groups that share at least one final node into a single cluster

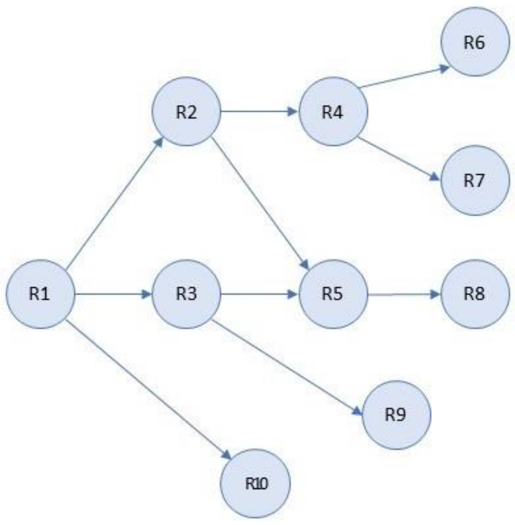


Figure 4.1: Example of monitored graph [19]

Considering the monitored network represented in figure 4.1, the groups that have to be created as first step to split the graph into clusters are:

1. (R1, R2) (R1, R3) (R1,R10)

2. (R2, R4) (R2, R5)

3. (R3, R5) (R3, R9)

4. (R4, R6) (R4, R7)

5. (R5, R8)

Groups should be merged as follows:

- Group 1 has no ending nodes in common with other groups and is left as it is: (R1, R2) (R1, R3) (R1,R10)

- Group 2 and 3 are merged because they have R5 in common: (R2, R4) (R2, R5) (R3, R5) (R3, R9)

- Group 4 is not merged: (R4, R6) (R4, R7)

- Group 5 too is not modified: (R5, R8)

4.2 Clustering optimization

Main requirements of optimization of network clustering is to produce balanced clusters, such that the number of nodes and edges that each cluster contains is as similar as possible. Possibly, it may also be desirable to use the smallest number of measure points in the network, so that computational resources' needs are minimized.
While in all previous works, measure points set was randomly built by selecting N nodes of extended graph, where N was selected by users themselves, in this work a more precise approach has been researched.
The idea that has been deepened is the one of training a neural network that is able to select the best nodes where to place the measure points via reinforcement learning algorithm.

4.2.1 Problem desiderata

The user's input of this approach are the maximum number of nodes and the maximum diameter that a cluster should have. Nodes are considered as the total number of nodes in the cluster, among the monitored ones and non monitored ones. The diameter has been computed as the maximum number of links in the extended graph that has to be traversed from the input of the cluster and its output.
First of all, the domain of these values have been analysed: the lowest value for both can be obtained by computing those metrics for the current network where all nodes are monitored, while the upper bound can be found by selecting only the border interfaces as measure points.
Since no analytical relation exists among these quantities, in order to choose these values coherently with each other, a difficulty coefficient has been introduced. Its value is between 0 and 1 and represents the distance between the minimum value and the one chosen as shown in figure 4.2.

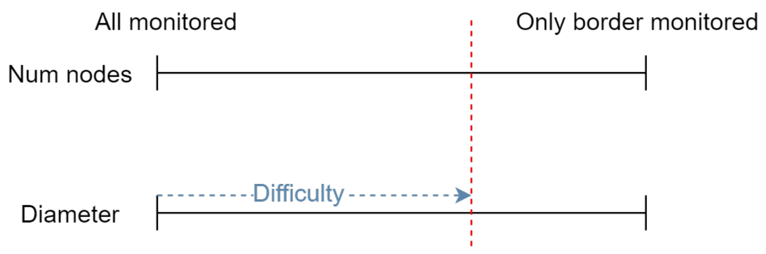


Figure 4.2: Difficulty coefficient definition

As an example, by considering a difficulty coefficient d, the number of nodes that will be selected is:

$$numNodes_{min} + d \cdot (numNodes_{max} - numNodes_{min}) \tag{4.1}$$

where $numNodes_{min}$ is the number of nodes that is obtained with the network completely monitored and $numNodes_{max}$ is the one obtained considering only border nodes as monitoring point. The same equation can be applied to the diameter value.

4.2.2 Clustering approach

Reinforcement learning is a machine learning framework that is used in contexts where decisions have to be taken one at a time, with the aim of maximizing a reward function. This is the technique that has been chosen to allow the optimization of network clusterization.
In this section, the clustering problem will be formalized so that reinforcement learning algorithms can be applied to maximise some quantities that will be exposed later on. The approach that has been pursued has been defined as follows:

1. consider the network as fully monitored
2. select a node that will be removed from the set of measure points
3. compute the new clusterization of the network
4. evaluate maximum number of nodes and diameter that have been obtained
5. if computed values are still below the desired ones, repeat from point 2
6. otherwise, reinsert last node in the set of measure points and terminate

This approach has been preferred to the opposite one, that would suggest to place one measure point at a time in the network only for computational reason. Algorithm 3 can be used incrementally: the value of `list_3` of previous iteration can be stored and passed to clustering algorithm of following iteration. Thus, it is possible, starting from values present in `list_3` to merge only the links in which the removed measure point is involved.

4.2.3 Clustering evaluation

It is necessary to define also a metric that will be the target of the optimization problem. As said before, the aim of this analysis is to obtain clusters that share the same characteristics. During this work, it has been formalized as:

$$uniformity = \left(\frac{\sigma_i[numNodes_i]}{\mathbb{E}_i[numNodes_i]} + \frac{\sigma_i[diameter_i]}{\mathbb{E}_i[diameter_i]} \right) \tag{4.2}$$

where $numNodes_i$ and $diameter_i$ are respectively the number of nodes and the diameter of cluster i. Standard deviation is by itself a good indicator of variability in data, but is not normalized on the scale of data. Standard deviation values have been consequently scaled by the average of the data, making these two values comparable and avoiding that one of the two parts involved in the sum would be more relevant than the other.

4.3 Clustering via reinforcement learning

In this section, all the theoretic foundations that are behind the reinforcement learning algorithms that allow the optimization of the clustering problem and the results that have been obtained will be exposed.

4.3.1 Reinforcement learning

Reinforcement learning allows to solve sequential decision-making problems. In this context it is defined *agent* the component that has to learn how to behave in the *environment*, which is a formalization of the problem itself.

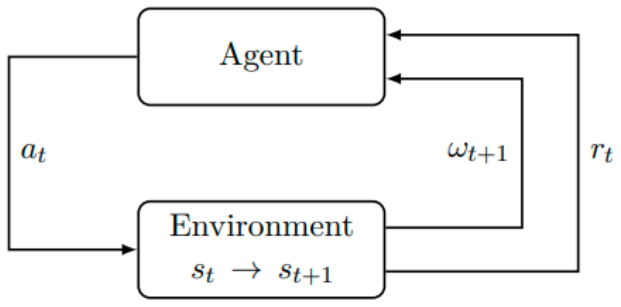


Figure 4.3: Explanation of components involved in reinforcement learning problems [27]

As shown in figure 4.3, the agent iteratively interacts with an environment. The aim of the agent at each step t is to produce an action a_t that has to be executed in the environment. This produces a state transition in the environment, that goes from state s_t to state s_{t+1}. The environment also produces a reward r_t for the current action to evaluate its quality and an observation of the new state ω_{t+1}[1].

[1]In case of environment that is fully observable, which is the simplest condition and the one that has been considered, $\omega_{t+1} = s_{t+1}$

By receiving as input the values, the agent is now able to proceed a step forward, by selecting a new action a_{t+1} and so on.

Markov Decision Process The simplest problem's conditions are the one of *Markov Decision Process* (MDP), where environment evolves only basing on previous action and state, without looking to the whole history of actions and states. Thus the agent can just receive the current state to base its decision without the need of receiving the whole history. As stated in [27] in order to completely define a MDP, it is necessary to provide:

- a set $\mathcal{S}$ of all possible states that the environment can assume
- a set $\mathcal{A}$ of all possible actions that can be performed on the system
- a transition function $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$ that given the current state s_t, an action a_t and the possible future state s_{t+1} gives the probability that the transition $s_t \implies s_{t+1}$ will happen, having applied the action a_t
- a reward function $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathcal{R}$ that takes as input the same parameter of the transition function and produces a continuos value in the finite range of $\mathcal{R}$, that depends on the problem, as reward of the current action
- a discount factor $\gamma \in [0,1)$, to weight the give reward basing on the time that has passed from the beginning. Total reward, finally will be given as

  $$\sum_{i=1}^{\infty} \gamma^t r_t$$

  the range [0,1) that γ can assume is the only possible one that will guarantee that the reward given is finite.

MDP definitions for clustering problem It is now possible to expose how a Markov Decision Problem has been fit to the clustering problem during this work:

- $\mathcal{S}$ corresponds to every possible configuration of monitoring points in the network. Practically every node in the reduced extended graph will get value 1 to indicate an active monitoring point, otherwise will be 0.
- $\mathcal{A}$ is the set of nodes of the extended graph. The one among them that is selected will be considered as measure point to be removed from the state. It has to be noticed that at each step the action has to be selected only among nodes that are currently measure point and do not belong to the set of border interfaces

- T is a deterministic function and has no stochastic behaviour. Starting from a state s_t and having selected a node a_t to be removed from measure points set, there will be only a state $\hat{s}_{t+1}$ for which the transition function will be equal to 1. All other future states will have value 0.
- R only gives non-zero rewards whenever the clustering process with a uniformity coefficient u is terminated and value will be equal to

  $$\frac{1}{1+u} \tag{4.3}$$

  this formulation will allow to give greater rewards whenever the uniformity is lower and vice versa. In addition, reward will be limited to the range (0,1]
- discount factor γ is equal to 1. Despite it is in contrast with MDP formulation, unitary value always provides a total reward that is bounded and simply corresponds to the final reward of equation (4.3). During this work, some experiments run with $\gamma > 1$ (e.g. 1.001) have been tried to force the optimization process to remove as many measure points as possible. Even these values of γ allow the reward to be non-infinite: number of steps of the decision making process is at most the number of nodes that are not border nodes. This however did not give satisfying results, uniformity was higher than simply using $\gamma = 1$, and higher reward is obtained with non uniform clusterization, but with smaller number of measure points used.

4.3.2 Deep Q-learning

Q-learning is an approach that can be followed to solve sequential decision-making problems by exploiting a Markovian Decision Problem. It finds its foundation in *Bellman* equation, that has been defined in [42]:

$$Q^*(s,a) = \mathbb{E}_{s'\sim\varepsilon}\Big[r + \gamma \max_{a'} Q^*(s',a')|s,a\Big] \tag{4.4}$$

where s' is the distribution of possible states that the transition function would produce.
This equation estimates the potential total reward that can be obtained from current state s by performing action a as the sum of reward given at current step, that is added to the maximum potential reward that can be obtain by choosing the optimal action that could be taken at following step, rescaled for the discount factor. As a result, by knowing Q^* function is it possible at every step to choose as action the one with highest Q-value.
Equation (4.4) can be simplified in the case under analysis, since the transition

function is not stochastic, thus the following state will be deterministically obtained (s will always evolve to s' when action a has been applied):

$$Q^*(s,a) = r + \gamma \max_{a'} Q^*(s',a') \tag{4.5}$$

The idea of Q-learning is to estimate this value function Q^* iteratively. By minimising the error

$$\Big(Q_{i+1}(s,a) - \big(r + \gamma \max_{a'} Q_i(s',a')\big)\Big)^2 \tag{4.6}$$

it has been demonstrated in [42] that $Q_i \to Q^*, i \to \infty$. The concept is to keep two estimations of the value function Q_i and Q_{i+1} and periodically update $Q_i = Q_{i+1}$. First choice of Q can be randomly carried out.
Deep learning has been applied to Q-learning, thus allowing to approximate value function with complex non linear functions. [35] is the most common example where a neural network has been used to solve a problem through Q-learning. It also introduces the concept of replay memory, where to store tuples (s_t, a_t, r_t, s_{t+1}) that will randomly sampled in batches to perform neural network optimization through stochastic gradient descent.
Algorithm 4 represents in pseudo code the approach that has to be followed to train an agent in a generic environment with deep Q-learning.

4.3.3 Graph processing through Neural Networks

Among all the available examples of graph processing exploiting deep reinforcement learning that can be found in literature[2], none of them perfectly matches the need of clustering problem. It is necessary to introduce some concepts and formalizations to allow to fully understand the problem and the resolution that has been proposed during this work.

Graph formalization In order to correctly and completely represent a graph, some elements are necessary:

- a set $\mathcal{V}$ that contains all the nodes v, that possibly have features $\mathbf{h}_v \in \mathbb{R}^D$ that convey information about them
- a set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ that includes edges $e = (v_i, v_j)$ to indicate that node v_i is linked to node v_j (depending on graph nature this may be directed or not). Also edges may have additional features $\mathbf{h}_e \in \mathbb{R}^C$ to describe the edge, providing for example a weight

[2]https://github.com/SunQingYun1996/Graph-Reinforcement-Learning-Papers is a quite updated overview of possible techniques that have been recently implemented

Algorithm 4 Algorithm to perform deep Q-learning optimization. Adapted from [35]

```
1:  $\mathcal{D} \leftarrow$ memory with capacity $N$
2:  $policyNetwork \leftarrow$ randomWeightFunction
3:  $targetNetwork \leftarrow policyNetwork$
4:  for $episode = 1, M$ do
5:      ▷ Initializing state to the initial value for the problem
6:      $s_t \leftarrow s_0$
7:      while not isEpisodeCompleted do
8:          ▷ Computing next action
9:          if random() $< \epsilon(episode)$ then
10:             ▷ Taking random action with probability that depends on episode number
11:             $a_t \leftarrow$ randomAction()
12:         else
13:             $a_t \leftarrow \underset{a}{\text{argmax}}\ policyNetwork(s_t, a)$
14:         end if
15:         $s_{t+1}, r_t \leftarrow$ executeActionOnEnvironment$(s_t, a_t)$
16:         ▷ Storing current action in memory
17:         $\mathcal{D}$.add$(s_t, a_t, r_t, s_{t+1})$
18:         ▷ Getting batch of $S$ samples from memory
19:         $samples \leftarrow \mathcal{D}$.sample$(S)$
20:         $y \leftarrow [\ ]$
21:         ▷ Compute Q values with previous estimation
22:         for $j \in samples$ do
23:             if $s_j$ is terminal then
24:                 ▷ Whenever the action is terminal, the reward is simply the one given at current step
25:                 $y_j \leftarrow r_j$
26:             else
27:                 $y_j \leftarrow r_j + \gamma \cdot \max_{a'} targetNetwork(s_{j+1}, a')$
28:             end if
29:         end for
30:         ▷ Updating policy network to minimise error
31:         gradientDescentOn$(y_i - \max(policyNetwork(samples.s_j, samples.a_j)))$
32:     end while
33:     if $episode \% U = 0$ then
34:         ▷ Updating value function every $U$ episodes
35:         $targetNetwork \leftarrow policyNetwork$
36:     end if
37: end for
```

Clustering framework A possible network architecture that may be built is shown in figure 4.4: it will receive the network graph, which embeds as node feature a single value that can be 0 or 1 to represent respectively the absence or presence of a measure point inside it. Edges, instead does not convey additional information apart from the link itself.

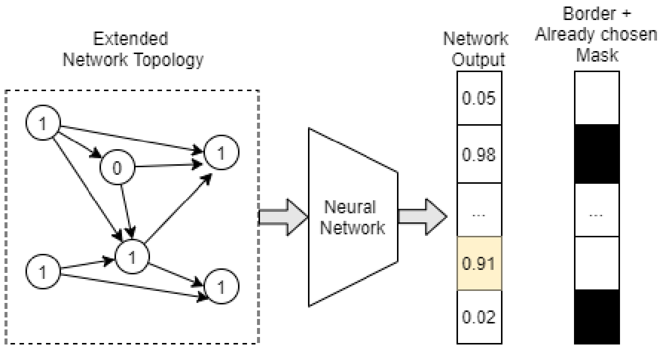


Figure 4.4: Possible architecture that can be used for processing optimization task

The output of the neural network is an array that represent the estimate Q function for the input graph and for every possible actions. As reported in section 4.2.2, the action for the element i in output array consists on removing the measure point from the i-th node. The output should be masked: with a mask that varies over time, it includes the elements that are border routers (which is a fixed part of the mask and by definition are measure points) and nodes that have no more a measure point. Values that have not been masked have an estimation of Q value that, according to algorithm 4, represent the reward that can be obtained at the end of the problem by choosing current action at this step. The best solution can be found by selecting at every step among valid elements, the one with highest Q value.

Message passing architecture Graph processing with neural networks is quite a recent topic. The approach that in the latest years has shown its performance and prevails in several works related to this kind of problem is message passing and was firstly applied in deep learning in 2015 in [33] and some years later in [29]. The work that claims to provide best performance with very deep networks and that has been chosen as base layer to build the neural network for this optimization is [22]. The strengths that this implementation provides are the resistance to vanishing gradient effect and to over-fitting when building very deep architectures, that potentially may be built.

Message passing is based on transferring information towards a node from its neighbours, in order to provide a compact, complete and structured graph representation in node features to convey some value or to be further processed by a traditional neural network architecture.
Firstly, in the implementation by [22] each neighbour u of node v *sends* a message $\mathbf{m}_{vu}$ that depends on features of both nodes and of the linking edge:

$$\mathbf{m}_{vu} = \rho(\mathbf{h}_v, \mathbf{h}_v, \mathbf{h}_{e_{uv}}) \tag{4.7}$$

The set of all the messages that have been *received* from node v has to be aggregated as a single message $\mathbf{m}_v$:

$$\mathbf{m}_v = \zeta(\{\mathbf{m}_{vu_i}\}_i) \tag{4.8}$$

A new set of features of the node will be generated basing on the aggregated message and the original features:

$$\mathbf{h}'_v = \phi(\mathbf{m}_v, \mathbf{h}_v) \tag{4.9}$$

Functions ρ, ζ and ϕ are learnable and non differentiable. This allows to build a neural network layer that takes as input a graph structure, with nodes' features $\{\mathbf{h}_{v_i}\}_i$ and generate a new aggregate set of features of nodes $\{\mathbf{h}'_{v_i}\}_i$ that can be a compact graph representation or can be further processed. The power of this structure is that by learning different sets of function, it is possible to fit different kinds of tasks.

4.3.4 Optimization framework implementation

After having exposed all the parts that compose the proposed framework, it is possible to provide an overview of the implementation of the optimization technique proposed in this work. Several lines of code were already available from previous works and have been collected in a repository[3].
Provided the availability of functions already implemented, machine learning frameworks and graph processing libraries, Python has been chosen as programming language to implement this technique.
The neural network has been built with `torch_geometric` library[4], which is fully integrated with Pytorch framework[5]. `torch_gemotric` library contains the implementation of several layers that are able to process graphs, including the one by [22].

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

[4]https://pytorch-geometric.readthedocs.io/en/latest/index.html

[5]https://pytorch.org/

The neural network model has been built with 6 graph convolutional layers previously exposed, interleaved with a *tanh* activation function to provide additional non-linearity to the model. For the last layer, instead, a sigmoid activation function has been used, so that the output of the network will be in range (0,1), that is the correct range of reward as defined in equation (4.3).
This architecture has not to be intended as the best possible one. It has been chosen since it performs quite well, but architectures that are more complex, deeper or that include also fully connected layers may be further analysed.
Tests were performed taking a single network topology in consideration. A more general approach has been considered, by using an environment in which the network has to learn a group of topologies that were sampled one at a time: this requires to define a reward that is not topology-specific and the uniformity coefficient should be in some way normalized. Some attempts have been done in this direction, but results are not good. Additionally, providing a network trained by reinforcement learning in this way, it is not a trivial solution: in literature, no work that exploits a direct training on different environments together has been found. Despite several works can be found in the field of multi-task reinforcement learning, this has not been considered as the desired solution: the task to learn is always the same, but applied on different environments.

4.4 Optimization results

Optimization process has been carried out with different network topologies. Uniformity of the obtained clusterization has been computed every time the target network is updated with the policy weights (that is to say every U episodes according to algorithm 4). The trend followed by this metric during the training phase has been observed and has been compared to the value that can be obtained in average by applying a random policy (choosing nodes with a random function). Obtaining a value that is stably under this threshold is considered as good result for this problem.
First trial of the technique has been carried out on a network topology derived from `Abilene` one from [36], which is quite a small network for 1000 episodes (it corresponds to value of M in algorithm 4). A difficulty value of 0.7 has been chosen in order to maximise the number of measure points that should be removed from the network. Obtained results have been reported in figure 4.5: the trend is quite good and converges rapidly. The execution time required is about 1.5 h relying only on a CPU[6] and the optimal clusterization has been obtained twice (episodes 150 and 435). This shows the validity of the solution, despite the fact that during

[6]`Server2` exposed in chapter 7 has been used

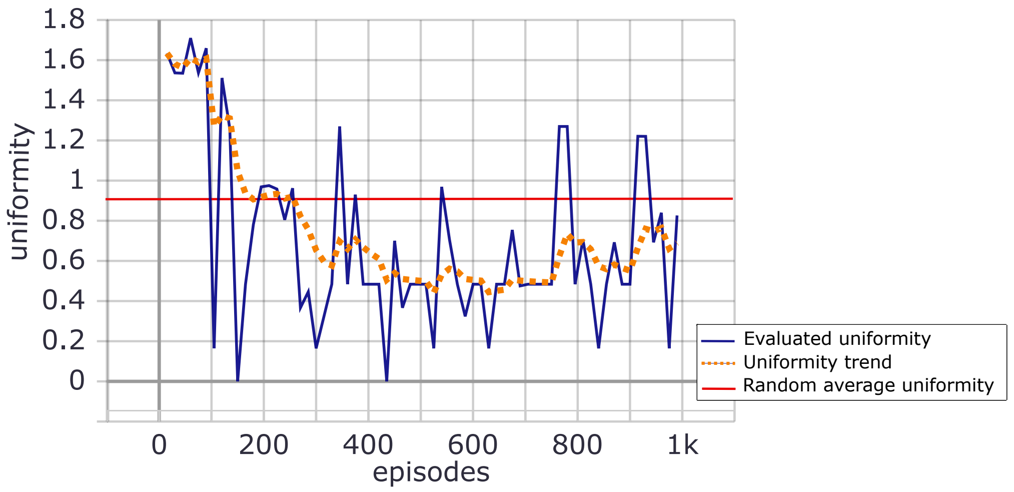


Figure 4.5: Results obtained on `AbileneExtended` network with difficulty coefficient of 0.7

last hundreds of episodes the trend has started growing.
A second trial has been carried out on a topology derived from `Geant2012`, the model of the network of an European project aiming to link research and instruction networks across the continent[7]. This is the network that has been used during previous works and will be the reference one for all of this work due to the dimensions and topological structure that is similar to the TIM Italian core network. The difficulty has been chosen by trying different values (from 0.3 to 0.7): 0.5 seems to be a good compromise between the number of measures points that are used in the clusterization and the number of generated clusters (that are considered acceptable in a range $[10, 20]$). The simulation took about 1 day, due to greater complexity of the generated network. Results are reported in figure 4.6: the trend is slightly more fluctuating and has a lower margin towards the threshold. Good results however can be found: although at step 930 there is a better solution, this is not been considered as the best one, since number of clusters was very small (these value has not been taken into account in the optimization metric). The solution that has been taken into account for future experiments is the one of steps 780, which provides 14 clusters with a uniformity of 1.66. Surely this has not the best possible solution that can be obtained, but it can be a good starting point for possible future works and improvements.

[7]https://www.geant.org/

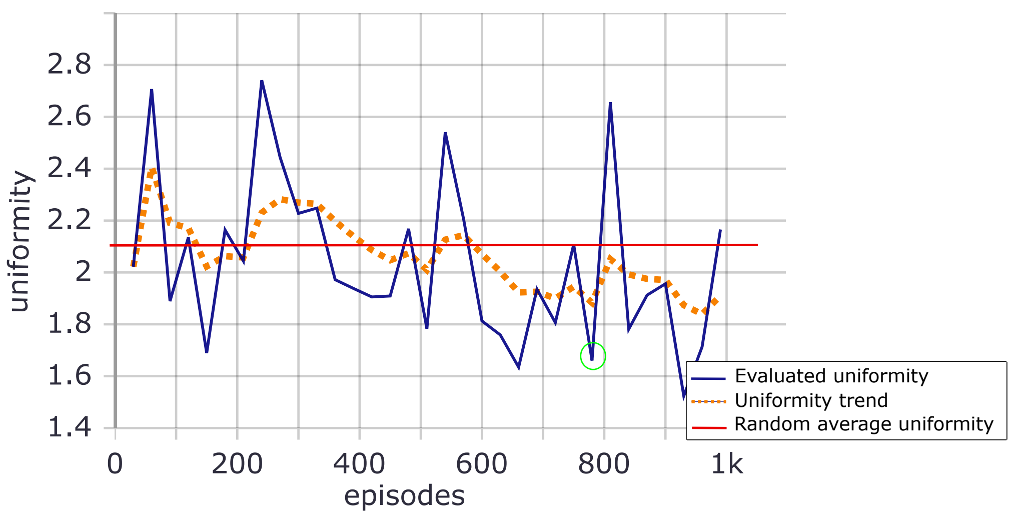


Figure 4.6: Results obtained on `GeantExtended` network with difficulty coefficient of 0.5. Considered solution is highlighted in green

4.4.1 Transfer learning application

Considering the execution time that may be huge, an additional trial has been carried out to evaluate the possibility to apply previously acquired knowledge on another topology to speed up following run of the algorithm.
Some works that apply transfer learning in reinforcement learning exist and a good global review, although it has been written about ten years ago, has been given by [34]. In several works, different effects on performance' improving have been observed and have been shown in figure 4.7:

- *learning speed*: performance becomes stable in a reduced number of epochs
- *asymptotic improvement*: performance that can be reached when the agent converges are better with respect to a simple training
- *jumpstart improvement*: initial performance obtained without any training on transferred environment is better, but the asymptotic result is the same

However, it is not possible to define *a priori* whether some of these effects will be observed or even whether the outcome of this technique will be positive or negative. In order to evaluate the effect of transfer learning the Garr network topology[8] has been extended and used: it is the network that links together Italian research centers and universities and has a geographical distribution that quite overlaps the

[8]https://www.garr.it/it/infrastrutture/rete-nazionale/infrastruttura-di-rete-nazionale

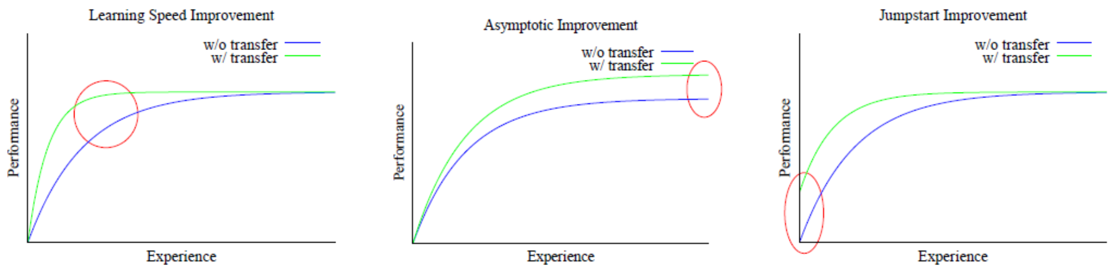


Figure 4.7: Possible improvements that transfer learning may bring in a reinforcement learning context [34]

one of TIM [21].
Two separate runs have been carried out on this topology, which is more computationally expensive with respect to the previous case (simulations took about 2.5 days): one with an agent that has been been initialized with random weights and another one using the weights in step 780 of run performed on `Geant` topology. Uniformity results' trend has been reported in figure 4.8: it seems that no positive effects can be obtained with this approach. A slight *jumpstart improvement* can be noted during the first 300 episodes: first episodes however depend on random choices that have been taken (at the beginning decisions are kept with more stochasticity to explore every possible solution) and this results in high initial performance variability. In conclusion, this has to be considered as a trial towards transfer learning application and future deeper studies in this direction may reveal a real benefit.

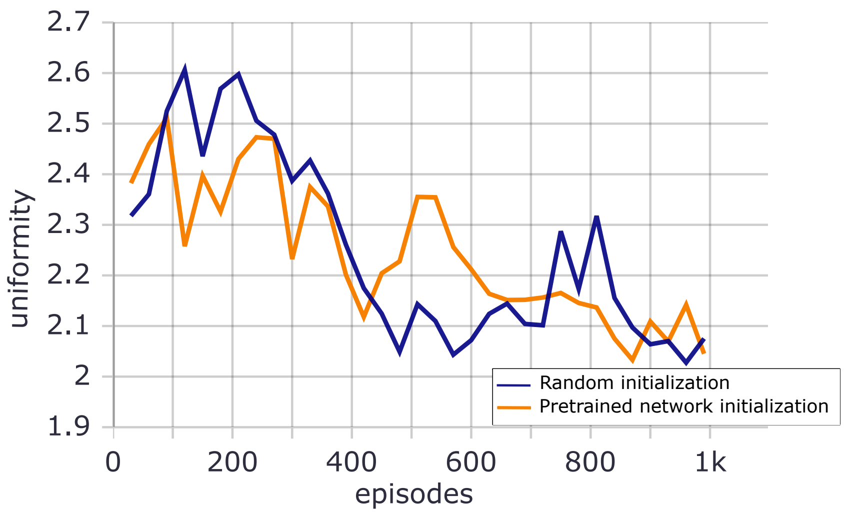


Figure 4.8: Trend comparison between pretrained agent and randomly initialized one

4.4.2 Final considerations

In conclusion to this chapter, it is possible to state that the optimization technique that has been proposed shows its validity. Since there is no ground truth to compare, the only consideration that can be done is that results obtained through the solution given are better than the *random* ones after an initial *learning* phase, the duration of which hugely depends on the complexity of the network.
Some weak points, however, have been highlighted:

- the behaviour is not completely stable under the *random* threshold and even from a certain point, the trend starts increasing, showing that the agent is no longer learning the right optimization of the problem
- the uniformity measure that has been provided does not take into consideration the number of generated clusters. Furthermore the oscillating behaviour of the uniformity, requires a manual search of a solution that can be considered the best one to apply
- no generalization is provided to work with every possible topology and it is not possible to exploit knowledge acquired during the training on a network topology to solve the same problem on another one

Chapter 5

Overall system architecture

In this chapter, an overview on how the system developed during this work should be deployed in a real environment will be given. All the technicalities that stand behind each component will be later on exposed in chapter 6.
The base structure of the system is the one provided by previous thesis work [19] and formalized in an `IETF` draft [18], with the adjustment that have been provided in [16]. Some weak points have been found and analysed, providing in this work possible improvements.

5.1 Network probe

Probes are the key component for *Alternate Marking* that provides data for each monitored network interface. Specifically, generated data will be sent to Network Management System (NMS), that will process them to extract network performance metrics. Those values that are generated may be divided into:

- **aggregate** that encloses statistics for each period. Among the values, a counter for marked packets and the corresponding mean timestamp is provided coherently with the metrics exposed in `RFC8889` [20]
- **sampled values** that report relevant information about packets. Probabilistically up to N entries will be sampled thanks to dynamic hashing for each period at every interface. These data will allow to compute more precise values for cluster metrics and also some of them will be stored to allow post-processing to obtain measures that are relative to a single flow
- **keep-alive messages** have been introduced during this work: previous data are generated only in case of traffic flowing through that interface. Whenever no data are coming from an interface, it is not possible to distinguish whether

no traffic has been captured or the probe is not running properly any more on that interface. The easiest solution is to introduce a message for every period that signals that current data for a specific interface and period were correctly generated

In order to perform correct network measurement, probes should be configured on the basis of the needed clusterization, providing them with:

- reference hash on which to perform dynamic hashing filtering
- (probabilistic) maximum number of packets N to collect for each period
- period duration after which to switch label value
- values for marking labels, to select packets that have matching label and assign them to correct block
- possibly, also a filter to select only a subset of packets to monitor. This value should be meaningful only whenever is not possible to distinguish a marking label within the traffic that is not marked

Those values must be equals in the entire network to provide coherent data. Thus aggregates will refer to the same packets and values that are sampled will target the same packets.
In addition probe may be configured by providing a list of interfaces that should be monitored: as stated before, there is the need of capturing traffic only on some of them.

5.2 Message queue

During previous thesis work [16] the usage of a publish-subscribe mechanism to send data to the NMS has been introduced. This has became necessary to overcome the temporary storage into routers' filesystem used in [19], which represents a quite unusual situation on this kind of devices. This mechanism have also introduced data decoupling, thus allowing the probe to generate data independently from when and how the data will be consumed.
Data that are generated by probes are saved into different queues, depending on their content. In an ideal case, when no transmission delays occur, data of different periods will be correctly enqueued. In a real network instead, where latency may be relevant, transmission time cannot be considered negligible. Due to these delays and also to generation times on different routers that may vary, data of consecutive periods may be interleaved. In order to overcome this issue, messages have been further dispatched to different queues.

For each type of data, N queues may be defined, assigning them an index from 0 to $N - 1$. Data belonging to a period P will be enqueued on the queue that have index P % N as can be shown in figure 5.1.

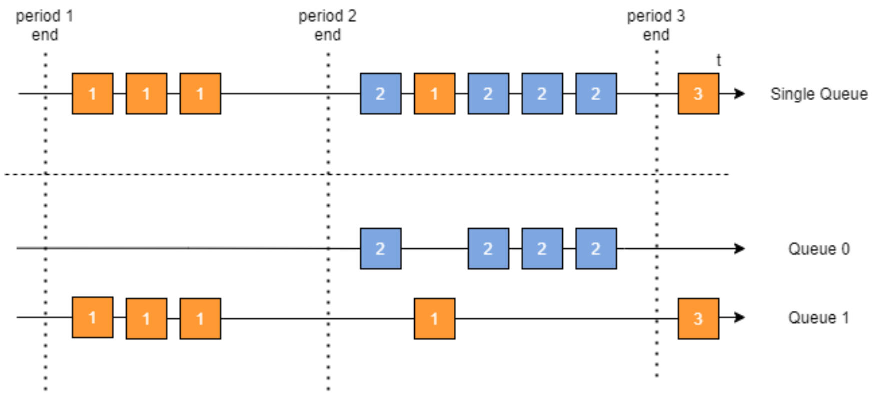


Figure 5.1: Example of data assignment to queue in case of 1 and 2 queues

This organization of queues will allow the preprocessing component to avoid reading multiple times the same data: whenever the processing for period P starts, it will start reading from the end of this period, possibly without having to read data that comes from other periods. Value of N should be considered on the basis of period duration and on the delays to reach the message queue that are considered acceptable in the network.

5.3 Cluster manager

A new component has been implemented during this work. It has been designed to satisfy different requirements:

- manage currently active probes: by analysing keep-alive messages, it will understand if a probe is faulty and remove it from the measure points set
- provide the current clusterization of the network under analysis
- management of measure points: they could be manually reconfigured while the system is running. This component will take care of this kind of situations
- manage the topology: network can evolve in time and the system should be able to deal with it. By providing to the cluster manager a change in the topology, it should respond, updating the clusterization and possibly also selecting the best placing of measure points to minimize the uniformity coefficient

This part has been designed to be a separate component, to provide low cohesion between system parts, facilitate each implementation (providing more freedom in technological choice for each of them) and provide modularity in the whole architecture.

5.4 Real time preprocessing

The principle on which this software is based is the *BigData approach* for *Alternate Marking* that has been standardized in the IETF draft [18]. This should work as NMS and has the aim of collecting results from all the probes that captured traffic and, due to the huge amount of data that will be generated, process them with *BigData* techniques, providing a scalability of the system. Processing of data should provide useful information that highlights the status of the network and the performance of flows of traffic, both minimizing the storage size and maximizing their value as shown in figure 5.2.

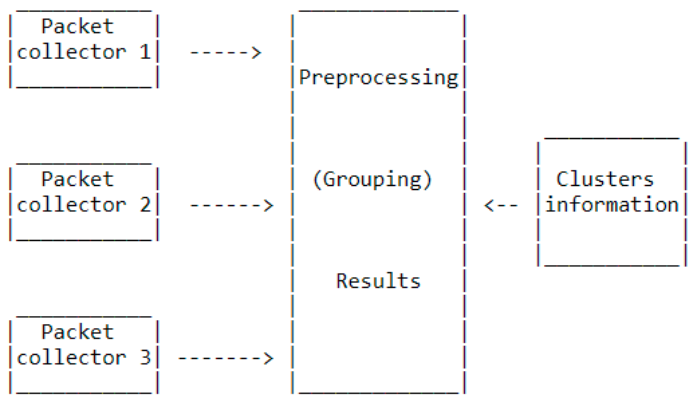


Figure 5.2: Organization of preprocessing component [18]

This component has been designed during the work of [19] to be run periodically, processing data that had been previously stored in the filesystem. Its aim was to aggregate packet captures by following their paths in the network and no metrics were actually computed.
During this work, this component has been improved so that:

- data are processed as soon as they arrive, without the need to store temporary files: this requires that data processing is completed in single period or that some pipelining mechanism is adopted. In this way, the system will always be ready to new data of each period
- results that involve per cluster metrics are computed just once, since they

are relevant to detect the status of the network in *near real-time*, and this structuring allows to run some kind of predictive maintenance algorithm that uses those results, so that possibly failures can be detected immediately and corrective actions can be taken as soon as possible, minimising network malfunctioning

5.4.1 Data synchronization

As for every distributed system, it is necessary to provide some synchronization among different components. In order to build an online component that aggregates data from different probes, in this work a synchronization mechanism has been implemented to correctly wait for the generation of all the data for every period.
The preprocessing system will implement the logic to detect whenever all data have been correctly collected. Assuming that keep-alive messages are sent by each probe after every other messages of the same period, it is possible to store which interfaces have correctly processed data for that period, by looking at keep-alive messages in the relative queues. Whenever all confirmations have been received for a period, its data is stable and performance measures can be extracted.
The processing can be assigned to a specific thread, so that the component is ready to process other data and does not block and keeps the pace even though the processing requires more than a single period duration.
Expecting data from all interfaces that are considered active, however, may produce unbounded waiting: probes or devices executing them can have unrecoverable failures and data from a certain period may be missing. In order to deal with those cases, it has been introduced a timeout for each period that will be renewed at each new data belonging to that block. Whenever the timeout expires, the processing will be started in any case: by knowing that data from a given node is missing, it is possible to signal for data belonging to cluster involved in data loss a warning flag, so that those data can be considered with a lower level of confidence.

5.4.2 Results organization

The design of results carried out during this work will be exposed in following paragraphs, basing on their content and on data used to produce them. As already highlighted, the main idea about the preprocessing phase is to produce detailed data about the status of each cluster in the network that has received traffic and also to filter out data, storing only those that can be useful in the postprocessing phase.

Aggregated measures

First part of results that are generated and defined *aggregated measures* can be extracted by simply looking at the aggregated data generated by network devices. This will mainly represent the implementation of what have been theorized in `RFC8889` [20]. It has been organized in two different data types:

- **AggregatedMeasures**: table 5.1 shows how global metrics for each cluster can be produced:
  - `date` and `period` univocally identify the period to which the datum belongs to
  - `clusterId` is the identifier of the cluster that the datum refers to
  - `loss` has been represented as percentage of lost packets (obtained using equation (2.6)) with respect to the number of packets that entered the cluster (`numberPacketsIn`) to provide results that may be easily interpreted without any further computation with respect to the absolute value
  - `meanDelay` will be computed by aggregating mean timestamp as exposed in equation (2.9)
  - `warning` identifies whether data can be wrong due to missing data from a probe in the cluster

  One value for each cluster that has encountered traffic will be produced at every period

AggregatedMeasures						
date	period	clusterId	loss	meanDelay	numberPacketsIn	warning

Table 5.1: Details of measures reports for each cluster

- **AggregatedProbability**: table 5.2 shows how loss probability can be exposed for each non internal node in every cluster:
  - `routerId` represents the interface identifier the metric refers to
  - `lossProbability` will be computed as exposed in section 2.3.3 to indicate the number of lost packets that have been lost upstream or downstream that interface
  - `direction` will represent the direction for which data are computed: depending on whether the node is an ingress one or it is an egress one, it will have value `D` (downstream) or `U` upstream

One or two values (depending on whether the interface is at the border on the network) for each interface that has encountered traffic will be produced at every period

AggregatedProbability						
date	period	clusterId	routerId	lossProbabillity	direction	warning

Table 5.2: Details of measures reports for each non internal device

Sampled measures

Another part of the results that are generated, has been defined as *sampled measures* and can be derived from information that are extracted from packets' sampling. These results have been inspired by [18, 19] and further extended to provide useful information at different level of granularity across the network:

- **link level**: this is the most detailed metric that can be obtained. However, links cannot be considered at physical level, but at a logical one. A *monitored* link corresponds to a directed edge in the monitored graph and practically may involve more physical links and also network devices traversing. Table 5.3 shows how those quantities can be organized:
  - `routerStart` and `routerEnd` represent the interfaces that define the monitored link
  - `meanDelay`, `jitterMean` and `jitterStd` represent the performance metrics. Also the value of the standard deviation of jitter has been introduced, since it can give a clearer idea on delay variation. These values can be further extended. By having a series of packets' data, it is possible to compute several statistics basing on the specific requirements, without the need to modify the architecture of the whole component
  - `packetCount` represents the number of packets on which those statistics have been computed. Due to the dynamic sampling mechanism, this does not correspond to the number of packets that traversed the link. Surely, the greater it is, the more those values can be considered as reliable

  One value for each monitored link that is not definable also as path (following the definition that will be given in the subsequent point) and has encountered traffic that has been sampled, will be produced at every period

- **path level**: it is an intermediate detail level. A path has been defined as the route followed by packets from an ingress node in a cluster to an egress one in

PerLinkMetrics/SampledClusterPath								
date	period	cluster	router Start	router End	mean Delay	jitter Mean	jitter Std	packet Count

Table 5.3: Details of metrics obtainable at monitored link or path level

the same cluster. The same structure defined in table 5.3 can be reused for this level of detail. Additionally, some link metrics may result to be also valid for per path ones: a link in case involves both ingress and egress nodes may also be a path. Since data would be replicated unchanged in this case, it has been decided, in order to save data storage, to consider this case only for path metrics, not including it in link level results.
One value for each path encountering traffic will be generated at every period

- **cluster level**: this is the highest level of detail that has been considered during this work. Despite it is possible to obtain metrics for all the network or by group of clusters, those have been seen as not so relevant. Traffic can vary rapidly and analysing it at level of the entire network can be misleading.

SampledClusterMetrics					
date	period	cluster	meanDelay	jitterMean	numberPackets

Table 5.4: Details of metrics obtainable at cluster level

Table 5.4 shows how this metrics can be organized. While `meanDelay` can be obtained as average of all delays of packets that flow through the cluster, the same cannot be done for `jitterMean`. It is not possible to give a global sorting mechanism that looks coherent and then compute jitter between packets. For this reason, jitter should be computed as weighted mean of per path jitter values. Contrarily, it has been considered that is not possible to compute a meaningful standard deviation for jitter values.
One value for each cluster with traffic will be produced at every period

Loss metrics have not been taken into account starting from sampled data, since in case of per link and per path metrics this problem is not trivial. In order to detect a packet that have not been detected at the end of the link or of the path, it is necessary to know which are the interfaces that should have captured it. Afterwards, the packet hash has to be compared with the reference one and together with the matched bits of this target interface, detect whether the packets would have been captured or not by the probe running on it. This mechanism is only possible by knowing routing information. Due to its dynamic and distributed nature, it has not been considered as a viable solution.

Cluster metrics, however, may involve packet loss information, by following an approach similar to the one that will be exposed in section 5.5.1. This metrics may result as quite useless: loss on packet sampling, as will be reported in chapter 8, will be quite imprecise. Since packet loss computed on aggregated data produces good and reliable results, this would be a preferable solution for per cluster metrics.

Packet captures

Information obtained from packet sampling can be collected to be later on processed to satisfy custom queries on data traffic. Storing all captures that have been generated by each probe may demand too much storage.
Measures related to network status are generated in *near* real-time. Thus, as a big improvement with respect to previous work [19], it is not necessary to store all captures.
Only a small subset of sampled data has to be stored by this component. To perform analysis at flow level, only the packets captured on border interfaces are relevant for further analysis that will be exposed in section 5.5.

OutPacketCapture											
date	period	hash	capture Router	direction	src Ip	dst Ip	src Port	dst Port	proto	capture Timestamp	useFor Loss

Table 5.5: Structure of packet capture information to store

Table 5.5 shows how those data have been organized:

- `hash` is necessary to distinguish captures of different packets and to couple the ones that belong to the same packet
- `captureRouter` indicates the interface that captured data
- `direction` indicates whether the capture has been done when packet was flowing in (`I`) or out (`O`) the monitored network
- `srcIp`, `dstIp`, `proto`, `srcPort` and `dstPort` represent the 5-tuple that is needed to identify the flow of traffic in a TCP/IP stack. These information should be extracted by each packet that has been captured
- `captureTimestamp` indicates the time in which packet has been captured
- `useForLoss` is a boolean value that will be needed in section 5.5 to compute loss value

A greater amount of data will be generated for these values with respect to the results that have been previously exposed: for every border interface, probably at most N packets will be stored.

5.5 Per flow post processing

This component has been designed to produce, when required, statistics relative to specific flows of traffic. Differently from work done in [19], it has been considered that metrics that are relevant at this level, are the end-to-end ones. A possible application of that is to produce periodical reports for business customers to summarize performance that has been measured for their traffic to proof the compliance with Service Level Agreement. In order to produce this kind of data, it is necessary to build a component that can be queried on demand to produce them.
A flow of traffic that will be analysed can be seen as a group of packets that share common characteristic. In IP networks, that have been considered during this work, a unique identifier for a flow is given by the 5-tuple that is commonly used: source and destination address, protocol and source and destination port.
Different levels of detail can be given also for flows: by considering, for example, a subnetwork as address, it is possible to include packets that comes from several elementary flows. This will provide a flexibility in the use cases of this component: for instance, queries can be generated both for a single user querying a web server and also for a business site that has set a VPN tunnel towards another site of the same company. Furthermore, this part of the system, when queried about a flow, will produce statistics separately for each elementary flow that matches the request filter: it will be always possible for client applications that use those data to aggregate data depending on their needs.
Results that will be generated during this phase, by using packets' information collected during previous stage, can be divided between loss and time based, for which it is necessary to have a packet capture at the ingress and the egress of the network. Packet loss will necessary rely also on packets that have been captured only at one end of the network. In the following sections the approach that has been pursued during this work for each category of metrics will be exposed.

5.5.1 Loss computation

Lot of attention has to be taken when computing loss for each flow. Due to the randomicity of sampling process and independence of number of bits in different measure points, a packet that has not been detected in output interfaces can be lost or can have crossed an interface when exiting the monitored network that was running a filter too selective (that is to say an higher number of matched bits with respect the interface crossed at the ingress of the network) that would have not captured the current packet.
In order to limit this problem, it is possible to define an algorithm to detect packets that will be surely lost. A packet with a given hash `h1` that has been detected only in input, can be surely considered as lost whenever there is the guarantee that the

interface that should have been crossed at the egress of monitored network was configured with a number of bits n such that `h1` matches at least n bits.
As stated before, knowing which interface should have captured this packet is not trivial at all: routing information should be queried, but this is not considered feasible. As possible workaround, all the egress interfaces can be considered as possible target. By following this approach, a number of bits of match equal to the maximum value (that will be defined n_{max}) among all these interfaces should be considered.
As a result, the number of potentially matching bits of packets hashes has to be computed, by knowing the reference hash, and compared with n_{max} value. Whenever this is greater than n_{max}, the information about packet loss will be reliable and the value `useForLoss` exposed in table 5.5 will be set to `true`, otherwise will be `false`.
In conclusion, considering only packets that have `useForLoss` value set to `true` will result in an underestimation of the loss, while considering all packets will produce a result where loss is overestimated. In this latter case, packets can be marked as lost even though they have reached the destination correctly, since they have not been sampled by any measure point while exiting the network. These two different approaches will provide bounds to the estimation of loss value and their precisions will be evaluated in chapter 8.

5.5.2 Time based measurements

Once that packets' captures have been isolated for each flow, they have to be coupled between input and ouput, then sorted by ingress timestamp. After this processing, one way mean delay and jitter values can be computed as by their definition.
As already stated, only one way metrics will be provided with this kind of architecture: considering the different portions of the network that are monitored in the two directions of the same flow, round trip delays would be biased by this problem and for this reason have not been implemented.

Chapter 6

System implementation

During this chapter, components that have been previously described in chapter 5 will be analysed from a technological point of view. Software, framework and implementation measures that have been deepened during this work will be exposed.

6.1 Network probe

Probe that was used during this work was developed by [23] and already integrated by [16], with improvements to support message publication in queues and updating the configuration.
During this thesis, some little changes have been done to support the *keep alive* messages, to fully integrate packet captures with the architecture of cluster manager and the synchronization mechanism of preprocessing part.
The development of the probe is based on eBPF, to provide access to network data at a kernel level, and BCC tools to easily integrate kernel modules with high level Python code.

6.1.1 eBPF and BCC

Extended Berkley Packet Filter (eBPF) [7] is an extension of BPF framework, that was developed for packet capture analysis. The improved version provides generalization for all kind of system tracing in Linux and the possibility to inject code at runtime, providing customizable applications.
eBPF has been developed in the context of IOVisor open source project [9], that provides also tools to help developers in implementation of this kind of tools. Among them, *BPF Compiler Collection* (BCC) [6] provides a way to inject limited C code that can be compiled and injected in kernel through a Python interface.
Limitations about the code are due to the fact that this code will run in kernel mode.

Thus it must be safe, provide high-performance in order not to block indefinitely the whole device and must not interfere with other programs: as an example, it cannot have loops, while it must have a limited size and memory accesses that are executed must be always valid.

6.1.2 Architecture

The organization of probe can be seen in figure 6.1 and it is mainly composed by:

- a REST API that will receive a configuration file to run the capture coherently with other instances of the probe in the network. Specifically, during this work, it has been configured as shown in listing 6.1
- PNPM Manager (*Packet Network Performance Monitoring* was a previous definition for AMPM framework) is responsible to interface the REST API to the other parts of the system, to collect data and process their storage, on the base of the chosen communication protocol
- BPF Manager that is in charge of configuring the actual behaviour of the probe starting from the configuration. It sets up the eBPF program that will be injected in the kernel space and initialises maps in which to store results

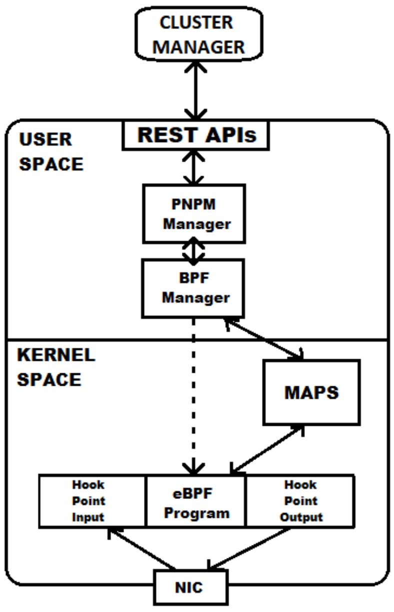


Image derived from [23]

Figure 6.1: Probe code structure

```
{
    //dynamic hashing program id
    "prog_id": 3,
    //duration of a period
    "mp": 30,
    //total duration of a run
    "mpc": 120,
    //alternate marking values
    "starter_mark": 1,
    "next_mark": 2,
    //filter on UDP traffic
    "proto": "udp",
    //reference hash (as decimal value)
    "match_value": 383146267,
    //starting number of matching bits
    "match_length": 0,
    //hash function to use
    "hash_function": "bob",
    //probabilistic upper bound on packets to sample
    "npkts": 100,
    //the interfaces on which capture traffic
    "netifs": [...]
}
```

Listing 6.1: Configuration file used for current system implementation

- maps, which can be accessed both from user and kernel, are the part that allows to store and access results about captures. Those are particular structures, whose primitives are provided by eBPF giving a controlled access to kernel space

- eBPF program is the code properly configured by BPF manager, that will be compiled at runtime to be run on each packet capture. This will store its results in maps, that will be later on read from the user space. The main advantage of running this part of the probe directly in kernel mode is from a point of view of performance. Continuous privilege changes from kernel (where packets are received and initially processed) to user space and vice versa will require hundreds of CPU clock cycles for each of them, greatly reducing the throughput of packets that can be analysed. This is one of the key point that lead to the development of BPF framework and its extension[1].

[1]For more details visit https://ebpf.io/what-is-ebpf

Another possibility to exploit this performance would have been to modify the kernel code, which can be quite challenging, especially from a maintenance perspective

- hook points are kernel event handlers to which code that is running is attached. Whenever one of these events is triggered, the program is executed. Two different hook points have been used during the development of the probe: one for packets that flow in the network interface and one when packets flow out. The latter case is the one that caused degraded performance during probe development: no primitive has been provided to get the timestamp of packet and it is not directly provided among the information as happens in input case. As already explained before, the output hook is not used during this work
- the Network Interface Card through its driver triggers BPF program that is linked to hook points. From a logical point of view, probe distinguishes each interface into two different parts: the input one, that will have `_IN` suffix and output one, with `_OUT`

Marking bits

A brief analysis on bits that have been used to provide marking while dealing with IPv4 traffic has to be carried out. Two possibility have been provided while developing the probe [23] and are represented in figure 6.2:

- using two bits: a good candidate is the Type Of Service header field. Historically, this field has evolved over time: with `RFC2474` [41] it was split into DSCP field (first 6 bits) and 2 unused bit (later on defined ECN bits in `RFC3168` [40])

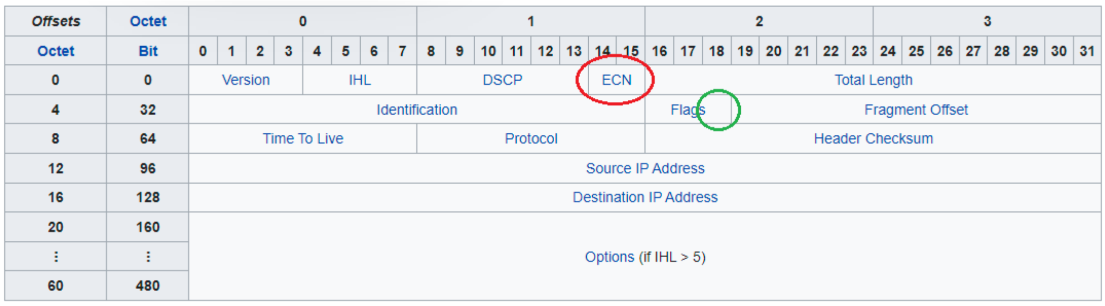


Image derived from https://en.wikipedia.org/wiki/IPv4

Figure 6.2: Viable solutions for packet marking: one bit in green and two bits in red

to provide a coherent definition of Differentiated Service between different versions of IP protocol. These unused bits have been marked with values `0x01` and `0x10` or left untouched as `0x00` when packet has not to be monitored

- using a single bit: by providing a filter to exclude some kind of traffic (that corresponds to `discarded` probe's configuration parameter) it is possible to distinguish traffic that won't be marked. Exploiting the last bit of *flags* field in IP header, it is possible to mark alternately the traffic with 0 and 1. This approach has not been considered as scalable as the previous one since a filter has to be set up in every probe, so the first one has been preferred during this work

6.2 Message queue

During previous work [16], the message queue has been added to the system. Among the possible implementations of a publish subscribe software, such as MQTT or ZeroMQ, the one that fits the requirement of the system is Apache Kafka [4]. Additionally, libraries for several programming languages are available and will be exploited when integrating following components.

6.2.1 Apache Kafka

Apache Kafka is an open source software, firstly developed by Linkedin and then it has become part of projects of Apache Software Foundation. It has been designed as a message queue that aims to provide high throughput, to be scalable, persistent and highly available. It is usually used, among several use cases, for messaging, metrics sending, log aggregation and commit log. These design choices and possible use cases are the ones that meet the scalability and robustness of AMPM system. In order to provide these properties, Kafka is provided as cluster of servers, that are defined *brokers*. As for all distributed system, a coordinator has to be present, and Zookeeper plays this role in a cluster as central manager. It is in charge of monitoring the status of each node and of all the structures that each of them manages and it has to synchronize all of these elements. It is also responsible of restoring data whenever a failure of a node occurs.
Messages have been defined as *events* in Kafka terminology. They are composed by a key, a message (both these value should be represented as string), a timestamp and optionally also headers to describe them. Events are generated by client applications (*producers*) and used by other ones (*consumers*).
Queues are defined as *topics*, which are divided into several partitions. Inside a single partition, messages are processed in their generation order. Events are assigned to partitions basing on event's key, so its definition is relevant for the

order in which events will be processed.
A key point in Kafka features is partition replication along different brokers. Whenever one of these fails, information is not lost and system can continue working. Additionally, replicas can be used to span data geographically to make them more easily available to data consumers.
When dealing with replicas, it is necessary to set some level of guaranties on acknowledgment from brokers. Data can be considered sent without any ack message from the servers (just relying on TCP acknowledgments), with just one from a server (so that there is the certainty that event has been correctly processed by the application itself) or with acks from all the servers that have to process the event. These options represent an increasing level of confidence on persistence of data, but also show decreasing performance.
Another feature provided by Apache Kafka is data retention. It is possible to preserve events for a fixed amount of time instead of deleting them whenever they have been processed by consumer application. In this way it is possible to provide robustness in case of failure of client applications, even for a quite long time.

6.2.2 Configuration

Once that features provided by Kafka have been described, the configurations that involve the message queue and used during the implementation, will be analysed.

PNPM manager

Probe has been set up to be a Kafka producer. Particularly, different topics have been defined to distribute various data accordingly to what exposed in section 5.2:

- **avg_metrics**$_i$ (with $i \in [0,1]$) will contain aggregated data. The key that is used is `period_interfaceID` and the value is the aggregate itself
- **metrics**$_i$ will contain sampled values. The same key-value definition of previous point has been used
- **probe-alive** have been added during this work to support the management of active probes. Differently from other topics, these data will be sent with key `period` and value `interfaceID`, so that whenever multiple instances of consumers applications will be set up, data coming from the same period will be processed by the same instance. The same reasoning was not necessary for other topics: as will be seen in following sections, accesses to them will be subordinated by an access to `probe-alive` topic, so that only an instance will be responsible to process it. Additionally, those kind of data can be read from other applications in every order, so it is not necessary to use multiple queues for this kind of data

Additionally, the producer has been set up to wait a single ack message from brokers: it is a good compromise between performance and reliability. Messages that are not acknowledged correctly will be retransmitted up to twice before signalling an error about cluster not reachable.

Brokers

Some configurations, that can be seen in listing 6.2, have been customized on Kafka cluster:

- `num.partitions` is set to 3 since data processing part will read topics, as will be explained in devoted section, in a distributed manner, by using several workers
- `default.replication.factor` equal to 3, that is the suggested value from Kafka developers, so that up to 2 brokers may fail without loosing data. This introduce the need of at least 3 brokers
- `log.retention.hours` is set so that data are persisted for 12 hours. These can be customized on the basis of the available space on brokers.
  By considering B brokers, given P the number of monitored interfaces and data equally distributed across partitions, data occupation for each period on each of them will be made up of several contributions, weighted by a factor $\frac{3}{B}$:
  - at most P aggregated messages, since not all interface may encounter traffic
  - probably at most $P \cdot N$ sampled data, where N is max number of packets to sample in each interface

```
############################# Log Basics #############################
# A comma separated list of directories under which to store log files
log.dirs= #to be set up on every node
# The default number of log partitions per topic. More partitions allow greater
# parallelism for consumption, but this will also result in more files across
# the brokers.
num.partitions=3
default.replication.factor=3
#############################Internal Topic Settings#############################
offsets.topic.replication.factor=3
transaction.state.log.replication.factor=3
############################# Log Retention Policy #############################
# The minimum age of a log file to be eligible for deletion due to age
log.retention.hours=12
```

Listing 6.2: Extract for the configuration file used to set up Kafka nodes. Modified from [16]

  - P probe alive messages

By further investigating the maximum occupation of data per period, it is possible to precisely compute the possible amount of time that can be set as retention period according the available space on each broker

6.3 Cluster manager

In order to satisfy requirements that have been previously highlighted for this component, the solution that has been implemented can be mainly divided into two parts:

- a REST API that has been considered as a valid solution for the purpose of giving access to information about the clusterization of the network and also allows modifying its state
- a Kafka listener that interacts with keep-alive messages from probes and manages the activity of each of them

Considering these requirements and the availability of some functions' implementation as part of clustering optimization, Python has been chosen as base part for this implementation. Additionally, some libraries have been targeted to provide a fast and reliable implementation: Flask[2] for the implementation of the web server part and Flask Kafka[3] for message handling.

6.3.1 REST endpoints

The REST interface that has been designed during this work is just a proof of concepts of possible operations that this component can carry out and may be further extended. Basically, it involves three main separate resources:

- `clusters` allows to retrieve all the information about clusterization. It provides a single `GET` to browse its status. All modifications can be done by modifying following resources
- `topology` represents the underlying network that the manager should handle. For the sake of simplicity, it allows a `PUT`, to which to send the filename of the topology `graphml` that should be used to load the current topology of the system. Surely, it is not a definitive solution: file should have been

[2]Full documentation is available at https://flask.palletsprojects.com/en/2.0.x/

[3]Details at https://github.com/nimzymaina/flask_kafka

previously loaded in the filesystem of the server that runs this component. In a real environment it would be necessary to introduce endpoints to manage a complete set of CRUD[4] operations at level of nodes and edges of the graph

- `measure_points` represents the status of measure points in the network. It provides methods to retrieve, add, delete or update measure points. Starting from this information, the clusterization can be derived. A part that has not been developed concern the configuration of the probes: for current implementation, cluster manager assumes that probes have been configured by hand. When the system will be deployed in a real environment, configuration of the probes will represent a task that is in charge of this component. Knowing addresses of probes, their status and credentials, it would be possible to configure probes, interacting with routers (whenever they are already running the web server) or remotely connecting to them (via protocols such as `ssh`, `telnet` or also others)

At every change of clusterization, the cluster manager must notify, through a specific topic, other components that are interested in the current topology of the network. As for the implementation of this work, only the preprocessing system will be interested in these updates.

6.3.2 Message handling

Cluster manager follows a timeout handling mechanism that is similar to the one described in section 5.4.1. For every period, a timeout is set at each new message received, so that when it expires and no other messages have arrived, a check is performed to detect which interfaces have not sent the keep alive message: whenever this occurs for N periods consecutively, the probe is considered as not running anymore and the clusterization of the network is recomputed.

6.4 Real time preprocessing

This component has been heavily updated, also basing on the modification that has been carried out in [16]. Apache Spark has been considered a choice that can be maintained in current implementation. Some additional considerations instead have been carried out on the persistence layer, to fully provide scalability and efficiency, also by looking at the needs that the new implementation of the post-processing part requires.

[4]Create, Read, Update and Delete

6.4.1 Apache Spark

Apache Spark [5] is a well known framework currently part of Apache Software Foundation's projects, that is commonly used when dealing with huge amount of data. It is based on MapReduce paradigm, that processes intermediate data directly in memory, thus providing performance up to 100 times faster than basic applications implemented in Hadoop environment, and providing several high level APIs that allow functional programming which facilitates the implementation of applications.

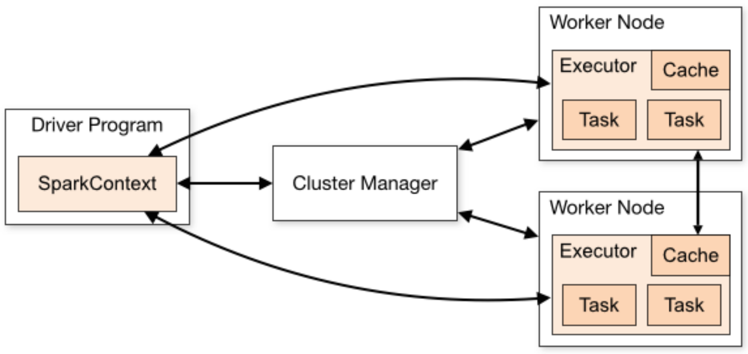


Image derived from
https://spark.apache.org/docs/latest/cluster-overview.html

Figure 6.3: Organization of a Spark Cluster

As shown in figure 6.3, Spark is organized in a series of nodes, that are called *workers*, that interact with a manager that coordinates their work. This architecture allows to exploit parallel processing of data that the application (that is called *Driver Program*) needs to perform.
Additionally, it offers natively the integration with Apache Kafka, so that some integration work will be directly done to optimise the pipeline that data has to traverse to produce results.
Using Spark framework, it is possible to manage large collections of data that are distributively stored in the memory of the workers in object defined as *RDD* (Resilient Distributed Dataset) that are one of the main abstraction that the framework provides. It is possible to write programs that exploit these functionalities in several programming languages (Java, Scala and Python).

6.4.2 Hadoop Distributed File System and Hive database

Some studies about data persistence have been carried out to improve the system that has been integrated in [16], which has left this component untouched, to allow this thesis work to be run in parallel:

- firstly, data coming from the message queue have no need to be persisted again (after having been stored in Apache Kafka server) before the preprocessing phase
- secondly, results that have been generated by the application itself, have to be persisted. During work by [19] the choice that has been pursued is to use Hadoop Distributed File System (HDFS), which provides good reliability of persistence, scalability and capability to process great amount of data. Some weak points of this choice will be highlighted later on, but as shown, it will be a good starting point to evolve the system

HDFS [8] has been developed as part of Hadoop environment to allow persistence of data in a distributed manner. It has been designed to provide horizontal scalability. Several servers (that in Hadoop terminology are defined *datanodes*) can be merged to form a cluster: at high level, this can be seen as a unique persistence layer, that can be evolved over time by providing additional hardware. As for all distributed systems, some coordination elements should be inserted, to provide the management and coherency of the system. In Hadoop environment, *namenode* is the server devoted to this task, which manages blocks' replication and logical filesystem organization, thus allowing to hide the complexity that stands behind real space organization.
Files that have to be stored are split in chunks of data called *blocks* with fixed dimension. Each of these will be replicated in R replicas (R depends on the configuration of the system) and stored into R datanodes as it is represented in figure 6.4.
HDFS usage presents some drawbacks that have been analysed during this work. Each block (which is by default 128 MB) can be assigned only to a single file: whenever a file is smaller or simply it does not fit exactly N blocks, some space will be left unused. Especially when the system has to deal with small files, a huge overhead is introduced in the storage system. Looking at results by [19] file generated are greatly below the block size. Whereas those behaviour may be due to the limited amount of data that is generated by the simulation, results that will represent cluster metrics will not exceed at most thousands of entry per each periods: this will surely result in a small file.
Additionally, in order to perform query on data to extract later on per flow metrics, a full search in data will be needed. This will surely degrade performance.
The solution that turns out to solve these problems is to rely on a Relational Database Management System. Among the one that provides data distribution and replication, the one that best fits problem requirement is Apache Hive [3]. A great integration with other Apache products is given and it can also be built on top of the HDFS.
This choice allows to maintain all the benefits of HDFS that have been previously

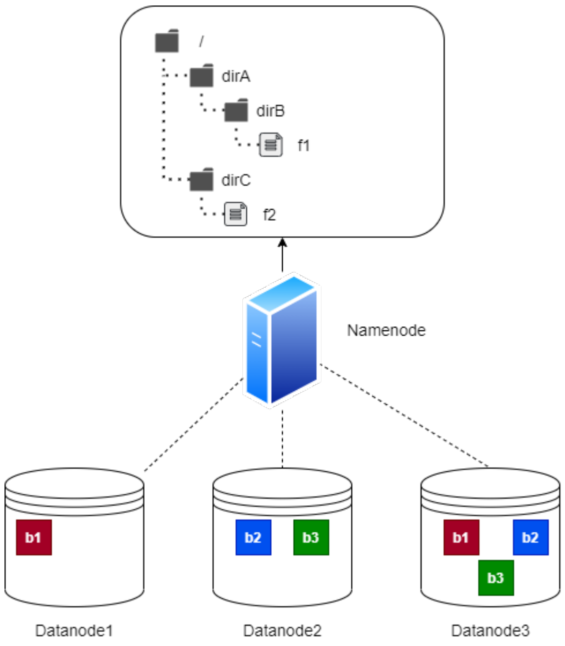


Figure 6.4: Example of HDFS file organization ($R = 2$)

exposed, leveraging on a product that tries to overcome weak points: the Hive server will allow to manage data organization, that will no more be divided in files, but it is managed by the database management system. Additionally, although some functionalities are not fully supported yet, standard SQL structures can be defined:

- data will be organised in tables with the structure that has been exposed in section 5.4.2
- indexes may be built to improve performance of data retrieving without the need to perform a full data scan

Furthermore, it is possible to implement a solution to clean periodically data, which can reach very huge amount of data: leveraging on triggers, built-in SQL procedure it is possible to easily delete data that can be considered too old.

6.4.3 Architecture and implementation choices

The choice of the programming language to use has been Java, due to great availability of libraries and also to be easily maintained in possible future works. The main part of the application has been implemented as simple Java program that listen on `probe-alive` topic to detect whether data of a given period has

been completely generated accordingly to algorithm that has been described in section 5.4.1. It also listen on `clusters-details` to be notified about clusterization of the network.
For this part, the choice not to use Spark framework is given by the fact that using batch processing in a *stateful* manner is not trivial (whenever new data arrives, an *RDD* should be updated and this operation is quite slow). Also the amount of data that has to be analysed has influenced the choice: considering a decade of periods that has to be completed yet (it is a very pessimistic assumption) and some thousands of active interface, this will results in some MBs of data, which is quite a reasonable value to be analysed in a single node, without adding useless synchronization overhead.
Whenever a period is ready to be processed, the analysis is assigned to a thread from a thread pool: this allows the main code to be executed without blocking and also to process several periods in parallel keeping the pace of new generated data. In order to keep all the necessary variables inside a single thread, `ThreadLocal` class has been exploited[5]. This allows easily to store value that can be accessed with a *thread scope*. Each thread will have different values for its configuration variables: those are stored in an object of class `ThreadContext`, that contains the period number and current clusterization of the system. The overall architecture of this part of the component is shown in figure 6.5.

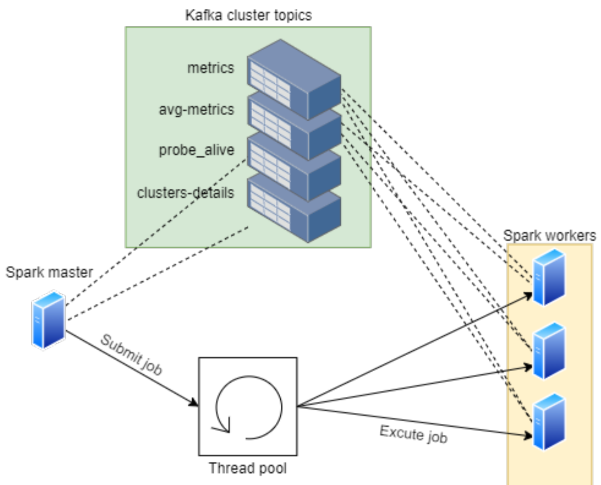


Figure 6.5: Overall architecture of thread organization in preprocessing part

The processing of a period is done through Spark framework in `Analysis` class.

[5]An interesting article that explains their behaviour is https://www.baeldung.com/java-threadlocal

In order to read data from the topics (for class `TopicReader`), a tricky solution has been implemented, due to limited offset management in Spark libraries: the ending timestamp of a period is computed and data are read from that instant on. It is possible that some data of following periods is read, but filtering them, using multiple queues and having periods of 30 s, limit this effect.
Some expedients have to be found also for final result storing in `Storage` class. The API to interact with Hive database is quite recent and not sufficiently mature: when saving an *RDD* into an Hive table, it is necessary to convert it into a `Dataset`, which has a behaviour similar to *RDD*s, but has been designed to store data in a tabular format. In order to later on write the data in the table, the function that allows this, considers the columns' order and not the variable name. This will not allow to write a generic and robust function. A workaround for this problem is to create a temporary table (which is visible only by current thread) and later on insert all its values in the main one (this operations relies on column name).

6.5 Per flow post processing

As already described in chapter 5, this component is designed to support queries that provide performance of flows of traffic. The choice that seems more adequate for this kind of task is to provide a REST API, so that this component will be always up and running, ready to process new query requests.
Basing on the huge number of data that will be generated by preprocessing phase and technological choices previously done, it has been necessary to deploy a software that relies on Apache Hive through Apache Spark.
In order to deploy a web server, the choice that provides a fast and reliable implementation of the system has been Spring Boot[6]. It is a very common Java framework that allows to build robust applications, without the need to write low level and repetitive code to parse HTTP requests.

6.5.1 Queries' and results' format

The parameters that the postprocessing query will receive are:

- temporal range (start and end) for which data have to be analysed: it should be specified as `date_period` to provide a variable granularity (from a single period to several days)
- topological information as source and destination IP subnets. Traffic that matches the filters that have been specified in one of the two directions will

[6]https://spring.io/projects/spring-boot

be carried out. This allows to provide different levels of detail that satisfy the various requirements exposed in section 5.5

Data that match those filters will be analysed separately for each period by grouping them into separate flows (basing on the IP 5-tuple identifier). Only flows that result to be relevant, with an overall number of captured packets greater than a given threshold, are considered for further processing. A coupling phase is then necessary, to analyse together group of packets that belong to the same flow, but to different directions. Afterwards, statistics are computed for these elementary flows and returned as result of the query. No aggregation of flows is performed, so that it is possible to manipulate the result as needed to extract custom reports without modifying this part.
Consequently, a possible result that may be returned by the postprocessing component will look like listing 6.3.

```
[{
    "period": "2021-11-20_15",
    "flows":[{
        // indicating the elementary flow
        "flowId": "10.0.10.5_10.1.56.2",
        "numPacketsUp" : 55,
        "numPacketsDown" : 51,
        // bounds for upstream flow
        "lossLowerUp" : 0.052,
        "lossUpperUp" : 0.250,
        // bounds for downstream flow
        "lossLowerDown" : 0.013,
        "lossUpperDown" : 0.185,
        // time-based  measures for up and down-stream
        "jitterMeanUp":-318478.049,
        "jitterMeanDown":201400.000,
        "jitterStdUp":35214205.089,
        "jitterStdDown":41430559.985,
        "delayUp":100956119.048,
        "delayDown":110403926.829,
        // packets used to compute loss lower bounds
        "numPacketsLossUp" : 28,
        "numPacketsLossDown" : 27
    }]
}]
```

Listing 6.3: Possible result returned by postprocessing phase

After having described all the components it is possible to give a complete overview of the system as depicted in figure 6.6

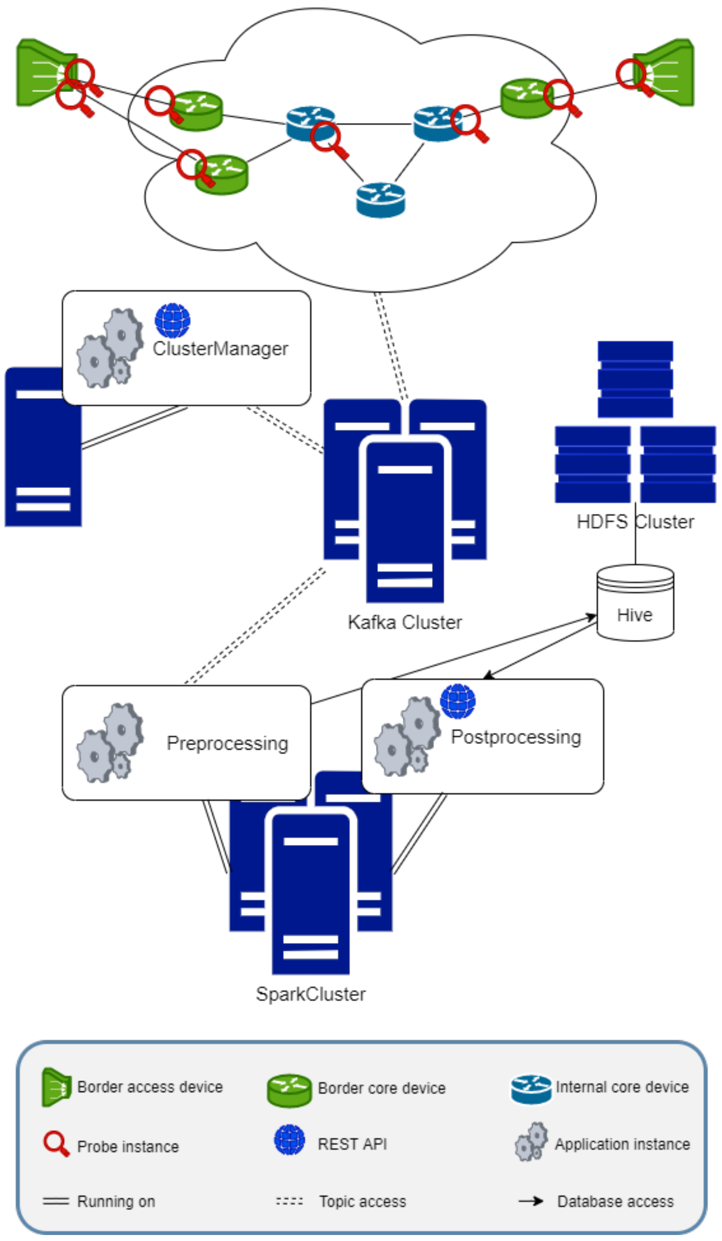


Figure 6.6: Overall architecture of the system

Chapter 7

Test environment

Similarly to previous works, there was not the possibility to deploy the whole system on a real architecture: this would have required several routers to execute the probe on them and also a huge number of server to deploy clusters that where needed (Kafka, HDFS, Spark and the cluster manager). Even though the gap between a development and standardization environment and a real deployment one has been closed during all of these works, the system is not mature enough to justify such an investment that would bring significant revenues in a short time. For these reasons it has been necessary to create an emulated environment on which perform tests to extract results that will be exposed in the following chapter. This part of the thesis will be devoted to describe this environment and provide all the adjustments that were needed.

7.1 Network emulation

As previously stated, it is necessary to introduce a virtual environment that mirrors the network topology and allows to perform traffic simulation and apply AMPM techniques. During previous work [19], a simulation based on Mininet framework has been proposed. This, however has some limitations and adopts some tricks to allow to perform packet captures as desired. During this work and the one by [16], the usage of a more recent library based on previous one has been introduced.

7.1.1 Mininet

Mininet [11] is a framework that has been developed to allow fast and realistic network simulations on a single machine. It provides a custom command line interface with sample topologies or it allows to built custom networks through a dedicated Python API. On top of this simulation, network exchanges and tests can

be carried out as for a real network.
Python API allows to create network devices on which run custom application (both on user or kernel space) to emulate different kind of network hardware. Additionally it is possible to run the OpenFlow [14] protocol on them to built Software Defined Networks.
Also links are emulated in the API, to allow to customize them, setting up parameters such as delay, bandwidth and loss that would have been measured on a corresponding physical link in a real network.

Artifacts and limitations of previous implementation

In previous work [19], the probe used [28] does not allow to capture output traffic on network interfaces. Although this does not represent a real limitation as for the current requirements, in this implementation a trick was built in order to emulate an output capture. Each physical link that has to be inserted between two routers has to be mapped as depicted in figure 7.1: a router (in the figure `R_1_2`) has been introduced in order to capture on its input interfaces the same traffic that flows out the output ones and switches have to be configured in order to forward packets as shown.

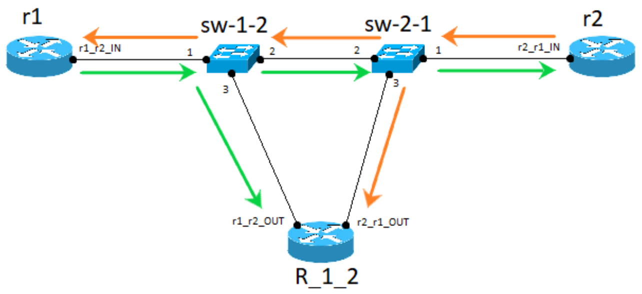


Figure 7.1: Mapping of physical link, with traffic flowing trough [19]

This artifact is not anymore necessary in this work: traffic that flows out interfaces has not to be considered. Furthermore, the new version of the probe [28] provides, although with degraded performance, also output traffic capture.
The principal limitations of the implementation provided with Mininet concern scalability of the solution:

- addressing plan as it was proposed implies that the number of devices in the network does not exceed 256
- in order to provide reachability between devices across the network routing mechanism has been set up: an instance of RIP protocol [15] has been run on each device (except for the ones that were built for capturing output traffic).

Distance vector routing protocols have some limitation, among them number of hops and number of nodes that can manage (in case of the used RIP implementation a maximum of 15 hops has to be used and no more than 40 routers should be put in the network)

The simulation that has been carried out during this work requires a number of routers that is around 3 times the one by [19], so a solution was found to overcome this problem.

7.1.2 IPMininet

IPMininet [12] is a framework that has recently been proposed as extension of Mininet API that among its features provides:

- simpler device configuration, providing several built-in types, as `Host` and `Router` classes that implements custom common behaviours in real networks
- automatic addresses assignment
- `Router` class provides natively routing mechanism: among them OSPF [13] is available

Current work integration

Using IPMininet allows to introduce automatic address assignment, loosing some control on mapping between hosts and addresses. This has not been considered as a main issue: hosts are directly reachable by using their name. Additionally, the native routing functionalities allows to build robust network with full reachability without having the need of setting up instances of routing protocols.
Links in the network topology have been directly mapped as virtual links (similarly to what has been done in [16]) on which custom values of loss, delay and jitter have been configured. Those values will be used to compare with the measures that will be obtained by the system.
The topology that have been used for the simulation is `Geant2012` from the *Zoo* dataset [36], which has been extended using the approach described in section 3.3.1, exploiting the optimal clusterization that has been found in section 4.4.

7.1.3 Naming conventions

In order to assign meaningful names to all routers and interfaces during all the processing of this work, some naming conventions have been taken.
Routers that were originally in the topologies have been named `Ri`, where `i` is the identifier that was already present in the `graphml` file. Border routers that

have been added as exposed in section 3.3.1, have been named as `Bi`, with `i` as incremental integer identifier.
Interfaces, have been named, similarly to what done in previous work [19], by concatenating the name of router that they belong to and the router that is directly reachable through that interface. As an example `R1_B2` is the interface of router `R1` that is linked to router `B2`.

7.2 Traffic generation

Traffic marking for a real development environment that satisfies requirement of AMPM has not been developed yet. [19] has already dealt with this problem: alternate marking has been carried out through `iperf` tool [10].

7.2.1 Iperf network measurement tool

`iperf` is a common network bandwidth measurement tool. It has a client-server architecture: flows of traffic are generated from the client towards the server.
Flows can be customized in several ways, such as protocol (TCP, UDP), number of packets per seconds, duration of the test. Among other tools that generate flows of traffic, `iperf` also allows to customize the value of Type Of Service (TOS) byte value in the IP header, which contains marking bits during this work.

7.2.2 Flows emulation

In order to emulate correctly several flows of marked traffic in the network, it is necessary to take some adjustments. Every client `iperf` applications should send marked traffic, taking care to swap the marking label at every period.
A script that handle client applications has been implemented by [19] to emulate this situation. However, it has to be run by hand on each desired virtual node. Although period duration has been set to 30 s and delays are admitted at period change, this may result in imprecise measurements due to time that is needed to run each flow. To provide a more structured and precise approach, the network simulation file has been set up to start all the necessary flows sequentially. Surely in this way, flows start time are closer than running each of them separately by hand.
In section 6.5.1 the need of bidirectional traffic has been highlighted. Script that was originally developed does not consider this situation: traffic is only generated by client instance towards the server one. The first approach that has been adopted is the usage of `-d` flag while starting the client application: this allows to perform the performance test in both direction, thus making the flow approximately bidirectional (this is not a real client-server interaction, since traffic in the two

direction is independently generated). However traffic that goes from the server to the client instance, has the value of TOS field that has been set in the server instance (that is by default 0). This solution would require to set up a new server instance with corresponding value of TOS: this may require lot of synchronization and report sent by `iperf` that can be useful for result comparison may be lost.
The other pursued approach was to built two different unidirectional flows between hosts: each of them act both as client and server for different communications. Although traffic in the two direction is not correlated as previous case, it is quite simple and provide marked traffic in a bidirectional way. For these reasons, it is the solution that was adopted for result generation that will be exposed in the next chapter. This expedient however has a drawback: flows are emulated in a bidirectional way, but UDP ports are not coherent. For this reason in the postprocessing analysis, flows has been considered only on the basis of their IP addresses and not on the 5-tuple in the simulations.
Additionally, there is the need of simulating traffic between hosts. Border routers have been considered as flows' endpoints: although from a logical point of view traffic should be generated from host attached to them, adding new hosts would not bring any benefit in terms of results obtained. Contrary, it would just provide a greater load on the machine that hosts the simulation.
Flows will be chosen among all the possible combinations of nodes in the network in a variable number, having the possibility to emulate different levels of load.

7.3 Test hardware setup

During this work, some servers were made available from TIM to build an emulation environment, as exposed in previous sections. Detailing, 3 servers which hardware specifics are shown in table 7.1 were shared with the development of [16]. Those have been organized as shown in figure 7.2:

- `Server1`, defined *Atreides*, contains the part related to the IPMininet simulation. All its softwares run inside a virtual machine due to isolation needs: thesis by [16] was being developed at the same time and network simulation running in parallel or specific configurations require separate environment

Server	# CPUs	CPU model	RAM (GB)
Atreides	56	Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz	192
Gesserit	48	Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz	384
SrvQuic	32	Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10GHz	64

Table 7.1: Hardware specifics for test servers

- `Server2`, defined *Gesserit*, contains all the high level components of this work: HDFS, Hive and Spark has been emulated through Docker containers that have been derived by [1] and [2]
- `Server3`, defined *SrvQuic*, runs a Kafka cluster that has been hosted as set of containers, differently from work done by [16], in order to facilitate the deployment. Although on the same physical machine, 3 brokers have been instantiated in order to provide data replication, which does not give any additive guarantee, but allows to emulate a system that will be closer to a real deployment

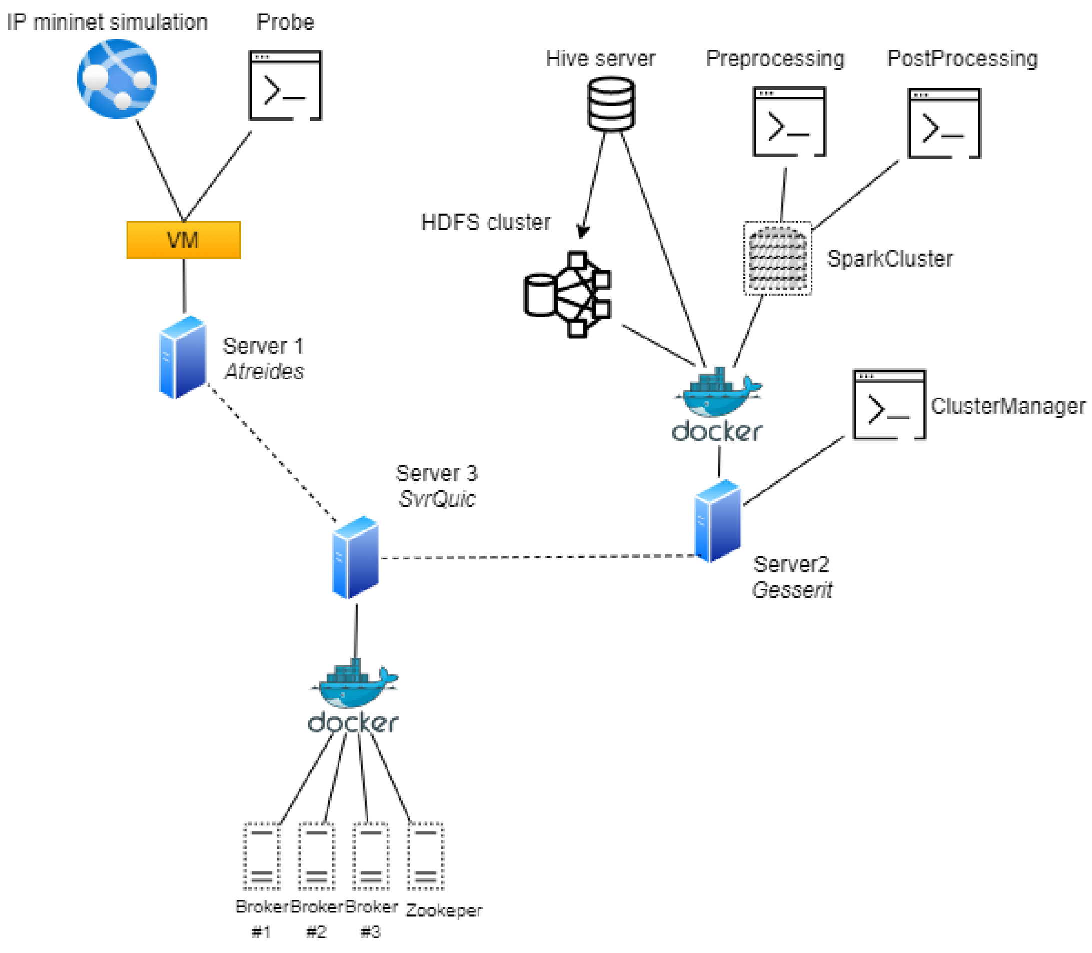


Figure 7.2: Architecture deployed in lab environment to perform tests

Chapter 8

Results

In this chapter, practical results that have been obtained during this work will be reported. It will involve the components that exploits the *BigData* frameworks, which represent the main part of this thesis.

8.1 Preprocessing

The analysis of the result of this component will follow two different flows: a first simple one will evaluate the time performance and the second one will be related to the analysis of the precision of the data that are generated during the preprocessing step, whenever it is possible to extract expected values to perform a comparison.

8.1.1 Timing results

Time performance has been measured for the preprocessing component. This has been necessary to correctly dimension the number of threads that have to be allocated in the thread pool, so that data will be always processed in time and no resource are left unused. In order to perform this kind of measures, the execution time of each thread's code has been measured and reported. Different loads in the network have been emulated in order to evaluate the response of the execution time and the results have been reported in figure 8.1. The simulations have been carried out by generating different conditions of traffic (varying number of flows) for 4 consecutive periods and after each execution the system has been restarted. Specifically, `1 flow` and `5 flows` produce each of them 40 packets per second, `10 flows` is limited to 20 packets per second and `20 flows` has only a bitrate of 10 packets. This is due to limitation in amount of data that the probe can process (it has to be considered that all routers' traffic is processed by a single computational unit inside the virtual machine that hosts the network simulation).

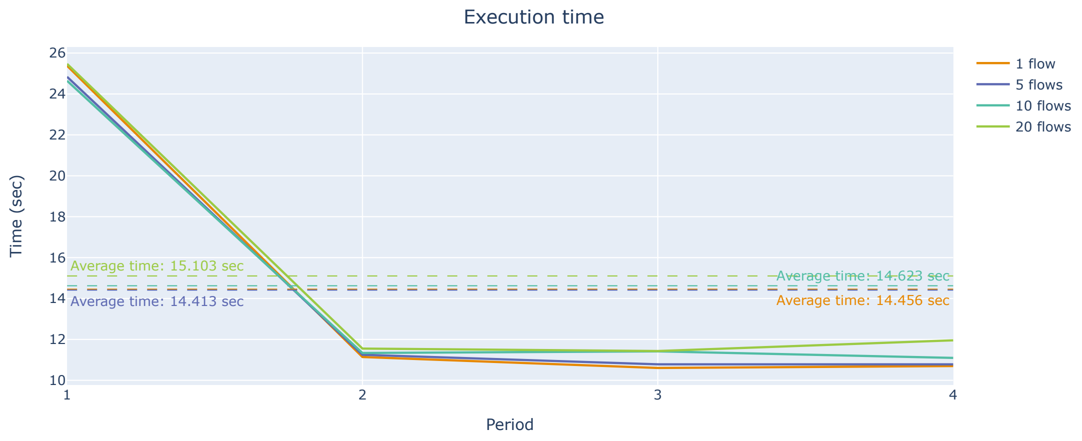


Figure 8.1: Execution time for preprocessing component under different load components.

As can be seen, each first period analysis requires much more time than the other ones: this is due to the fact that the system needs to connect to the database the first time, while the connection is cached for the other ones. On average instead, the time needed for the computation is greatly smaller than the period value (30 s). Timings do not change significantly when traffic in the network is increased, thus a pipelined mechanism seems to be unnecessary. However, it will may be useful whenever the system will be updated and more complicated computation will be carried out or whenever a smaller period duration will be chosen. The number of thread that will compose the thread pool is 2, so that the system is able to overcome some slowing that may occur in some part of the system: all the preprocessing units run on the same machine, so that transmission time between nodes (database and workers) are near to zero. This will not be the same in a real environment: transmission time has to be taken into account and possibly these tests should be carried out when deploying the system.

8.1.2 Loss related measures

First part of the result analysis will concern loss metrics. It is not trivial to compute an expected value that should be measured by the system. Traffic in each cluster may follow different paths and computing the loss that may be encountered in each of them will require lots of computation. For this reason, a simpler approach has been followed for this analysis and the following ones: it is possible to give bounds to this value. Inside a cluster, the minimum and maximum metrics will be the ones expected respectively on the shortest and on the longest path. For this reason, for every cluster of interest, the minimum and maximum length of paths inside it have

been computed and stored.
As a result, on a path of p links, the loss that will be measured whenever all the links suffer from a loss l will be equal to:

$$1 - (1 - l)^p \tag{8.1}$$

The computation does not include losses that may occur in routers' queues themselves which has not been set up in simulation but may be present in small percentages, so this value is a little bit underestimated.
Figure 8.2 reports results obtained in the first 6 clusters for number of traffic. The simulation that has been carried out considers 4 consecutive periods (generically numbered from 1 to 4, as will be done for all the following simulations) and setting loss at every link equal to $l = 1$ %. Flows are generated with a rate of 40 packets per second for each of them. It can be seen how measures result quite accurate with respect to expected values.

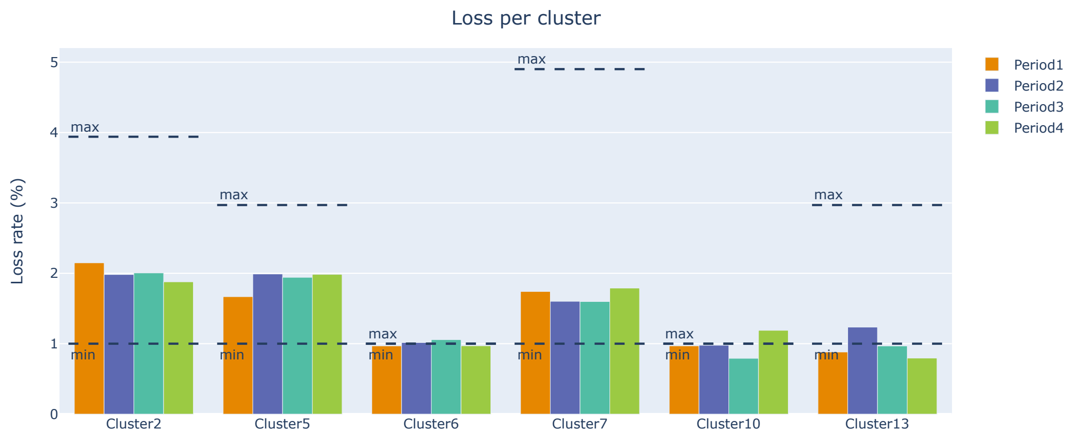


Figure 8.2: Measured loss in clusters with more traffic ($l = 1$ %)

Probability of loss

The precision of probability of loss depends only on the value that has been calculated in previous point, since it is simply a computation that has involved those values and packet counters, which are already used at previous point. Providing a comparison value will be quite complex, not giving any particular benefit. Thus, to validate results for this metric, a simple verification has been carried out. Probabilistic counters have been aggregated for each cluster to verify that both upstream and downstream values sum up to the number of packets lost in the cluster. Absolute value of the difference between these two values has been computed for the two approaches and have been reported in table 8.1.

	Max absolute value	Mean value
Upstream	1	0.412
Downstream	1	0.500

Table 8.1: Lost packets per cluster reconstruction precision ($l = 1$ %)

Values show that some differences exist, but considering that hundreds of packets flow into clusters and that results are obtained by casting values from float numbers to integer, some representation error can be encountered, thus providing a quite good precision of values.

8.1.3 Delay measures

Time based measures are reported at different levels of detail inside the network during the preprocessing phase: separate analysis will be carried out for each of them, providing different results' visualizations.

Cluster level

Similarly to loss evaluation, it is not easy to obtain the mean delay value that a clusters suffered from and the same approach as for loss has been followed. Consequently minimum and maximum delay in a cluster will be extracted, and those correspond to the delay of shortest and longest path. On a given path of p links with a delay d the resulting measured one will be equal to $p \cdot d$. Also this value will be quite understimated, since it does not consider transmission time inside the routers, but with the order of magnitude of milliseconds that will be used in the simulations, it may be considered negligible.
A simulation with a delay $d = 100\,\text{ms}$ on every link has been carried out. Despite this value is not realistic for a physical link, it is the one that made the assumption of negligibility of transmission delays valid. It has been noted that using a lower value gives results with higher variability of delays. This effect has been attributed to the performance of the probe. It has to be considered that the whole simulation has been carried out using a single instance of the probe, which is not a realistic condition. Tests carried out during its development [23] show good performance of the probe. Considering a situation where a great number of interfaces should be monitored is not a real situation, so this effect should be only limited to the test environment that has been set up. However, before a real deployment, it is appropriate to perform a complete performance analysis on dedicated hardware, which was not available during these works. Additionally, it has been noted that this effect depends also from the packet rate of flows that it is used, so the rate for time related measures analysis has been lowered to 20 packets per second in every

flow.
The same measure will be obtained both via aggregated and sampled data: the former measure, however, results to be biased by losses (mean timestamps will be computed on different sets of packets), providing values that are also negative or with incoherent orders of magnitude. The latter strategy to compute delay will not suffer from this effect, although the number of packets on which the metrics will be computed will obviously result lower than in case of no loss.
Additionally, it has been noticed that in case of loss in the network, some packets may suffer of greatly higher delays. This can probably be caused by routing information that is not consistent during transient periods that can last more in case of some routing packets are lost and causes non optimal path to be followed. For these reasons, the simulations to estimate time related measures will be carried out with no loss on links, so that all packets will with high degree of confidence follow the same and optimal path and a greater number of packets that can be exploited for measures will be sampled.

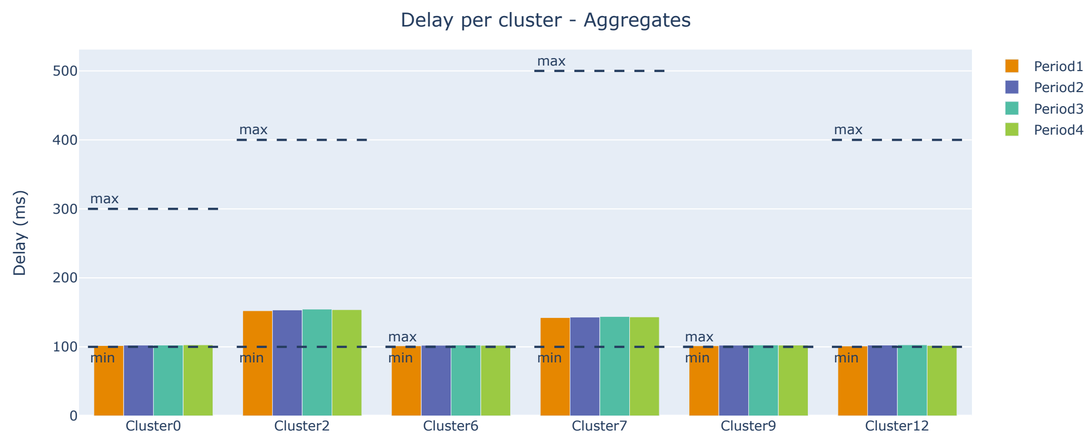


Figure 8.3: Delay per cluster measured through aggregated data ($d = 100\,\text{ms}$ and $l = 0\ \%$)

Figures 8.3 and 8.4 report results that have been obtained with both techniques on clusters that encountered more traffic considering the one used to compute the metrics (figure 8.3 considers all marked packets, while figure 8.4 only number of sampled ones, thus represented clusters do not perfectly correspond). These are quite promising: aggregate ones may be considered as more convenient to apply, since less data have to be collected, however, it is a valid solution only in a simulated environment with no loss. The ones obtained via sampled data require more computational resources, but are robust enough to be deployed on a real network.

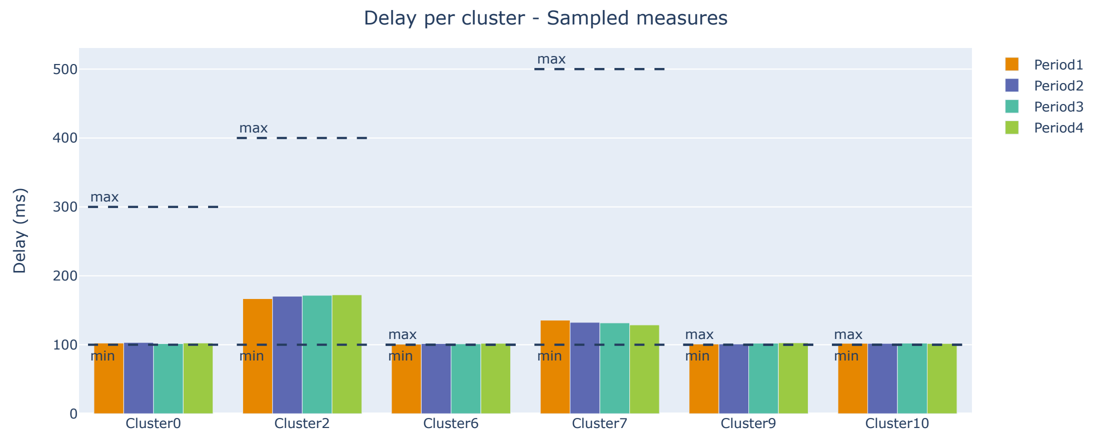


Figure 8.4: Delay per cluster measured through sampled data ($d = 100\,\text{ms}$ and $l = 0\ \%$)

Path and link level

A different approach can be followed when dealing with path and link delay measures. Although these values represent two different zooming levels, they can be estimated in the same way as a single value and no more as a range. For a path or link that involves p links with delay d, the overall average delay will be, as for previous point, $\mu_d = p \cdot d$. Thus, it is possible to compute the relative error in the metric computation as

$$e_{\%} = 100 \cdot \frac{\hat{\mu}_d - \mu_d}{\mu_d} \tag{8.2}$$

where $\hat{\mu}_d$ is the delay measures estimated during preprocessing phase.

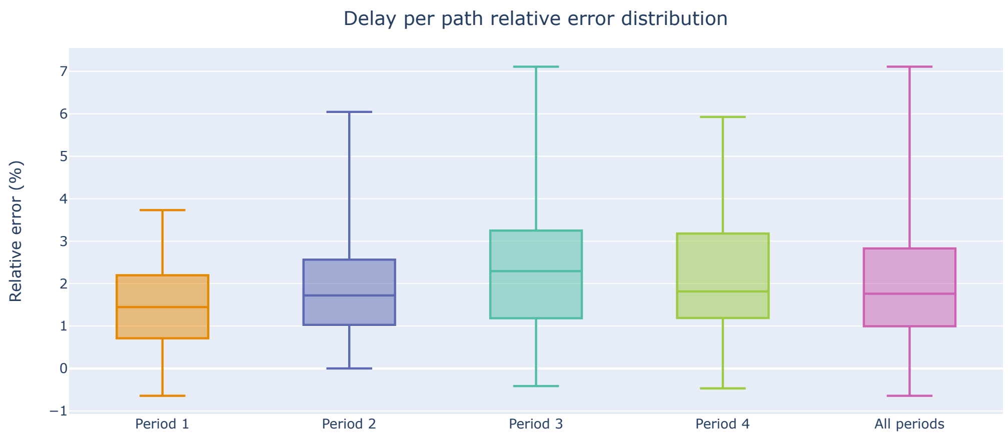


Figure 8.5: Relative error in per path delay computation ($d = 100\,\text{ms}$)

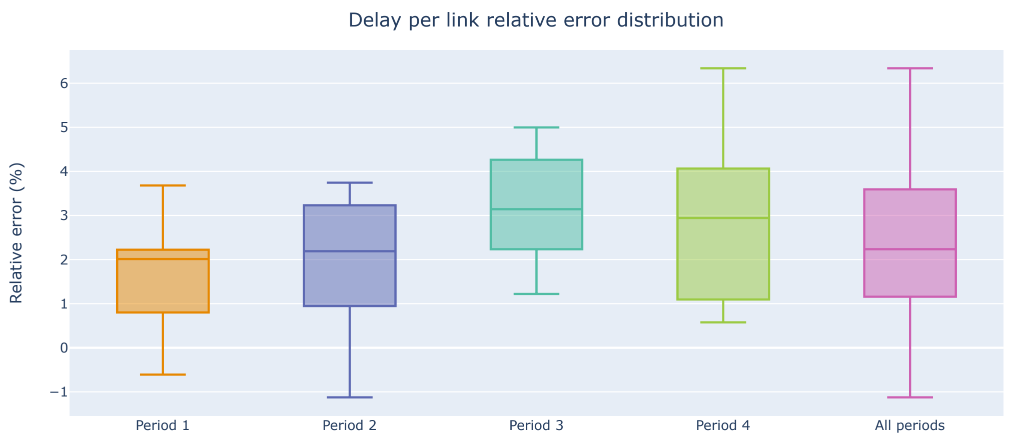


Figure 8.6: Relative error in per link delay computation ($d = 100\,\text{ms}$)

Figures 8.5 and 8.6 show the relative error distribution computed period by period in the simulation, reporting the distribution through boxplots, where extreme whiskers represent minimum and maximum values that have been encountered in the distribution. Results are quite encouraging: median values (and also the average ones) are globally around 2 %, and error never exceed 7 %.

8.1.4 Jitter metrics

Simulations have been carried out setting a constant jitter value for every link set to $j = 10\,\text{ms}$. This value, however, follows a different definition: this represents the standard deviation that will be used when generating the delay for a single packet. Delay distribution consequently will be a normal distribution with mean d and standard deviation j[1]. Jitter on a single link, in the definition of [39] that has been followed during this implementation, will be the difference among two independent normal distributions with parameters defined above, thus having 0 mean and standard deviation equal to $\sqrt{2}j$.

Cluster level

Cluster's mean jitter will have expected values that should be close to 0, since the resulting jitter will be the sum of normal distributions with 0 average. Figure 8.7 confirms this hypothesis in clusters that encountered the great part of traffic, providing values that are almost zero, even though some small fluctuations (with

[1]According to https://www.man7.org/linux/man-pages/man8/tc-netem.8.htm this is the default distribution that is applied whenever it is not specified and this is done in Mininet call (https://github.com/mininet/mininet/blob/master/mininet/link.py at line 300 and following)

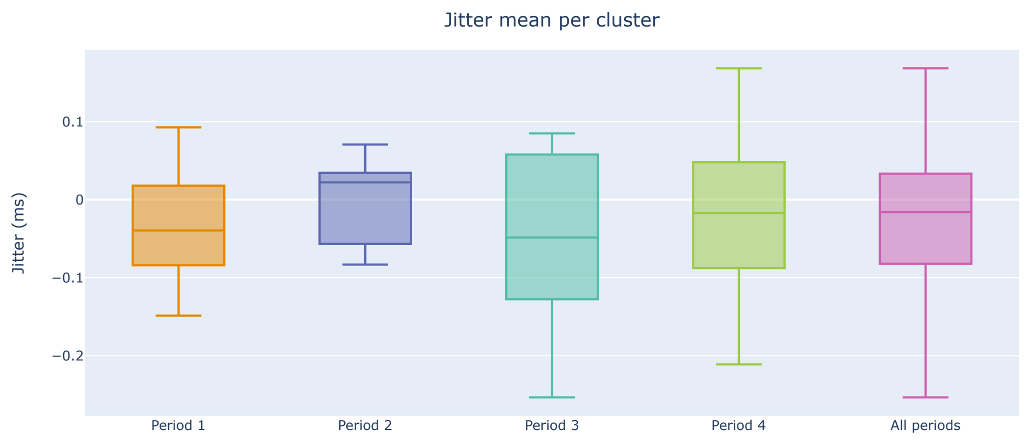


Figure 8.7: Jitter mean values per cluster ($d = 100\,\text{ms}$, $j = 10\,\text{ms}$)

respect to the order of magnitude of the standard deviation value) exist and they are due to the randomicity of packet sampling and synthetic jitter generation. It has been noticed that those fluctuations usually occur when a smaller number of packets is involved.

Path and link level

The same consideration made for cluster mean jitter can be carried out at path and link level.
Figures 8.8 and 8.9 show the behaviour of path jitter mean measurements: with respect to figure 8.7, path values are a little bit greater. This may be attributed

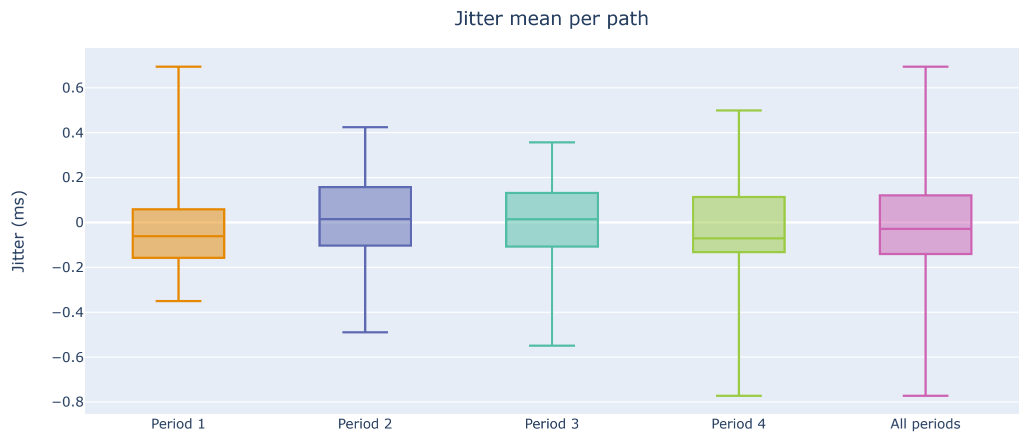


Figure 8.8: Jitter mean per path ($d = 100\,\text{ms}$, $j = 10\,\text{ms}$)

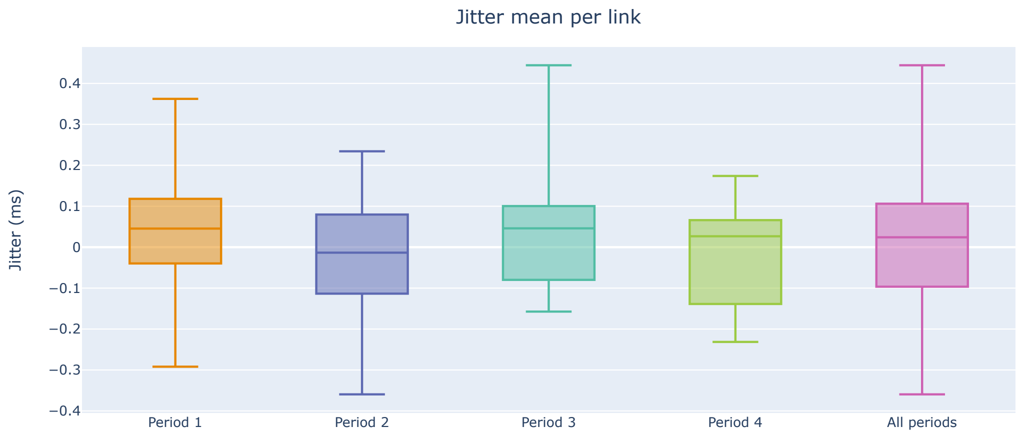


Figure 8.9: Jitter mean per link ($d = 100\,\mathrm{ms}$, $j = 10\,\mathrm{ms}$)

to the fact that over a single path (which is usually longer than simple links) a smaller number of packets is observed, thus producing results that may result more variable.
Different reasoning should be carried out for the standard deviation values. For every path, the resulting standard deviation will be the sum of the ones on each link. Given that, on a single link, jitter will have a distribution with zero mean and standard deviation $\sqrt{2}j$ and a path of length p, the overall standard deviation value will be $\sqrt{2}j \cdot p$.
In this way it is possible to collect and evaluate relative errors that can be obtained period by period and results can be seen in figures 8.10 and 8.11. It can be seen

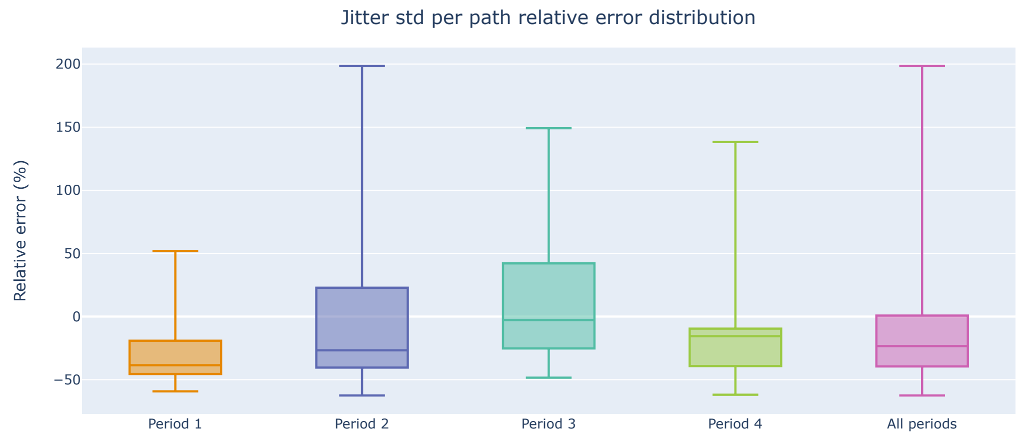


Figure 8.10: Jitter standard deviation relative error per path ($d = 100\,\mathrm{ms}$, $j = 10\,\mathrm{ms}$)

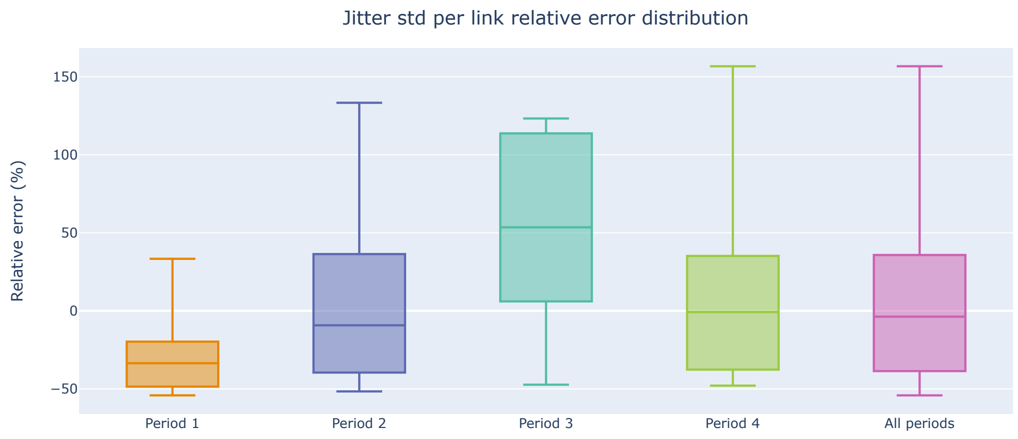


Figure 8.11: Jitter standard deviation relative error per link ($d = 100\,\text{ms}$, $j = 10\,\text{ms}$)

that the errors reach relatively high peaks, but the overall distribution has the central quartiles that are bounded in range $(-50\%, 50\%)$: it has been observed that those imprecisions are even more evident when the simulation is carried out with smaller delays and jitter values (using $d = 10\,\text{ms}$ and $j = 1\,\text{ms}$ peaks reaches 2000 %). As said before, this effect has been attributed to probe loads, causing packets to suffer from additional jitter. It has been noticed that considering greater order of magnitude in the simulation's timings, the effect decreases a lot, so there should be a systematic error that is introduced by the probe itself. Simulation with higher delay and jitter values, will even produce better results, but this may not be relevant at all since they use link configuration values that are too unrealistic. The effect of the system on a real environment should be measured to definitely evaluate the power of jitter standard deviation that has been proposed in this work.

8.2 Post processing

For post processing performance measurement, the metrics obtained for most relevant flows have been considered to report results. Simulations have been carried out following the same approaches and network simulation's configuration of previous section.

8.2.1 Loss related measures

Losses have been reported in terms of lower and upper bounds and compared to the expected one (obtained via equation (8.1)) in figure 8.12.
Results report the correct order of magnitude, although the precision is lower with

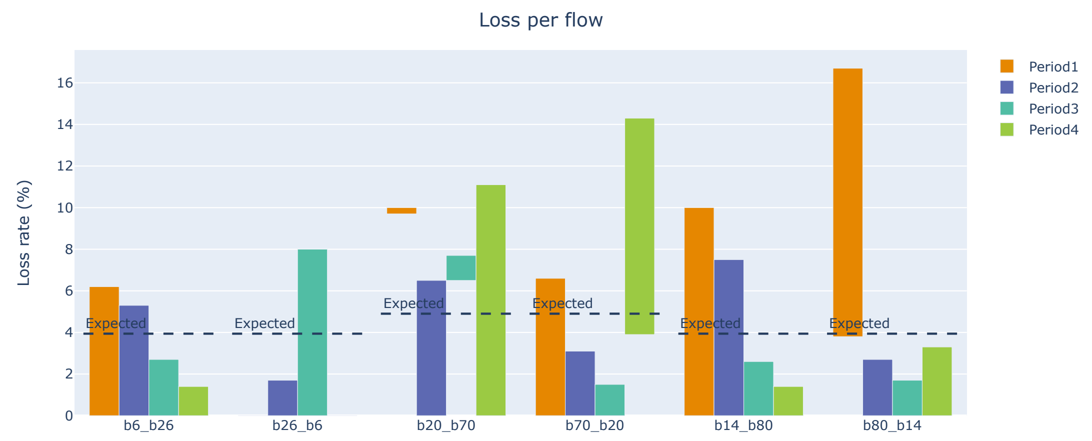


Figure 8.12: Loss comparison for most important flows ($l = 1\ \%$)

respect to the measures that have been obtained at cluster level. It can be seen that even though the range of values is acceptable, it does not always include the expected one. Packet sampling, as already analysed, causes this kind of effect that cannot be easily reduced.

8.2.2 Delay measures

Delay measures that have been obtained in the simulation have been reported in figure 8.13: a good overall precision can be found with no particular unexpected behaviour.

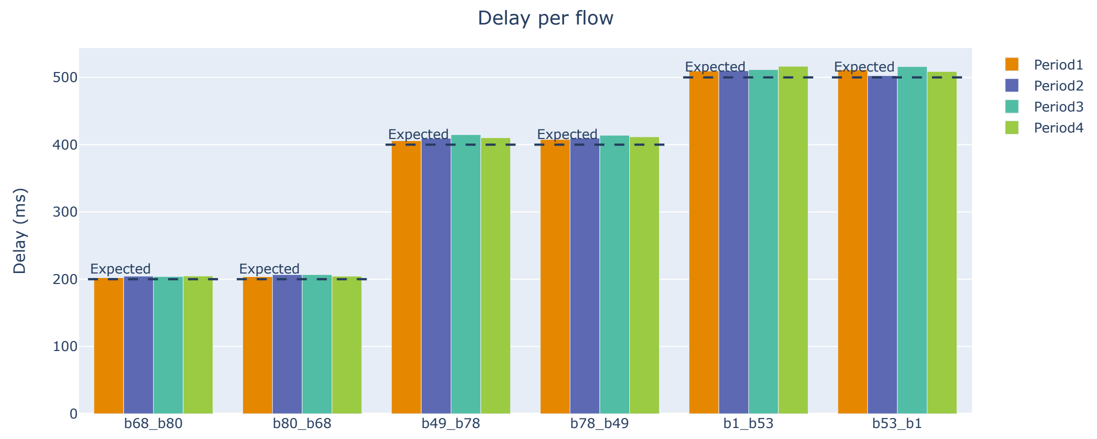


Figure 8.13: Delay comparison for most important flows ($d = 100\,\mathrm{ms}$, $j = 10\,\mathrm{ms}$)

8.2.3 Jitter metrics

Jitter result has been reported in terms of mean value (figure 8.14) and standard deviation (figure 8.15) for most important flows: the former is fully as expected providing values that are close to 0, while the latter produces values that are smaller than the expected one. This shows the same imprecision that has obtained at link and path level.

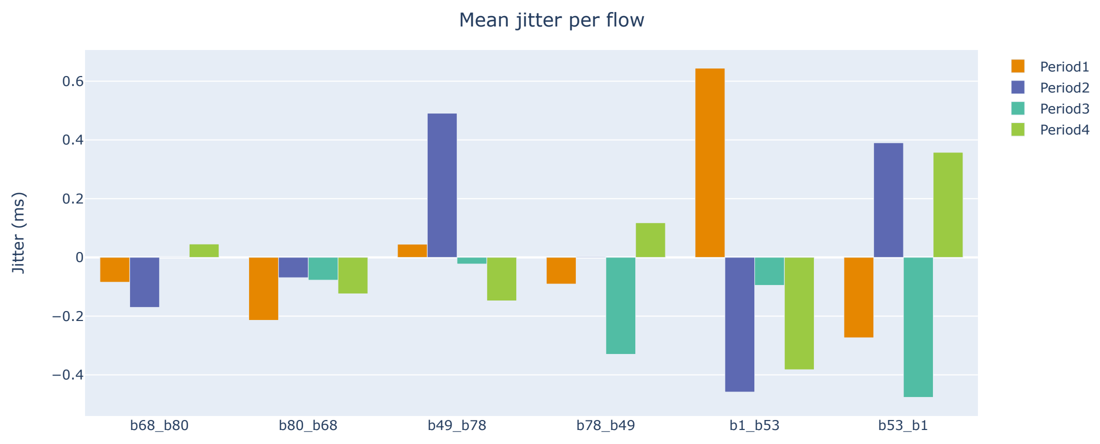


Figure 8.14: Jitter mean comparison for most important flows ($d = 100\,\text{ms}$, $j = 10\,\text{ms}$)

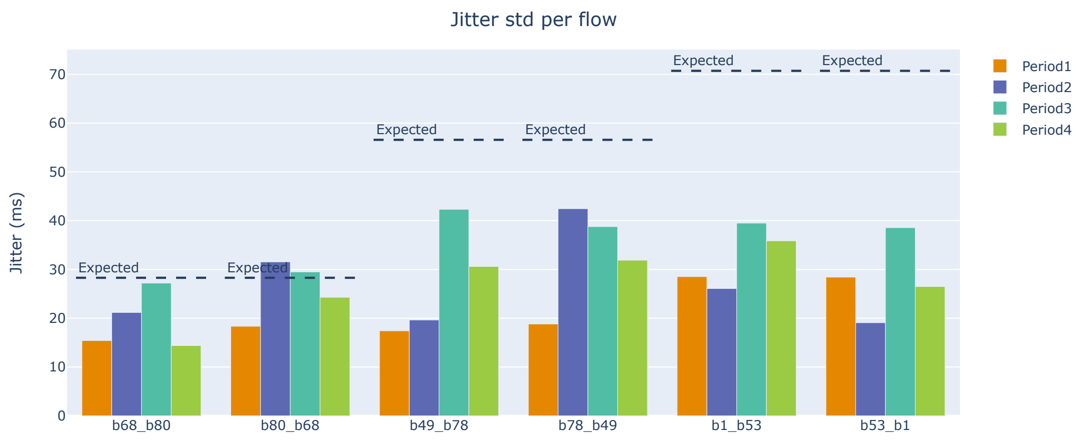


Figure 8.15: Jitter standard deviation comparison for most important flows ($d = 100\,\text{ms}$, $j = 10\,\text{ms}$)

8.3 Overall results considerations

By looking collectively at results that have been generated during this chapter, it is possible to to demonstrate the validity of the *Alternate Marking* techniques, together with the *BigData* approach and the improvements that have been carried out on this thesis.
However some values that do not provide a good precision have been found. This probably are strictly related to the simulation environment that has been used and should be further investigated by providing more robust environment on which to carry out this kind of tests. In details:

- online measurement of the standard deviation of jitter values provides an error that with high degree of confidence is related to a systematic error that has been introduced by the probe itself. This probably introduces a jitter that is on the same order of magnitude of the one introduced in the network simulation
- delay values are not so precise, but the effect is very limited (relative error is around 2 %). This also may be an effect of the probe, which can delay a little bit packets when is too overload
- loss values in post processing, even though 2 different estimations were proposed, present a variable imprecision. This has been attributed mainly to the fact that sampling techniques introduce a level of uncertainty that cannot be removed. It has to be noticed that however, number of packets that has been sampled for each period on each interface is quite limited: values greater than 100 cannot be used, since the load of the probe does not allow to process a greater number. Considering that each flow of traffic generates at least 600 packets on each period, the sampling would not be so representative. When dealing with a real environment, a greater number of packets can be sampled, thus allowing to better evaluate the precision of the proposed techniques.
- in conclusion, the standard deviation value for jitter in post processing is also not so precise. The considerations that have been carried out for this problem are the same that have been exposed for the same value in the online measurement

Chapter 9

Conclusions

The activities that have been carried out during this work were challenging for several reasons: understanding all the technicalities and difficulties that stand behind previous thesis, facing new technologies, with all the problem that this implies and integrating different software products may sometimes require lot of effort.
Despite all these difficulties, the work done has produced quite good results in every part of the system. However, criticalities has been found in results, some of them intrinsic in the *AMPM* techniques and others related to the simulation environment. Surely the latter can be deepen in the future to evaluate the framework in a real environment: efforts have shown that the methodology is quite promising and requirements that has been posed in the design of the *Alternate Marking* can be deployed in a real system.
This experience has been very positive and inspirational: I had the possibility to apply my knowledge in a real world problem and face out with challenges that has to be defined step by step during this problem. Despite the whole activity has been carried out in remote, I had the possibility to continuously discuss problems that arose with the supervisors and with another thesis student ([16]), trying to find out possible resolutions.

9.1 Possible future works

Surely this work cannot be considered as the final chapter of the group of thesis and works that have been carried out during the years. The main aim ot the thesis is to close the gap between the development environment and the deployment one. This has been partially done, especially in the components that has to deal with *BigData*. However, this opened a series of possibilities that can be further investigated in possible future thesis:

- probe has been already improved during the various work. However, it has been designed to run for a finite period of time: in a real environment this can work theoretically forever. The possibility of marking traffic inside a probe should be given, since tests have been carried out only with traffic simulators that have been configured *ad hoc*. To perform a more realistic simulation, real network traffic should be marked and used to perform this kind of simulation. Additionally, using JSON format to send data to the broker is a little bit verbose, it would be possible to investigate some binary format to serialize data. This will also imply modifying other components, such as preprocessing and cluster manager, but the implementation that has been given has tried to provide a modularity to support these kind of improvements over time. In order to deploy the system in a real network, it is also necessary to provide a correct synchronization between several probe instances. A possible solution would be to start probes only at a time that is multiple of the period duration (this will concern with all time synchronization issues)

- clustering optimization has been explored as possible solution. The implementation given can be surely improved and even optimized in terms of time needed to extract a viable solution. Additionally, it would be a more scalable approach if the agent implemented as neural network would be able to work using different network topologies. As highlighted in chapter 4, this is a complex and open topic in neural network field, so lots of studies will be necessary to provide such a solution

- cluster manager has been designed to be further extended. By updating properly the probes, it is possible to integrate an automation of measure point activation and deactivation and to keep an history of all the clusterization that has occurred in the network. Furthermore, also the clustering optimization may be integrated in this component. In a possible work, this component may work continuously to optimize the clusterization and to be able to automatically react to a probe failure, trying firstly to recover and eventually later on, to replace the measure point that cannot be activated anymore with other ones that provide a similar clusterization

- surely the preprocessing part is the one that has required the great part of this work. It is quite mature, but some problems have not been addressed. The component has been designed in such a way that it is possible to easily replicate it in multiple instances, but no tests have been performed in this direction. Additionally, no mechanism to recover from failures during the processing of data from a period has been set up. Moreover, data that are generated can be exploited in several ways: a machine learning algorithm that is able to predict eventual network failure and a dashboard to represent the status of the network may be developed. Furthermore, as far as the system is implemented, data are persisted forever. It is possible to study and develop automation to periodically removed old data, keeping a constant disk occupation

- data generated by the postprocessing component should be relevant for additional client applications. The main idea that motivates this design is to produce custom reports for customers, where to show aggregated network performance metrics related to their usage. This is just one of the possible applications that can be developed

Surely TIM will deepen these and other topics in future works in collaboration with Politecnico di Torino to build a system that is robust and scalable to be deployed in an internet network at large scale, such as the infrastructure that provides connectivity to tenth millions of users all over Italian country.

Appendix A

Practical simulation instructions

A.1 Reinforcement learning training

Clustering optimization has been run on `Server2`. A first try to run it exploiting a GPU has been carried out through Google Colab servers[1]. Resources that are made available are time limited and do not allow to perform a complete run of the algorithm. Even though a recovery mechanism has been setup, GPUs were not available and so the usage of servers provided by TIM has been preferred. This however can require up to the double of time in training.
The algorithm has been implemented in a Python notebook[2] that provides a way to train the network with parameters that can be easily customized. Among them the one that has been considered mostly during this work have been:

- `BATCH_SIZE` which provides the number of samples to use to perform stochastic gradient descent to train the network at every step. 32 seems to be a good trade-off between fast but imprecise training (small value) and slow, but precise one (higher value)

- `TARGET_UPDATE` that indicates the number of episodes after which updating target network with weights of the policy one. A constant value that outperforms well in all the situation cannot be found. Empirically, it has be seen that for episodes that require more steps (with complicated networks,

[1]It is a common platform to run machine learning algorithm, exploiting dedicated GPUs for free online (https://colab.research.google.com/)

[2]`clustering/clustering_with_reinforcement_single_net.ipynb`

such as `GeantExtended` or higher difficulty greater than 0.7) 15 provides good performance, while in case of episodes that are made up of a smaller number of steps 30 is more accurate

- episodes decay (`decay`) provides the amount of randomicity that has to be used during the training phase: the greater it is, the huger number of randomicity will be used. 1000 is the value that has been used in all the runs and provides good results for every situation
- learning rate (`lr`), widely used in all deep learning algorithms, indicates how much the network has to learn at every gradient descent step. 10^{-4} has been considered as good value for all the runs that has been carried out

Graphs about uniformity and loss that have been detected during the training phase can be queried while the software is running thanks to Tensorboard[3], which is a powerful tool that helps to visualize easily performance while developing machine learning algorithm.
A second part of the notebook provides a way to evaluate performance of every target network that is saved in a dedicated file, providing also the list of all the interfaces that the algorithm has selected as useful. Those values have to be inserted in the probe configuration file (`probe/Run/config_filter.json`) and in `cluster_manager_service.py` file[4], so that at bootstrap, the optimal clusterization is already provided.

A.2 Probe run

A probe instance must be activated on each machine running the simulation. In case of IPMininet simulation, a single instance is sufficient, since all the interface of routers in the simulation will be seen as virtual interface of the host machine.
In order to correclty instantiate a probe the command `sudo python run.py` (pay attention that probe code should run with Python at version 2, which is the default one in the virtual machine that has been setup) should be run in `probe/Software` directory. The Kafka brokers' addresses should be specified inside `probe/Software/app/kafkabrokerIP` file.
Configuration file for packet capture programs (`config_filter.json`) that has to be used should be customized inside `probe/Run`. This directory contains also all the necessary scripts to correctly configure the probe. These should not be directly

[3]https://www.tensorflow.org/tensorboard/

[4]It is in `cluster_manager/python-flask-server-generated/swagger_server/services` directory

run, since this procedure has been automatized inside the network simulation that will be explained in next section.

A.3 Network simulation

The emulation of traffic in the network has been completely automatized. File `mininet/netSwitch.py` contains all the necessary code and can be executed with `sudo python3 netSwitch.py ../topologies/GeantExtended.graphml` in `mininet` directory (whatever network that has border synthetically generated can be used).
Lines of code that are shown in listing A.1 allows to define link parameters that can be customized on demand to simulate different network conditions.

```python
for edge in G.edges():
    # iterating over topology edges and extracting involved nodes
    edge0_id = edge[0]
    edge1_id = edge[1]
    # creating link in the simultated network
    # (loss should be indicated with percentage value as integer)
    link = self.addLink(self.nodes_mine[edge0_id],\
      self.nodes_mine[edge1_id],intfName1=edge0_id+"_"+edge1_id,\
      intfName2=edge1_id+"_"+edge0_id, bw=100, delay="100ms",\
      jitter="10ms", loss=1)
```

Listing A.1: Simulation configuration of links (`mininet/netSwitch.py`)

Flows generation can be customized in terms of number of them or adopted policy to extract them. The snippet of code that is shown in listing A.2 should be modified in order to do that.

```python
# extracting 5 flows by all possible combination of border routers
flows = random.sample(list(combinations(mytopo.borders,2)), 5)
# selecting all nodes involved in the simulation
test_nodes_distinct = list(set(chain(*flows)))
...
# starting an iperf server on each node
for node in test_nodes_distinct:
    command = f"iperf -s -u -B {extract_ip(net, node)} &"
    net[node].cmd(command)
```

Listing A.2: Extraction of simulation flows (`mininet/netSwitch.py`)

Simulation file may require some seconds to be correctly loaded and configured: a manual synchronization has to be given by typing `y` or `yes` to start the effective traffic generation. As shown in listing A.3, this will run the `iperf_test` script (that has been implemented in [19], to which has been added the source address to use) on each node that has been chosen toward the other end of the flow.

```
command = input("Run the test?").lower().strip()
if command == "yes" or command == "y":
    # configuring probe progam automatically
    subprocess.Popen(["./start_capture.sh"], cwd="../probe/Run").
   wait()
    # starting the traffic for each flow
    for flow in flows:
        for i in range(2):
            # generating the right command with actual parameters
            # (script will be run in detached mode
            #     and will save results into a devoted file)
            test_cmd = f"../probe/Test/iperf_test.sh 40 {
   extract_ip(net,flow[i])} 4 30 1 2 {extract_ip(net,flow[1-i])}
   > result/{flow[1-i]}_{flow[i]} &"
            net[flow[i-1]].cmd(test_cmd)
```

Listing A.3: Simulation start (`mininet/netSwitch.py`)

A.4 Message broker

Kafka cluster can be started by running `docker-compose up` in a privileged terminal inside `kafka` directory[5].
It is possible to customize the properties of each broker as exposed in listing 6.2 by setting environment property as indicated in the images' documentation[6] by using properties name in upper case and replacing . with _

A.5 Cluster manager

In order to document well this component, that presents several REST endpoints, Swagger editor[7] has been used. This software uses an OpenAPI 3.0[8] specification file in `yaml` or `json` format that describes REST endpoints and allows to generate

[5]A guide on functioning of this setup can be found at https://www.baeldung.com/ops/kafka-docker-setup

[6]It can be found at https://docs.confluent.io/platform/current/installation/docker/config-reference.html#confluent-ak-configuration

[7]It is available for download at https://github.com/swagger-api/swagger-editor

[8]Format of OpenAPI 3.0 files can be found at https://swagger.io/blog/news/whats-new-in-openapi-3-0/

rapidly a server skeleton. `cluster_manager/swagger.yaml` is the file that has been used for the development of cluster manager component .
To start this module, command `python -m swagger_server` should be run in `cluster_manager/python-flask-server-generated` directory.
At its startup, by default `GeantExtended` topology is loaded, considering as measure points the ones that have been obtained by clustering optimization. This allows to have a component that is coherent with the whole system without any additional configuration. Obviously, this is just for demonstration purposes.

A.6 Big Data clusters

Before running big data components, it is necessary to setup the cluster that contains Spark, Hive and HDFS elements. This has been automatized in a single group of Docker containers. By running `sudo docker-compose up` in `docker-hive` directory.
At the first execution it is necessary to initialize the database and all the tables that are needed. An `SQL` script (`bigdata/create_db.sql`) has been provided for this purpose. Although it is possible to run this script inside the `hive-metastore` container shell, the solution that has been preferred during this work is to use directly graphic tools that are already integrated in advanced IDEs[9]. This integrated tools also provide a fast access to all the tables.

A.7 Preprocessing

The preprocessing component has to be run in a shell of Spark master container. The software has been developed with command line parameters to customize the behaviour, exploiting PicoCLI library[10]. Default values have been already provided so that it is not necessary to write all of them every time. However, a script (`bigdata/run.sh`) to be run with admin privileges as been declared to run all necessary code in the attached shell. Parameters' usage can be queried by modifying the script and passing `--help` option to the application.

A.8 Postprocessing

The base architecture of postprocessing part is the same of previous component. A similar script has been setup also here (`postprocessing/run.sh`), that provides

[9]During the development IntelliJ Idea has been used (https://www.jetbrains.com/idea/)

[10]https://picocli.info/

command line parameters.
The programs runs it REST API on port `8088`, which is remapped on same port by the `spark-master` container. The API is accessible as single entrypoint `filter?from=<yyyy-mm-dd_period>&to=<yyyy-mm-dd_period>&sourceIp=<ip[/mask]>&destIp=<ip[/mask]>`.
Configuration of the system can be given by properly modifying the configuration file that is commonly used while developing Spring Boot applications (`application.properties`).
The integration of this framework with Hive and Spark has been quite challenging:

- the Spark session has been recreated at each HTTP request (thanks to `@RequestScope` annotation). Despite it degrades the performance of each request, this allows to browse a fresh copy of the database: without this, a single Hive connection is created at first request and all the modifications to the database that have been performed after postprocessing start won't be seen correctly by the postprocessing application
- the deployment on the cluster gave some problems, due probably to some incompatibilities of libraries or some of them were missing on worker nodes, thus the choice of deploying the system only on a single machine exploiting all available cores with `local[*]` master indication was necessary. This, in a simulated environment where all the cluster runs on the same machine, does not affect the performance at all, but when the system will be deployed on different nodes, this should be in some way resolved or the usage of Spring Boot framework should be abandoned

A.9 Results' comparison

A Python notebook has been set up to reproduce in an easy way results that have been shown in chapter 8. It has been made available as `result.ipynb`. It can be customized to provide query for specific simulations and by setting values that have been used, such as loss, delay and jitter per link, to provide correct graphs.
Results about the preprocessing phase will be obtained by directly querying the database and extracting needed information. It is necessary to provide the information about the clusterization of the system in terms of minimum and maximum diameter of each cluster. These information can be included in `cluster_dimensions.json`, that should be extracted through a dedicated Python script (`extract_cluster_dimensions.py`).
Additionally, providing files about the simulation (`cluster_dimensions.json` and `flows.json` that are generated in `mininet/result` directory) also post processing results can be shown, by exploiting REST API calls about the flows that contains the major part of the traffic in the interest period.

Bibliography

[1] URL: https://github.com/big-data-europe/docker-hive.

[2] URL: https://github.com/big-data-europe/docker-spark.

[3] *Apache Hive*. URL: https://hive.apache.org/.

[4] *Apache Kafka*. URL: https://kafka.apache.org/.

[5] *Apache Spark - Unified engine for large-scale data analytics*. URL: https://spark.apache.org/.

[6] *BCC*. URL: https://github.com/iovisor/bcc.

[7] *eBPF*. URL: https://ebpf.io.

[8] *HDFS architecture guide*. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[9] *IO Visor Project*. URL: https://www.iovisor.org/.

[10] *iPerf - The TCP, UDP and SCTP network bandwith measurement tool*. URL: https://iperf.fr/.

[11] *Mininet*. URL: http://mininet.org/.

[12] *Mininet extension to make experimenting with IP networks easy*. URL: https://github.com/cnp3/ipmininet.

[13] *Open Shortest Path First*. URL: https://en.wikipedia.org/wiki/Open_Shortest_Path_First.

[14] *OpenFlow*. URL: https://en.wikipedia.org/wiki/OpenFlow.

[15] *Router Information Protocol*. URL: https://en.wikipedia.org/wiki/Routing_Information_Protocol.

[16] Francesco Palmieri. «Alternate Marking Performance Monitoring: experimental evaluation of the Big Data approach». MA thesis. Politecnico di Torino, 2021.

[17] Mauro Cociglio. *Statistical Tracking of Population Dynamics Over an Area*. Patent. 2020. URL: https://patentscope.wipo.int/search/en/detail.jsf?docId=WO2020127920.

[18] Mauro Cociglio, Calogero Corbo, Giuseppe Fioccola, Massimo Nilo, and Riccardo Sisto. *The Big Data Approach for Multipoint Alternate Marking method*. Internet-Draft draft-c2f-ippm-big-data-alt-mark-01. Work in Progress. Internet Engineering Task Force, Oct. 2020. 12 pp. URL: `https://datatracker.ietf.org/doc/html/draft-c2f-ippm-big-data-alt-mark-01`.

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

[20] Giuseppe Fioccola, Mauro Cociglio, Amedeo Sapio, and Riccardo Sisto. *Multipoint Alternate-Marking Method for Passive and Hybrid Performance Monitoring*. RFC 8889. Aug. 2020. DOI: `10.17487/RFC8889`. URL: `https://rfc-editor.org/rfc/rfc8889.txt`.

[21] *La rete per l'Italia - Netbook 2021*. Dec. 2020. URL: `https://rete.gruppotim.it/sites/default/files/download/NetbookFY2020_1.pdf#`.

[22] Guohao Li, Chenxin Xiong, Ali Thabet, and Bernard Ghanem. *DeeperGCN: All You Need to Train Deeper GCNs*. 2020. arXiv: `2006.07739 [cs.LG]`.

[23] Marino Urso. «High performance eBPF probe for Alternate Marking performance monitoring». MA thesis. Politecnico di Torino, 2020.

[24] Mauro Cociglio, Giuseppe Fioccola, Guido Marchetto, Amedeo Sapio, and Riccardo Sisto. «Multipoint Passive Monitoring in Packet Networks». In: *IEEE/ACM Transactions on Networking* 27.6 (2019), pp. 2377–2390. DOI: `10.1109/TNET.2019.2950157`.

[25] Tal Mizrahi, Gidi Navon, Giuseppe Fioccola, Mauro Cociglio, Mach Chen, and Greg Mirsky. «AM-PM: Efficient Network Telemetry using Alternate Marking». In: *IEEE Network* 33.4 (July 2019), pp. 155–161. ISSN: 1558-156X. DOI: `10.1109/MNET.2019.1800152`.

[26] Giuseppe Fioccola, Alessandro Capello, Mauro Cociglio, Luca Castaldelli, Mach Chen, Lianshu Zheng, Greg Mirsky, and Tal Mizrahi. *Alternate-Marking Method for Passive and Hybrid Performance Monitoring*. RFC 8321. Jan. 2018. DOI: `10.17487/RFC8321`. URL: `https://rfc-editor.org/rfc/rfc8321.txt`.

[27] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. «An Introduction to Deep Reinforcement Learning». In: *CoRR* abs/1811.12560 (2018). arXiv: `1811.12560`. URL: `http://arxiv.org/abs/1811.12560`.

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

[29] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. *Neural Message Passing for Quantum Chemistry.* 2017. arXiv: `1704.01212 [cs.LG]`.

[30] Guy Almes, Sunil Kalidindi, Matthew J. Zekauskas, and Al Morton. *A One-Way Delay Metric for IP Performance Metrics (IPPM).* RFC 7679. Jan. 2016. DOI: `10.17487/RFC7679`. URL: `https://rfc-editor.org/rfc/rfc7679.txt`.

[31] Guy Almes, Sunil Kalidindi, Matthew J. Zekauskas, and Al Morton. *A One-Way Loss Metric for IP Performance Metrics (IPPM).* RFC 7680. Jan. 2016. DOI: `10.17487/RFC7680`. URL: `https://rfc-editor.org/rfc/rfc7680.txt`.

[32] Al Morton. *Active and Passive Metrics and Methods (with Hybrid Types In-Between).* RFC 7799. May 2016. DOI: `10.17487/RFC7799`. URL: `https://rfc-editor.org/rfc/rfc7799.txt`.

[33] David Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P. Adams. *Convolutional Networks on Graphs for Learning Molecular Fingerprints.* 2015. arXiv: `1509.09292 [cs.LG]`.

[34] Alessandro Lazaric. *Transfer in Reinforcement Learning: a Framework and a Survey.* 2013.

[35] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning.* 2013. arXiv: `1312.5602 [cs.LG]`.

[36] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. «The Internet Topology Zoo». In: *Selected Areas in Communications, IEEE Journal on* 29.9 (Oct. 2011), pp. 1765–1775. ISSN: 0733-8716. DOI: `10.1109/JSAC.2011.111002`.

[37] Maurizio Molina, Fredric Raspall, Saverio Niccolini, Dr. Nick Duffield, and Tanja Zseby. *Sampling and Filtering Techniques for IP Packet Selection.* RFC 5475. Mar. 2009. DOI: `10.17487/RFC5475`. URL: `https://rfc-editor.org/rfc/rfc5475.txt`.

[38] Jennifer Rexford, Derek Chiou, Matthias Grossglauser, Benoît Claise, Albert Greenberg, and Dr. Nick Duffield. *A Framework for Packet Selection and Reporting.* RFC 5474. Mar. 2009. DOI: `10.17487/RFC5474`. URL: `https://rfc-editor.org/rfc/rfc5474.txt`.

[39] Carlo M. Demichelis and Philip Chimento. *IP Packet Delay Variation Metric for IP Performance Metrics (IPPM).* RFC 3393. Nov. 2002. DOI: `10.17487/RFC3393`. URL: `https://rfc-editor.org/rfc/rfc3393.txt`.

[40] Sally Floyd, Dr. K. K. Ramakrishnan, and David L. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. Sept. 2001. DOI: `10.17487/RFC3168`. URL: `https://rfc-editor.org/rfc/rfc3168.txt`.

[41] Fred Baker, David L. Black, Kathleen Nichols, and Steven L. Blake. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. RFC 2474. Dec. 1998. DOI: `10.17487/RFC2474`. URL: `https://rfc-editor.org/rfc/rfc2474.txt`.

[42] Richard Bellman. «A Markovian decision process». In: *Journal of mathematics and mechanics* 6.5 (1957), pp. 679–684.