POLITECNICO DI TORINO

Master's Degree in
Computer Engineering – Data Science

Master's Degree Thesis

Alternate Marking Performance Monitoring: experimental evaluation of the Big Data approach

Supervisors:
Prof. Riccardo Sisto
Prof. Guido Marchetto
Company supervisors:
dott. Mauro Cociglio
dott. Massimo Nilo

Candidate:
Francesco Palmieri

October 2021

Table of Contents

Introduction 5

1.1 Goal of the Thesis 6

1.2 Chapters Content 7

Performance Monitoring 9

2.1 Alternate Marking 9

2.1.1 Packet loss measurement 11

2.1.2 Timing Aspects 12

2.1.3 Delay measurement 13

2.2 Multipoint Alternate Marking 14

2.2.1 Packet loss 15

2.2.2 Clustering algorithm 16

2.2.3 Delay 18

The Big Data Approach 21

3.1 The Working Principles 22

3.2 Achievable results 24

Starting Architecture 27

4.1 Implementation 28

4.2 Possible Improvements 31

Technologies 35

5.1 Probe 35

5.1.1 eBPF 35

5.1.2 Overall architecture 36

5.1.3 Possible Configurations 39

5.1.4 Differences with previous probe 40
5.2 Kafka 42
5.2.1 Functioning 42
5.2.2 Main use cases 46
New Architecture 47
6.1 Mininet Configuration 47
6.2 Implementation 51
6.2.1 Kafka Configuration 53
6.2.2 Sink VM 55
6.2.3 Mininet VM 56
6.2.4 Probe(oldHDFS) 59
6.2.5 Final Probe 60
6.3 Workflow 63
Lab Simulation 65
7.1 Model 65
7.1.1 Flow between servers 68
7.2 IPMininet 70
7.3 Workflow 73
7.4 Tests and Performance 75
7.4.1 Kafka Performance 76
7.4.1 Traffic Variability 77
Conclusions and future works 79
Appendix 81
Bibliography 85

Chapter 1

Introduction

More and more in recent years, Service Providers have to manage an ever-increasing traffic on their networks, due to the spread of various types of services such as streaming video, audio, video games, etc. A concrete example can be found in the just passed year, where the pandemic has forced us in the majority of cases to use online services to make up for the inability to work, study or do recreational activities, making the networks increasingly busy. In this scenario, Service Providers need to consider issues such as packet loss, delay and jitter since the majority of traffic is now highly sensitive to these metrics. In fact, many applications, for example all video conferencing tools, but also all real-time applications, do not work properly if the delay and packet loss are greater than certain thresholds. Therefore, too high values of these metrics translate into a bad user experience which is then reflected on the customer satisfaction towards the service provider.

In this scenario, while a lot of work has been done by the Internet Engineering Task Force (IETF) for what concern fault detection and connectivity verification, everything related to performance monitoring has not evolved in the same way. Performance monitoring instead is a key point for the service providers and is easy to understand why. “A good way to monitor traffic means easy maintenance and less effort to understand what and where do improvement, since it could potentially detect faults and weak points of their network infrastructure” [6]. For these reasons, ISPs were incentivized to design new methods for real-time performance monitoring that were as simple and effective as possible. From this need is born TIM's RFC8321 [1], described in detail below, which introduces the concept of alternate marking, the pivot on which this work is based. The process described in this RFC is based on passive/hybrid monitoring technique which, as described by RFC7799 [2], in opposition to active monitoring, which would need to generate

special packets to monitor the traffic, is limited to observing the existing traffic or at most to modify unused fields. Furthermore, the alternate marking is potentially applicable to any type of packet-based traffic such as IP, MPLS, and Ethernet, both unicast and multicast. This technique mainly addresses the measurement of packet loss, but as we shall see, it is easily applicable to the measurement of the delay variation as well. The advantages of this approach are:

- Ease of implementation: can already be implemented on existing network nodes
- Applicability: as already mentioned it can be implemented on any type of packet network.
- Low computational demand: there is a minimum additional load
- Accuracy: you can choose between different degrees of granularity, from the single package to larger groups.
- Robustness: can handle out-of-order packet

At this point, to obtain a monitoring divided by flows (where a flow is a set of packets that share some parameters, for example the same header field of the IP source in the IP packets) both the RFC8321 [1] and the RFC8889 [4], talk about a filter-based approach. In practice, a filter is installed on a network device for each flow we want to monitor. However, this approach raises two problems:

1. The number of filters we can set is limited and the total number of flows can increase dramatically making this approach impractical.
2. Setting up and configuring the filters requires considerable effort. The increase in monitored flows would require new filters making this approach non-scalable.

1.1 Goal of the Thesis

To overcome the problems described above, TIM in collaboration with the Politecnico di Torino has started various projects aimed both at the creation of two probes, both based on the eBPF [23] (Enhanced Berkeley Packet Filter), and at the

drafting of a first architecture capable of implement a new type of traffic analysis. This architecture, described in the draft The Big Data Approach for Multipoint Alternate Marking method [3] and in a previous thesis [6], consists in installing the first developed probe [5], both on border routers, which must collect data on packets entering and leaving the network, and secondly also on some routers within the monitored network, thus allowing us to have finer or coarser measurements.

The idea is to consider a single filter instance per network node that captures all passing traffic and obtains performance details based on the flow at a later time, sending the collected data to a Big Data server, which will analyze it, revealing the criticalities of the network. This makes the measurement effort independent from the total number of flows. It is also possible to divide the results by "groups of nodes" (or clusters), which simultaneously with a flow-based approach, makes this architecture extremely flexible to different types of measurement.

My work starts from these assumptions. In fact, this first architecture is not definitive and mainly served as a starting point to demonstrate the feasibility of this type of measurement. My task is to modify this model in some of its parts in order to get closer and closer to a final architecture actually applicable on real networks. To do this, a new probe [7], again developed by the collaboration between TIM and the Politecnico, will be analyzed and new technologies aimed at making the model easier and more practical to apply are considered.

1.2 Chapters Content

The thesis can be split in two main parts: a first theoretical part that illustrates the methods behind the starting architecture, and the architecture itself in detail, and a second one where the new technologies used, the new architecturesimulations and implementation of the latter are explained carefully.

The following chapters are organized as follow:

- Chapter 2 presents an overview about performance monitoring with a special focus to RFC8321 and multipoint draft, that constitute starting point of the thesis.
- Chapter 3 exposes the principles behind the big data approach in performance monitoring.
- Chapter 4 describes the starting architecture deployed for simulation environment. It aims to describe its constituent components and all the possible improvements that can be done.
- Chapter 5 provides a general view about some of the technology involved in this thesis to develop new components.
- Chapter 6 illustrates the workflows of the new architecture, the modification applied to the starting one and how each component interacts with the other one.
- Chapter 7 exposes, in addition to the simulated model, a more realistic configuration, implemented in the TIM laboratories and the results obtained.
- Chapter 8 summarizes the overall results obtained and lays the foundation for the possible next steps.

Chapter 2

Performance Monitoring

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

2.1 Alternate Marking

The idea behind Alternate Marking is to count the packets passing through a node on the network and compare this number with that counted by the previous or next node in order to identify a possible packet loss. To make the comparison make sense, you need to make sure that the nodes count the same group of packets. There is therefore a need for a way to group the packets that must be counted together.

The solution, which is described in RFC 8321 [1], is to mark packages with different colors so that those with the same color are counted together. It is logical to think that two colors are enough to make this approach feasible. In fact, by changing color after a certain period of time, the colored packages during two consecutive periods will have different colors, making it possible to distinguish the packets of one period from the next or the previous one. Hence the name Alternate Marking.

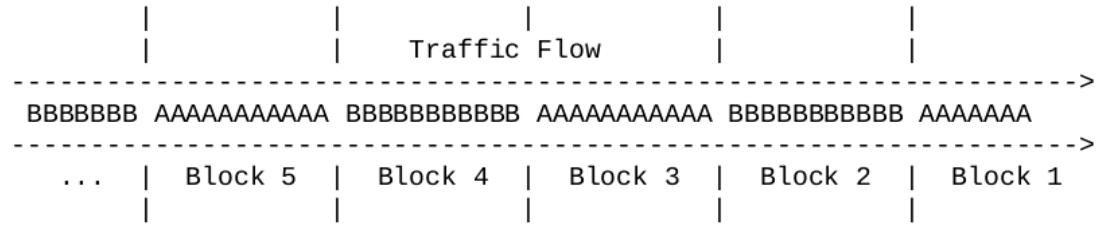


Figure 2.1 - Traffic coloring [1]

It is clear at this point that the marking operation is crucial, and to do so you can follow two strategies:

1. Mark the packets according to the flow
2. Mark packages according to link

This second case is to be used if we want to monitor the traffic that passes through a certain link. In this mode, all traffic must be colored before it passes through the link to be monitored by the nodes that use that link.

The flow-based marking, on the other hand, is to be preferred when we are interested in monitoring certain traffic flows, and therefore avoiding marking the traffic that we are not interested in monitoring. The flows to be marked can be traced through some header fields (for example in IP the flows are identified by source and destination addresses and their respective ports). Since marked traffic will be able to pass through various points in the network, a large number of nodes will have to be able to recognize the marked traffic in order to follow the path of the packets.

In this modality, the problem of where to mark packages is raised. In general, the most advantageous solution is to mark packages as soon as possible. Thinking about the network of an ISP, the best choice will be to mark the traffic on the border nodes, so that the traffic generated by the clients will be marked as soon as it enters the network. In cases where there are multiple marker nodes, these must be synchronized to avoid inconsistencies in the marking of the packets.

For what concerns instead the color change, it can be done on the basis of a fixed number of marked packets, or as already mentioned, on the basis of a certain period of time. Although the first method is easier to implement, it can lead to synchronization problems between markers as we do not know after how much a node will change color. It is therefore recommended that the markers change color after a certain period of time.

2.1.1 Packet loss measurement

The division into groups of packets serves, as mentioned, to measure the loss of packets. The idea is simple: each node counts the packets of a block and compares the value obtained with the one calculated by the next node. If the two values do not match, it means that there has been a packet loss between the two nodes. This is always true in a point-to-point path, like the one shown below.

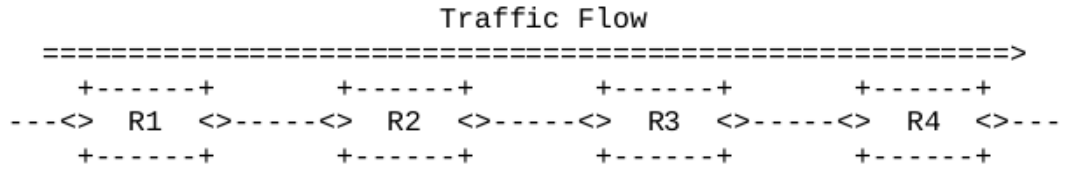


Figure 2.2 – Point to point network [1]

So, imagining that we have two colors A and B, and want to monitor the traffic between R1 and R2, it will be necessary that the two nodes keep four counters each, Ci(A) and Co(A) to count packets of color A in input and output and Ci(B) and Co(B) to count those of color B.

At this point, when traffic marked with A crosses R1 and R2, the counters C (A) of the two nodes will be updated respectively. When, after a given period of time, a color change occurs, the counters C (A) are stopped and the counters C (B) start counting and so on in a loop, resetting the counters each time it starts counting.

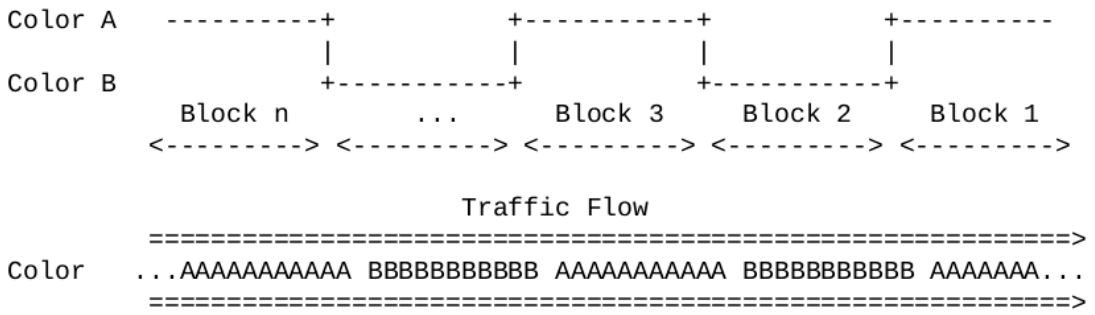


Figure 2.3 – Color change [1]

The comparison between the counters of the same color allows to identify the packet loss in a period of time.

2.1.2 Timing Aspects

What has been described so far leads to an important consideration, namely the one concerning when to read the counters. R1 and R2 need to choose the best time to read the counters, in fact reading them too early can lead to incorrect data due to out-of-order packets. Furthermore, between R1 and R2 there is a network delay, calculated as the difference between the moment in which the packet arrives at R1 and the time in which it arrives at R2. Another factor not to be ignored is the misalignment of the clocks between the nodes.

What we want then is to identify a time interval in which we are sure that if all nodes read the counter, it is guaranteed that:

- The value read refers to the same block in the whole network
- The value is not affected by out-of-order packets and its counter is stopped.

Therefore, considering a period of duration L, and a network delay d, in general, the most advantageous solution is to read the meter at L/2.

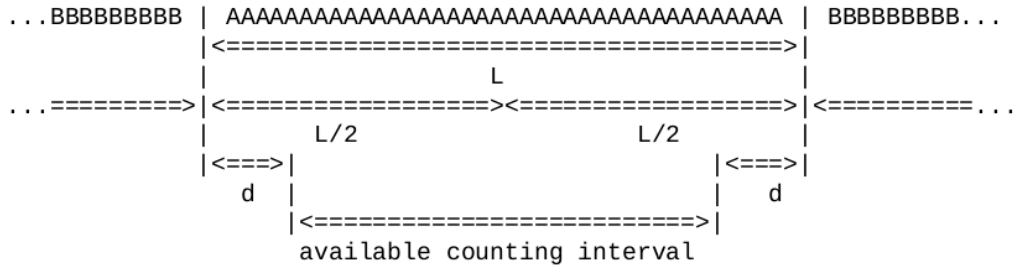


Figure 2.4 – When read counters [1]

However, these considerations suggest that there is a minimum limit for the length L in order to have d<L/2, and that it must be evaluated in every implementation. Choosing L long enough increases our chances of getting the right value.

2.1.3 Delay measurement

As mentioned above, RFC 8321 [1] can be easily adapted to calculate network delay. In theory, it would be enough to compare the timestamps of the same packet recorded by two different nodes. In practice, there are several alternatives.

In the Single-Marking method, a router retains the timestamp of the first received packet after a color change. This timestamp compared with the timestamp of the first packet received by the next node, gives us the delay to cross that piece of the network. For example, if router R1 stores the timestamp of the first packet of the block with color A, TS(A)R1, and router R2 does the same with the first packet marked with A, storing TS(A)R2, the delay $TS(A)R2 - TS(A)R1$ is the delay associated with that packet. To measure the delay of multiple packets, it is sufficient to store timestamps of different packets, for example every N received packets, or even the timestamps of each packet. This approach, however, is sensitive to packet loss and out-of-order reception, as the router cannot know if the sampled packets are the same as the previous routers (they may arrive in a different order), or if a packet has been lost in the path between R1 and R2 (in which case it can never be captured by R2).

A more feasible approach is the Mean Delay method, which solves the problem of sensitivity to packet loss and out-of-order reception. In practice, the router collects N timestamps from N packets and calculates an average by dividing the sum of these timestamps by N; in this way we get more robustness to packet loss and out-or-order packets, and it also saves disk space because the number of timestamps to send to the NMS is one per period. However, this method does not have only advantages; in fact, only one measurement is obtained for an entire period even if the duration of the block is high. We also lose the maximum, minimum and median delay, where the last is necessary if we want a statistical distribution of the delay.

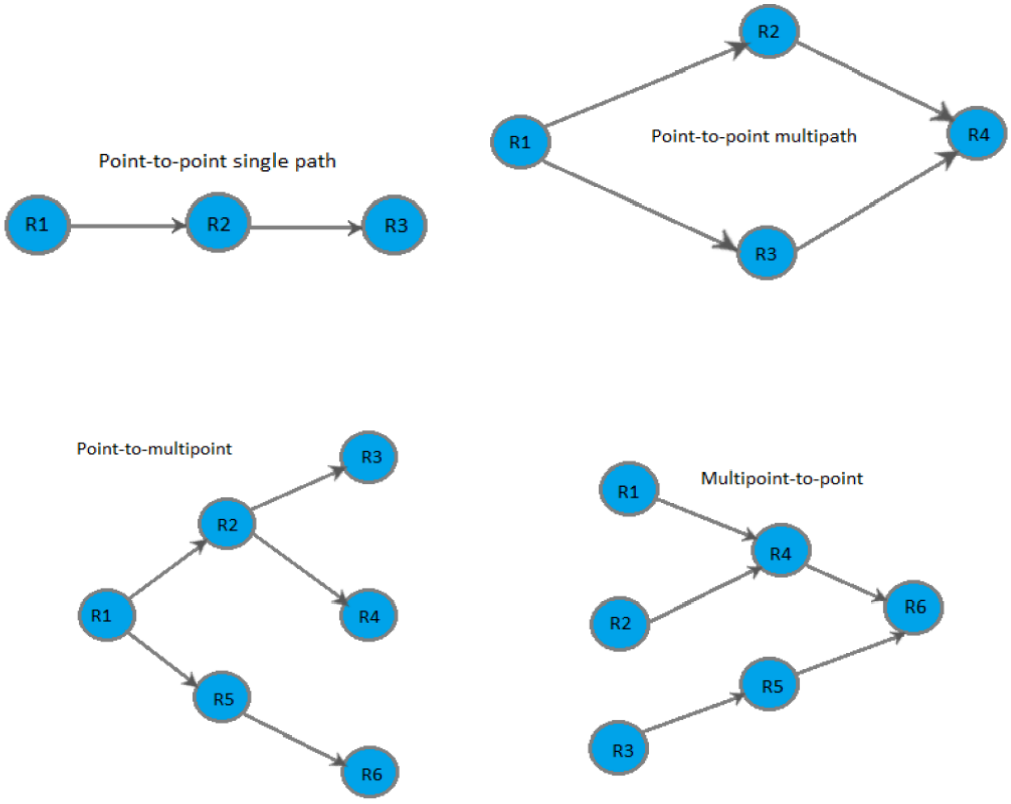
A last approach, which solves all the problems mentioned above, is based on the Double Marking method. In practice some packets are marked only once and are used to calculate the average delay, while others have a double marking to obtain other measures that give us a static distribution of the delay.

2.2 Multipoint Alternate Marking

The Alternate Marking Method as it has been described so far, is only applicable to point-to-point traffic as it is assumed that all packets captured by a node will then be captured by a single subsequent node. This may not always be true on a real network, where a packet can follow different paths due to routing mechanisms.

The Multipoint-Marking-Method [4][8] aims to extend Alternate Marking to all possible unicast flows. A unicast flow means that the packet is sent to one and only one destination, and the flow is a collection of packets having a common set of characteristics (for example same source IP and same destination IP).

All the possible paths that we can encounter in a unicast flow and which are considered in the Multipoint-Marking-Method are showed in the figure below.



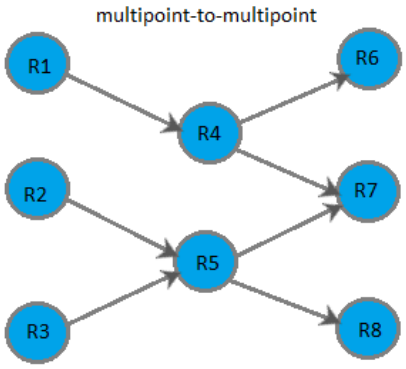


Figure 2.5 - Possible paths [6]

What has been done in this approach is to introduce the concept of cluster. A cluster is the smallest subnet that guarantees the condition that the number of incoming packets is equal to (or greater than) the number of outgoing packets. With the concept of cluster, we can monitor the network with a different degree of detail, we can analyze a large portion of the network and if packet loss occurs, we can detect where the problem occurred, with an in-depth analysis.

2.2.1 Packet loss

In a monitored network, nodes can be of three types: input nodes, output nodes, or intermediate nodes. The input nodes are those in which the traffic passes first while the output nodes are the nodes through which the traffic leaves the monitored network. Obviously, the input and output nodes, being at the edge of the monitored network, are reversible depending on the direction of the traffic. The intermediate nodes, on the other hand, are used to define what happens within the monitored network and to provide details, but they are not necessary to monitor the network unlike those of inputs and outputs.

To calculate the packet loss we consider a principle already mentioned: the number of packets counted by the input nodes must always be greater than or equal to the number of packets counted by all the output nodes.

We can therefore define the packet loss within the monitored network or within a single cluster as the difference between the number of packets counted by all the input nodes and the number of packets counted by all the output nodes, in a period:

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

where:

- n is the number of input nodes
- m is the number of output nodes
- PL is the network packet loss (number of packets lost)
- PIi is the number of packets passed through the i-th Input node in the period
- POj is the number of packets passed through the j-th Output node in the period

2.2.2 Clustering algorithm

The monitored network mentioned so far can be seen as a graph whose nodes are represented by all the network devices that are Measurement Points (MP) and the arcs are all the links that connect directly or indirectly (if cross a node that is not a MP), each MP to another. In a fully monitored network, each network device is an MP and the monitoring network corresponds to a real network. A cluster is a subnet obtained from the graph of the monitored network and which maintains the packet loss properties just seen.

The simplest algorithm for composing the smallest possible clusters, as also mentioned in the RFC8889 [4], is organized in two steps:

1. Group the arcs that share the same initial node;
2. Merge the groups with those that share at least the same end node.

Let's assume to have the following monitoring network and to apply to that the clustering algorithm.

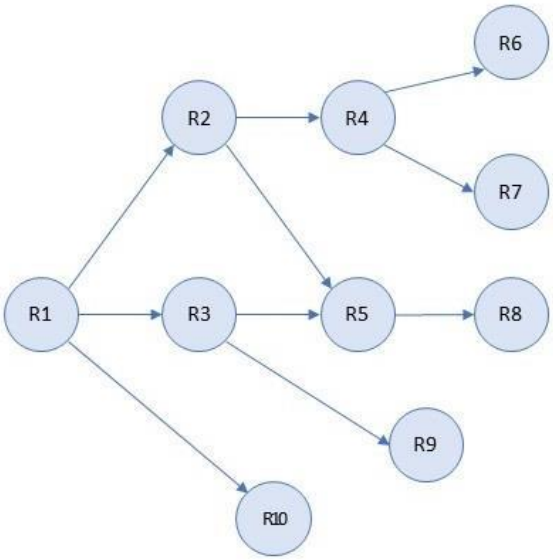


Figure 2.6 – Example of Monitored Network [6]

After the first step we will obtain 5 groups:

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

Finally, applying the second point we will get the real clusters which in this case will be the following four:

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

- Cluster 4: (R5-R8)

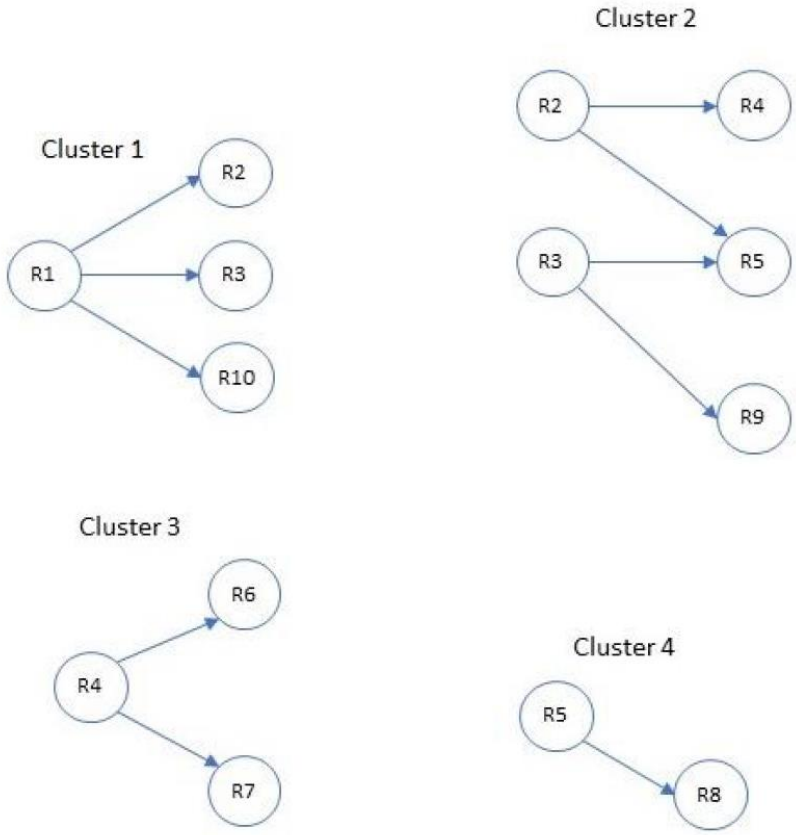


Figure 2.7 – Clusters [6]

Obviously, by joining these clusters into larger clusters, using the output nodes of one with the input nodes of another as a junction point, it is possible to obtain bigger clusters (called Super Cluster), where the packet loss equation is still true, or regain the entire monitored network.

2.2.3 Delay

The calculation of delay and jitter obviously remains of great importance. Since we are considering multipoint networks, however, a new approach must be found, as the Double Marking Method, described as the definitive solution in point-to-point networks, in multipoint networks is not applicable in the same way because packets

can take different paths and reach different network interfaces. From this need, new approaches based on hashing techniques [9][10] have been used, which allow to uniquely identify packets within the network and to select a small number, based on the hash. Subsequently, the delay will be calculated from the selected packets, using the associated timestamps.

In particular, the possible methods are the basic hash and the dynamic hash.

The basic hash consists in setting a certain number of bits of the packet hash that must be compared with the reference hash value, so that if the values are the same the packet is selected. This approach, however, leads to the capture of a very variable number of packets depending on the traffic.

The dynamic hash solves this problem and is in fact the method applied within the existing architecture. In practice, the initial number of bits n (typically a few bits, to sample a high percentage of traffic) that must be compared with the reference hash, is provided together with NMAX, that is the maximum number of packets to be captured in a marking period. A hash function is applied to each packet entering a node and the first n bits are compared with the reference hash. If the first n bits are the same, the packet is selected and a value in a counter that keeps track of how many packets have already been captured, is updated. It is called dynamic hash as the length of the hash is dynamically adapted to the amount of traffic: when NMAX + 1 packets are sampled, the length of the hash to be compared is increased by 1 bit so that, statistically, only NMAX/2 matches again to the reference hash. This mechanism is applied in a loop until the end of the period.

This method is therefore to be preferred because it converges to a predetermined number of packets, which is between NMAX/2 and NMAX, also limiting the amount of data to be stored.

Furthermore, these methods are resistant to packet loss, in fact if some packets are lost, the corresponding hash will not be present in the nodes that did not receive those packets, but all the others will be correctly coupled thanks to the hash value.

Chapter 3

The Big Data Approach

The Big Data approach [3] is designed for performance measurement based on a posteriori calculation and is based on the principles, just discussed, of Alternate marking [1], Multipoint alternate Marking Method [4][8] and Hash Sampling [9][10].

This method allows you to perform two types of measurement:

1. The per cluster approach, that allows to obtain a list of values, such as packet loss or average delay, which characterize the performance of each single cluster;
2. The end-to-end approach, that aims to collect instead the packet loss and the average delay over an entire path.

The results are calculated, as just said, not in a real-time scenario, but on request and for a certain marking period. This approach is based on packet sampling applied to all incoming traffic without flow distinction. Sampling takes place through hashing techniques, which speed up and facilitate the task of following the path of each packet in the network. The Big Data server deals with the splitting of data into flows, after having collected the identification field of each sampled packet (as well as timestamp, hash value and cluster identifier). As already mentioned, to make this mechanism feasible, the backbone network of an Internet service provider must be surrounded by routers, equipped with a running probe that collect the packets, as they are the first to handle traffic of customers. Packets must therefore be marked (via Alternate Marking [1]) outside the monitored network, as only marked traffic will be monitored (can be seen from the diagram below).

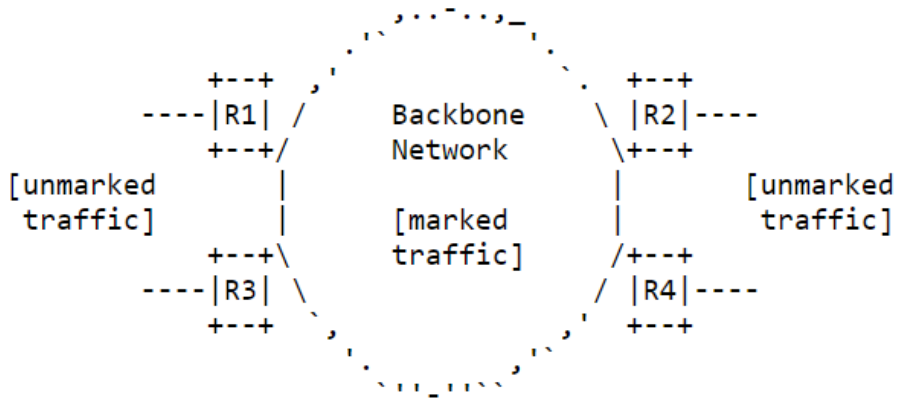


Figure 3.1 – Monitored Network Example [3]

It is possible to mark even only partially the traffic and the results will not be affected by the unmarked packets and will refer only to the marked ones. The marking can be done both by the customers devices and by the border routers themselves, remembering however that the markers must be synchronized.

3.1 The Working Principles

The method consists of several stages:

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

The following diagram is representative:

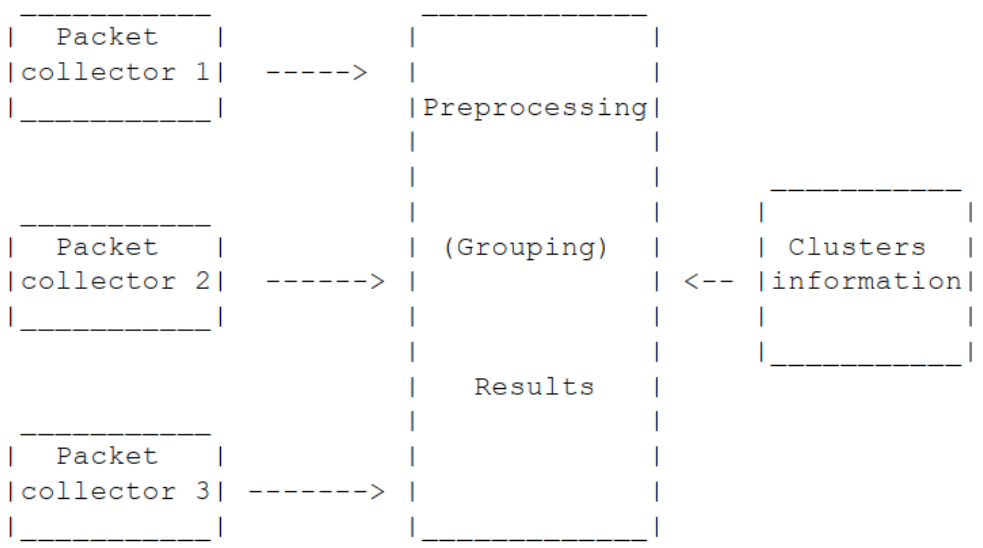


Figure 3.2 - Big Data method scheme [3]

The probes must be placed in each router that we want to monitor in order to analyze the data that passes through the monitored interfaces. In fact, during the configuration phase, it is necessary to pass to the program a series of parameters such as:

- the set of interfaces to be monitored;
- the reference hash;
- the maximum number of packets to be stored;
- the duration of the period;
- the two values that identify the marked traffic

As regards the flows to be monitored, it is possible to monitor all flows without distinction. The packet collector only checks if the packet is consistent with the filters and if its hash value matches the reference hash, and if so, stores it. If the number of stored packets reaches the maximum NMAX, the number of bits to be matched is increased by one and a variable number of packets is discarded (this number is statistically approximately NMAX / 2).

The probes can send two different types of data to the management system:

1. Detailed packet data including fields identifying the stream, packet hash value, timestamp, and period.
2. Aggregated data on the period including, for each interface, the interface ID, the total packets counted, the packets captured by the hash system, average timestamp calculated on all the timestamps of the packets passed for the interface and the period.

At this point, after the probes have sent the data to the management system, it is useful to have a preprocessing phase to produce, from the input records, new data ready to be effectively processed and analyzed more easily to obtain performance parameters. In addition to this, a further advantage is the decrease in the total amount of data to be stored.

In this phase it is possible, by aggregating the incoming data from all the devices, to calculate the path followed by each sampled packet; this is possible by grouping the records by hash and sorting them by timestamp. The network management system must also know the topology of the network and the nodes that make up each cluster, so that it can determine the clusters traversed by packets by comparing the interface id with the nodes belonging to clusters. This is possible thanks to the cluster algorithm already implemented in a previous work [24] at Politecnico di Torino, that given the topology of a network and the desired node to monitor, returns a file with all the clusters and nodes that belong to them.

The preprocessed records are stored on Big Data servers and can be queried as needed, to obtain desired results.

3.2 Achievable results

The results are provided by querying the storage system and giving as input the identifying parameters of the flow that we want to analyze plus the identifier of the required time period.

In addition to measuring the packet loss through the formula that considers all the input and output nodes seen in the previous chapter

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

another obtainable result is the average delay of cluster D_i (referred to cluster i).

This is calculated as the sum of the delay of each d_j record (relative to j record), which is the difference between the output timestamp (when the packet left the cluster) and the input timestamp (when the packet entered the cluster). The result is then divided by the number of records belonging to the same cluster, obtaining:

$D_i = [d_0 + d_1 + ... + d_ (N_i - 1)] / N_i$

where:

- D_i is the delay associated with cluster i,
- d_j the delay associated with record j
- N_i the number of records captured in cluster i.

It is also possible to calculate the end-to-end mean delay, AD, as the sum of all delays belonging to all records that pass from the two considered points, divided by the total number of records:

$AD = [ad_0 + ad_1 + ... + ad_ (M - 1)] / M$

where:

- AD is the average end-to-end delay,
- ad_j the delay for record j
- M is the number of all records.

Other possible values that can be calculated, comparing all the timestamps relating to all the records, are the min / max / avg delay of a link.

Chapter 4

Starting Architecture

The drafting of the Big Data approach just seen is the continuation of a first model conceived and built in a thesis work [6] at the Politecnico di Torino. This first architecture was designed to test and prove the functioning of the Alternate Marking Method applied to Multipoint Measurements with a subsequent post processing phase based on Big Data.

The architecture at issue consists of the following components:

- Mininet [11]: a network emulation software that creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux networking software and its switches support OpenFlow [12] for highly flexible custom routing [13]. It was used to simulate a network (more realistic than a laboratory network) whose traffic will be monitored.
- Probe: a software called IOVisor-PNPM [5] (IOVisor Packet Network Performance Monitoring) based on eBPF [14], as already mentioned, it is the first probe developed by the collaboration between Politecnico di Torino and TIM, and was used to collect the data from marked traffic generated by iperf (software for the active available bandwidth for IP networks) [15].
- Apache Flume: a distributed, reliable service for efficiently collecting, aggregating and moving large amounts of log data. It has a simple and flexible architecture based on streaming data flows. It is robust and fault tolerant with tunable reliability mechanisms and many failover and recovery mechanisms [16]. It is used in two instances, to send data captured by the probe directly to the HDFS cluster.
- HDFS: Hadoop Distributed File System (HDFS) is a distributed file system designed to run on common hardware. It has many similarities to existing

distributed file systems, with some significant differences. HDFS is highly fault tolerant and is designed to be deployed on low-cost hardware. It enables high data throughput and is designed to store and process huge amounts of data [17]. It can also scale horizontally by adding new clusters or new nodes to existing clusters. In this architecture it is clearly the file system where all the collected data will end up.

- Spark: the framework that provides a faster alternative to the MapReduce paradigm for querying in a cluster environment. Not only is it up to 100x faster than MapReduce, but it includes some key features like data parallelism and fault tolerance. It was originally developed at the University of California, the Berkeley AMPLab. Then, the Spark code was taken over by the Apache Software Foundation, which currently maintains it [18]. Used to obtain significant data for the actual network analysis.

4.1 Implementation

The entire model was implemented using two virtual machines, logically separating the two main parts that make up this architecture:

1. the network emulation part with relative data collection
2. the data storage part with related processing

This choice allows you to have two machines that are independent of each other, favoring the modularity of the project and therefore a possible updating of the latter. The two machines are organized as shown in the next figure.

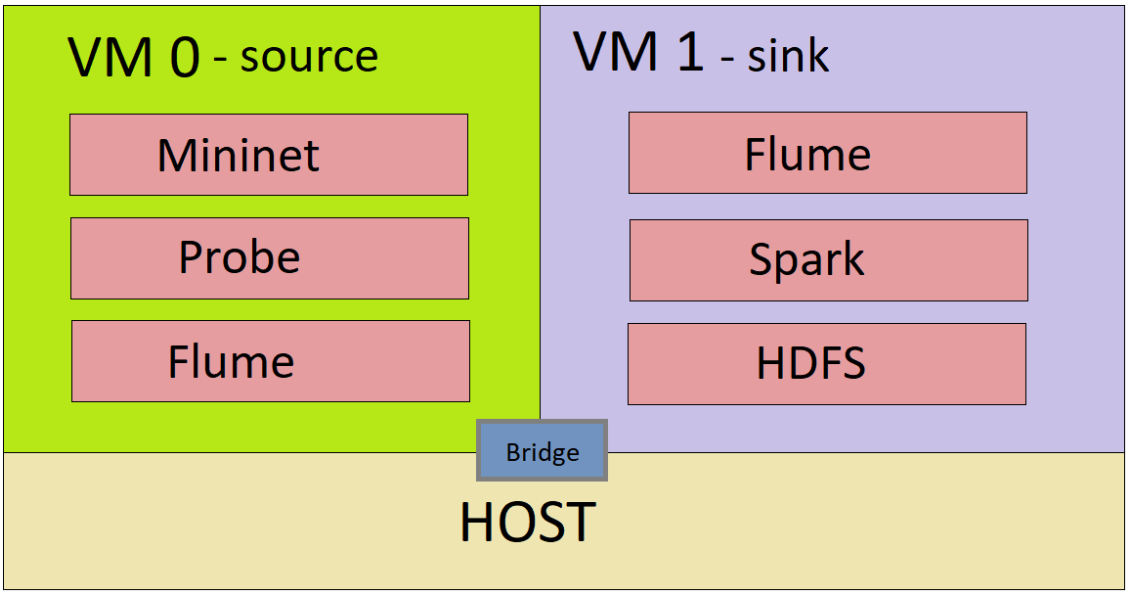


Figure 4.1 - Starting architecture [6]

In practice we have VM0, also called source, and VM1, called sink, both based on Ubuntu 18.04 LTS. In VM0 we have Mininet which deals with the emulation of a network, precisely the GEANT2012 (chosen from the Internet Topology Zoo [19]), in which traffic is generated via iperf. The routing of traffic on this network is managed through the RIP [20] protocol, in conjunction with POX [21], the Open Flow controller, that manages the traffic on the switches. This solution, required to ensure that the routers find by themselves where to direct the packets, is however a limit since, if in reality the final version of the RIP protocol is already obsolete, and more efficient protocols such as IS-IS or OSPF are preferable, the version of RIP that we managed to integrate into the routers of the network emulated by Mininet, is a simplistic version where some known problems, such as split horizon, are not solved. This in fact conditioned the choice of the network to emulate, namely the GEANT0212, as the latter has only 40 routers and is more easily manageable by this implementation of the RIP.

The traffic on this emulated network is then captured by the probe [5] running on VM0 which, through the dynamic hash mechanisms, mentioned in the previous chapters, captures the statistics of the packets and stores them within files with

name PERIOD_INTERFACE in a folder on the file system. So for what concerns the emulated network we will have a situation as represented below.

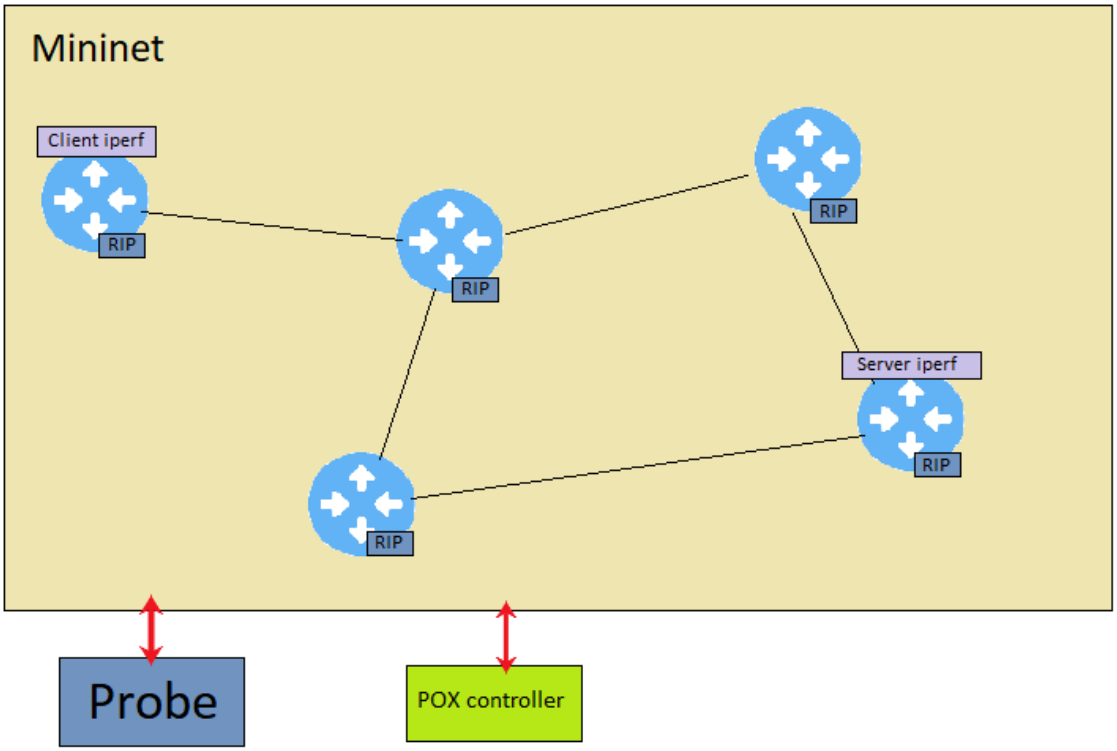


Figure 4.2 [6] - Example of Network emulation on VM0

The choice to save the data in a folder is justified by the fact that Flume, among other options, can send the data present in a specific location on the file system. In essence, we have an instance of Flume that checks every 30 seconds, that is the duration of the period chosen for the probe, in this directory and if there are new files, sends them. At this point to make the two VMs communicate has been used the bridge, a setting available in Oracle Virtual Box that allows a virtual machine to obtain an IP address reachable on the local network.

On the other hand, in VM1 we have another instance of Flume which in this case is configured to receive data on an IP address (the one assigned to the VM from the bridge). Given Flume's strong integration with HDFS, it can save received data directly to HDFS servers running on VM1. The HDFS server is implemented through docker [22], an open-source platform for building, deploying and managing

containerized applications, in order to host a Hadoop cluster with three Data Nodes [23]. Within the HDFS servers, it will be possible to perform spark jobs that will pre-process the data in first place to make them easier to manipulate, and then extract the information necessary to understand the progress of the network.

In a possible real implementation, as thought by the colleague who built this model, we would have as many instances of the probe as there are interfaces to monitor and a flume instance, both running on the routers that we want to monitor in the network. Another instance of flume would then run on the HDFS server where the data is actually collected, as showed below.

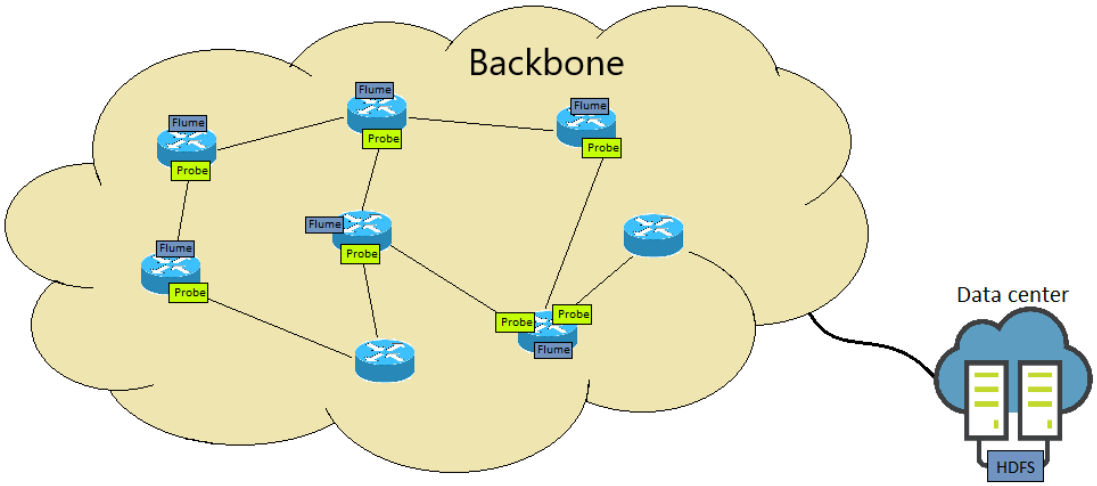


Figure 4.3 – Possible real architecture [6]

4.2 Possible Improvements

This model brings with it a series of problems and therefore of possible improvements, both as regards the emulation part, and as regards a subsequent real implementation.

The component that first of all in this model certainly needs to be updated is the probe. In fact, the probe used was, as already mentioned, the first born from the collaboration between TIM and Politecnico di Torino and although based on Alternate Marking, it was not designed to be used in this scenario. In particular, the probe as it has been programmed cannot filter traffic on virtual interfaces (and therefore on those emulated by Mininet). It has, for this reason, been modified to capture traffic only from the interfaces that you actually want to monitor, modifying the python part, resulting in a deterioration in performance. In fact, since at the kernel level the traffic will be captured from all the interfaces and only at the user level this will be filtered, there is obviously a waste of time and resources. However, the feature for which it was really necessary to have a new probe is the fact that the probe used in this model captures only incoming traffic, while it would be appropriate to also capture the output one to determine more accurately, for instance, where a packet loss occurs (whether on the link or in the router). This feature has also led to a particular configuration of the Mininet network, as shown below.

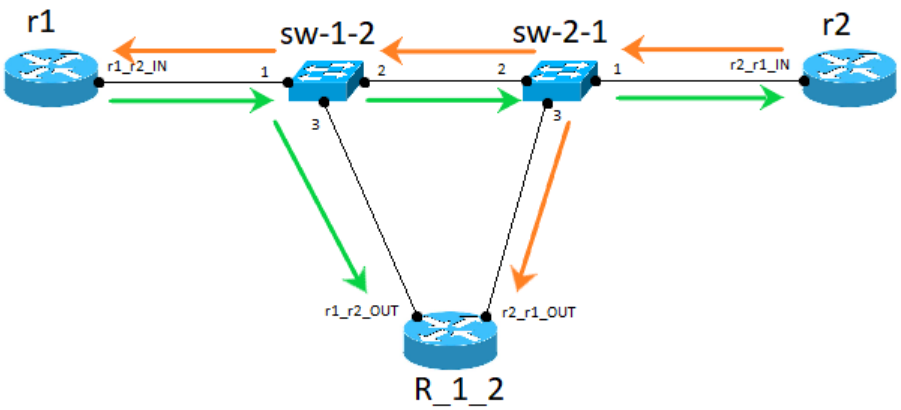


Figure 4.4 [6] - Package path

In practice, to connect two routers, R1 and R2, two switches were used that pass traffic from R1 to R2 and vice versa. In addition to this, however, the traffic entering the switches from port 1 (from R1 for switch 1-2 and from R2 for switch 2-1) is sent to port 3 where a router has been specifically connected to intercept outgoing traffic

from R1 and R2. In fact, since this router will only have incoming traffic, and having purposely named its interfaces as if they were the output ones of R1 and R2 routers, the probe used will function as if it were able to intercept the outgoing packets from R1 and R2.

Other possible improvements concern the sending and saving of the data collected by the probe. In fact, although the used architecture is functional, it is not optimal for various reasons:

1. In a realistic case, saving the data in a directory, asking in practice to a router to access its file system, is a quite unusual choice as it can lead to delays due to access to memory and does not take into account of the actual memory available on the router (which usually have limited capacity).
2. For how it has been configured, sending data through Flume does not allow to have a good level of synchronization as the Flume instance running on the router checks the directory at issue every 30 seconds and not when actually new data are ready. Therefore, a change in the duration of the period would also require a modification to Flume, wanting to respect the idea that between one check and another there is a wait as long as a period. Obviously, from this situation also derives the obligation of having an instance of Flume that runs on the routers (and therefore having installed java on the latter).
3. As regards the Flume instance on the sink machine, or on the HDFS servers considering a real case, this has no way to differentiate the aggregated data on a period from the specific data of the single interfaces (hence the choice in the simulation not to send at all aggregate data).
4. Furthermore, given the need for preprocessing the data, in order to be able to manipulate them in a simpler way at a later stage, the condition whereby the data are somehow duplicated on the Big Data servers is created. In fact, in addition to the raw data that are saved directly from Flume on HDFS, there will be also the data preprocessed, effectively creating a duplication and therefore inefficiency from the memory point of view.

In addition to all this, there are other possibilities for improvements (some of which are also mentioned in the original work [6]), like: enhancing the Spark Jobs, improving the way in which they are executed (since now we have to run a series of scripts and you could instead think of a REST interface) and the possibility of creating a graphical interface where view in a more readable form the information extracted from the data.

In this paper, my task was to solve some of the reported problems, replacing the probe, with the relative modifications to the emulated model, and finding a more efficient method of sending the data to the HDFS server.

Chapter 5

Technologies

The development of this thesis required the use of different technologies, some already used in the previous work, others totally new. Therefore, in addition to the use of software already seen as Mininet, the HDFS and Flume, although all configured differently, as we will see below, the main innovations are to be found in the use of a new probe [7], born from a further collaboration between TIM and the Politecnico di Torino, and Apache Kafka [27], both detailed below.

5.1 Probe

The development of the probe at issue was carried out through BCC [25], a toolkit developed by IOVisor, an open source project formed by a community of developers to innovate in the field of security and networking, aimed at simplifying the writing, validation and compilation of code eBPF [14].

5.1.1 eBPF

The new probe, in fact, like the previous one, is based on BPF [26] (Berkley Packet Filter). Developed in the late 80's, BPF was designed to make packet filters. In particular, the peculiarity was to do packet filtering at the kernel level thus avoiding copies of packets in the user space, a very inefficient practice, and thus improving performance. Operation in kernel mode was possible thanks to a new pseudo machine language that could be injected directly into the kernel.

Subsequently in recent years BPF has been improved leading to a new version, eBPF (extended BPF) which is the one on which the probe is actually based.

This new version allows you to inject code at runtime so you can create and execute code in the kernel at any time, moreover it is no longer necessarily tied to packet filtering events as the code in eBPF can be hooked to any kernel event. It can then be executed as soon as this event occurs.

The programming language used in this new version is a restricted C (to avoid malicious code and ensure the security of the CPU)

A further addition to the classic version (essential for the implementation of the probe) are the Maps. Maps are data structures, organized in key-value pairs, which reside in the kernel space and can be accessed by different BPF programs but also by programs in the user space. This allows for an exchange of data between the kernel and the user space.

To access Maps an eBPF program must use a helper. Helpers are a set of functions, available in the Linux kernel, which allow you to use some library functions of the operating system, bypassing the limitations given by eBPF. These functions act as a proxy between the BPF code and the kernel. Basically, a helper calls functions that could not be executed or called directly from a BPF program. However, only a subset of the operating system functions is accessible through the helpers, and adding new ones requires a long consultation process within the Linux community.

5.1.2 Overall architecture

The probe [7] is made up of two parts:

- the part executed in user space
- the part executed in kernel space.

The frontend, that is the part in user space, is written in python and is accessible from a REST interface that allows you to configure it or give commands (e.g. start / stop). In particular, the frontend is composed of two parts: a PNPM manager (which, as we will see, will be the one most subject to changes) and a BPF manager.

The PNPM Manager interacts with the NMS via the REST interface. The NMS can configure the probe as needed by sending a configuration file in json format that contains all the parameters necessary for the configuration, from the duration of the period to the maximum number of packets to be captured. The PNPM Manager also interacts with the BPF manager by passing the configuration parameters received from the NMS.

The BPF manager is connected to the PNPM manager from which it receives the parameters to enable the execution of the BPF program; on the other hand, it interacts with the eBPF program in kernel space. For example, when the NMS sends the start command to the PNPM manager, in chain the BPF injects the eBPF program into the kernel by connecting it to the desired hook point.

On the kernel side, we have the eBPF code. Its execution on the kernel side is, as mentioned, the main advantage, as the packet processing speed is really high compared to user space programs, thanks to the absence of context switch that slow down the execution.

On the kernel side we obviously also find the NIC (Network Interface Card), the physical component that manages the sending and receiving of packets and the placing of the latter within the network stack. Each physical interface is split into two logical interfaces (one that receives packets with the suffix _IN, and another that sends packets with the suffix _OUT).

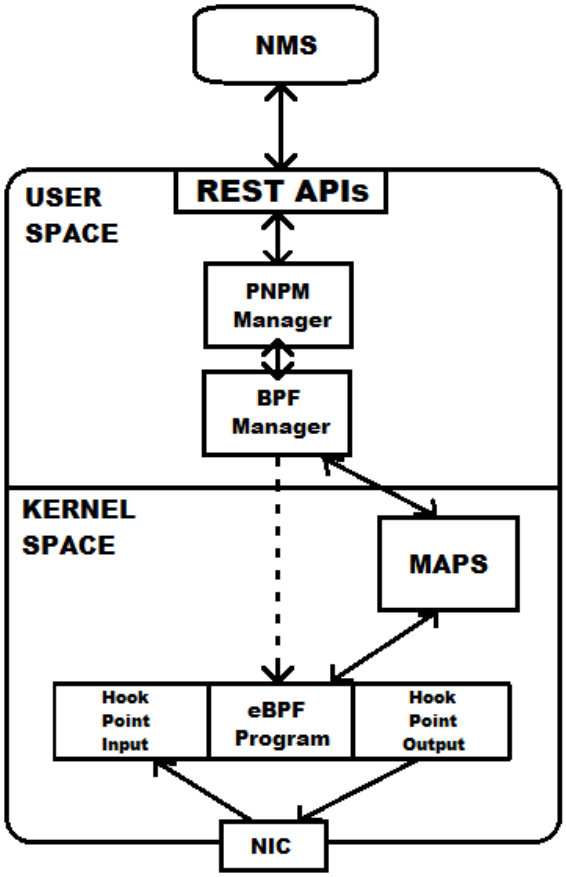


Figure 5.1 [7] – Overall probe architecture

The eBPF program is connected to two Hook Points. A Hook point is a kernel event that calls a function when the above event occurs. The probe is configured on two hook points. In fact, the functions are called both when a packet arrives in the kernel network stack (which the previous probe already did) but also when a packet leaves the stack (in order to also capture the output traffic).

The program analyzes the packet fields to check if it is compatible with the filters set (for example if it comes from a monitored interface or not) and then, since the selection of the packets to monitor is based on dynamic hash, calculates the hash, discarding or saving the packet depending on whether the hash matches the reference one. At this point it stores all the information inside the maps.

Maps, as mentioned, are data structures in memory, introduced by eBPF, necessary to interact from the kernel to the user space and vice versa. In this context, the probe saves the data of the sampled packets in the maps and at the end of the period, the PNPM Manager reads them and collects the aggregated values in a data structure, while the values of the individual sampled packets are saved in files divided by period.

5.1.3 Possible Configurations

The probe has been designed to work in two marking modes.

Using two bits, alternately 0 or 1, if the traffic is marked externally (with the possibility that not all flows are marked) so that routers can understand which packets need to be scanned. The choice of bits was made thinking of the less used fields of the IP header. The two least significant bits of DSCP (Differentiated Services Code Point) have been chosen for this purpose.

Since some service providers can use the DSCP field to guarantee a quality of service to their customers, an alternative solution has also been thought to avoid using that field.

There is a bit in the IP header that is useless, it is called the unused bit or the evil bit and it corresponds to bit 0 of the flags field. Using a single bit to mark traffic leads to a number of trade-offs. In fact, since the reported traffic can be confused with unmarked traffic, the principle that guarantees the functioning of this method is that all the traffic that enters the network must be marked and that the probes can only contain a filter to discard the traffic generated by the internal routers (routing protocols, protocol configuration, etc.). This filter is based on the IP subnet of the backbone.

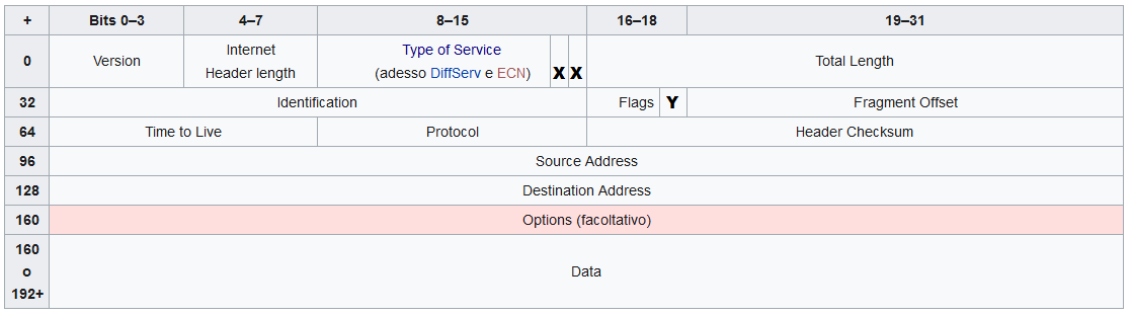


Figure 5.2 [7] – Used fields to mark packets (X first case, Y second case)

To choose which of the two methods to use, just enter in the configuration file a subnet from which to discard the packets, and in this case the second method will be chosen.

Another way is to explicitly define the marking values. Basically, if you want to use the first method, it will be defined as an initial value 1 and a subsequent value 2 (that are the values of the DSPC field). Conversely, if you want to use the second method, just define 0 and 1 as initial and subsequent values (that are the values of the unused bit). During my tests I always used the first mode.

5.1.4 Differences with previous probe

The reasons that led to the development of this new implementation are many and coincide with the differences between the new probe [7] and the previous one [5].

In particular, the main difference is that the new probe can, as already mentioned, also capture the output traffic from a router allowing better traffic monitoring, something that the previous one could not do.

Further differences are to be found in the data collected. The new probe, in fact, for each captured packet, in addition to the fields collected by the previous probe (ip source and destination, source and destination port, protocol, hash, timestamp, departure and arrival cluster and period), keeps track of the number of bits matched with the hash, and the “color” (so 1 or 2 for example) of the packet. Furthermore, as

regards the aggregated data, it provides (in addition to the color, total number of packets, number of captured packets, length of the hash reached, and initial timestamp) also the average timestamp of the period.

Last analysis must be made with regard to performance. The new probe allows filtering directly at the kernel level and this generally guarantees superior performance compared to the previous probe which for our use required filtering at the user level, a practice that led to context switches that certainly affected performance.

As can be seen from the development work of the probe [7], and comparing the data with the previous one [5], it can be seen that if the improvement in terms of maximum data speed at which the probe can work without packet loss is marginal in single core, in multi core there has been a considerable improvement, reaching 10Gbit without loss already with 700byte packets, and in any case significantly improving performance with smaller packets.

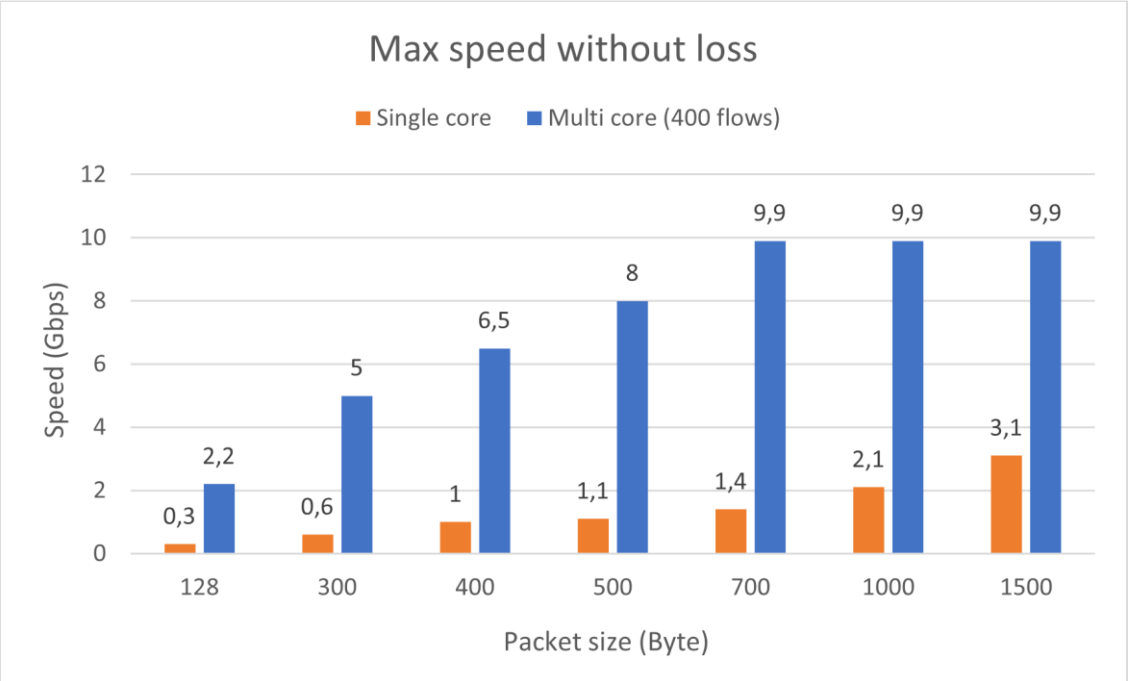


Figure 5.3 [7] – Max speed without loss in the new probe

5.2 Kafka

Among various possible options, examined during my thesis work, to replace Flume in the exchange of messages, or at least solve the problems mentioned in the previous chapter, Kafka, unlike other software such as Mosquitto broker or 0MQ, was the one that convinced me the most and in which I saw the greatest potential.

Apache Kafka is an open-source flow processing software platform, initially created by LinkedIn and later left in open source to the Apache Software Foundation, which then developed it. Written in Scala and Java, the project aims to provide a unified, high-speed, low-latency platform for handling real-time data feeds [27].

5.2.1 Functioning

Kafka is a distributed system consisting of servers and clients that communicate via a high-performance TCP network protocol. Its operation is based on a message queue managed with an extremely scalable publish/subscribe pattern, which makes it extremely valuable to process streaming data.

For what concern the server side, Kafka runs as a cluster of one or more servers that can extend on multiple datacenters or cloud regions. In other words, multiple server instances can run on multiple machines and form a cluster. Some of these servers form the storage tier, called the broker. In order to support mission-critical use cases, Kafka clusters are highly scalable and fault tolerant. In fact, if one server fails (for example, a disk failure), the other servers will take care of its work to ensure continuous operations without any data loss. In this context Kafka is supported by Zookeeper.

Zookeeper "is top-level software developed by Apache that acts as a centralized service and is used to maintain naming and configuration data and to provide flexible and robust synchronization within distributed systems" [28]. Zookeeper keeps track of the status of cluster nodes, topics and partitions. It is essentially the brain that allows the various brokers to be synchronized with each other and allows

continuous operation even in the event of failures. Zookeeper also keeps track of the configuration related to all topics, including the list of existing topics, the number of partitions for each topic, the location of all replicas, which node is the preferred leader, etc. Access control lists for all topics are also maintained within Zookeeper, and it also maintains a list of all brokers that are operating at any given time and are part of the cluster.

On the client side we have distributed applications and microservices that read, write and process event streams in parallel, on a large scale and with fault tolerance even in the event of network problems or machine failures. Kafka comes with some of these included clients, which are augmented by dozens of clients provided by the Kafka community: clients are available for Java and Scala, for Python, C / C ++, and many other programming languages.

The clients are divided into:

- producers, those who write / send data to Kafka servers
- consumers, those who sign (read and process) these data.

Data on Kafka is saved in the form of events. Conceptually, an event has a key, a value, a timestamp, and optional metadata headers.

In Kafka, producers and consumers are completely decoupled and independent of each other, which is a key design element for achieving high scalability. An example of this is the fact that producers never have to wait for consumers, but can continue to publish events even if at a given time there is no active consumer. Kafka offers synchronization and guarantee mechanisms which, for example, allow a consumer to process events exactly once, or give certainty to a producer that an event has actually been received and stored by the Kafka brokers. In the latter case, again by configuring the clients and servers, you can decide whether:

- have no acknowledgment mechanism
- have an asynchronous mechanism, where the producer waits for at least one broker to have actually received and stored the message, ignoring whether the copy to other brokers is successful or not

- have a synchronous mechanism, where the producer waits to receive an acknowledgment from all brokers (assuming a replication factor of 3 means receiving a message from all 3 brokers).

Although the last method is the one that guarantees the greatest safety in terms of data storage, a good compromise between speed and security remains the asynchronous mechanism.

The events are organized and archived in a lasting way in topic. Simplifying, topics are similar to a folder in a filesystem, with events representing the files in that folder. Kafka topics can always be multi-producer and multi-subscriber. A topic can have zero, one, or many producers writing events, as well as zero, one, or many consumers listening. The events in a topic can be read as often as necessary because, unlike traditional messaging systems, the events are not necessarily deleted after consumption. In fact, it is possible to define how long Kafka must keep the data or even a memory threshold after which start deleting the oldest data. Kafka's performance is actually constant relative to the size of the data, so storing data for a long time isn't a factor that degrades performance significantly.

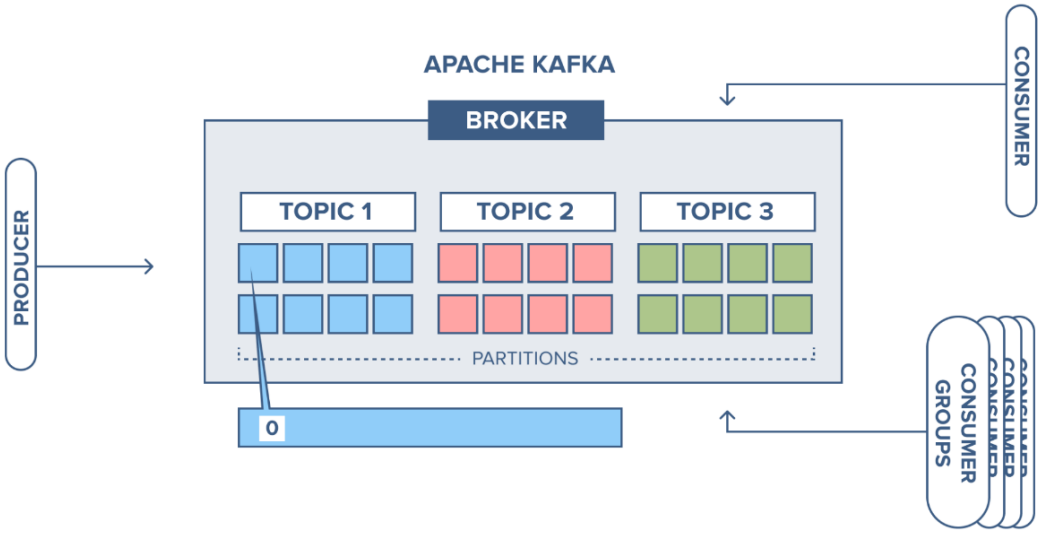


Figure 5.4 [27] – Kafka Example Model

As we can see from the previous image, topics are partitioned, which means that a topic is spread across a number of "buckets" located on different Kafka brokers. This distributed data placement is very important for scalability because it allows client

applications to read and write data from / to multiple brokers at the same time. When a new event is published on a topic, it is actually added to one of the latter's partitions. Events with the same key are written to the same partition and Kafka guarantees that any consumer will always read the events from that partition in the exactly same order they were written.

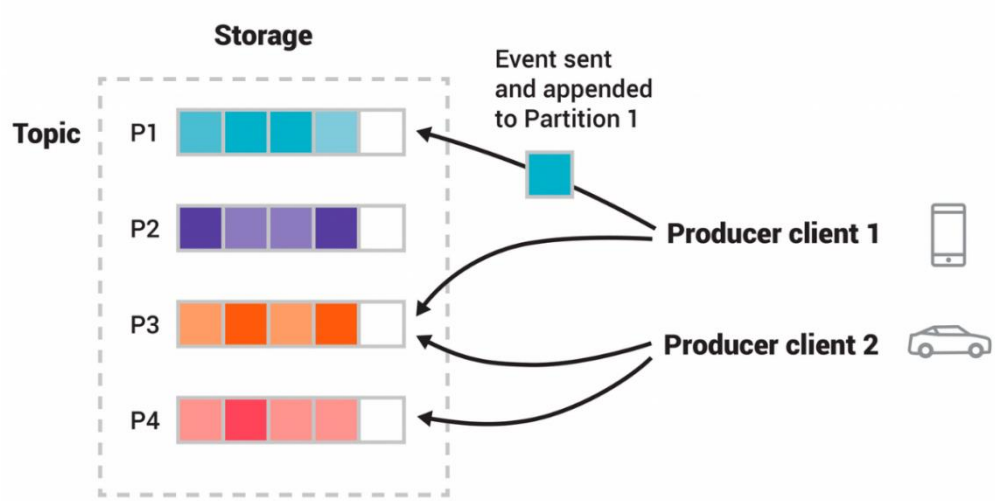


Figure 5.5 [27] – Topic Partitions

Looking at the figure we have a topic with four partitions P1 – P4. Two different producer clients are publishing, independently of each other, new events on the topic. Events with the same key (indicated by color in the figure) are written to the same partition. Note that both producers can write to the same partition.

To make the data fault-tolerant and highly available, every topic can be replicated, even between geo-regions or data centers, so that there are more and more brokers who have a copy of the data in case a problem is encountered in a broker or you want to do maintenance. A replication factor often suggested, and which I applied within the model that we will see below, is a replication factor equal to 3, meaning there will always be three copies of the data.

5.2.2 Main use cases

In the light of its characteristics, the uses that can be made of Kafka can be many, and among the main ones we can find:

- Messaging: compared to most messaging systems, Kafka has better throughput, integrated partitioning, replication, and fault tolerance making it a good solution for large-scale message processing applications.
- Activity Monitoring: Kafka can rebuild a user activity monitoring pipeline as a set of publish-subscribe feeds in real time.
- Metrics: Kafka is often used for operational monitoring data, leading to the aggregation of statistics from distributed applications to produce centralized feeds of operational data.
- Log Aggregation: Collect physical log files from servers and place them in a central location (a file server or perhaps HDFS) for processing. Compared to log-centric systems like Flume, Kafka offers equally good performance, but with greater guarantees thanks to data replication and much lower end-to-end latency.

The use that was made of Kafka within the project, follows, as we will see, a mix of the cases mentioned above.

Chapter 6

New Architecture

In this chapter we will explore the new simulation architecture that aims to replicate as much as possible what would happen in a real network. This simulated model serves to lay the foundations for what will be a possible real model of performance monitoring through Alternate Marking Method and Big Data approach.

6.1 Mininet Configuration

Before looking at the architecture in detail, I focus on the configuration of the Mininet network. As mentioned in the previous chapters, and as can be seen from the work done by those who preceded me [6], the configuration of the simulated network with Mininet was designed ad hoc to use the old probe [5], as the latter did not provide the capture of the outgoing traffic.

First of all, therefore, I dedicated myself to modifying the configuration of the Mininet network to make it work with the new probe which instead also captures the output traffic.

The problem to be solved basically lies in the addition of a third node connected to the two switches between two network routers, as shown in the figure 4.4. This additional node is therefore superfluous and would create confusion in packet capture. For this reason, the first necessary change was to eliminate this additional node by modifying the net2switchRandomLoss.py, the file in which the network was created. The rest of the file, such as the creation method based on reading a graphml file, and how the ip addresses are assigned to the nodes has remained unchanged

compared to the previous implementation. This led to the creation of a new file named net2switchRandomLossNew.py, which differs from the previous one only for the modification mentioned. The same goes for the Pox controller of the switches. In this case also, a minor change has been made, in order to forward traffic to the switches correctly, as these no longer have 3 ports, as before, but 2. This change can be found in the controller2switchNew.py file. The changes made have led to a configuration of this type:

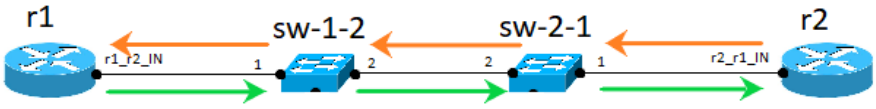


Figure 6.1 – Updated Mininet Configuration

Desired choice was to leave two switches anyway as in this way I could only use the middle link to set the loss and delay between the two routers.

At this point, however, the use of the new probe has raised a further problem. In fact, the new probe works in a way that doesn't capture the outgoing traffic from the node that generates that traffic. To be clear, if I have 2 nodes R1 and R2, and R1 sends traffic to R2, the probe only captures the traffic entering R2 but not the one leaving R1. This happens because R1 is the generator of the traffic. Clearly if connected to R1 we imagine a third node R3 that wants to send traffic to R2 passing through R1, in this case the outgoing traffic from R1 will be captured without problems, because it will not be the one generating the traffic. This behavior is due to the hook point at which the probe for the output traffic is hooked, which probably does not see the packets generated by the device itself (probably because they are generated at a lower level, e.g. by the NIC directly). However this is not to be considered a defect because thinking about the use of the probe in a real context, clearly the traffic will almost never be generated by the routers inside the monitored network, but will come from external users.

This behavior has therefore created the need to add external nodes from which to start the traffic to not reduce the monitored network, which is already not excessively extended. In fact, the network emulated by Mininet is still the GEANT2012, a choice motivated, as before, by the limited routing capacity of the RIP algorithm that runs on the emulated hosts.

The choice of which and how many nodes to add was based for continuity on the previous work. In that case, in fact, all the tests had been done by analyzing three flows in particular, which are highlighted below.

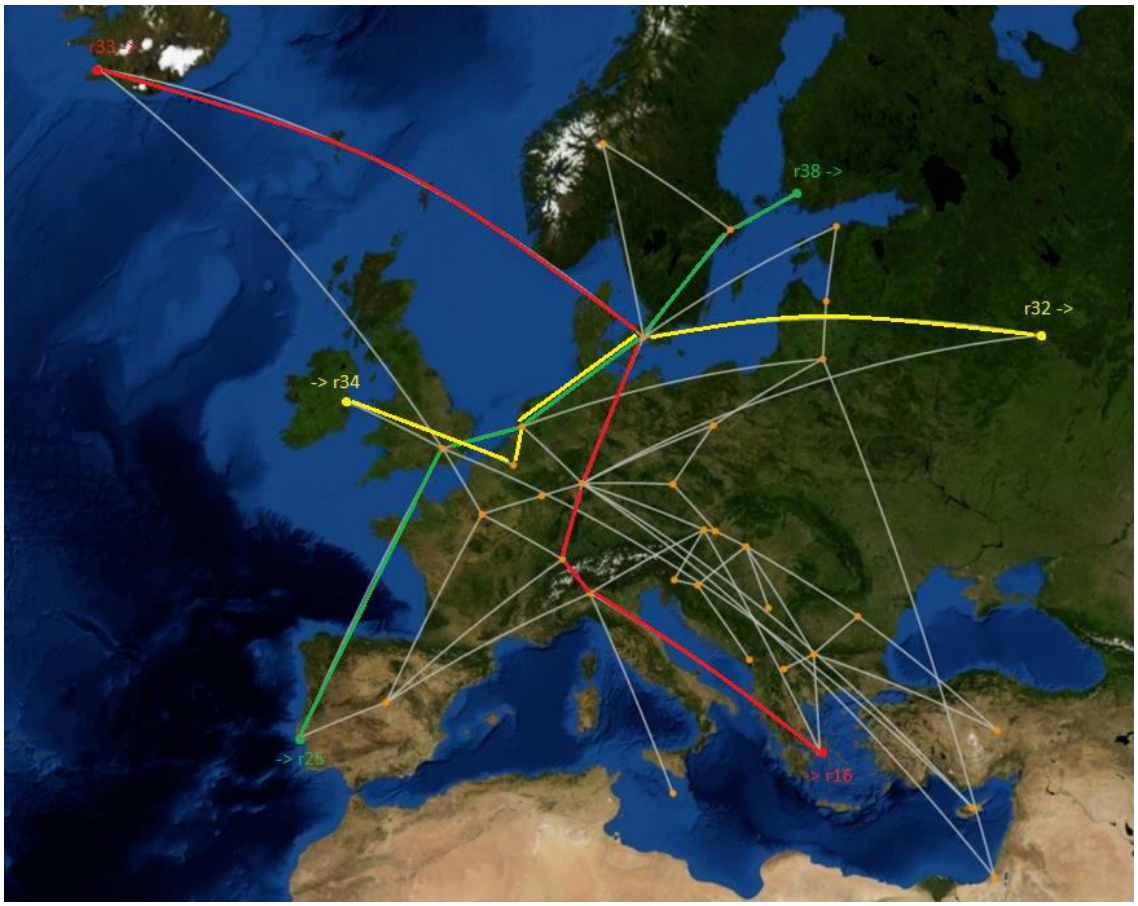


Figure 6.2 – Analyzed Flows

In fact, we can see the flow highlighted in green from R38 to R25, the one in yellow from R32 to R34, and the one in red from R33 to R16. To continue working on these

flows, three nodes have been added, respectively called Ext1, Ext2 and Ext3, respectively connected to R38, R32 and R33, as we can see in the next picture.

Figure 6.3 – External nodes

From these nodes, from this moment on, the traffic will start, considering them external to the monitored network. The addition of these three nodes was carried out by modifying the GEANT2012.graphml file.

In particular, a new GEANT2012MOD+3.graphml file, copy of the original, was created, where the three nodes and the respective three links that connect them to the three routers R38, R32, R33 have been added, as we can see in the next figure.

```
<node id="100">
  <data key="d28">1</data>
  <data key="d29">56.946</data>
  <data key="d30">External1</data>
  <data key="d31">100</data>
  <data key="d32">24.10589</data>
  <data key="d33">E1</data>
</node>
<node id="101">
  <data key="d28">1</data>
  <data key="d29">56.946</data>
  <data key="d30">External2</data>
  <data key="d31">101</data>
  <data key="d32">24.10589</data>
  <data key="d33">E2</data>
</node>
<node id="102">
  <data key="d28">1</data>
  <data key="d29">56.946</data>
  <data key="d30">External3</data>
  <data key="d31">102</data>
  <data key="d32">24.10589</data>
  <data key="d33">E3</data>
</node>
```

```
<edge source="37" target="100">
  <data key="d37">e2</data>
  <data key="d38">0</data>
</edge>
<edge source="32" target="102">
  <data key="d37">e2</data>
  <data key="d38">0</data>
</edge>
<edge source="31" target="101">
  <data key="d37">e2</data>
  <data key="d38">0</data>
</edge>
```

Figure 6.4 – Modified code in GEAN2012.graphml

In this way, passing the modified file to the net2switchRandomLossNew.py script, nodes, links and the RIP routing protocol on those nodes will be automatically added, creating the emulated network without further modifications.

6.2 Implementation

At this point, having solved the compatibility problems between the previous Mininet network and the new probe, we can move on to what is the new architecture. The fundamental change is to be found in the way the probe sends the collected data. As already explained, until now the system expected the probe to write into a folder on the file system. At this point, an instance of Flume periodically checked the directory to send, to another instance of Flume running on the server, the data which was then permanently stored on HDFS to be finally processed.

Therefore, to prevent the probe from writing to the file system and to guarantee

- a higher level of synchronization,
- safety in case of failures,
- to be able to send information on aggregated data to the server
- to avoid the problem of data duplication due to preprocessing,

the choice inevitably fell on Kafka.

The basic idea is to have a Kafka broker that receives data from the probes on one or more topics. After that a Kafka consumer that reads the data and saves it to HDFS is necessary. For what concern the monitored network, rather than having an instance of Flume running on the routers in addition to the probe, the latter will be modified in a way that it also behaves as a Kafka Producer by sending the collected data to the Kafka broker.

This leads to the architecture I developed which consists of 3 virtual machines, as we can see below.

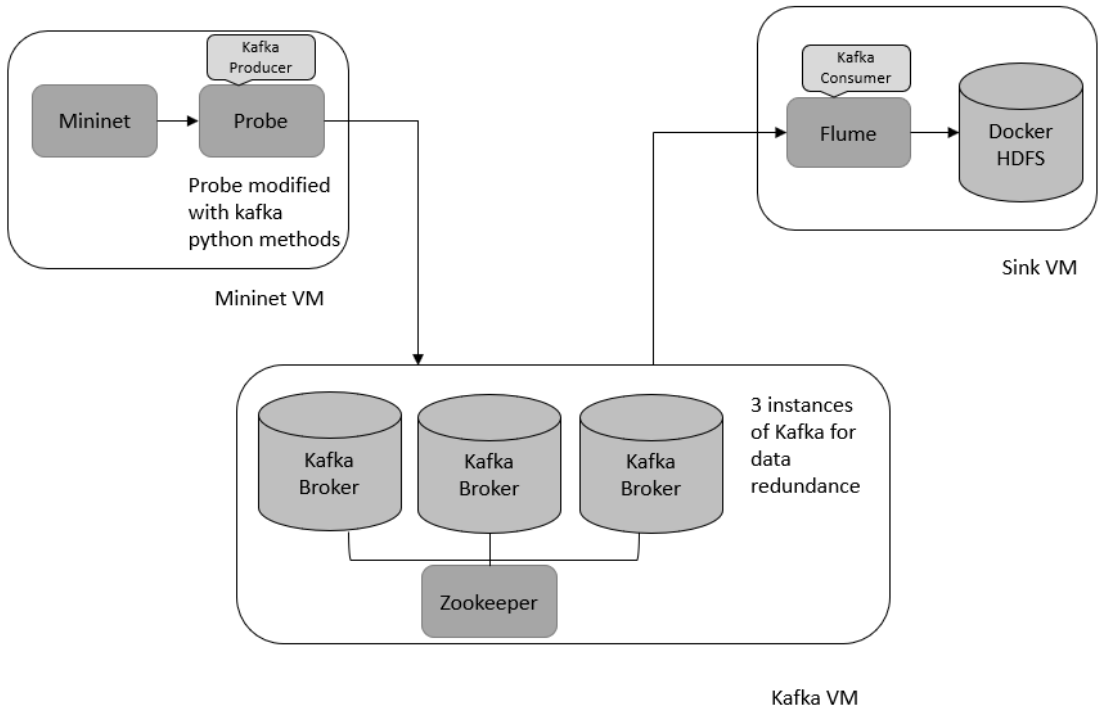


Figure 6.5 – The new architecture

In practice, in addition to the two virtual machines present in the old model, whose role has basically remained the same (the Mininet machine simulates the network and generates the data, while the Sink machine collects them), a third has been added in which a Kafka server has been implemented. The latter acts as an intermediary and synchronization agent for the other two. In fact, the Mininet machine will act as a producer, while the Sink machine as a consumer through an instance of Flume modified ad hoc. The three machines communicate with each other thanks to a bridge, a setting within the Oracle Virtual Box (the software used to manage virtual machines) that allows the machines to have an address on the local network. This allows the three machines to be reached between them and, if necessary, also from PCs connected to the same local network.

The detail of the three machines is explored below.

6.2.1 Kafka Configuration

The machine that in the graph I called Kafka VM is a virtual machine on which Ubuntu 18.04 LTS is installed, and on which a Kafka cluster consisting of three Kafka Brokers and an instance of Zookeeper has been set up. Although the usefulness of having three instances of Kafka on the same machine is lacking if you think about data security and their redundancy, this choice is motivated by wanting to create a configuration as much realistic as possible (hence the choice to remain with only one instance of Zookeeper, as in reality it makes no sense to have the same number of instances of Zookeeper and Kafka Broker). Once I have installed Kafka on the machine, I have configured the cluster, putting Zookeeper to listen on the address of the machine, which in my tests has always been 192.168.1.95, at port 2181 (that is the default one), and I have created three Kafka instances reachable on the same address but respectively on ports 9092,9093 and 9094. In addition, I have assigned to each of the instances a directory where to keep a copy of the logs and data, wanting to simulate three different disks for each instance. In detail, the directory of each broker is /home/ubuntu-kafka/kafka/logs, ../kafka1/logs and ../kafka2/logs. These settings are visible in the respective configuration files in /home/ubuntu-

kafka/kafka/config/server.properties, ../server1.properties and ../server2.properties of which we can see an extract below.

```
############################# Log Basics #############################

# A comma separated list of directories under which to store log files
log.dirs=/home/ubuntu-kafka/kafka2/logs

# 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=1
default.replication.factor=3
# The number of threads per data directory to be used for log recovery at startup and flushing at shutdown.
# This value is recommended to be increased for installations with data dirs located in RAID array.
num.recovery.threads.per.data.dir=1

############################# Internal Topic Settings  #############################
# The replication factor for the group metadata internal topics "__consumer_offsets" and "__transaction_state"
# For anything other than development testing, a value greater than 1 is recommended to ensure availability such as 3.
offsets.topic.replication.factor=3
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=1
delete.topic.enable = true

############################# Log Retention Policy #############################

# The following configurations control the disposal of log segments. The policy can
# be set to delete segments after a period of time, or after a given size has accumulated.
# A segment will be deleted whenever *either* of these criteria are met. Deletion always happens
# from the end of the log.

# The minimum age of a log file to be eligible for deletion due to age
log.retention.hours=12
```

Figure 6.6 – Kafka server.properties file

As you can see, further settings have been defined. In particular, a default replication factor equals to 3 has been set. In this way, every time a new topic is created automatically (for example when a producer sends data on a topic that does not yet exist), it will have three copies by default. The number of partitions instead has been set to 1. The motivation behind this choice is that, having in the current state of things, a single consumer reading the data from the topics, having more partitions would not bring any benefit in terms of performance. Finally, both to save space on the virtual machine and as a sensible choice in a possible real context, I set the log retention hours to 12 hours, so that the data is kept 12 hours after its arrival, after which it will be deleted to make place to the most recent ones. In this way, even in the presence of a network error or a failure on the server, the data of the last 12 hours will still be available.

Everything concerning the topics, like how many and which to create, although strictly connected to the Kafka brokers, will be discussed in the next paragraphs in order to clearly justify the choices made.

6.2.2 Sink VM

The virtual machine Sink VM is none other than the machine used in the previous model [6], that is a machine with Ubuntu 18.04 LTS on which an HDFS cluster and an instance of Flume are installed. This was possible, thanks to the modular approach of the machines in the previous architecture, an approach also maintained in this architecture, which allows you to replace some parts and reuse others.

While the emulation of the HDFS system and the injection of Spark queries has remained unchanged, what has changed is the way in which data is collected. As mentioned above, this machine has an instance of Flume that originally communicated with another instance of Flume to receive data. The solution I used was to reconfigure Flume on the server machine to act as a Kafka consumer. This was possible due to the strong integration of the two products, both developed by Apache.

The main differences from the previous configuration are, as we can see, the type of source, that is now set as a Kafka source, and consequently the address and the port of the data source, which is now the address and the port of the Zookeeper instance running on the Kafka machine. In addition, a further field has been added that indicates on which topics you need to be listening. The address of the data saved on HDFS has also changed, adding the name of the topic to the path, in order to read and save data from multiple topics without overlapping problems. Finally, the readSmallestOffset option allows the client to keeps track of the data already arrived and read from the Kafka broker only those not yet received.

The changes made are available in the configuration file at /home/ubuntu-sink/Dropbox/TESI/apache-flume-sink/conf/myconf/avro-source-hdfs-sink.conf and an extract can be seen in the next figure.

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

# Describe/configure the source
a1.sources.r1.type = org.apache.flume.source.kafka.KafkaSource
a1.sources.r1.zookeeperConnect = 192.168.1.95:2181
a1.sources.r1.topic = metrics
a1.sources.r1.batchSize = 100
a1.sources.r1.channels = c1

# Describe the sink
a1.sinks.k1.channel = c1
a1.sinks.k1.type = hdfs
a1.sinks.k1.hdfs.writeFormat = Text
a1.sinks.k1.hdfs.fileType = DataStream
#a1.sinks.k1.hdfs.filePrefix = test-events
a1.sinks.k1.hdfs.useLocalTimeStamp = true
a1.sinks.k1.hdfs.path = hdfs://localhost:9000/user/root/input/%y-%m-%d/%{topic}/%{key}
#a1.sinks.k1.hdfs.path = /data/kafka/%{topic}/%y-%m-%d
a1.sinks.k1.hdfs.rollCount=100000
a1.sinks.k1.hdfs.rollInterval=30
a1.sinks.k1.hdfs.rollSize=0
a1.sinks.k1.hdfs.minBlockReplicas = 1

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

Figure 6.7 – Flume configuration as Kafka consumer

6.2.3 Mininet VM

The last machine to be analyzed is the Mininet VM. This is also a machine on which Ubuntu 18.04 LTS is installed and which, for reasons due to the probe, needs a kernel version that is not higher than 5.3.0. In fact, a higher version does not allow the probe to work because the latter uses system functions that have probably been removed or modified in subsequent versions.

Naturally, this machine emulates the network via Mininet and Pox controller, which, as explained at the beginning of this chapter, has been modified to work better with the new probe.

The probe at issue has been modified to make it compatible with this architecture, because, although, as we have seen, it solves many critical issues of the old

implementation, it was designed to be compatible with the previous architecture, and therefore basically saves the collected data inside a folder of the file system exactly as the previous probe did.

To make the probe compatible with Kafka and solve the problem of saving data on the file system, I modified the pnpm.py file in the Software/app folder of the probe. This file represents, as the name suggests, the PNPM manager described when I deepened the probe, that is the part performed in the user space, the one that is waiting to receive commands from the REST interface to actually start the probe and that takes care of saving the data received from the BPF manager.

First of all, I therefore eliminated all the code inherent to saving the data in a folder of the file system, and then I implemented a solution to make the probe, or at least the user part of the probe, a Kafka producer. As the user part of the probe was completely written in python, the most sensible solution was to use Kafka-Python[29].

Kafka-Python is an open-source community-based library. Provides a Python client for the Apache Kafka system, with interfaces for producers and consumers. It was designed to work very similar to the official Java client, with clearly some added Python interfaces (for example, consumer iterators). It works both with the now old version of Python 2.7, and with the most recent ones, and is compatible with all versions of Kafka brokers, from the most recent (being constantly supported by the community) to the oldest such as the 0.8.0.

For this reason I then installed the kafka-python library and included it in the pnpm.py file.

Therefore, I added some methods to the pnpm.py file to make it a Kafka producer and send data to the Kafka broker.

```python
def publish_message(producer_instance, topic_name, key, value):
    try:
        key_bytes = bytes(key)
        value_bytes = bytes(value)
        producer_instance.send(topic_name, key=key_bytes, value=value_bytes)
    except Exception as ex:
        print('Exception in publishing message')
        print(ex)

def flush_messages(producer_instance):
    try:
        producer_instance.flush()
        print('Message flushed successfully.')
    except Exception as ex:
        print('Exception in flushing message')
        print(ex)

def connect_kafka_producer():
    _producer = None
    kafkaIPs=readKafkaIP()
    try:

        _producer = KafkaProducer(bootstrap_servers=[kafkaIPs[0],kafkaIPs[1],kafkaIPs[2]],acks = 1, retries = 2)
    except Exception as ex:
        print('Exception while connecting Kafka')
        print(str(ex))
    finally:
        print("Connected")
        return _producer

def readKafkaIP ():
    f=open("app/kafkabrokerIP","r")
    lines=f.readlines()
    count=0
    for line in lines:
        count+=1
    f.close()
    if count >= 3:
        return lines
    else:
        print ("There are less than 3 addresses in Kafka Broker IP file!")
        exit(-1)
```

Figure 6.8 – Kafka-python methods

In particular, as can be seen from the figure, I have implemented 3 methods.

The connect_kafka_producer is used to connect to the Kafka Broker. We can see that the 3 addresses are passed calling a method readKafkaIP. This method read a file in the directory and return the addresses and respective ports of the 3 Kafka brokers present in the Kafka VM machine, so that we can contact all three in case the first or the second are not working. Reading the addresses from a file allows us to change the address at runtime, modifying the kafkabrokerIP file that the readKafkaIP checks. Clearly if the Kafka server addresses are saved in a DNS, you can enter the server name rather than the IP address.

As the name suggest the bootstrap servers is a list of Kafka servers used to bootstrap connections to Kafka. Therefore the order in which these addresses are passed to the connect method only affects which server will be contacted first to connect to the cluster and which ones will be contacted if the previous one does not respond. The broker contacted when sending the data will instead depend on the topic and

partition leader and will therefore always be the same for all producers and consumers.

Another element to notice are the acks and retries settings which respectively refer to what type of acknowledgment to receive (in this case 1 indicates that it is enough that only one broker has received and saved the message), and how many times to retry sending in case no acknowledgment is received.

Then we have the publish_message method which takes care, after the connection, of sending/publishing messages on the Kafka Broker, taking as input the topic to publish on, and the key and value to be published, converted in bytes objects.

The flush_messages method instead, is a blocking function used at the end of a period, which serves to make sure there are no pending messages waiting to be sent.

At this point it remains to be discussed in what format the data is sent and on which and how many topics. My work in this case resulted in two versions of the probe. A first version, which was used to be integrated more than anything else with the pre-existing HDFS model, in order to have a functioning architecture starting from data collection up to their processing, and a second one, which was the one originally thought, developed instead with an eye to what will be in the future (a more complex and efficient HDFS and Spark query system). At the time of writing, in fact, another student is developing this part and some choices have been made in relation to the work he is developing. The two versions are both present within the Mininet VM and are respectively located in the Probe(oldHDFS) and Probe folders.

6.2.4 Probe(oldHDFS)

In the version designed to be adapted to the old HDFS model, the probe sends the collected data in each period, spreading them all on a single topic. The topic in question was called *metrics*. This choice is justified by the fact that in the previous version the data in the HDFS was collected under a single directory, and to work the

Spark queries needed and need to have all the data of a period available in the same folder.

```
if first_time:
    period = get_period_from_tstamp(l.tstamp, mp)
    ml.info('Period: %u',period)
    first_time = False
    publish_message(kafka_producer,'metrics',str(period)+'_'+str(fileN),'ip_src, ip_dst, port_src, port_dst, proto, hash, timestamp, cluster1, cluster2, period')

netif2 = str((cast (k.netif, c_char_p)).value+"_"+in_out)
if netif2 == netif:
    hash_and = l.hash_low & and_val
    match_and = int (match_value) & and_val
    if hash_and != match_and:
        continue

    hash_hex = ''
    if hash_function == 'md5':
        hash_hex += format (l.hash_high, '02x')
        hash_hex += format (l.hash_low, '02x')
    else:
        hash_hex += format (l.hash_low, '02x')

    pkt_info = Marked_Flow_Pkt_Info_Extended (l.tstamp, hash_hex, str (netaddr.IPAddress (l.sip)),
                                              str (netaddr.IPAddress (l.dip)), l.proto, l.sport, l.dport)
    #ml.info ("Netif: %s Pkt_info: Timestamp: %s Hash: %s IP_src: %s IP_dst: %s Protocol: %s Sport: %s Dport: %s" % (
    #    netif, ns_to_date (l.tstamp), hash_hex, pkt_info.sip, pkt_info.dip, pkt_info.proto, pkt_info.sport, pkt_info.dport))
    pkt_list.append (pkt_info)
    block_info.count_measure += 1
    sum_timestamp += temp_tstamp

    if netif not in clusters_per_node.keys():
        clusters_per_node_1 = str(-1)
        clusters_per_node_2 = str(-1)
    else:
        clusters_per_node_1 = str(clusters_per_node[netif][0])
        clusters_per_node_2 = str(clusters_per_node[netif][1])
    publish_message(kafka_producer, 'metrics', str(period)+'_'+str(fileN), str(pkt_info.sip)+'_'+str(pkt_info.dip)+'_'+str(pkt_info.sport)+'_'+str(pkt_info.dport)+'_'+str(pkt_info.proto)+','
                +hash_hex+','+str(l.tstamp)+','+clusters_per_node_1+','+clusters_per_node_2+','+str(period))
```

Figure 6.9 – Probe(oldHDFS) implementation

A choice consistent with the old model, but which is also sensible in terms of data cataloging, is the choice of the key. In fact, the key consists of the period plus the name of the interface from which the data comes, as can be seen in the figure. This choice also allows, having set in Flume that the name of the directory where to insert the data is also composed of the key, to find the data in the HDFS exactly as if they had been loaded with the previous architecture (where the directory was the name of the file from which data was sent), allowing complete compatibility with the new message sending system. As for the value, this includes all the data that were already sent with the previous version, excluding the additional data that the new probe can capture, since they would still not be considered in post processing.

6.2.5 Final Probe

A problem that is not solved using the probe in the version just described, is the failure to send aggregate data over the period, and the exclusion of the additional

data just mentioned. In fact, leaving the HDFS part unchanged, it makes no sense, even if it is possible, to send them.

From this need and from that of having a probe ready for future changes that will affect the Sink VM machine, this second version was born. In this version, the collected data is spread over a total of 4 topics.

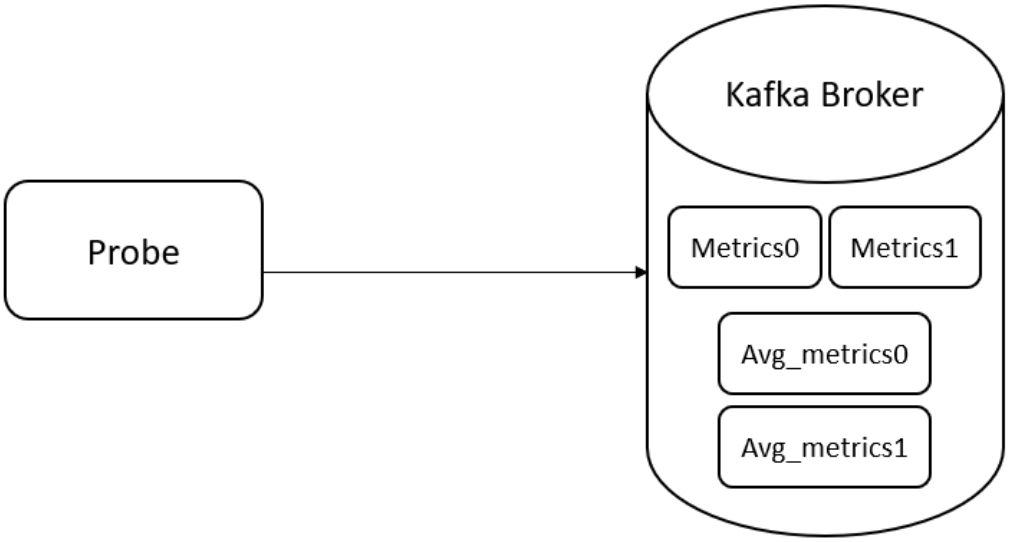


Figure 6.10 – Topics in the final version of the probe

The four topics as we can see in the figure are Metrics0, Metrics1, Avg_metrics0 and Avg_metrics1. The topics Metrics0 and Metric1 contain the data collected by the probe on an interface in a period, and in particular in Metrics0 those whose division of the period by 2 gives remainder 0, and in Metrics1 those with remainder 1. The same goes for Avg_metrics0 and Avg_metrics1, where instead aggregated data are sent divided by interface over a period. The message key is unchanged from the previous implementation, both for Metrics and Avg_metrics topics. For what concerns the value part of the Metrics topics, this is comprehensive of the field left out before. The value part of the Avg_metrics topics instead consists of aggregated information collected by the probe in a period, like the total number of packets

passed through an interface, the number of collected packets and others that can be seen in the figure below.

```
    if netif2 == netif:
        hash_and = l.hash_low & and_val
        match_and = int (match_value) & and_val
        if hash_and != match_and:
            continue

        hash_hex = ''
        if hash_function == 'md5':
            hash_hex += format (l.hash_high, '02x')
            hash_hex += format (l.hash_low, '02x')
        else:
            hash_hex += format (l.hash_low, '02x')

        pkt_info = Marked_Flow_Pkt_Info_Extended (l.tstamp, hash_hex, str (netaddr.IPAddress (l.sip)),
                                                  str (netaddr.IPAddress (l.dip)), l.proto, l.sport, l.dport)
        #ml.info ("Netif: %s Pkt_info: Timestamp: %s Hash: %s IP_src: %s IP_dst: %s Protocol: %s Sport: %s Dport: %s" % (
        #    netif, ns_to_date (l.tstamp), hash_hex, pkt_info.sip, pkt_info.dip, pkt_info.proto, pkt_info.sport, pkt_info.dport))
        pkt_list.append (pkt_info)
        block_info.count_measure += 1
        sum_timestamp += temp_tstamp

        if netif not in clusters_per_node.keys():
            clusters_per_node_1 = str(-1)
            clusters_per_node_2 = str(-1)
        else:
            clusters_per_node_1 = str(clusters_per_node[netif][0])
            clusters_per_node_2 = str(clusters_per_node[netif][1])
        publish_message(kafka_producer, 'metrics'+str(topic_calculation(period)), str(period)+'_'+str(fileN), str(pkt_info.sip)+'_'+str(pkt_info.dip)+'_'
                        +str(pkt_info.sport)+'_'+str(pkt_info.dport)+'_'+str(pkt_info.proto)+','
                        +hash_hex+','+str(l.tstamp)+','+clusters_per_node_1+','+clusters_per_node_2+','+str(period)+','+str(l.hash_len)+','
                        +str("%x" % l.dscp))

        #del pkt_map[k]

    ml.info (
        "Block_info: Block_number: %u Color: %u Total_pkts: %u Count_measure: %u Block_hash_length: %u \nFirst_tstamp: %s " % (
            len (measures_dict[netif]), color, total_pkts, block_info.count_measure, block_hash_length,
            ns_to_date (first_tstamp)))
    value= str(color)+','+ str(total_pkts)+',' +str(block_info.count_measure)+','+ str(block_hash_length)+','+ str(ns_to_date (first_tstamp))+','
           +str(ns_to_date (int((sum_timestamp/float(block_info.count_measure))+boot_ns)))
    publish_message(kafka_producer,'avg_metrics'+str(topic_calculation(period)),str(period)+'_'+str(fileN),value)
    flush_messages(kafka_producer)
ml.info ("Mean_timestamp: %s\n" % (ns_to_date (int((sum_timestamp/float(block_info.count_measure))+boot_ns))))
```

Figure 6.11 – Final Probe implementation

The motivation behind the choice to divide the data on 2 topics according to the period, is to be able to analyze the data of a period without having to worry about separating the data of the previous and subsequent periods. In an ideal scenario, a consumer, listening on all the topics, will receive at a given moment the data of the period just passed on a topic, and at the end of their processing he will receive data on the topic that previously had not provided data and so on, respecting hence the duality of marking also in the processing time.

Thanks also to the information received on the Avg_metrics topics, it will be possible to carry out further checks (such as the verification of the packets actually arrived at the Kafka Broker compared to those captured by the probe) or even obtain information that are less accurate but faster in the processing.

Use cases of this type, and more, will be the subject of the work that will happen to this.

6.3 Workflow

At this point, after having explored the various machines, it is possible to outline the workflow which can be represented as a sequence of steps listed below.

1. In the Mininet VM the GEANT2012 network emulation (with the addition of the nodes previously seen) takes place via Mininet and Pox controller. Multiple streams are simulated by starting various iperf servers and iperf clients communicating with each other.
2. The probe, when enabled, is waiting for marked packets arriving on the various emulated interfaces. When this happens, it calculates the first raw measurements, based on timestamps, hashes and identification fields. When it finishes analyzing the data coming from a specific interface in a period, it sends them through the kafka-python library, practically acting as a kafka producer, to the Kafka brokers (which in this case are all on the Kafka VM machine), within a single topic metrics.
3. Meanwhile on the Kafka VM, the three Kafka brokers, already started previously, are waiting to receive data. When this happens, as soon as one of the brokers, realistically the leader of the topic/partition, receives and stores the data, it sends an ack to the producer (the probe). At the same time Zookeeper, while the data between the various brokers will synchronize to have data redundancy, will update all the information about the index and the number of replicas.
4. The Flume agent on the Sink VM is waiting for new data arriving on the topics incoming data on the topics on which it is listening on. When the Kafka broker sends the arrived data, Flume immediately stores it within HDFS, in the path / user / root / data / input / topic / key.

From this point on, if we used the probe compatible with the old HDFS processing, the execution follows exactly what happened previously, namely the storage in the HDFS and the injection methods of the preprocessing and post processing Spark jobs in order to extrapolate relevant information from the data.

Chapter 7

Lab Simulation

At this point in my work, it became necessary to test this model in an environment that is as much realistic as possible. In fact, until now, both my work and the one before [6] mine have been tested in a highly controlled and emulated environment as all the tests were carried out on the three, or two in the previous case, virtual machines that all ran on the same PC. In addition to this, we wanted to test the potential of Kafka and its scalability. To take a step forward in this sense, thanks to the collaboration with TIM, I had some server machines on which to test this mechanism in a more realistic way.

Although the initial idea was to test the probe directly on programmable routers, this turned out not to be feasible given the lack of availability of many routers of this type to be used at the time of my tests. We therefore opted for server machines (some of which were already used in the development of the new probe [7]) to set up a more realistic test environment.

7.1 Model

I therefore had 3 machines available. Two running Ubuntu 18.04 LTS and one running Debian 9, all with a kernel version lower than or equal to 5.3.0.

The idea behind this model is to have a Kafka cluster, an HDFS cluster, a consumer, and then have more probes that act as producers, and consequently more flows to monitor, in order to generate more traffic for the Kafka cluster.

To do this, we have chosen to have a hybrid solution, that is, on the one hand a flow that passes through the physical machines available, on the other an emulation of the GEANT2012 network emulated with Mininet. In this way, since the machines are connected to each other with 1 gigabit ethernet cables, we can on the one hand have simple flows, i.e. passing through 2 or 3 nodes at most but with high throughput, and on the other hand more complex flows but with minor throughput (those emulated). In addition, this solution allows us to have 3 Kafka producers, that is the maximum number we can have with the servers at our disposal, in order to test the broker.

To generate the flows that will pass from the real machines, the ideal solution would have been to use traffic generators, as happened in the work [7] on the new probe. At the time of my tests, however, these were not available and we therefore opted for the use of the iperf tool, already used previously to generate traffic in the emulated network.

All these considerations led to the following model.

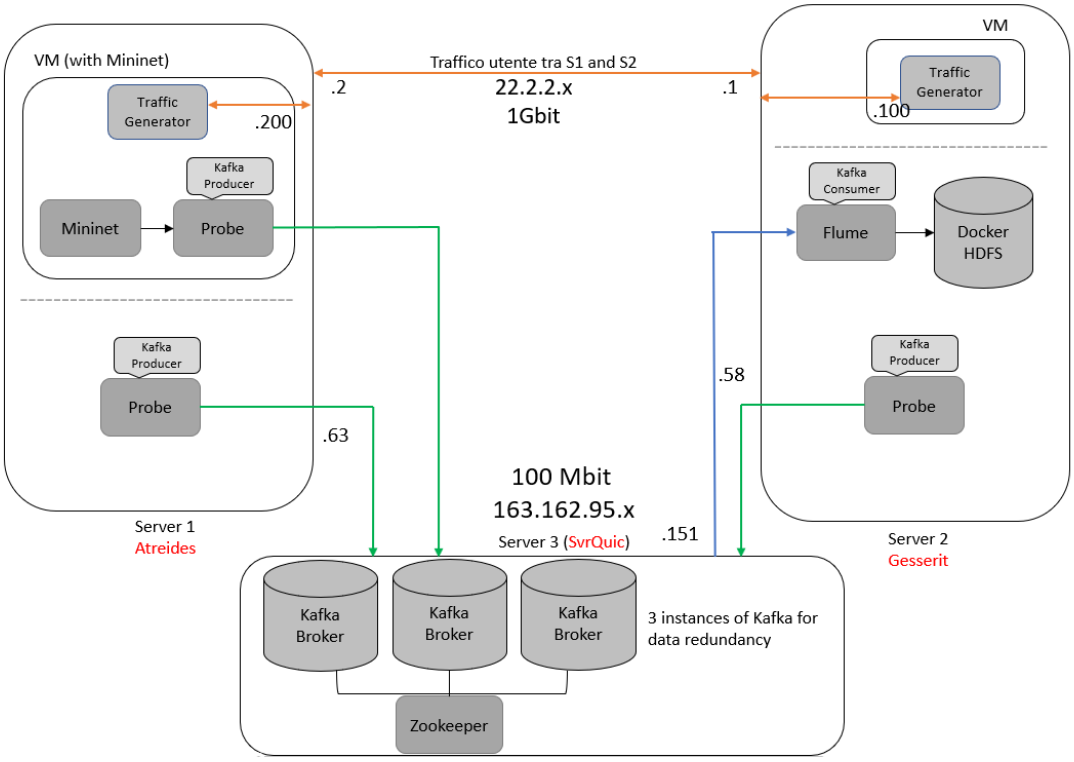


Figure 7.1 – Lab Schema

As we can see, the model is based on the use of three server machines, respectively called Atreides, Gesserit and SvrQuic.

The task of acting as a Kafka server broker fell on the SvrQuic (Server3) machine, the one with Debian on it. It will therefore be the server to which all the data, collected by the probes placed in listening to the various flows, both real and emulated, will arrive. The choice was forced since this machine, as we can see in the figure, does not have a direct connection with the other machines but can only be reached through the laboratory network 163.162.95.x, which is a 100Mbit network. However, since the other two machines were connected directly by a gigabit ethernet, the most sensible idea was to pass the flow of data to be captured through the other two.

On Server3 we therefore have 3 instances of Kafka and one of Zookeeper, which is a scheme that strongly recalls the configuration detailed in the previous chapter on the Kafka VM. In fact, the main difference is the address at which the brokers are reachable, which in this case has become 163.162.95.151 with the respective ports 9092, 9093 and 9094. All other settings such as topic redundancy and retention time remained unchanged. Clearly compared to the configuration seen previously, in this case we are on a real machine with 48 cores, which allows us to have a cluster that can certainly handle a more realistic traffic. However, it must be said that the three instances, in this case, all work on the same disk, while in a realistic environment each broker would have disks at exclusive disposal.

On the Gesserit machine (Server2), the HDFS cluster has been set up instead. This will obviously have the task of collecting data in a lasting way and processing them through Spark jobs. As happened in the Sink VM, also in this case a docker container was used to virtualize the HDFS cluster as the latter needs multiple instances to function and in this case the performance could be neglected. To enter the data within this cluster, I also opted here for an instance of Flume modified to behave like a Kafka consumer. Its configuration follows the one seen in the virtual machine, with the only change of the address to contact.

On the Atreides machine (Server 1), the Mininet VM machine, running in VirtualBox, has been replicated, for the emulation of the GEANT2012 network, on which the probe, as seen before, will capture the marked traffic and send it to the Kafka broker, which in this case will be located at 163.162.95.151. To allow this virtual machine to be able to communicate with the local network, to which Server1 is actually connected, the Mininet VM has been bridged, through a setting of Virtual Box, with the interface on which Server1 is actually connected to the network. In this way the virtual machine will appear to the network as an additional physical machine, reachable at an address that will be assigned directly by DHCP. In this way the Mininet VM can therefore receive and send traffic on the 163.162.95.x network.

7.1.1 Flow between servers

At this point, however, we can see that a second function has been neglected in the last two server machines. In fact, as shown in the figure 7.1, there is a probe and a traffic generator in both.

The two machines are connected to each other via a direct link, gigabit ethernet, which is addressed as a separate network that is 22.2.2.x. The two servers therefore have, in addition to the addresses on the network 163.162.95.x (.63 for Atreides and .58 for Gesserit), respectively, addresses 22.2.2.2 and 22.2.2.1.

Since, as mentioned, the model also provides for the capture of real traffic, in addition to the emulated one, this traffic pass on this link. Hence the presence of the two probes on the two servers, so that they can intercept the traffic that passes between the two machines.

Thanks to the lack of traffic generators and the characteristic of the probe, which I have already talked about previously, of not capturing the outgoing traffic from the device that generates that traffic flow, it was not possible to generate traffic via iperf, directly on the Atreides and Gesserit servers, as we would not have been able to capture the traffic exiting these two machines. It was therefore necessary to generate traffic from different machines or at least from different IPs. For this

reason, an iperf traffic generator has been inserted inside the Mininet VM on Server1 and inside a virtual machine created for the occasion, on Server2. In fact, for both machines a bridge has been set up on the respective interfaces that connect the two servers, so as to be connected to the 22.2.2.x network. Both were manually assigned an IP address, as there is no DHCP. We therefore have that the Mininet VM in addition to the address on the network 163.162.95.x will also have a second address 22.2.2.200, while on the other hand the virtual machine created exclusively to generate traffic will have an address 22.2.2.100.

In the two virtual machines, a path was then set up to reach the other, which passes through the two servers Atreides and Gesserit. In essence, therefore, we will be able to create a bidirectional flow through two clients and two servers iperf located within the two virtual machines.

From the point of view of the network the two virtual machines will appear connected as in the following figure.

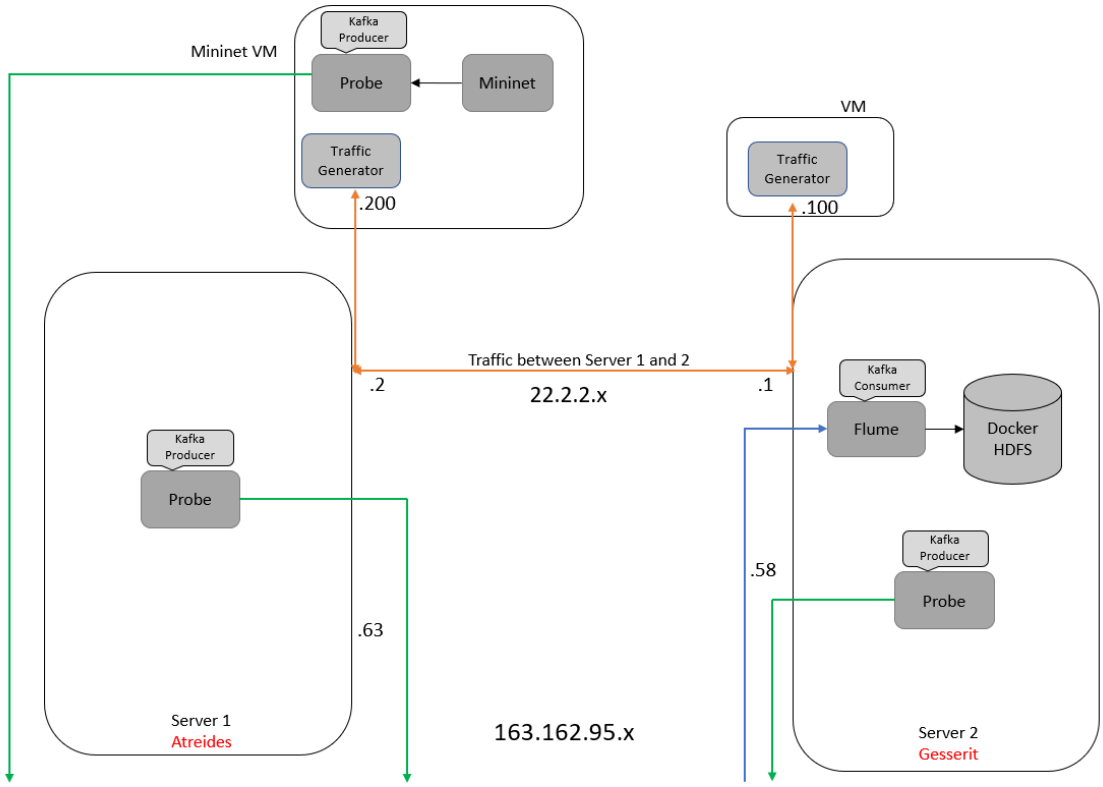


Figure 7.2 – Bridged Virtual Machine on 22.2.2.x network

The traffic from the Mininet VM will first pass from Server1 to address 22.2.2.2, then from Server2 to address 22.2.2.1 to finally arrive at the other virtual machine with address 22.2.2.100. On the other VM, the traffic path will instead be reversed. This will allow the two probes positioned on the two servers to capture both outgoing and incoming traffic without problems.

7.2 IPMininet

At this point in the design of the laboratory model, a further modification to the network emulated with Mininet was necessary. In fact, wanting to have bidirectional flows like the one set up between the two server machines, there was the need (always driven by how the probe works) to add another three nodes (considering the three flows analyzed previously).

As we have seen previously, the changes made to the GEANT2012 network to work correctly with the new probe had led to the configuration with three additional nodes seen in figure 6.3.

Therefore, in order to have the probe capture all the packets and wanting to have bidirectional flows, three further nodes must be added, respectively connected to r25, r34 and r16 in order to have a departure and arrival point of the flows external to the monitored network.

To do this, a further version of the GEANT2012.graphml file was created, called GEANT2012MOD+6.gramphl, where in a similar way to what we saw with the addition of the first three nodes we arrived at the configuration in the following figure, in which the 6 external nodes and the 3 flows (distinguished by color) with the possible paths they can take, are highlighted.

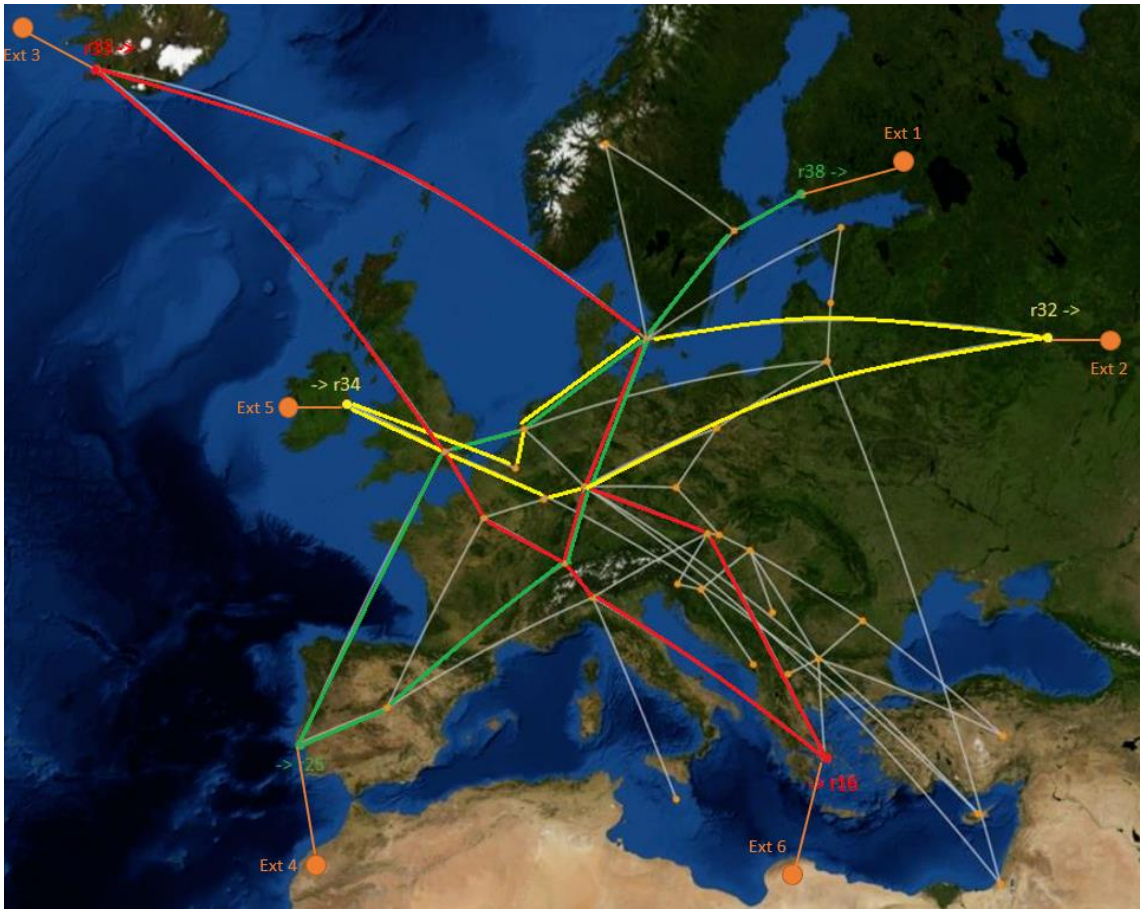


Figure 7.4 – Final GEANT configuration

In this configuration the RIP algorithm that is loaded on the routers during the creation of the network, showed his weakness. In fact, in most simulations, this increase in nodes leads to the failure of the entire network, which, due to the RIP routing algorithm, is unable to route packets from one end to the other.

Although a solution based on the manual addition of some paths, adding static routes in the routers during the creation of the network, has been implemented in the net2switchRandomLossNew6ExNodeStatic.py file, in order to make at least the analyzed flows functional, it is clear that any other future modification that passes for the emulation of a network with this implementation of Mininet, will lead to malfunctions.

The main problem to be solved is the using of a more modern and performing routing algorithm to replace the RIP.

A permanent solution to this problem has been found using IPMininet [30]. IPMininet is a Python library, which extends Mininet, to support emulation of complex IP networks. This library in fact provides new classes, including the Router one, which allows the automatic addition of the most various routing algorithms, such as OSPF or ISIS, to the network nodes.

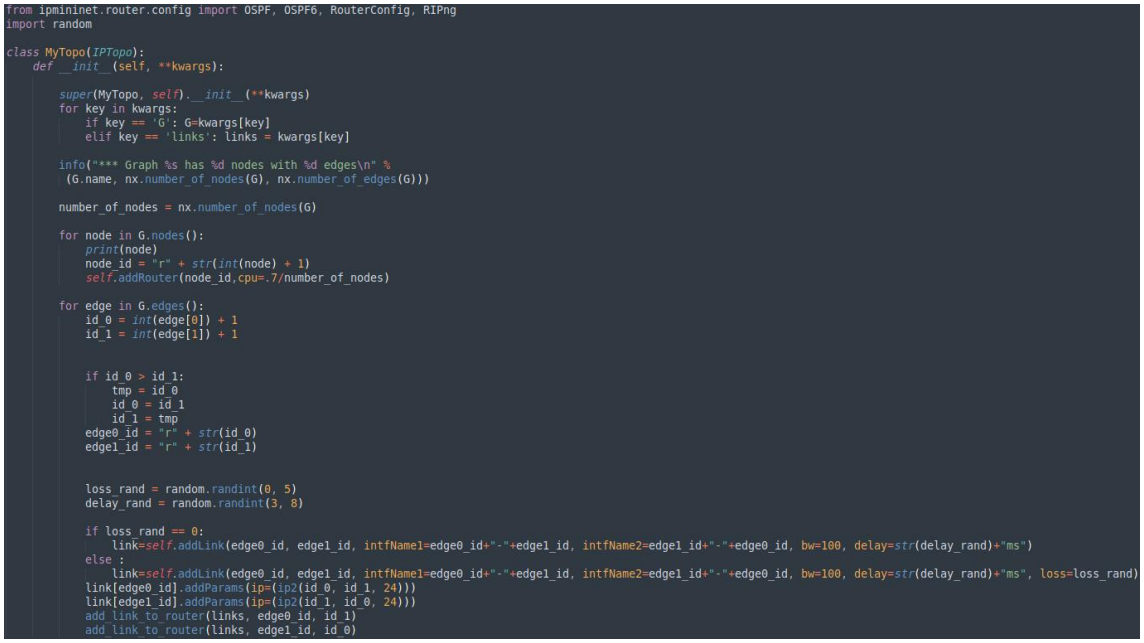
Through this library I have implemented a new solution that allows the creation of a network starting from a. graphml file, without node limits since OSPF [32] is now the default algorithm running on IPMininet nodes.

Furthermore, this implementation allowed me to lighten the structure of the emulated network, as the presence of switches between the routers was no longer necessary, thus also eliminating the need to use the POX controller. In this way between two nodes/routers of the network there will be only the link that connects them, and on which the loss and delay will be set, as in the figure.

Figure 7.5 – IPMininet node connection

This implementation can be consulted in the netIPMininet3.py file, of which there is an extract below.

By giving the GEANT2012MOD+6.graphml file as input to this script, the network in figure 7.4 will be emulated without problems, giving us the concrete possibility of having bidirectional flows as we wished.



```
from ipmininet.router.config import OSPF, OSPF6, RouterConfig, RIPng
import random

class MyTopo(IPTopo):
    def __init__(self, **kwargs):

        super(MyTopo, self).__init__(**kwargs)
        for key in kwargs:
            if key == 'G': G=kwargs[key]
            elif key == 'links': links = kwargs[key]

        info("*** Graph %s has %d nodes with %d edges\n" %
         (G.name, nx.number_of_nodes(G), nx.number_of_edges(G)))

        number_of_nodes = nx.number_of_nodes(G)

        for node in G.nodes():
            print(node)
            node_id = "r" + str(int(node) + 1)
            self.addRouter(node_id,cpu=.7/number_of_nodes)

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


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


            loss_rand = random.randint(0, 5)
            delay_rand = random.randint(3, 8)

            if loss_rand == 0:
                link=self.addLink(edge0_id, edge1_id, intfName1=edge0_id+"-"+edge1_id, intfName2=edge1_id+"-"+edge0_id, bw=100, delay=str(delay_rand)+"ms")
            else :
                link=self.addLink(edge0_id, edge1_id, intfName1=edge0_id+"-"+edge1_id, intfName2=edge1_id+"-"+edge0_id, bw=100, delay=str(delay_rand)+"ms", loss=loss_rand)
            link[edge0_id].addParams(ip=(ip2(id_0, id_1, 24)))
            link[edge1_id].addParams(ip=(ip2(id_1, id_0, 24)))
            add_link_to_router(links, edge0_id, id_1)
            add_link_to_router(links, edge1_id, id_0)
```

Figure 7.6 – IPMininet script

Another version of the netIPMininet3.py file, called netIPMininet3Test.py, was at the end created to automatize also the creation of client and server iperf on the external nodes and the subsequent traffic generation.

Clearly, the use of the IPMininet library will allow in the future, if needed, to use even larger networks than the GEANT2012 that was chosen in the previous work because it was compatible with the stability of the RIP algorithm used previously.

7.3 Workflow

At this point, after having described the model in all its parts, it is possible to outline the workflow which also in this case can be represented as a sequence of steps listed below.

1. First of all, the three Kafka brokers together with the zookeeper instance are started on Server3.

2. After that the Flume agent, on the Gesserit Server2, is started. From now on it will be waiting for new data arriving on the topics metric0, metrics1, avg_metrics0 and avg_metrics1.
3. On Server 2 the HDFS cluster also is started trough the docker virtualization.
4. At this point the 3 probes, those on the two servers and the one in Mininet VM, are enabled, in order to put them waiting for the start signal.
5. In the Mininet VM the GEANT2012 network emulation (with the addition of 6 external nodes) takes place via IPMininet. Multiple bidirectional flows are simulated by starting the various iperf servers and iperf clients that communicate with each other on the various emulated nodes.
6. At the same time, iperf servers and clients are started on the Mininet VM and on the virtual machine on Server2 to generate bidirectional traffic between the two servers Atreides and Gesserit.
7. At the same time probes are started through the start signal. From this moment on, they are waiting for marked packets to arrive on the various emulated and non-emulated interfaces. When this happens, the first raw measurements are calculated, based on timestamps, hashes, identification fields and at the end of a period when the probe finishes analyzing the data coming from a specific interface, these are sent through the kafka-python library, to the Kafka brokers (which in this case are all on Server 3).
8. On server 3 as soon as one of the brokers receives and stores the data, it sends acks to the producers (probes) and will synchronize the data between the various brokers, to have data redundancy, thanks to the help of zookeeper.
9. When the Kafka broker sends the arrived data to the listening consumer, Flume immediately stores it within the HDFS cluster, virtualized in the docker container, in the path / user / root / data / input / topic / key.

For further clarification on the tests carried out, in the Appendix there is a point-by-point explanation of all the command lines and settings used.

7.4 Tests and Performance

During this laboratory tests, considerations regarding the traffic generated, the performance and the scalability of Kafka in relation to both the generated loads and the possible real loads, have been made.

As for the laboratory model, two tests were carried out by having the probes on the servers capture at most first 10000 and then 20000 packets per period per interface. This means that having each server an input and output interface, where marked traffic pass, we have a total of 4 interfaces. As for the probe on the Mininet VM instead, this has been set to capture at most first 1000 and then 2000 packets per period per interface. In this case the number of monitored interfaces clearly depends on the routing and where the packets pass, but since there is a filter (that represent the interfaces that have to be monitored) we have approximately 20 interfaces. We therefore have 40000 or 80000 messages at most sent by the probes on the servers and 20000 of 40000 messages at most from the probe on the emulated network, for a total of about 60000 messages per period in the first case and 120000 messages per period in the second case.

Each message, looking at the length of key and value, currently weighs about 100bytes (upwards). This means that to monitor all these real and emulated flows, each period, which in our tests has always been 30 seconds, generates about 6 MB of traffic in the first test and 12MB in the second one.

For what concerns, instead, a possible real scenario, discussing with the TIM managers who supervised my work, we came to imagine a scenario in which in a network there are about 300 monitored interfaces and 1000 packets per period. This leads, with messages that are currently 100byte, to a traffic of 30MB per period, which on a network with links of at least 1Gbit is more than acceptable.

For this reason, it would have made sense to try to increase the number of packets captured to test the broker with a 30MB of traffic. However, since the network that connect the Kafka broker is a 100 Mbit, increasing the number of packets captured would not have led to an effective load increment on the Kafka broker respect to the

one tested with 6 and 12MB of traffic. The reason why the number of packets to be captured was not increased further was to not saturate the 100Mbit network "only", that connected the Kafka cluster with the other machines, in order to also test the strength of the latter without delays due to the network.

All these considerations do not account the traffic generated by the kafka broker to the Flume consumer on Server2. In this case we should consider a doubled traffic that for sure will saturate the 100Mbit network.

7.4.1 Kafka Performance

During my tests the broker Kafka did not give any signs of failures in receiving messages from the probes. Verification that there were no packet losses in this phase was done through a python script “check.py” which compare the number of messages received by the brokers on the two topics metrics with the one calculated by looking at the statistics in the two topics avg_metrics.

However, wanting to have an idea of the load that it could support, various papers on the performance of the Kafka brokers tested in various configurations came to my aid. Among these, the article I selected as a reference (but which is still compatible with other works that I have consulted) is the one [31] made in 2014 by LinkedIn (the company that gave birth to the first versions of Kafka).

In this work three consumer machines are used, as they have 6 core processors, 32 GB of Ram and 7200rpm SATA disks as storage, to create a Kafka cluster.

In this setup, according to the tests with 3 producers, and a replication factor 3 with asynchronous ack (i.e. the ack is sent when only 1 replica is ready) we get up to 2,024,032 records/sec or 193 MB/sec (with messages of at least 100bytes). This number can grow (clearly not linearly) as the number of producers and brokers increases. However, even if this number were an upper limit, we are well below the 30 MB per period, calculated for a real case study.

With regard to the choice of keeping the data saved on the Kafka brokers for 12 hours, this will mean having about 40GB of space available on each broker (always considering 100byte messages), a requirement now satisfied even by low-end machines. This implies that, if it were necessary in the future, the retention time of messages could be increased.

All this ensures that Kafka is a solution that can last firmly over time, even as the number of nodes or packets captured increases.

7.4.1 Traffic Variability

With regard to this last question, I would like to clarify. The numbers I have just provided refer to specific cases tested or that refer to what in the imagination will be the future of this performance monitoring system.

Clearly these numbers can vary greatly depending on three main factors:

1. Number of monitored interfaces
2. Packages captured per period
3. Length of the period

In fact, doubling (and therefore increasing by 100%) the number of monitored interfaces or the number of packets collected in a period, results in a directly proportional increase in traffic on the network (therefore an increase of 100%) and consequently in data arriving to brokers.

The length of the period, on the other hand, has a different effect. First of all because it does not necessarily imply an increase in collected packets. However, considering a proportional increase in packets, although at the point level (i.e. at the end of each period) we could measure more traffic, in general the traffic circulating on the network does not increase as we would simply have fewer periods of more packets, or vice versa.

Given the dependence of traffic on these values, one way to have a unique measure of how much traffic is actually generated is to consider the packets captured per

second. In this way, the tests carried out in the laboratory can be summarized as 666 packets/sec for each interface at most captured by the probes on the servers (considering the maximum of 20000 packets in a period of 30 seconds), and 66 packets/sec for each interface at most captured by the probe on the Mininet VM.

Chapter 8

Conclusions and future works

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

One of the most time-consuming parts that clearly did not have space in the previous chapters, was the search for the component that should have replaced Flume in the exchange of messages. In fact, various alternatives have been evaluated to then arrive at Kafka. In particular, the possibility of staying on Flume was evaluated, trying to use a data source other than that of the reading in memory. Another option that has been evaluated is 0MQ, an asynchronous messaging library that provides a message queue. It can be used as a publish / subscribe system without a dedicated message broker. Finally, MQTT broker, another publish/subscribe system, similar to Kafka in certain aspects, was also evaluated as a possible choice. If in the first case Flume was rejected due to the impossibility of both a good level of synchronization and to differentiate the data sent, for the other two cases, the choice was conditioned in particular by the fact that Kafka, being an Apache product, was of easier integration both with HDFS in general but also for future projects such as real-time streaming of data to a Spark server.

Even modifying the parts done by my colleagues previously took some time to understand in detail what to change and how. In this sense, however, the previous undergraduates Calogero Corbo and Marino Urso have always been at my disposal in case of doubts.

In the end, although it was done during the pandemic period, I still consider this work a positive experience because it helped me to deepen topics that I might not have addressed.

The aim of this work was to bring the network performance monitoring through Big Data approach closer to the use on real networks and this work, with the modification of the message exchange system which guarantees the possibility of differentiating the data sent and allows a greater level of synchronization between producer and consumer, certainly moves in this direction.

Clearly there are still a number of improvements to be implemented. In particular as regards the probe, this could be further modified to avoid peaks in sending data. At present, in fact, after that a period passed, the probe sends all the data collected on that period as quickly as possible. This behavior means that at the end of each period there is a traffic peak on the network in which all the probes send data. A solution could be to spread the sent of the data on the next period, while the probe collects the data of the new period. Furthermore, the probe needs an update due to incompatibility with kernels higher than 5.3

Another part that needs a change as soon as possible is the processing part of the collected data. Although in fact a version of the probe was created to maintain compatibility with the old HDFS model and the way in which the data were processed, it is clear that to fully enjoy the potential of this model it is necessary to rethink this part. An idea, which the graduate student who will succeed me is already carrying out, is to analyze the data in real time and save only the necessary ones.

In addition to these, other improvements, not explored in this work, such as the creation of routers that mark traffic, the improvement in the calculation of clusters, the choice of the points to be monitored, and the possibility of a graphical interface to consult the processed data, and others, will certainly give space for further refinement works.

I am sure that the Politecnico di Torino, together with TIM, will try to explore these fields and continue these works.

Appendix

Step by step test setup

First of all on Server 3, Zookeeper is started via zookeper-server-start.sh (in kafka directory) to which the zookeeper.properties file is passed, where the zookeeper address and port are set.

- sudo bin/zookeeper-server-start.sh config/zookeeper.properties

Then, always on Server 3, the three Kafka brokers are started through kafka-server-start.sh (in kafka directory) to which server.properties, server1.properties and server2.properties files are passed. In this configuration file, we set the brokers to have a redundancy of 3 and one partition. In addition we pass the address and port of zookeeper and set the address and port of each kafka broker which in this case is 163.162.95.151:9092,9093 and 9094 (detail of the server.properties in chapter 6).

- sudo bin/kafka-server-start.sh ../server.properties
- sudo bin/kafka-server-start.sh ../server1.properties
- sudo bin/kafka-server-start.sh ../server2.properties

The Flume agent, on the Gesserit Server2, is started running the run.sh file in the flume directory. From now on it will be waiting for new data arriving on the topics metric0, metrics1, avg_metrics0 and avg_metrics1, according to the configuration file /apache-flume-sink/conf/myconf/avro-source-hdfs-sink.conf seen in chapter 6.

- ./run.sh

The HDFS cluster, on Server 2, also is started running, in the docker-hadoop-master folder, the command “docker-compose up” that start the docker and create containers as indicated by the configuration file docker-compose.yml (also present in the folder).

- sudo docker-compose up

The 3 probes, those on the two servers and the one in Mininet VM, are enabled via the python file run.py (which puts them waiting for the start signal) present in the directory probe/software. The script takes as input the configuration file config_filter.json, present in the directory probe/run.

- sudo python run.py

The configuration file is set in this way on the servers 1 and 2:

- "prog_id": 3,
- "mp":30,
- "mpc":180,
- "starter_mark": 1,
- "next_mark": 2,
- "proto": "udp",
- "match_value":383146267,
- "match_length": 1,
- "hash_function": "bob",
- "npkts": 20000

On the Mininet VM the configuration is the same unless for the last value (that represent the max number of packet to capture) that is 2000, and the addition of a field “netifs” where there is the list of the interfaces to monitor.

At this point, in the Mininet VM the GEANT2012 network emulation (with the addition of 6 external nodes) takes place via IPMininet. Multiple bidirectional flows are simulated by starting the various iperf servers and iperf clients that communicate with each other on the various emulated nodes. All this happens automatically by running the python script netIPMininet3Test.py and passing it as an argument the file Geant2012MOD+6.graphml. The iperf traffic on the emulated network is generated automatically whit a marked traffic of 100 packets/sec.

- sudo python3 netIPMininet3Test.py topologies/Geant2012MOD+6.graphml

At the same time, iperf servers and clients are started on the Mininet VM and on the virtual machine on Server2 to generate bidirectional traffic between the two servers Atreides and Gesserit. This happens enabling on both machines a server iperf through command “iperf -s -u -B address.of.the.machine” and a client using the iperf_test.sh script, present in the probe/test folder, generating a traffic of 1000 packets/sec (how to use it is explained in the file itself).

- iperf -s -u -B 22.2.2.100 on VM on server2
- iperf -s -u -B 22.2.2.200 on MininetVM
- ./iperf_test.sh 1000 22.2.2.200 4 30 1 2 on VM on server 2
- ./iperf_test.sh 1000 22.2.2.100 4 30 1 2 on MininetVM

At the same time all the 3 probes are started through the start_capture.sh script.

- ./start_capture.sh

From this moment on, the simulation is running and following the settings in the configuration file config_filter.json, the probes are waiting for marked packets to arrive on the various emulated and non-emulated interfaces. When this happens, the first raw measurements are calculated, based on timestamps, hashes, identification fields and at the end of a period when the probe finishes analyzing the data coming from a specific interface, these are sent through the kafka-python library, to the Kafka brokers (which in this case are all on Server 3) following the schema reported in chapter 6.

Bibliography

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

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

[3] M. Cociglio, C. Corbo, G. Fioccola, M. Nilo, and R. Sisto. «The Big Data Approach for Multipoint Alternate Marking method». *IETF draft*. Vol. draftc2f-ippm-big-data-alt-mark-00. March 2020. url: https://datatracker.ietf.org/doc/html/draft-c2f-ippm-big-data-alt-mark-01.

[4] G. Fioccola, M. Cociglio, A. Sapio, and R. Sisto. «Multipoint Alternate Marking method for passive and hybrid performance monitoring». *IETF: RFC 8889. url: https://www.rfc-editor.org/info/rfc8889*

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

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

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

[8] M. Cociglio, G. Fioccola, G. Marchetto, A. Sapio and R. Sisto, "Multipoint Passive Monitoring in Packet Networks," in IEEE/ACM Transactions on Networking, vol. 27, no. 6, pp. 2377-2390, Dec. 2019.

[9] N. Duffield, D. Chiou, B. Claise, A. Greenberg, E. Grossglauser, and J. Rexford. «A Framework for Packet Selection and Reporting». *IETF: RFC 5474*. Vol. DOI 10.17487/RFC5474. March 2009. url: https://www.rfceditor.org/info/rfc5474.

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

[11] http://mininet.org/

[12] https://it.wikipedia.org/wiki/OpenFlow

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

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

[15] https://iperf.fr/

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

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

[18] https://spark.apache.org/

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

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

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

[22] https://www.docker.com/

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

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

[25] https://github.com/iovisor/bcc

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

[27] Apache Kafka, http://kafka.apache.org/documentation.html

[28] Zookeeper, https://www.cloudkarafka.com/blog/cloudkarafka-what-is-zookeeper.html

[29] Kafka-Python, https://github.com/dpkp/kafka-python

[30] IPMininet, https://ipmininet.readthedocs.io/en/latest/getting_started.html

[31] Kafka Performance, https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines

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