POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Thesis

Evaluation of passive monitoring techniques in multipoint networks

Supervisor
prof. Riccardo Sisto

Co-Supervisor:
prof. Guido Marchetto
dott. Amedeo Sapio

Candidate
Federica Mesolella

April 2018

To my family.

Contents

List of Figures

Listings

Chapter 1

Introduction

Nowadays one of the biggest problem for Internet Service Providers is how to deal with excessive packet losses, end-to-end delay and inter-packet jitter. Most service providers' networks carry traffic that is very sensitive to these network metrics. Modern applications, such as voice and video streaming, are more sensible to changes in the transmission features of data networks. These applications don't perform well if the end-to-end loss between hosts is larger than a threshold value. In addition an excessive packet loss may make it difficult to support certain real-time applications. This is a problem because it can affect the user experience and the customer satisfaction. A lot of work on fault detection and connectivity verification has been done by IETF, the Internet Engineering Task Force, but only few works concern the performance monitoring. So, in this scenario, service providers need methodologies and tools to monitor and measure the network performance with a proper accuracy in order to guarantee the quality of service which determines the customer satisfaction. The lack of proper tools to measure packet loss with the desired accuracy provides an incentive to think and design a new method for the performance monitoring of live traffic. This method simple to implement and deploy is described in the next chapter. It is a passive performance monitoring technique, potentially applicable to any kind of packet-based traffic, including Ethernet, IP, and MPLS, both unicast and multicast. The advantages of this method are:

- Easy implementation: It can be implemented either by using features already available or by applying an optimized implementation for both legacy and newest technologies.
- Low computational effort: There is an insignificant additional load on processing.

- Accuracy: Accurate packet loss measurement thanks to passive measuments.
- Applicability: Potential applicability to any kind of packet-based or frame-based traffic: Ethernet, IP, MPLS, etc., and both unicast and multicast.
- Robustness: The method can tolerate out-of-order packets, and it's not based on "special" packets whose loss could have a negative impact.
- Interoperability: The features required to experiment and test the method are available on all current routing platforms. Both a centralized or distributed solution can be used.

1.1 Goal of the thesis

This thesis aims at evaluating the Alternate Marking method as a passive monitoring technique in multipoint networks. There are different methods to perform packet loss measurements on a real traffic flow. The first one consists in numbering flow packets so that each router that receives the flow can immediately detect a missing packet. This approach uses a sequence number into each packet so the devices must be able to extract the number and check it in real time. This operation is difficult to implement on live traffic. The second approach, instead, is to count packets sent on one end, count the packets received on the other end, and compare the two values. In order to compare two counters, it is required that they refer exactly to the same set of packets. Since a flow is continuous and cannot be stopped when a counter has to be read, it can be difficult to determine exactly when to read the counter. A possible solution to this problem is to virtually split the flow in consecutive blocks by periodically inserting a delimiter so that each counter refers exactly to the same block. The alternate marking method follows this second approach, but it doesn't use an additional packet to virtually split the flow in consecutive blocks. It "marks" the packets so that the packets belonging to the same block will have the same color, while the packets belonging to different blocks will have different colors. The alternate marking method is applicable only to a point-to-point path so the extension proposed in this thesis is the Multipoint Alternate Marking method that, instead, is applicable to multipoint-to-multipoint path. This method permits to measure any kind of flow whose packets can follow several different paths in the network. It was evaluated the accuracy of this method to measure only the packet loss in a network topolgy but average delay and jitter metrics can be also evaluated. An application of multipoint alternate marking methodology is in the context of software defined networks (SDN) paradigm.

The SDN Controllers are the brains of the network, they control the hosts, the switches and the routers in the network and manage all the operation of performance monitoring. In an SDN architecture there is a clear separation between the application layer, the control layer and the infrastructure layer. The main modules in our deployment are:

- The control program: an application that implements the multipoint alternate marking method using POX Python API.
- The POX SDN controller: the network controller that communicates with switches using the Openflow protocol.
- The network devices. In order to emulate a real network topology we use the Mininet Emulator that allows to create a network of virtual switches (using Open vSwitch), hosts, one or more controllers and links.

To check the accuracy of the controller measurements, in the program used to emulate the real network topology two different scripts are running: the Packet Generator and the Packet Collector. The first one, running on each host, starts two processes. One sends and the other one receives UDP packets to/from all the hosts in the network, so we have traffic all-to-all. The second one, instead, collects the reports received from all the packet generators and writes in a file the number of packets sent and received from host X to Y. Separately we run the controller program implementing the multipoint alternate marking technique that, after reading input and output counters, writes the values in a json file. The first tests are performed on two real network topologies: Geant and Bics. Geant is an European network for the research and education community. Bics, instead is a global service provider. It provides many different services such as voice, connectivity, messaging, roaming to more than a thousand fixed and wireless carriers and service providers. We take the information about each topology from The Internet Topology Zoo dataset. This is a project that provides data on network topologies from around the world. It was created from the information that network operators make public. We performed two separate tests. The first without emulating a certain percentage of losses in the network. The second, emulating a percentage of losses equals to 1% in all links between switches in the network. In both cases we started the tests with L=120s, where L is the bit marking interval, then we progressively reduced the duration of this interval to 60s, 30s and 20s. The traffic generation is always all-to-all but the monitored traffic changes. It will be one-to-all, for example from a specific node to all the other nodes, or all-to-all.

So we have some graphs that show the total number of packets lost for each link in the topology, in each monitoring period. In all the graphs plotted we can notice that the counters values are the same both in results.json and in summary.dat. This means that the controller measurements are correct. This is the first part of the thesis, the second part concerns the clustering. In the first part we have seen how to determine the number of packets lost globally in the monitored network, exploiting only the data provided by the counters in the input and output nodes. But we can also exploit the data provided by the other counters in the network to converge on smallest subnetworks, that we call clusters, where packet losses occur. The second part of the thesis concerns the selection of clusters, given a network topology. We present two possible algorithms. One recursive and one iterative. Applying the clustering algorithms to all the network topologies present in the dataset of The Internet Topology Zoo, we noticed that the interface-level graph is much larger than the physical graph. It was clear in all the plotted graphs in which the relation is close to linear. The last part of the thesis concerns the study of the clusters features in a real situation. In particular are examined the cluster features in three network topologies: Geant, Colt Telecom and Cogent Communications. One of small size and the other ones of the medium size. We noticed that in all the topologies the number of clusters increases when increasing the percentage of the selected monitored nodes. We also notice that the values are all quite close to the median in the box plot, therefore there is a little variability. Another important examined cluster's feature is the diameter. The diameter d of a graph is the greatest distance between any pair of vertices. To find the diameter of a generic graph, first we find the shortest path between each pair of vertices, then the greatest length of any of these paths is the diameter of the graph. We noticed that the cluster's diameter both in the extended and in monitored graph is most often equal to 1. This means that the dimension of a cluster is small and this is important because this means that it is simple to learn where packet losses occur. Another feature examined is the execution time. We measure the time to calculate the edges in the monitored graph (monitored_edges_time) and the time to calculate the clusters, (clustering_time). This one is calculated both with recursive and with iterative clustering algorithm.

This thesis is structured as follows:

- **Chapter 2:** Proposes an overview on the alternate marking method on which this thesis is based and on the other methods used to detect a packet missing on live traffic: the TCP Congestion Control and the ITU-T Y.1731(Ethernet OAM).

- **Chapter 3:** Exposes how the multipoint alternate marking method works and how to calculate the multipoint packet loss. In addition it introduces the clustering algorithms.
- **Chapter 4:** Proposes an overview on the SDN architecture on which this thesis is based and on technologies and tools used.
- **Chapter 5:** Exposes the tests performed on some real network topologies. Describes how to calculate the clusters on a real network topology. Exposes the clusters' features useful to evaluate the scalability and the efficiency of the clustering algorithms.
- **Chapter 6:** Exposes the conclusions and provides some projects that will follow this thesis.

Chapter 2

The Alternate Marking method

2.1 Overview

One of the methods used to detect a packet missing on live traffic consists in numbering the packets that a single router receives. This is difficult to implement because each packet must have a sequence number and the network equipments must be able to extract and check this, in real time. An example is the TCP congestion control. Another method, simple to implement, consists to count the number of packet sent and received and compare the two values. The problem is that the devices must be in sync in order to read the counter of the same set of packets. To do this operation, the flow data is virtually split in consecutive blocks. So, each counter reads the same set of packets. The delimiter of these blocks could be a special packet, inserted artificially into the flow, called the OAM packet.

2.1.1 TCP Congestion Control

The TCP congestion control is a feature that allows to limit the amount of data transmitted on the network in the form of packets, adapting the data flow sent to the possible congestion state of the network. The four congestion control algorithms are respectively: slow start, congestion avoidance, fast retransmit and fast recovery[1]. The slow start and the congestion avoidance algorithms must be used by a TCP sender to control the amount of pending data being injected into the network. Instead, the Fast retransmit and Fast recovery (FRR) is used in order to quickly recover the lost data packets in the network. There are two main reasons that cause loss of packets: the transmission errors that generally affect a single packet or a congestion problem that causes the loss of more consecutive packets. The FRR algorithm works in this way.

A TCP receiver should send an immediate duplicate ACK when arrives an out-of-order segment in order to give to sender the information about the sequence number and the out of order incoming. Duplicate acknowledgement is the basis of FRR. It can be caused by a lot of problems. For example, by dropped segments, by the re-ordering of data segments and by replication of ACK or data segments. The "fast retransmit" algorithm knows, from the incoming of 3 duplicate ACKs, that a segment was lost. So, after receiving 3 duplicate ACKs, TCP retransmits the missing segment. After this operation, the "fast recovery" algorithm performs the transmission of new data until a non-duplicate ACK arrives. The FRR algorithm works most efficiently when there are isolated packet losses, instead, when there are multiple data packet losses in a short period time, it is not so efficient.

2.1.2 ITU-T Y.1731(Ethernet OAM)

To perform the packet loss monitoring, the International Telecommunication Union (ITU), the Institute of Electrical and Electronics Engineers (IEEE) and the Metro Ethernet Forum (MEF) had defined protocols and standards for Ethernet operations, administration, and maintenance (OAM) including ITU-T Y.1731[2]. Ethernet OAM provides fault and performance management mechanisms. In particular, it allows to detect the network connectivity, to locate faults on the network and to measure network trasmission parameters like packet loss jitter and delay. ITU-T Y.1731 is a service-level OAM mechanism that provides fault management functions. It is applicable to the Ethernet networks and services and supports a set of OAM tools for fault management and performance monitoring. In particular, it provides procedures and messages in order to perform the packet loss measurement and the on-demand (one/two-way) frame delay and delay variation measurements. The operations, administration, and maintenance (OAM) function for Y.1731 performance monitoring measures the following performance parameters: the Frame Loss Ratio, the Frame Delay and the Frame-Delay Variation. These performance parameters are defined for point-to-point Ethernet connections. The frame loss ratio (FLR) parameter, for example, is expressed as a percentage of the number of undelivered service frames divided by the total number of service frames during the time interval T. So, the number of undelivered service frames is the difference between the number of service frames arriving at the ingress Ethernet flow point (EFP) and the number of service frames delivered at the egress Ethernet flow point EFP, in a point-to-point connection. These performance parameters are applicable to service frames that are acknowledged at the ingress Ethernet flow point and should be delivered to the egress one.

In order to separate the packets blocks, the OAM packets are inserted, so the OAM packets are used as delimiters and boundaries between the blocks. In these OAM packets, the packet counters of the blocks created and the timestamp of the packet are inserted to allow the measurement of packet loss and latency. So, the difference between the alternate marking method and this one is that in the first method the blocks are created by coloring/marking packets, instead, in this case the blocks are created by delimiting them from special packets of OAM. Obviously the alternate marking method, as we can see after, is much better because the idea of inserting the OAM packets forces you to use a powerful hardware. So, if the packets move during the path you can have errors in the measurements.

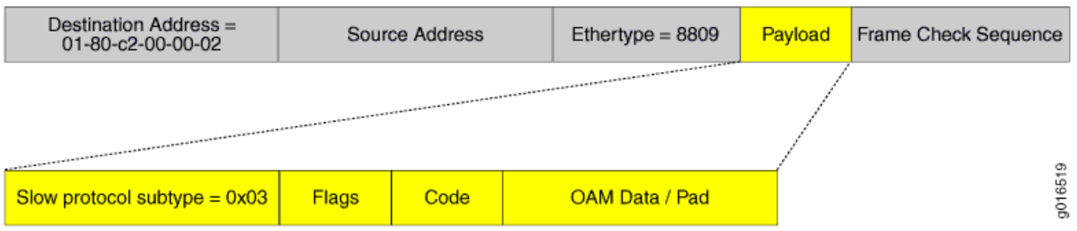


Figure 2.1. OAM message.[3]

2.1.3 Alternate marking method

The alternate marking method is similar to the second approach, ITU-T Y.1731, Ethernet OAM, but the difference is that the flow data is not virtually split in consecutive blocks by the addition of the other packets. It consists to mark the packets belonging to the same block with a color and the packet belonging to the other blocks with another color.

```
                      Traffic flow
  ========================================================>
    +------+       +------+       +------+       +------+
---<>  R1  <>-----<>  R2  <>-----<>  R3  <>-----<>  R4  <>---
    +------+       +------+       +------+       +------+
    .              .      .              .       .      .
    .              .      .              .       .      .
    .              <------>              <------->      .
    .          Node Packet Loss      Link Packet Loss   .
    .                                                   .
    <--------------------------------------------------->
                     End-to-End Packet loss
```

Figure 2.2. Measurements on different network segments.[4]

In this way the packets belonging to consecutive blocks have different colors[4]. In the Fig. 2.2 we can see the application of the method to the different network segments in order to perform the link monitoring, the node monitoring or the end-to-end monitoring.

2.2 How the method works

The alternate marking method, as we have seen before, is based on the idea to split the real traffic in consecutive different colored blocks. In this way it is possible to measure the packet loss counting the packets in each block along the path between two end points. To split the traffic the basic idea is coloring the blocks, so that the packets with the same color belonging to the same block and those with different colors belonging to different blocks. Let's assume we want to monitor the traffic between two routers R1 and R2, Fig. 2.3.

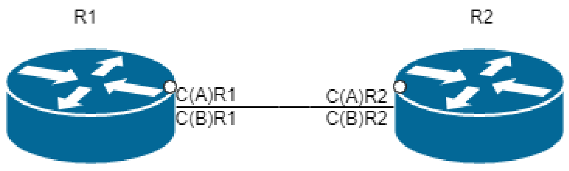


Figure 2.3. Monitoring the packet loss on link between R1-R2.

The real traffic between two routers is colored with two different colors: A and B. So, in this way, each block has a different color and the transition from a color A to a color B produces a square wave signal as we can see in the Fig. 2.4 below.

```
Color A   ----------+           +-----------+           +----------
                    |           |           |           |
Color B             +-----------+           +-----------+
           Block n        ...      Block 3     Block 2     Block 1
         <---------> <---------> <---------> <---------> <--------->

                             Traffic flow
         ============================================================>
Color ...AAAAAAAAAAA BBBBBBBBBBB AAAAAAAAAAA BBBBBBBBBBB AAAAAAA...
         ============================================================>
```

Figure 2.4. Packet coloring technique. [4]

The R1 router keeps, on its egress interfaces, two counters per interface, for each value of the marking bit: C(A)R1, that counts the packets with color A and C(B)R1, that counts those with color B. In the same way, the R2 router keeps, on its ingress interfaces, two other counters per interface, for each value of the marking bit. The packet loss within the block, for example the block colored with A, is calculated comparing the respectively counters of the R1 and R2 routers. The reading of counters is done not immediately. Because some packets can arrive out of order and we read wrong values of counters. So it is useful waiting for some seconds after the packet coloring. If L is the duration of the interval of the packet marking, the duration of the waiting time, before reading the counters, is equal to L/2, without considering clock error. In this way it is reduced the possibility that a packet could arrive later. The packet coloring/marking is important in order to equally mark all packets belong to the same block. In order to do this operation it is used a single bit of the ip packet header. The choice of the marking field depends on the specific application. For example a possible choice is to use the bit of the TOS field of the ip header. In the packets belong to the same block this bit has the same value, instead in the packets belong to the consecutive blocks this bit has different values. The value of this single bit changes periodically at regular time intervals of L seconds. So two counters are needed. One for each value of the bit marking. This operation requires a synchronization of the clock of the involved devices. In particular if we consider the R1 and R2 routers, their measurements are compared. If they have a maximum clock displacement of A seconds, such that:

$$L > 2(A + Dmax - Dmin) \tag{2.1}$$

where $Dmax$ and $Dmin$ are the lower and the upper bounds of the network delay between two envolved routers, the duration L of a marking block is chosen considering the maximum synchronization error and the maximum and minimum delay between two network equipments [5].

```
...BBBBBBBBB | AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA | BBBBBBBBB...
             |<====================================>|
             |                  L                   |
...=========>|<=================><=================>|<==========...
             |       L/2                 L/2        |
             |<===>|                          |<===>|
                d  |                          |  d
                   |<========================>|
                    available counting interval
```

Figure 2.5. The duration of the packet marking.[4]

Chapter 3

The Multipoint Alternate Marking method

The alternate marking method, as we have seen in the previous chapter, can be applied only to the point-to-point flows. It supposes that all the packets of the data flow measured on one node are measured again by a second node. In this chapter we describe a new methodology for passive performance monitoring. This permits to measure not only the point-to-point flows but any kind of unicast flow. A flow is defined as a set of packets having common features with regard to RFC 7011 [RFC7011] [6]. For example a flow could be a set of packets having the same source ip address or the same destination ip address. In particular, a flow is defined as a set of selection rules used in order to match the packets processed by the network equipments. An application of multipoint alternate marking methodology is in the context of software defined networks (SDN) paradigm where the SDN Controllers are the brains of the network, they control the hosts, the switches and the routers in the network and manage all the operation of performance monitoring.

3.1 Multipoint packet loss

Every interface of a single node must keep four counters, two on the ingress and the other two on the egress interfaces. Since all the packets of a specific flow that leave the network have previously entered the network, the number of packets counted by all the input nodes is always greater or equal than the number of packets counted by all the output nodes. Instead, if there is no packet loss in the network the number of packets is the same on all the ingress and the egress interfaces [7]. In a network the number of lost packets is equal to the number of packets counted by the input nodes minus the number of packets counted by the output nodes.

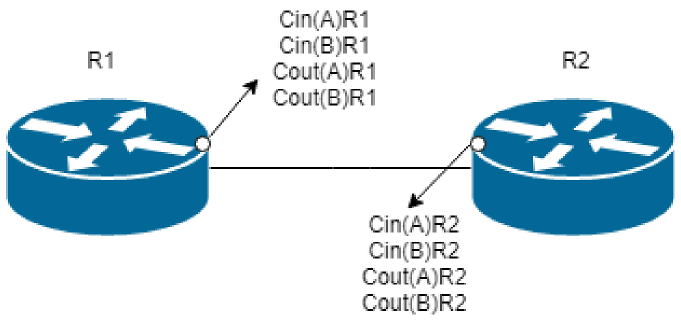


Figure 3.1. Packet couters for each interface.

In particular, if we considered n input nodes and m output nodes the Monitored Network Packet Loss is that:

$$PL = (PI1 + PI2 + ... + PIn) - (POi + PO2 + ... + POm) \tag{3.1}$$

Where PL is the Network Packet Loss. PIi is the number of packets flowed through the i-th input node in this period. POj is the number of packets flowed through the j-th output node in this period [7].

3.2 Network clustering

The equation (3.1) shows the network packet loss defined as the number of the packets globally lost in the monitoring network. This was done considering only the counters of the input and the output nodes. In order to localize where the losses occur we can consider the counters of the global monitoring network or we can split the network in the smallest subnetworks and considering the counters of only these subnetworks. These subnetworks are called clusters. In particular a cluster is defined as a subnetwork of the monitoring network graph that meet fully the network packet loss equation (3.1). However in this case the PL value is the number of packets lost in the cluster, not in the global topology. So, a cluster should contain all the arcs coming from its input nodes and all the arcs ending in its output nodes. This guarantees that we can count all the packets leaving an input node again at the output node regardless of the path followed. In a monitored network, where every interface is monitored, each node corresponds to a cluster and each physical link corresponds to two clusters, one for each direction. The clusters can have different sizes depending on the flow filtering criteria adopted.

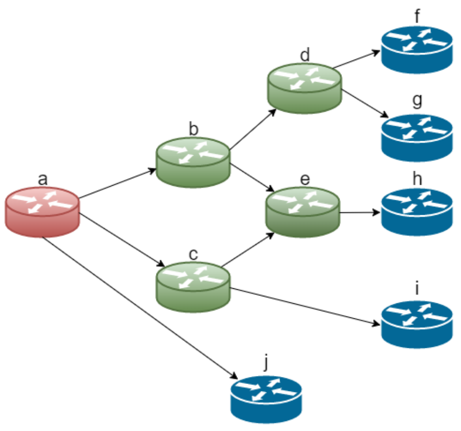


Figure 3.2. The monitoring network.

For example considering the monitored network in the Fig. 3.2 we have respectively that the input, the output and the intermidiate measurement points are coloured red, green and blu. So, in order to localize the losses we can split the monitoring network in the smallest subnetworks, maintaining the packet loss property for each subnetwork. We call these subnetworks clusters. From this monitoring network, for example, we can derived four clusters with only input and output measurement points, as we can see in the Fig. 3.3. In the two nodes cluster the loss is on the phisical link (Cluster 4). Instead, in the other clusters, the loss is on the cluster itself but we cannot know in which phisical link (Cluster 1, 2, 3).

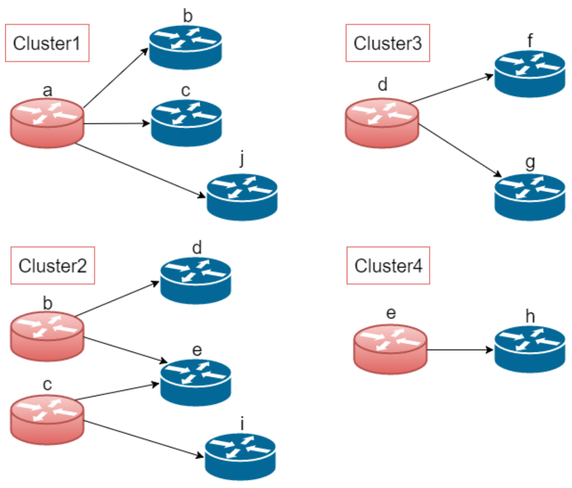


Figure 3.3. Clusters.

3.2.1 Clustering algorithm

We have to localize where the losses occur, so we use a 2 steps algorithm to split the network in clusters.

- We group the links (x->y) where there is the same left node.
- We join the link groups with at least 1 right node in common.

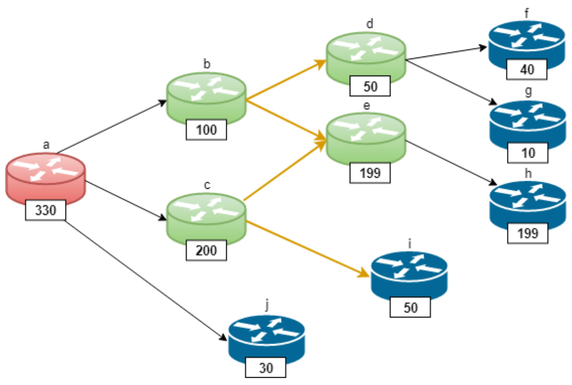


Figure 3.4. The clustering algorithm.

So, as we can see in the Fig. 3.4, we have 4 clusters (link groups). The number below each node is the packets number of a flow in a period:

- Cluster 1: (a(330)->b(100), a(330)->c(200), a(330)->j(30))
- Cluster 2: (b(100)->d(50), b(100)->e(199), c(200)->e(199), c(200)->i(50))
- Cluster 3: (d(50)->f(40), d(50)->g(10))
- Cluster 4: e(199)->h(199)()

In order to find the cluster Packet Loss firstly we sum the right numbers (counting once each node), then we do the same with the left numbers, and at the end we subtract the first result to the second one:

- PL Cluster 1= 330-(100+200+30)=0
- PL Cluster 2= (100+200)-(50+199+50)=1
- PL Cluster 3= 50-(40+10)=0
- PL Cluster 4= 199-199=0

One packet was lost in the Cluster 2 beacuse PL= [330-(40+10+199+50+30)]=1. We know that the loss occurred on yellow links but we don't know on which.

3.2.2 How the clustering works

In the previous section we have seen that we can determine the number of packets lost globally in the monitored network, exploiting only the data provided by the counters in the input and output nodes (3.1). In addition we can also exploit the data provided by the other counters in the network to converge on the smallest subnetworks, called clusters, where packet losses occur [5]. A cluster $\tilde{G} = (\tilde{N}, \tilde{A})$ is a subnetwork of $\bar{G}$ that still satisfies the packet loss equation but PL in this case is the number of packets lost in the cluster and not in the network. A cluster contains all the arcs emanating from its input nodes and all the arcs terminating at its output nodes. This guarantees that we can count all the packets exiting an input node again at the output node, regardless of the path they follow.
A monitored network is based on the assumption that each network interface can separately count packets in both the incoming and the outgoing directions. The monitored network is modelled as a directed network $G = (N, A)$ defined by a set N of n nodes and a set A of m directed arcs. Each network interface i is modelled by two nodes, $i_i \in N$ and $i_o \in N$, corresponding to the incoming and the outgoing directions, respectively.
Each physical link connecting two interfaces i and j , is modelled by two directed arcs, $(i_o, j_i) \in A$ and $(j_o, i_i) \in A$, corresponding to the two directions. As a result, if there are p links in the network and q network devices in total, each one with r_i interfaces, with $i \in \{1, ...q\}$, the number of nodes n and the number of arcs m are respectively:

$$n = 2 \times \sum_{i=1}^{q} r_i \tag{3.2}$$

$$m = 2 \times q + \sum_{i=1}^{q} r_i^2 \tag{3.3}$$

All the border interfaces of the monitored network must be configured to count respectively the incoming and the outgoing packets that match the chosen identification fields. But only a subset of the internal network interfaces can be selected for traffic minitoring.
So, we consider the directed network $\bar{G} = (\bar{N}, \bar{A})$ where:

- $\bar{N}$ is the subset of N containing only the nodes corrisponding to a monitored interface.
- $\bar{A}$ is a set of arcs, where the $arcs(i, j) \in \bar{A}$ corresponds to a possible directed path in G between two nodes $i \in \bar{N} \subseteq N$ and $j \in \bar{N} \subseteq N$, not crossing any other node in $\bar{N}$.

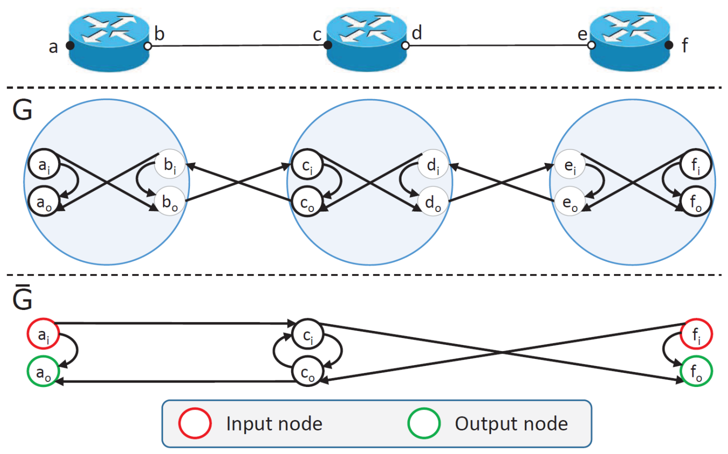


Figure 3.5. A simple netwok model. [5]

The Fig. 3.5 shows the directed network of a simple network with only 3 devices and 3 monitored interfaces. The set of input nodes of $\bar{G}$ are defined as the set of nodes in $\bar{N}$ without incoming arcs:

- $I = \{j \in \bar{N} \mid \nexists(i,j) \in \bar{A} \forall i \in \bar{N}\}$

The set of output nodes of $\bar{G}$ as the set of nodes in $\bar{N}$ without outgoing arcs:

- $O = \{i \in \bar{N} \mid \nexists(i,j) \in \bar{A} \forall j \in \bar{N}\}$

These sets are $I = \{a_i, f_i\}$ and $O = \{a_o, f_o\}$. In a monitored network, each network device corresponds to a cluster and each physical link corresponds to two clusters for each direction. In fact, when two monitored interfaces i and j are the endpoints of a physical link, one cluster would be made of $\bar{N} = \{i_o, j_i\}$ and $\bar{A} = \{i_o, j_i\}$, while the other cluster would have $\bar{N} = \{i_i, j_o\}$ and $\bar{A} = \{j_o, i_i\}$. This happens because on a physical link, all the packets that leave one interface either reach the other endpoint or are lost.
The clustering algorithm is that:

- Compose the sets $\bar{A}_i \forall i \in \bar{N}$ containing all the arcs $(i,j) \in \bar{A}$ with the common tail $i \in \bar{N}$.
- Join all the sets $\bar{A}_i$ and $\bar{A}_j$ containing at least one arc $(i,z) \in \bar{A}_i$ and one arc $(j,z) \in \bar{A}_j$ with a common head $z \in \bar{N}$.
- Each one of the composed sets of arcs, together with the nodes that acts as their endpoints, constitutes a cluster.

The corresponding pseudocode is presented below [5]:

Algorithm 1 Clustering algorithm

```
Require: $\bar{N}$ and $\bar{A}$

 1: $\bar{n} \leftarrow |\bar{N}|$
 2: for all $(i,j) \in \bar{A}$ do                    ▷ Group arcs by the tail
 3:     $\tilde{A}_i \leftarrow \tilde{A}_i \cup \{(i,j)\}$
 4: end for

 5: for all $\tilde{A}_i$ do
 6:     for all $(i,j) \in \tilde{A}_i$ do
 7:         for $i' \leftarrow i+1, \bar{n}$ do
 8:             for all $(i',j') \in \tilde{A}_{i'}$ do
 9:                 if $j = j'$ then
10:                     $\tilde{A}_i \leftarrow \tilde{A}_i \cup \tilde{A}_{i'}$
11:                     $\tilde{A}_{i'} \leftarrow \emptyset$
12:                     break                   ▷ Go to the next $\tilde{A}_{i'}$
13:                 end if
14:             end for
15:         end for
16:     end for
17: end for
```

The six line in the pseudocode above, should iterate also on the arcs of $\tilde{A}_i$ added in the process. When each cluster is made of only one arc, in the worst case, the clustering algorithm needs:

- $\bar{m}$ iterations, where $\bar{m}$ is the number of arcs in $\bar{A}$, in order to make the groups according to the arc tail.
- these iterations in order to test if any groups should be joined.

$$\sum_{i=1}^{\bar{m}-1} (\bar{m} - i) \tag{3.4}$$

So, the total number of iterations in the worst case is:

$$\bar{m} + \sum_{i=1}^{\bar{m}-1} (\bar{m} - i) = \bar{m} + \frac{\bar{m}(\bar{m}-1)}{2} \tag{3.5}$$

and the complexity is equls to $O(\bar{m}^2)$.

It is possible to use a second approach in order to identify the clusters in the network $\bar{G}$. This is the recursive clustering algorithm and consists in the using of the node-node adjacency matrix representation. This rappresents the network $\bar{G}$ as an $\bar{n} \times \bar{n}$ matrix $M = \{h_i j\})$. The matrix has a row and a column corresponding to each node in the network, and its entry $h_i j$ is equals to 1 if $(i, j) \in \bar{A}$ otherwise it is equals to 0. The matrix has $\bar{n}^2$ elements, where only $\bar{m}$ of which are different from 0. We can obtain the arcs emanating from node i by scanning the i-th row: if the j-th element in this row has a nonzero entry, (i, j) is an arc of the network $\bar{G}$. Instead, we can obtain the arcs entering node j by scanning the j-th column: if the i-th element of this column has a nonzero entry, (i, j) is an arc of the network $\bar{G}$. Using the adjacency matrix, for each arc $(i, j) \in \bar{A}$ we can make a cluster by grouping all the outgoing arcs of i, all the incoming arcs of j and, recursively, all the outgoing and incoming arcs of the new added nodes. The pseudocode is below [5].

Algorithm 2 Recursive clustering algorithm

```
Require: $\bar{N}$ and $M = \{h_{ij}\}$                    ▷ Node-node adjacency matrix

 1: for all $i \in \bar{N}$ do
 2:     $Current \leftarrow \emptyset$
 3:     InspectRow($i$)
 4:     $NewCluster \leftarrow Current$
 5: end for

 6: function InspectRow($i$)
 7:     for all $j \in \bar{N}$ do                                  ▷ Scan row $i$
 8:         if $h_{ij} = 1$ then
 9:             $Current \leftarrow Current \cup \{(i, j)\}$
10:             $h_{ij} \leftarrow 0$
11:             for all $i' \in \bar{N}$ do                         ▷ Scan column $j$
12:                 if $h_{i'j} = 1$ then
13:                     $Current \leftarrow Current \cup \{(i', j)\}$
14:                     $h_{i'j} \leftarrow 0$
15:                     InspectRow($i'$)
16:                 end if
17:             end for
18:         end if
19:     end for
20: end function
```

This algorithm needs a full scan of the adjacency matrix M ($\bar{n}^2$ operations) and a row or column scan for every arc $(i,j) \in \bar{A}$. The total number of iterations is, in any case:

$$\bar{n}^2 + \bar{n} \times \bar{m} \tag{3.6}$$

If $\bar{m} \gg \bar{n}$ the complexity is $O(\bar{n} \times \bar{m})$, so this means that the recursive algorithm is better than the previous one.

Extended graph

In this section it is provided a model of the monitored network where each network interface counts packets in both the incoming and the outgoing directions. In order to do this operation, first of all, it is modelled the monitored network as an extended network. This is a directed network G = (N,A) where N is a set of nodes and A is a set of directed arcs. As we have seen before, each network interface is modelled by two nodes corresponding to the incoming and the outgoing directions and each physical link, that connect two interfaces, is modelled by two directed arcs corresponding to the two directions. In order to model the routing process, we also consider one directed arcs for each ingress and egress node corresponding to two interfaces on the same physical device. This includes also some directed arcs that connect the ingress and egress nodes corresponding to the same physical interface, to allow a packet entering and leaving from the same interface.

```
#Physical Graph
graph= nx.read_graphml('Geant2012.graphml',str)
#Setup ingress, egress interfaces
#Setup external links
for edge in edges_iter(data=True):
        x=int(edge[0])
        y=int(edge[1])
        if x not in out_interfaces:
            out_interfaces[x]={}
        if y not in out_interfaces:
            out_interfaces[y]={}
        if x not in in_interfaces:
            in_interfaces[x]={}
        if y not in in_interfaces:
            in_interfaces[y]={}
        n0=(n,x,y,"out")
        n1=(n+1,y,x,"out")
        n2=(n+2,x,y,"in")
        n3=(n+3,y,x,"in")
        out_interfaces[x][y]=n0
        out_interfaces[y][x]=n1
        in_interfaces[x][y]=n2
        in_interfaces[y][x]=n3
        Enodes.extend([n0,n1,n2,n3])
        Eedges.extend([(n0,n3),(n1,n2)])
        n+=4
#Setup internal links
for node in in_interfaces:
    for intf1 in in_interfaces[node]:
        in_intf=in_interfaces[node][intf1]
        for intf2 in out_interfaces[node]:
            out_intf=out_interfaces[node][intf2]
            Eedges.append((in_intf,out_intf))
#Extended graph
extended_graph=nx.DiGraph()
extended_graph.add_nodes_from(Enodes)
extended_graph.add_edges_from(Eedges)
```

Listing 3.1. The extended graph.

Monitored graph

All the border interfaces of the extended network that we have seen before must be configured to count incoming and outgoing packets respectively. But only a subset of the internal network interfaces can be selected to monitor the traffic. For this it is created the monitored network $\bar{G} = (\bar{N}, \bar{A})$ starting from the extended network G = (N,A). So it's need to build another graph starting from the extended one. First of all we need to select a subset of the extended nodes: enodes. A possible choice is to select them randomly. For example, it is decided a percentage equals to 30% and the program randomly selects 30% of the nodes between the enodes. These selected are the nodes of the new monitored graph that we called mnodes. The edges of the new graph, medges, are all possible paths between two nodes in mnodes that don't cross other nodes in mnodes.

- Starting from enodes and eedges it is constructed the extended directed graph.
- For each pair of nodes (x, y) in mnodes, all the paths in the extended graph from x to y are calculated.
- Each of these paths that doesn't contain mnodes except x and y represents an edge (x, y) of medges.

So, as we can see in the code below, first of all, it is created the DiGraph() monitored_graph selecting only the enodes randomly.
The program randomly selects a percentage of the nodes between the enodes. This percentage changes at each iteration. Then these mnodes are added to the graph. After adding the monitored nodes to the monitored_graph, the same are removed from the extended graph. But when removing the nodes, automatically are removed the edges connected to the nodes, because you can't have pending links. So, before removing the nodes, it's need to save all the edges, in the extended graph, with a monitored node as one extremity. In this way, when adding n and d nodes to each cycle, are adding also these edges.

```
#Adding mnodes to the Monitored Graph
Mnodes = random.sample(Enodes,((len(Enodes)*rate)/100))
monitored_graph=nx.DiGraph()
monitored_graph.add_nodes_from(Mnodes)
#Save the links into the extended graph
links_to_mnodes=[]
for edge in extended_graph.edges():
    for mnode in Mnodes:
        if edge[0]==mnode or edge[1]==mnode:
            links_to_mnodes.append(edge)
#Remove From the Extended graph the monitored nodes
extended_graph.remove_nodes_from(Mnodes)
#Add edges to the monitored graph
for n in monitored_graph.nodes():
    for d in monitored_graph.nodes():
        if(n!=d):
            extended_graph.add_node(n)
            extended_graph.add_node(d)
            for link in links_to_mnodes:
                if link[0]==n or link[0]==d or link[1]==n or link[1]==d:
                    extended_graph.add_edge(link[0],link[1])
            path= my_has_path(extended_graph,n,d)
            if (len(path)>0):
                monitored_graph.add_edge(n,d)
                Dict[(n[0],d[0])]=path
            extended_graph.remove_node(n)
            extended_graph.remove_node(d)
```

Listing 3.2. The monitored graph.

Clusters

After the creation of the monitored graph, in order to identify the clusters in the network, the clustering algorithm is applied. In this case the iterative algorithm has been implemented. This algorithm is different from the recursive approach because it is used not the node-node adjacency matrix representation but a simple list with all the edges in the network in both the igress and egress directions.

Chapter 4

Scenario

4.1 SDN Network

The software defined networks (SDN) is a new paradigm to manage the networks. It is a new approach to cloud computing that allows to separate the control plane from the forwarding plane. This technology permits to improve network performance monitoring and network management because the network devices are programmable [8]. This is important because it permits to promote the innovation in the network management and to deploy a lot of new services using open network API without the need to modify the underlying hardware platform. The SDN suggests to centralize the network brain in one network device, the SDN controller, separating the forwarding process of packets (Data Plane) from the routing process (Control plane). The control plane is made up of one ore more controllers that are the brains of the network, they sends information to switches and routers and to applications via API. All switches are controlled by the same physical entity this means that the controller acts as a single point of failure for the whole network. A way to pass this problem is by connecting multiple controllers to a switch, allowing a backup controller in case of failure events. One of the most well-known protocols used by SDN controllers is openFlow. It is used for remote communication with network devices in order to determine the path of network packets across the switches [9].

The SDN architecture is :

- Directly programmable: Control plane is programmable.
- Agile: Control network wide traffic flow to satisfy changing needs.

- Centrally managed: The controller is centralized, appears as a single switch.
- Programmatically configured: Using open network API without the need to modify the underlying hardware platform. The programs don't depend on proprietary software.
- Open standards-based and vendor-neutral: Simple network design because not depend on the specific vendor or device or protocol.

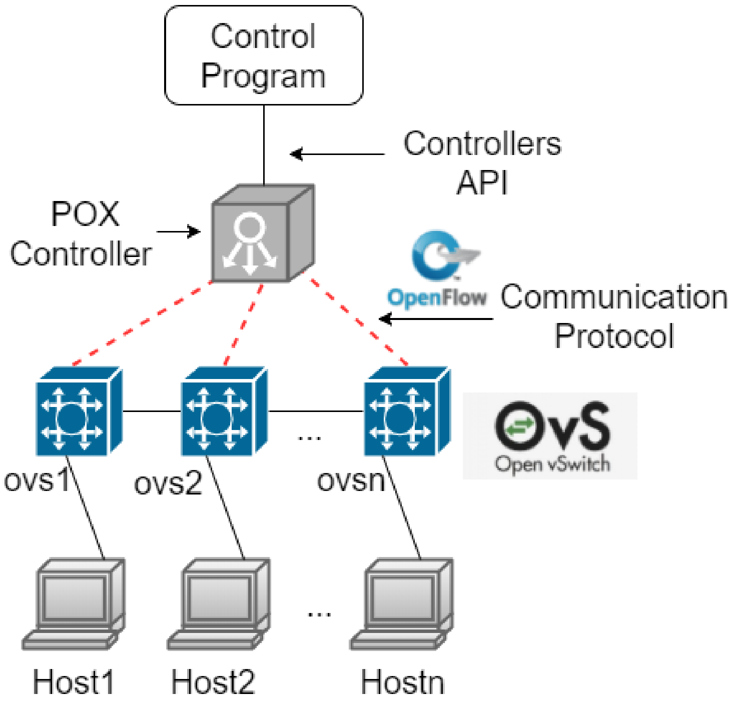


Figure 4.1. The SDN network architecture.

In the Fig. 4.1 it is showed the architecture of the SDN network in our deployment. It's clear the separation between the application layer, the control layer and the infrastructure layer. The control program is an application that implements the multipoint alternate marking method using, in this case, POX Python API. The controller SDN chosen is POX. It is the brain of the network that communicates with switches using the communication protocol openflow. Then there are the network devices. In order to emulate a real network topology we use the Mininet Emulator that allows to create a network of virtual switches (using open vSwitch), hosts, one or more controllers and links. The controller is the key point of the network, it allows users to write their own applications using the controller itself as an intermediary layer between the network applications and the network devices.

4.2 Openflow

OpenFlow is an instance of the SDN architecture, defined in the OpenFlow Switch Specification, published by the Open Networking Foundation (ONF). The ONF is a consortium of software providers, content delivery networks, and networking equipment vendors whose purpose is to promote software-defined networking [10]. It is a communication protocol that allows the communication between the controller and the network devices, in a software defined network. It is defined as the first standard communication interface between the control plane and the data plane of an SDN architecture. OpenFlow protocol allows direct access and manipulation of the forwarding plane of network devices both physical and virtual. It is important because in this way we no longer have monolithic and closed network devices but programmable devices whose control software is open source and locally managed [11]. The separation of the control plane from the forwarding plane allows a more sophisticated management of packet forwarding compared to the classic one that use only the access control lists and the classic routing protocols. The openFlow protocol is divided into four components [12]:

- **Message Layer**: The message layer is the core of the protocol stack. It defines the structure and the semantics for all messages used into the communication. Each openFlow message has the same header structure composed by 4 field and payload. The first is the version field, 8 bits, that indicates the version of the openFlow which this message belongs. The second is the length field, 16 bits, that indicates the message lenght. The third is the xid, 32 bits, it is a unique value used to match requests to responses. The type field, 8 bits, instead, indicates what type of message is present(Hello, EchoReq, EchoRes...) and how to interpreter the payload. It is version dependent.

- **State Machine**: The state machine defines the core low-level behavior of the protocol. Typically, it is used to describe some actions that need to be done before sending messages. OpenFlow has a simple finite machine model. Almost all the messages in this protocol are asynchronous, which does not require a state to handle. However, the connection establishment procedure need some actions as version and capability negotiation, capability discovery, flow control and delivery which has to be done before the messages can be exchanged.

- **System Interface**: System interface is the part of the openFlow protocol that provides services to the other components in the system.

It typically identifies some interfaces. The TLS and TCP interface that interacts with lower level protocols (TCP or TLV) in the protocol stack. The switch agent interface that interacts with the system kernel of an openFlow switch. It forwards messages from the controller to the switch's kernel. The controller application interface that interacts with controller applications running on the top of the openFlow stack. It accepts messages sent to switch by the controller application and forwards these to the openFlow stack for the processing and the transmission. The last is the configuration interface that provides services to system operator in order to configure the openFlow stack.

- **Configuration**: This layer covers anything about the configurations or setting of initial values in the protocol, from default buffer sizes and reply intervals to X.509 certificates an so on.
- **Data Model**: The openFlow data model is a set of relational structures that describe capabilities, configuration, and statistics for each openFlow abstraction.

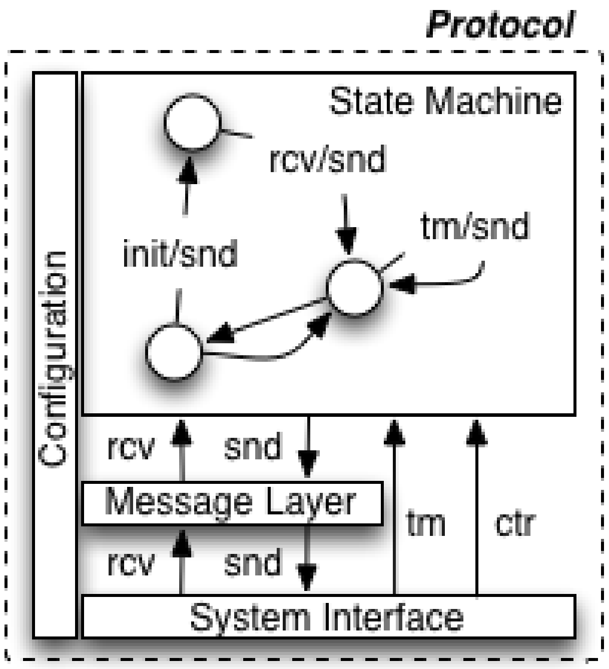


Figure 4.2. Openflow components.[12]

4.3 Open vSwitch design

The open vSwitch is commonly used as switch in a SDN network. It has two main components that work together in order to allow the packet forwarding. The first component is **ovs-vswitchd**, a userspace daemon that is indipendent from the operating system and the enviroment. The second component is a **kernel Datapath**, written specially for the host operating system [13]. The datapath module, in the kernel, receives, firstly, the packets from a physical NIC or a VM's virtual NIC. The ovs-vswitchd module ,instead, instructs or not the datapath how to handle packets of this type. In the former case, the datapath module simply follows the instructions, called actions, given by the ovs-vswitchd. It lists the physical ports on which send the packet or also specify the packet modifications, packet sampling, or packet droppping. When the datapath has not istruction on what to do with the packet, it delivers it to the ovs-vswitchd module. In userspace, the ovs-vswitchd determines how the packet should be handled, then it passes this to the datapath with the desired handling. The ovs-vswitchd also instructs the datapath to cache the action instructions, to handle similar packets in the future. The protocol used to control packet forwarding is openFlow. It allows a controller to add, drop, update, monitor, and obtain the flow statistics, to deviate selected packets to the controller and to send packets from the controller into the switch. The ovs-vswitchd receives openFlow flow tables from an SDN controller, matches any packets received from the datapath module with these openFlow tables, collects the actions applied, and finally caches the result in the kernel datapath [14]. This further simplifies the datapath module. OpenFlow controller doesn't know the caching mechanism and the separation into the user and the kernel space. In the controller's view, each packet visits a series of flow tables and the switch finds the highest priority flow whose conditions are satisfied by the packet and executes its openFlow actions [9].

4.4 Flow caching design

This section describes the design of the flow caching in the open vSwitch. In the ovs architecture, as we have seen in the previous section, the forwarding plane is composed by two modules. A **slow-path** userspace module called ovs-vswitchd and a **fast-path** module called kernel Datapath. Most of the complexity of the ovs is in the ovs-vswitchd module that handles all about the forwarding decisions and network protocol processing. The kernel module, instead, is responsable of only the caching traffic.

When a packet is received by the kernel module, it consults its flows cache. If an entry is found, there is an hit, so the actions are executed[9]. If there is a miss, it means that there is no entry, the packet is sent to ovs-vswitchd module that decides what to do. Then the ovs-vswitchd executes the pipeline on the packet, the actions, sends the packet iteself to the fast-path for forwarding, and at the end installs a flow cache entry to prevent the same expensive steps for the others packets. Until ovs 1.11 the kernel module was implemented as a microflow cache in which a single cache entry exact matches with all the packet header fields supported by the openFlow. In this design, the cache entries are extremely detailed and match at most packets of a single transport connection. For example, if it is considered a single transport connection, a change in the network path and in the IP TTL field would result in a miss. So it would send a packet to userspace, which consulted the actual openFlow flow table to decide how to forward it. This implies that the criticality is the flow setup time, the time, in the kernel view, to report a microflow "miss" to userspace and that for userspace to reply, limiting the global performance. The ovs 1.11, instead, introduced the megaflows. The microflow cache is retained as a first-level cache consulted before the megaflow cache. The megaflow cache is a single flow lookup table that supports generic matching, caching and forwarding decisions for larger aggregates of traffic. It does not have priorities and there is only one megaflow classifier, instead of a pipeline of them, so the userspace installs the megaflow entries that collapse together the behavior of all the openFlow tables. The introduction of the megaflows allowed the ovs to drastically reduce the number of packets sending to the slow path.

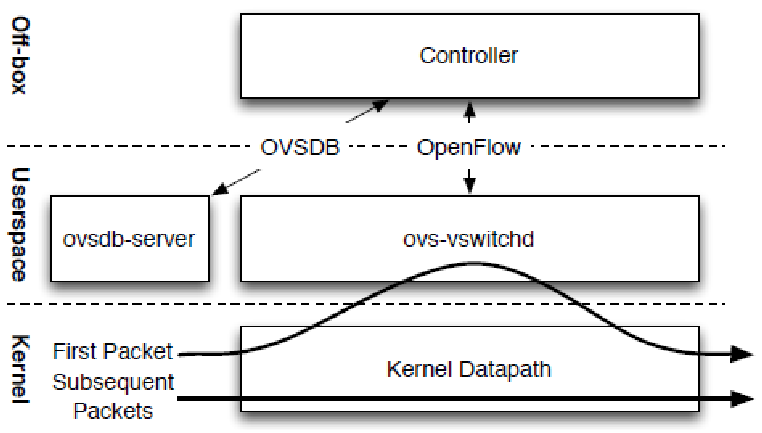


Figure 4.3. The components and interfaces of Open vSwitch.[9]

4.5 Mininet network emulator

To emulate a real network topology it is used the Mininet Emulator [15]. The Mininet Network Emulator allows to create a network of virtual switches (open vSwitch), hosts, one or more controllers and links. The hosts run a standard network software Linux-based, instead the switches support the openflow as communication protocol. The Mininet networks run real code including standard Unix/Linux network applications as the real Linux kernel and the network stack. Mininet has a lot of features compared to the others emulators, hardware testbeds, and simulators. If it is compared to full system virtualization based approaches, Mininet is:

- Boot faster: seconds and not minutes.
- Scalability: hundreds of hosts and switches.
- More Bandwidth: typically 2Gbps total bandwidth.
- Easy to install: a prepackaged VM is available with openFlow v1.0 tools already installed.

Instead, if it is compared to hardware testbeds, Mininet is:

- Inexpensive.
- Simply reconfigurable and restartable.

If it is compared to simulator, Mininet:

- Connects to real network topologies.
- Interactive performance.
- Runs real code. Application code, OS kernel code, openFlow controller code and open vSwitch code.

4.6 Emulation of real network topology

The first approach in order to emulate a real netwok topology is summarized in the Fig. 4.4. This configuration provides a controller c0 connected to N open vSwitches with point-to-point links and N open vSwitches connected to the controller and to the others switches and hosts.

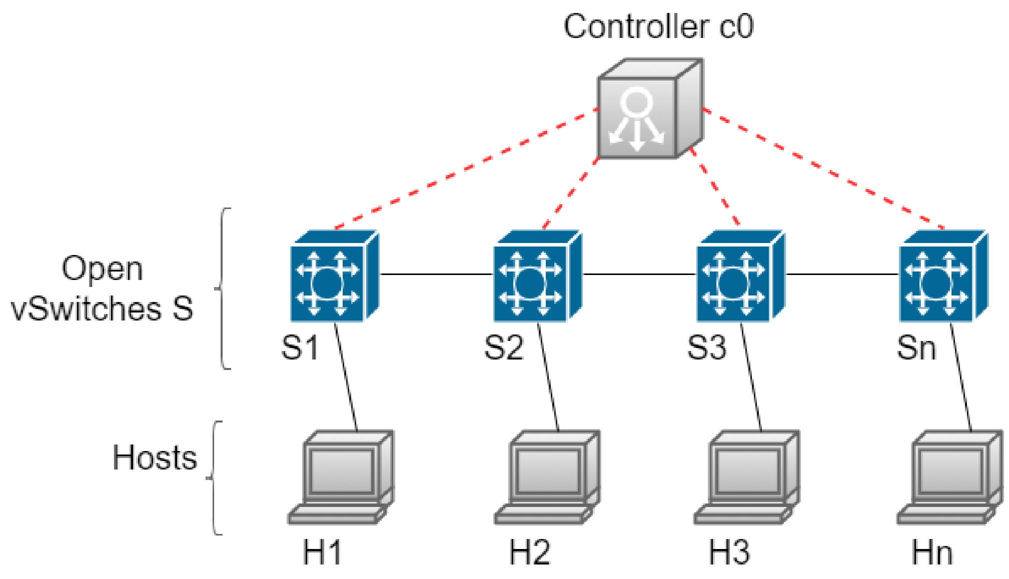


Figure 4.4. The first approach to emulate the network.

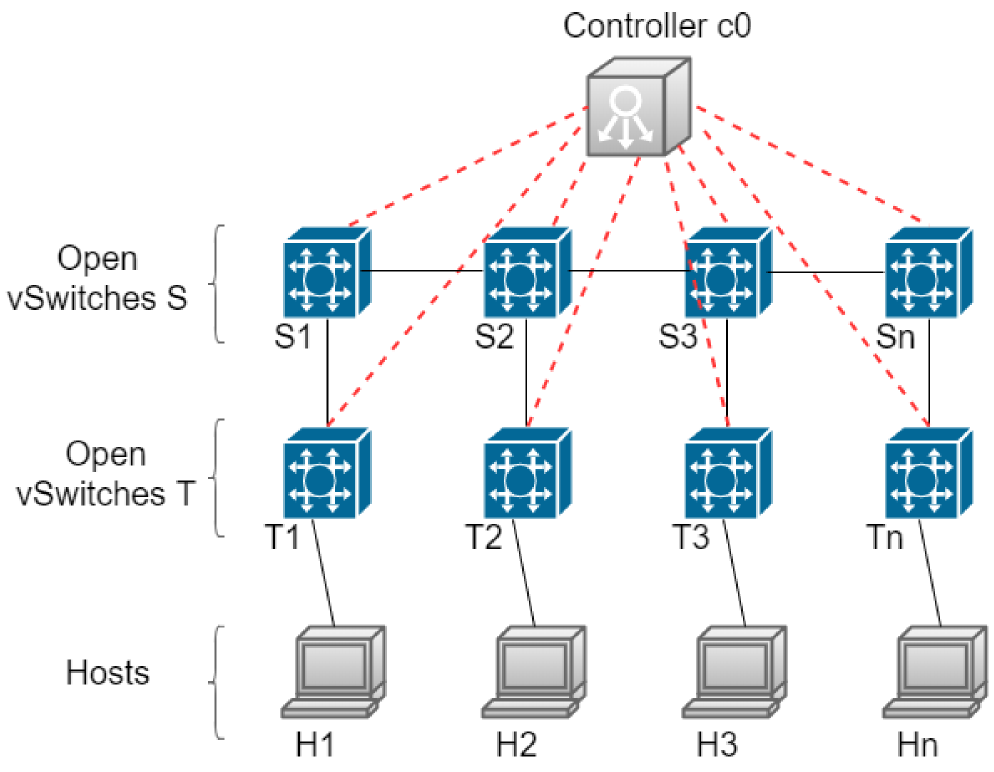


Figure 4.5. The second approach to emulate the network.

With this configuration it was observing an unexpected behavior. Newly marked packets were not always counted by the output counters.
This problem has been highlighted by considering the input and the output counters in two different switches connected with a point-to-point link. In fact, considering two switches s1 and s2, the output counters of s1 and the input counters of s2 didn't match. This problem is due to a flow caching mechanism presents into the ovs switches, as we have seen in the previous section. So, the second approach, that is the solution, consists in the distribution of the marking rules and the counting rules in separate switches. In fact, as we can see in the Fig. 4.5 there are the switches T in addition to the switches S. These are connected to the hosts and to the switches S using single point-to-point links.

4.7 The Internet Topology Zoo

The Internet Topology Zoo [16], is a project that allows to collect data network topologies from around the world. It is a dataset of network data created from the information that network operators make public. It is the most precise store of network topologies available. Now there are over two hundred and fifty networks in the Zoo, in a variety of graph formats for statistical analysis, plotting, or other network research. There are the network topologies of the most important internet service providers, ISPs, coming from all over the world, Europe, Asia, USA, Africa and so on. For example some ISPs are: Deutsche Telekom, Bell Canada, Sprint, ARPANET, GARR, BELNET, IBM, Hurricane Electric ad others.
The emulated topologies in Mininet are respectively GÉANT and BICS.

- **GÉANT 2012-03**: GÉANT [17] is the pan-European data network for the research and education community. It is the most advanced and well-connected research and education network in the world. It interconnects national research and education networks (NRENs) across Europe. It allows to connect Europe's researchers, academics and students to each other and linking them to a lot of countries in the world. The GÉANT project is a collaboration between 40 partners where each partner is a node in the topology.
- **BICS 2011-01**: BICS (previously known as Belgacom ICS or Belgacom International Carrier Services) [18] is a globally services provider. It provides a lot of different services such as voice, connectivity, messaging, roaming to more than a thousand fixed and wireless carriers and service providers. This topology includes 33 nodes between Europe and Asia [18].

4.8 Network topology setup

The tool used in order to emulate e real network topolgy is Mininet. The steps followed for the topology setup are:

- Reading information about the topology from the .graphml file.
  It is used the Python software package **NetworkX**. In particular it is used the **readgraphml()** method that allows to read a specific graph in graphml format from a path specified as first parameter. It is possible read information from different network topologies simply by changing the file name in the path. This file belongs to the Internet Topology Zoo's dataset seen in the previous section.

```
        G = nx.read_graphml('Bics.graphml',str)
```

- Creation of Mininet network.

```
       net = Mininet( topo=None,link=TCLink,
        autoStaticArp=True,
        build=False,
        ipBase='10.0.0.0/8')
```

- Adding remote controller c0 with ip 127.0.0.1 and port 6633.

```
       c0=net.addController(name='c0',
        controller=RemoteController,
        ip='127.0.0.1',
        protocol='tcp',
        port=6633)
```

- Adding ovsKernelSwitches S.

```
        for switch in G.nodes_iter():
            switch_id="s"+str(switch)
            id_node=int(switch)
            s=net.addSwitch(switch_id,
                dpid=hex(id_node)[2:],
                cls=OVSKernelSwitch)
```

- Adding ovsKernelSwitches T.

```
        for switch in G.nodes_iter():
            val=int(switch)+N
            switch_id="s"+str(val)
            id_node=int(switch)
            t=net.addSwitch(switch_id,
                cls=OVSKernelSwitch)
```

- Adding hosts H.

```
        for host in G.nodes_iter(data = True):
            host_id="h"+str(host[0])
            ris=int(host[0])+ int(10)
            host_mac="00:00:00:00:00:"+str(ris)
            host_ip="10.0.0."+str(host[0])
            h=net.addHost(host_id,cls=Host,
                ip=host_ip,
                mac=host_mac,
                defaultRoute=None)
```

- Adding links between ovsKernelSwitches S.
  The method called is **addLink()** passing as parameters the source and the destination node of a link itself. In addition, the links between nodes can also emulate a certain percentage of losses in the network. In this case it is necessary passing a third parameter, **loss**, that means the percentage of emulated losses.

```
        for edge in G.edges_iter():
            edge0_id="s"+edge[0]
            edge1_id="s"+edge[1]
            net.addLink(edge0_id,edge1_id,loss=1)
```

- Adding links between ovsKernelSwitches T and S.

```
        for nod in G.nodes_iter():
            val=int(nod)+N
            tswitch_ID="s"+str(val)
            switch_ID="s"+str(nod)
            port1=2
            port2=52
            net.addLink(tswitch_ID,switch_ID,port1,
                port2)
```

- Adding links between ovsKernelSwitches T and hosts.

```
        for hos in G.nodes_iter():
            val=int(hos)+N
            host_ID="h"+str(hos)
            tswitch_ID="s"+str(val)
            port1=51
            port2=1
            net.addLink(host_ID,tswitch_ID,port1,
                port2)
```

- Staring Mininet network.

```
        net.build()
```

- Staring controller.

```
        for controller in net.controllers:
            controller.start()
```

- Staring ovsKernelSwitches T and S.

```
        for sw in G.nodes_iter():
            s_id="s"+str(sw)
            net.get(s_id).start([c0])
        for sw in G.nodes_iter():
            w=int(sw)+N
            s_id="s"+str(w)
            net.get(s_id).start([c0])
```

- Running Packet Generator on each hosts in the topology.
  Running on each host, this script starts two processes: SENDER and RECEIVER. The receiver process receives UDP packets from all the hosts in the network and the sender process sends UDP packets to all hosts in the network, so we have traffic all-to-all. Before terminating, these two processes send a report to the Packet Collector with the number of packets sent and recv.

```
            H1.cmd('rm -r /tmp/pnpm')
            H1.cmd('screen -d -m python -m trace -t
                /home/fmesolella/Desktop/Server.py'+"
                   "+str(H1.IP())+" ")
```

- Running Packet Collector on a specific host, for example host H1.
  It collects the reports from all the Packet Generators and writes them in a file.

```
        for h in net.hosts:
            arrayIpHost.remove(h.IP())
            s=""
            for h1 in arrayIpHost:
                s=s+h1+" "
            arrayIpHost.append(h.IP())
            h.cmd('python
               /home/fmesolella/Desktop/Client.py'+"
               "+
                str(H1.IP())+" "+str(h.IP())+"
                   "+str(s)+" >> error.log 2>&1 &")
```

4.9 Controller accuracy

To check the accuracy of the controller measurements, in the program used to emulate the real network topology, as we have seen in the previous section, two different scripts are running. The first is the packet generator. When it is running, on each host in the topology, it starts two different processes:

- SENDER: This process sends UDP packets, with packet size equals to 42B to all the hosts in the network.
- RECEIVER: This process receives UDP packets, with packet size equals to 42B, from all the hosts in the network.

In this way all-to-all traffic is generated.

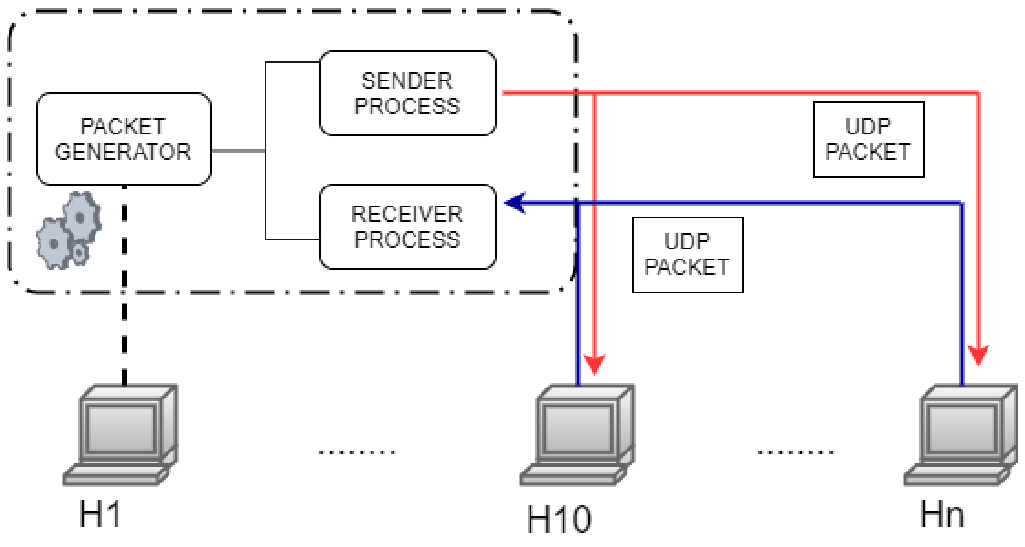


Figure 4.6. Packet Generator.

In the sender() function the tasks performed are respectively:

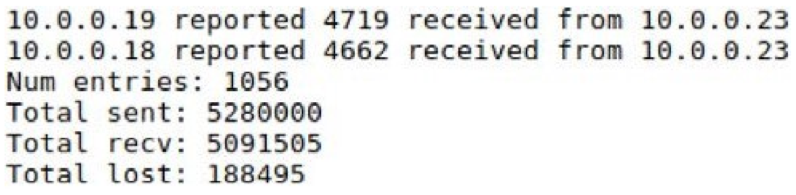
- The connection of the socket to the specified endpoint.
- The creation of the UDP socket. Since udp sockets are not connected sockets, the communication is done using the sendto() method. This function doesn't require the connection of the socket to the peer. This just sends directly to a given address.
- The sending of the UDP packets.

- The close of the socket.

Instead, in the receiver() function the task performed are:

- The connection of the socket to the specified endpoint.
- The creation of the UDP socket. The communication is done using the recvfrom() method. This function doesn't require the connection of the socket to the peer. This just recvs directly from a given address.
- The binding of the socket to the local host and port.
- The receiving of the UDP packets.

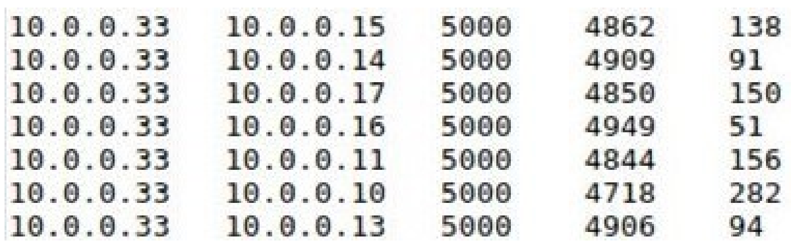
The second script, instead, is the packet collector. Before terminating, the sender and the receiver processes send a report to the packet collector with the number of packets sent and received. It collects the reports from all the packet generators and writes in a file the number of packets sent and received from the host X to Y and the number of the lost packets calculating the difference between the packet sent and the packet received. The collector generates two files. Summary.dat has the total number of packets sent, recv and lost.



```
10.0.0.19 reported 4719 received from 10.0.0.23
10.0.0.18 reported 4662 received from 10.0.0.23
Num entries: 1056
Total sent: 5280000
Total recv: 5091505
Total lost: 188495
```

Figure 4.7. Summary.dat file.

Results.dat has the information about the packets sent, recv and lost for each pair source and destination ip address.



```
10.0.0.33    10.0.0.15    5000    4862    138
10.0.0.33    10.0.0.14    5000    4909    91
10.0.0.33    10.0.0.17    5000    4850    150
10.0.0.33    10.0.0.16    5000    4949    51
10.0.0.33    10.0.0.11    5000    4844    156
10.0.0.33    10.0.0.10    5000    4718    282
10.0.0.33    10.0.0.13    5000    4906    94
```

Figure 4.8. Results.dat file.

In the first listing is showed the sender function. To manage the operation of sending messages it is used the ZeroMQ framework.
It looks like an embeddable networking library but it acts like a concurrency framework. It gives you sockets that carry atomic messages across various transports like in-process, inter-process, TCP, and multicast.
Zeromq opens the tcp connections so the created messages are sent by zeromq within these connections.
The packet generator starts and sends the SYN message to say that it has started, it does the experiment and then it sends, with the RECV message, the number of packets received and, with the SENT message, the number of packets sent. At the end, it sends the FIN message and ends.
The packet collector, instead, receives these messages and for each pair of packet generators it calculates the difference between the packets sent and received and then the losses.

```
def sender(dests,host):
    for i in xrange(0,NUM_PKTS):
        for d in dests:
                udp_sock.sendto("", (d,UDP_PORT))
                num_sent[d]+=1
                sleep(interpacket_gap)
```

Listing 4.1. Packet generator: sender function

In the second listing is showed the receiver function.

```
def receiver(stop,host):
    while (not stop.value):
        ready = select.select([udp_sock], [], [], 10)
        if ready[0]:
            msg, (src_ip, src_port) =
               udp_sock.recvfrom(1500) #Buffer size is
               1500 bytes
            if src_ip not in srcs:
                srcs[src_ip]=1
            else:
                srcs[src_ip]+=1
```

Listing 4.2. Packet generator: receiver function

4.10 Controller program

The controller program is a network application that uses the POX software definded network controller. It is a Python program that uses the POX API in order to implement the multipoint alternate marking method. As we have seen in the chapter 3 all the packets of a specific flow that leave the network have previously entered the network. So, in a network the number of lost packets is equal to the number of packets counted by the input nodes minus the number of packets counted by the output nodes. If L is the duration of the interval of the packet marking, the duration of the waiting time, before reading the counters, is equal to L/2. This means that he reading of the input and output counters of all the switches into the network is done at L/2. The packet marking is alternated. It means that if L = 60s, as we can see in the Fig. 4.9, when t = 60s the tos flag is marked to 0, when t = 120s it is marked to 1, when t = 180s it is marked to 0 and so on. So, at L/2 I will read the counters with the tos bit marked in the previous period. It means that when t = 60s the tos bit is marked to 0, so when t = 90s I will read 1, equally when t = 120s tos bit is marked to 1, so when t = 150 I will read 0 and so on.

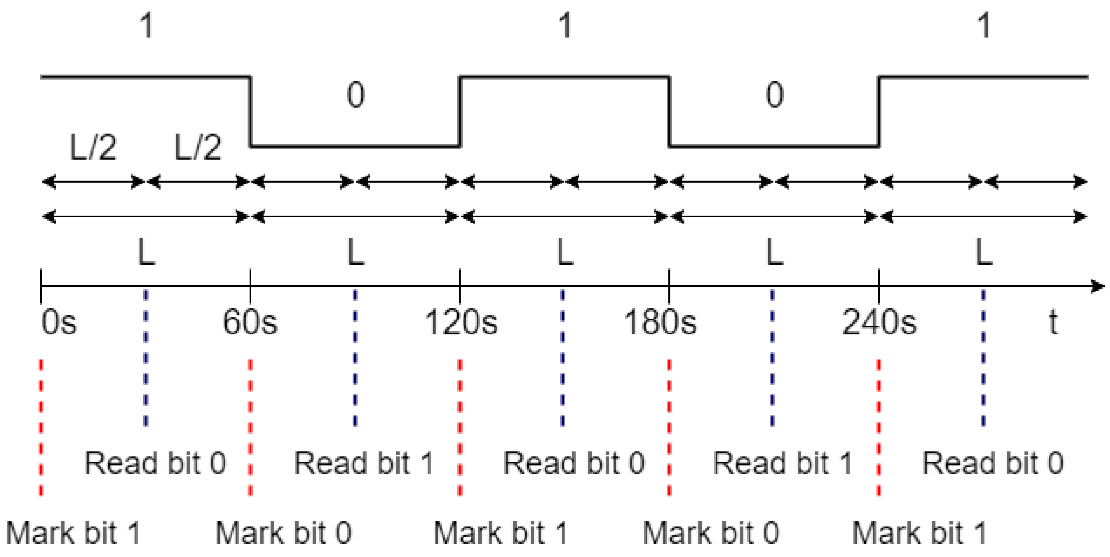


Figure 4.9. Controller timeline.

This mechanism is implemented using a flow table pipeline into the open vSwitch. The pipeline in a openflow switch has at least one flow table and for each flow table there are multiple flow entries.

The pipeline processing allows to understand the interaction between the flow tables and packets. The flow tables are sequentially numbered, starting at 0. In fact the pipeline processing always starts at the first flow table. A flow entry can only submit a packet to a flow table with a number greater than its own number. It is not possible submit a packet to a flow table with a number smaller than its number. During the pipeline processing, the packet is matched against the flow entries of the flow table. If there is a table hit, the instructions set in the selected flow entry is executed. These instructions can submit the packet to another flow table where the previous step is repeated. If the matching flow entry doesn't submit packets to another flow table, the pipeline processing stops. In this case it is executed the actions set and the packet is forwarded. At the end if a packet doesn't match a flow entry, this means that there is a table miss. So, the packet is dropped or it is submit to another table or it is sent to the controller.

4.10.1 Pipeline processing switch T

The pipeline processing in switches T is very simple. There is only one flow table in the pipeline. In this case the pipeline processing is greatly simplified. When a packet arrives into the switch can take two different paths. The first path represents the monitored traffic, with high priority. In this path it is matched the UDP packet and, before forwarding, it is set the tos bit of ip header to 0 or 1.

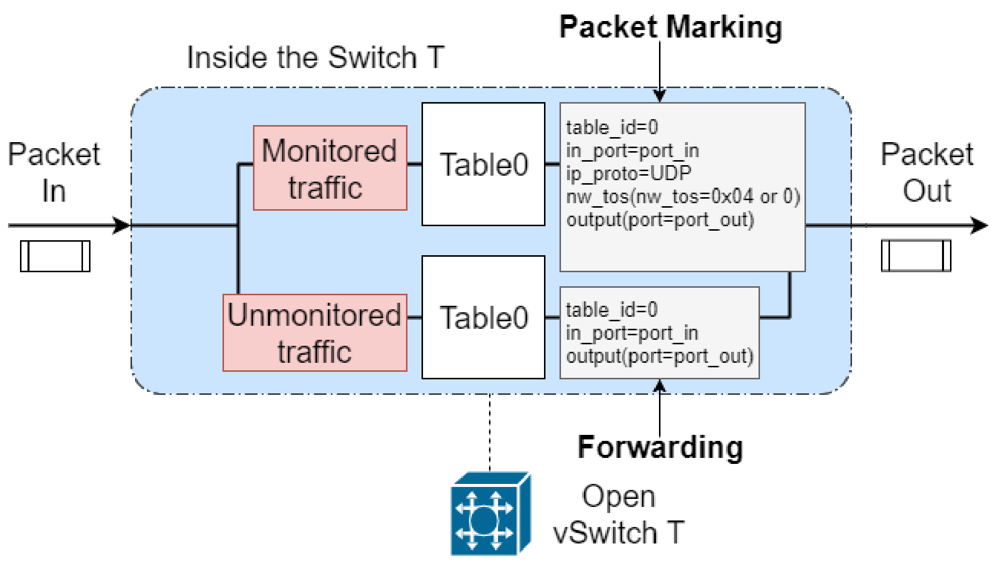


Figure 4.10. Pipeline processing in switch T.

After the execution of the actions set the packet is forwarded to the output port. Instead, the second path represents the unmonitored traffic, with low priority. In this path it is matched all traffic that is not UDP and it is forwarded to the output port.

4.10.2 Pipeline processing switch S

The pipeline processing in switches S is totally different. There are always two paths. The first represents the monitored traffic, with high priority. In this path it is matched the UDP packet with the tos bit marked to 0 or 1. After this, the packet is submitted to the next table where it is written the output port number into the tos field and then it is submitted to the last table where it is resetted again the value of tos field deleting the port number. After this step, the packet is forwarded to the output port. The second path, instead, represents the unmonitored traffic, with low priority. In this path it is matched all traffic that is not UDP and it is forwarded to the output port.

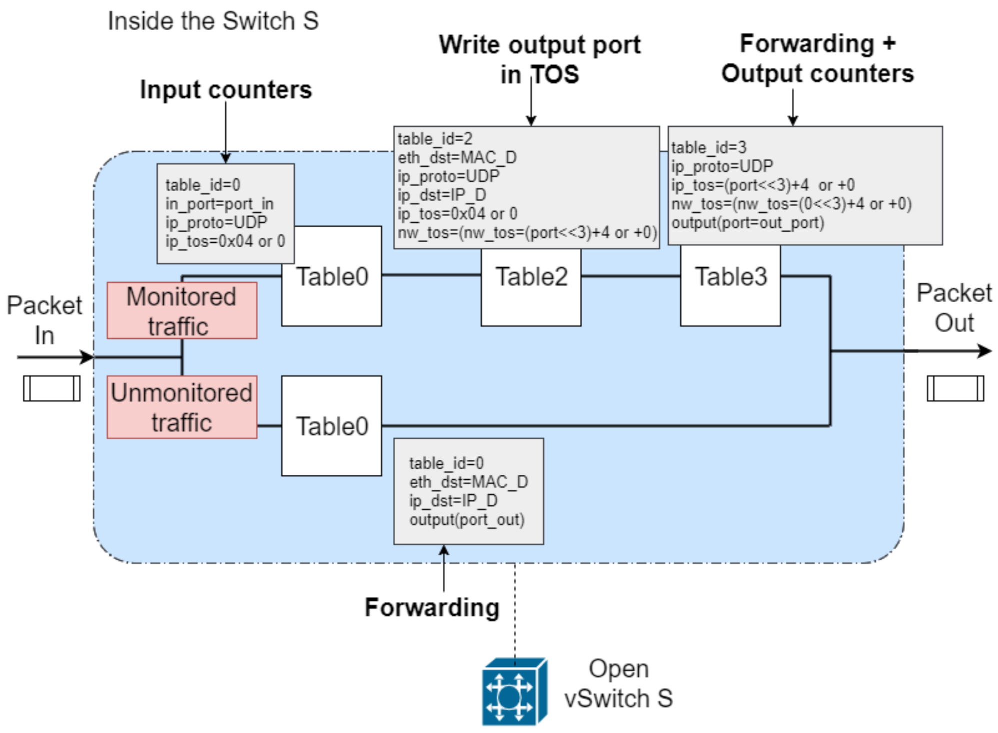


Figure 4.11. Pipeline processing in switch S.

4.10.3 Packet marking

If L is the duration of the interval of the packet marking, the duration of the waiting time, before reading the counters, is L/2. The packet marking is implemented with the flow table 0 in switches T. It consists to set a bit in the tos field of the ip header. In the 8-bit tos field the third bit is used for packet marking because the first two bits are reserved. So, the tos field is equal to 0x04 when the bit is marked to 1 and 0 when it is marked to 0. Since openflow supports the match on the tos field of the ip packet, when it's need to count the number of marked packets, it is read the value of the tos field to see if it is equal to 1 or 0. The reading of counters is implemented with the flow tables 2 and 3. The counters on the input port are read with the flow table 0, implemented in the switches S. Considering all the UDP packets that match against the input port having the tos field equals to 0x04 or 0. The situation is totally different for the output port because openflow doesn't provide counters on the output port. In the first approach, not implemented, there are two tables. The first is a classic forwarding table in which for each destination mac address there is an output port. The second one, instead, is used to count the output packets. For each output port there is a packet counter. So, for each port of each switch in the network there is the corresponding packet counter. This idea is not implementable because openflow doesn't provide a field match on the output port. The solution is to use the tos field of the ip header as a temporary metadata in which is written the output port number. Some bits of the tos field are used for the output port and another bit is used for the marking. The flow tables are always two. In the forwarding table, for each destination mac address it is written the correponding output port in the tos field. In the table 3, instead, for each tos field (with the information about the port) there is a packet counter. In this way I can read both input and output counters. Before forwarding the packet to the output port the tos field is resetted, deleting the port number and keeping the bit marking.

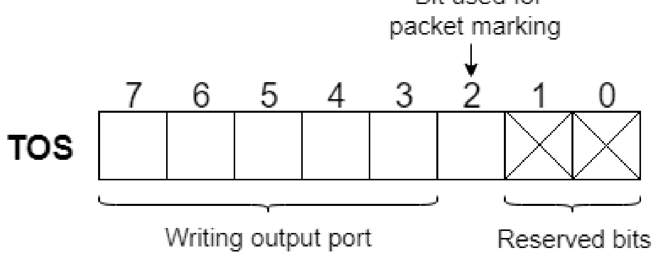


Figure 4.12. TOS field.

Chapter 5

Experimental evaluation

The tests are performed on two real network topologies: GÉANT and BICS. On the one hand, in Mininet, it is emulated the specific topology and then the packet generator and packet collector scripts are running, in order to check the accuracy of controller measurements. As we have seen in the previous chapter the packet generator, running on each hosts in the topology, sends and receives the UDP packets from all the other hosts in the network and sends the report to the collector. It generates the UDP traffic all-to-all. In the packet generator script are defined two costant variables: NUM_PKTS and TEST_DURATION. These represent respectively the total number of UDP packets sent and the duration of the interval of traffic generation. It means that if NUM_PKTS=1000*5 the total number of packets sent is equal to 5000, while if TEST_DURATION=60*5 it means that the duration of the interval of traffic generation is equal to 300s. Separately we run the controller program that implements the multipoint alternate marking technique. After reading input and output counters, the values are written in a json file (results.json). At the end when no more traffic is generated, the program is interrupted and the values of the controller (in results.json) and collector (in summary.dat) counters are matched. In addition the json file is also used to draw some graphs. We performed each test first with loss=0%, without emulating a certain percentage of losses in the network, and then with loss=1%. This means that it is emulated a percentage of losses equals to 1% in all links between switches S in the network. We started the tests with L=120s, where L is the bit marking interval, then we progressively reduced the duration of this interval to 60s, 30s and 20s. We noticed that with values smaller than 20s the counters in the results.json, on the Controller side, and those in summary.dat, on the collector side, didn't match. This happens because the duration of the packet marking interval is too small so some UDP packets are not marked, or not counted in the right period.

5.1 BICS topology

5.1.1 Test with loss=0%

The first table shows the results of the testing with loss=0% and L=120s. The counters values are the same both in results.json and in summary.dat. This means that the controller measurements are correct.

Total packets	Total packets counter
Total packets sent	5.280.000
Total packets recv	5.280.000
Total packets lost	0

The second table shows the results of the testing with loss=0% and L=60s. The counters values are the same both in results.json and in summary.dat. This means that the controller measurements are correct.

Total packets	Total packets counter
Total packets sent	5.280.000
Total packets recv	5.280.000
Total packets lost	0

The third table shows the results of the testing with loss=0% and L=30s. The counters values are the same both in results.json and in summary.dat. This means that the controller measurements are correct.

Total packets	Total packets counter
Total packets sent	5.280.000
Total packets recv	5.280.000
Total packets lost	0

The last table shows the results of the testing with loss=0% and L=20s. The counters values are the same both in results.json and in summary.dat. This means that the controller measurements are correct.

Total packets	Total packets counter
Total packets sent	5.280.000
Total packets recv	5.280.000
Total packets lost	0

5.1.2 Test with loss=1% and monitored traffic type=1:N

Fig. 5.1 shows the results of the testing with L=120s, loss=1% and filter on Milan node with ip_src=10.0.0.17. The filter type is 1 to N. This means that the traffic generation is always all-to-all but the monitored traffic is one-to-all. Only from the Milan node to all the other nodes. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

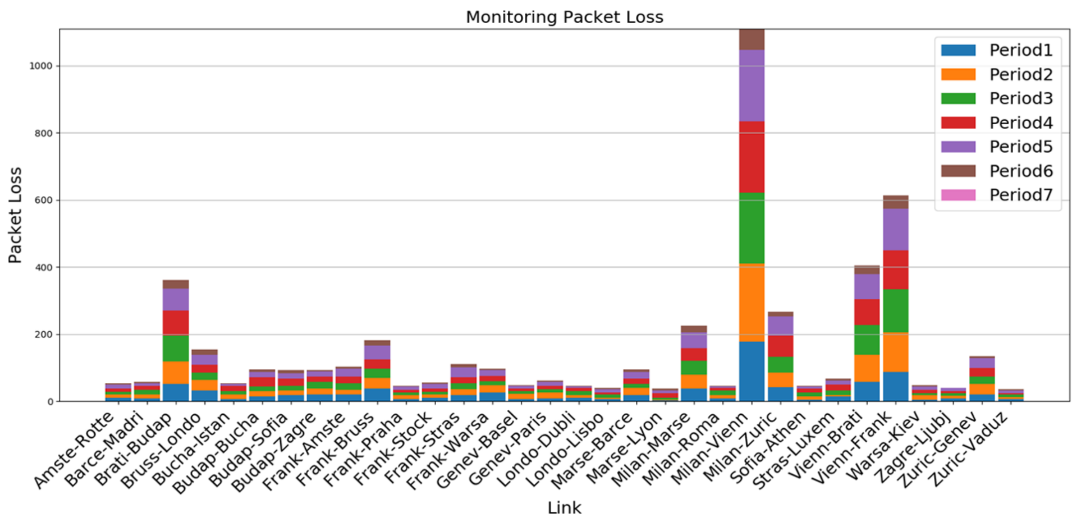


Figure 5.1. Packet Loss, BICS, L=120s, monitored traffic=1:N.

Total packets	Total packets counter
Total packets sent	5.280.000
Total packets recv	5.092.818
Total packets lost	187.182
Total packets lost with filter	4.912

Fig. 5.2 shows the results of the testing with L=60s, loss=1% and filter on Milan node with ip_src=10.0.0.17. The filter type is 1 to N. This means that the traffic generation is always all-to-all but the monitored traffic is one-to-all. Only from the Milan node to all the other nodes. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

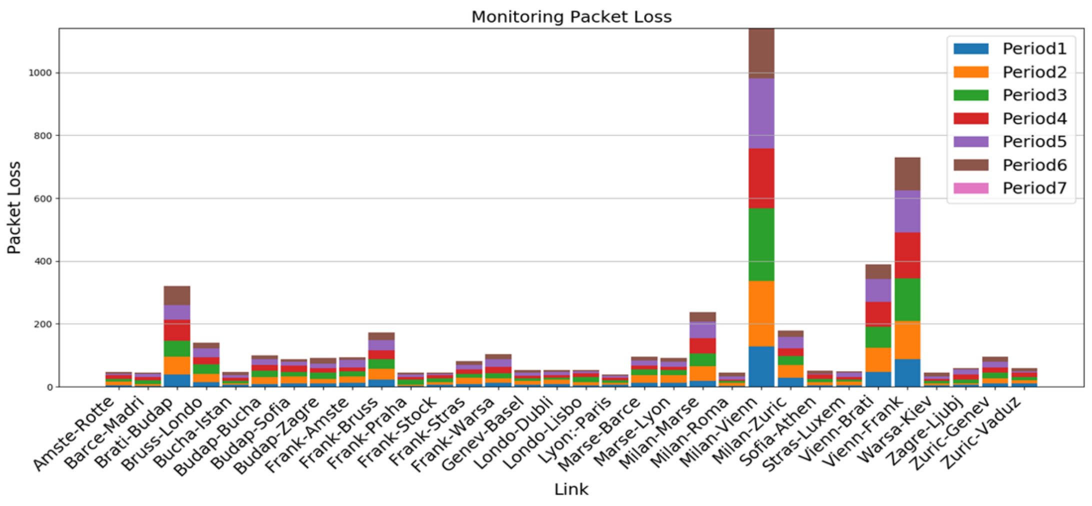


Figure 5.2. Packet Loss, BICS, L=60s, monitored traffic=1:N.

Total packets	Total packets counter
Total packets sent	5.280.000
Total packets recv	5.091.505
Total packets lost	188.495
Total packets lost with filter	4.876

Fig. 5.3 shows the results of the testing with L=30s, loss=1% and filter on Milan node with ip_src=10.0.0.17. The filter type is 1 to N. This means that the traffic generation is always all-to-all but the monitored traffic is one-to-all. Only from the Milan node to all the other nodes. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

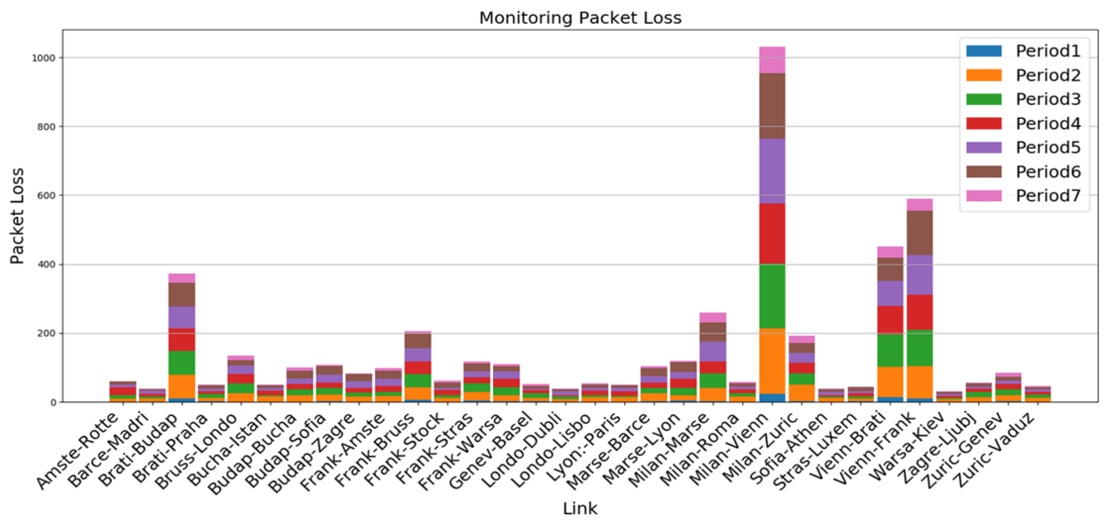


Figure 5.3. Packet Loss, BICS, L=30s, monitored traffic=1:N.

Total packets	Total packets counter
Total packets sent	5.280.000
Total packets recv	5.093.047
Total packets lost	186.953
Total packets lost with filter	4.883

Fig. 5.4 shows the results of the testing with L=20s, loss=1% and filter on Milan node with ip_src=10.0.0.17. The filter type is 1 to N. This means that the traffic generation is always all-to-all but the monitored traffic is one-to-all. Only from the Milan node to all the other nodes. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

Total packets	Total packets counter
Total packets sent	5.280.000
Total packets recv	5.090.923
Total packets lost	189.077
Total packets lost with filter	5.037

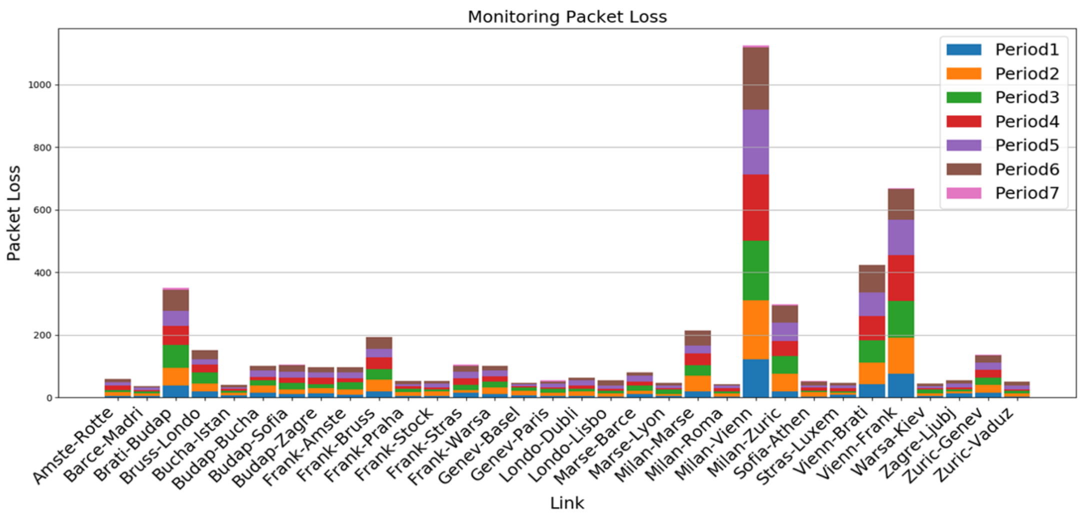


Figure 5.4. Packet Loss, BICS, L=20s, monitored traffic=1:N.

5.1.3 Test with loss=1% and monitored traffic type=N:N

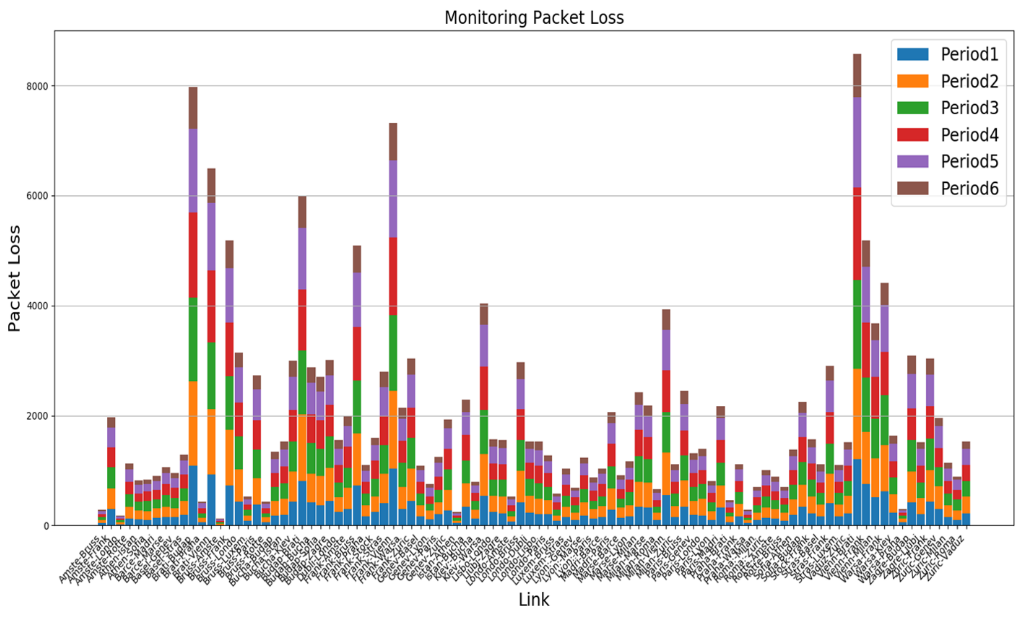


Figure 5.5. Packet Loss, BICS, L=120s, monitored traffic=N:N.

Total packets	Total packets counter
Total packets sent	5.280.000
Total packets recv	5.091.259
Total packets lost	188.741

Fig. 5.5 shows the results of the testing with L=120s, loss=1%. There is no filter. This means that the traffic generation is always all-to-all and the monitored traffic is all-to-all. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

Fig. 5.6 shows the results of the testing with L=60s, loss=1%. There is no filter. This means that the traffic generation is always all-to-all and the monitored traffic is all-to-all. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

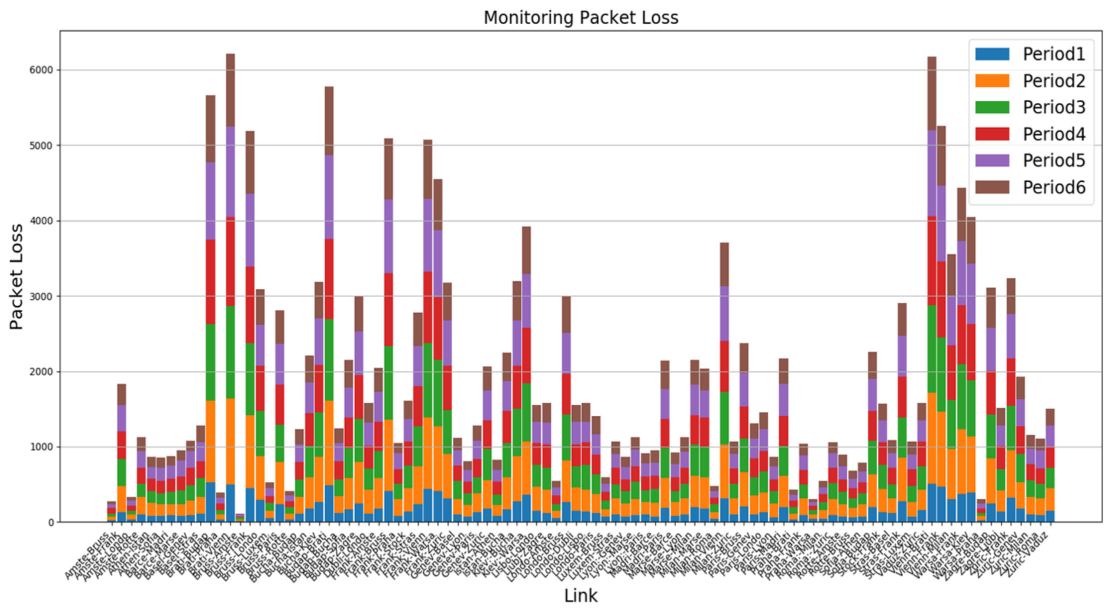


Figure 5.6. Packet Loss, BICS, L=60s, monitored traffic=N:N.

Total packets	Total packets counter
Total packets sent	5.280.000
Total packets recv	5.093.011
Total packets lost	186.989

Fig. 5.7 shows the results of the testing with L=30s, loss=1%. There is no filter. This means that the traffic generation is always all-to-all and the monitored traffic is all-to-all. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

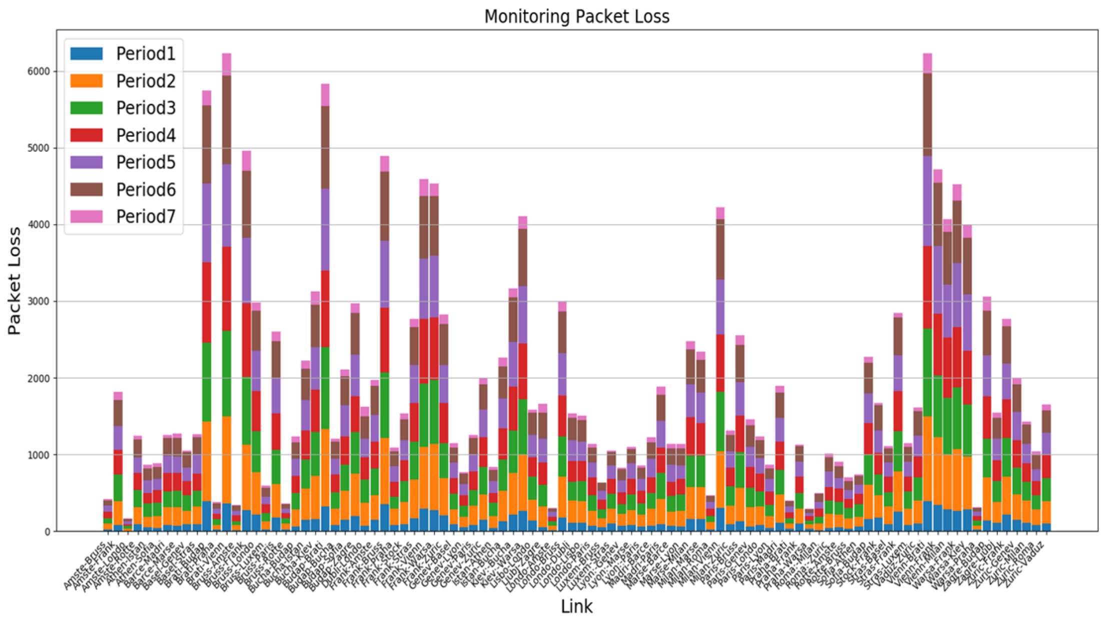


Figure 5.7. Packet Loss, BICS, L=30s, monitored traffic=N:N.

Total packets	Total packets counter
Total packets sent	5.280.000
Total packets recv	5.092.665
Total packets lost	187.335

Fig. 5.8 shows the results of the testing with L=20s, loss=1%. There is no filter. This means that the traffic generation is always all-to-all and the monitored traffic is all-to-all. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

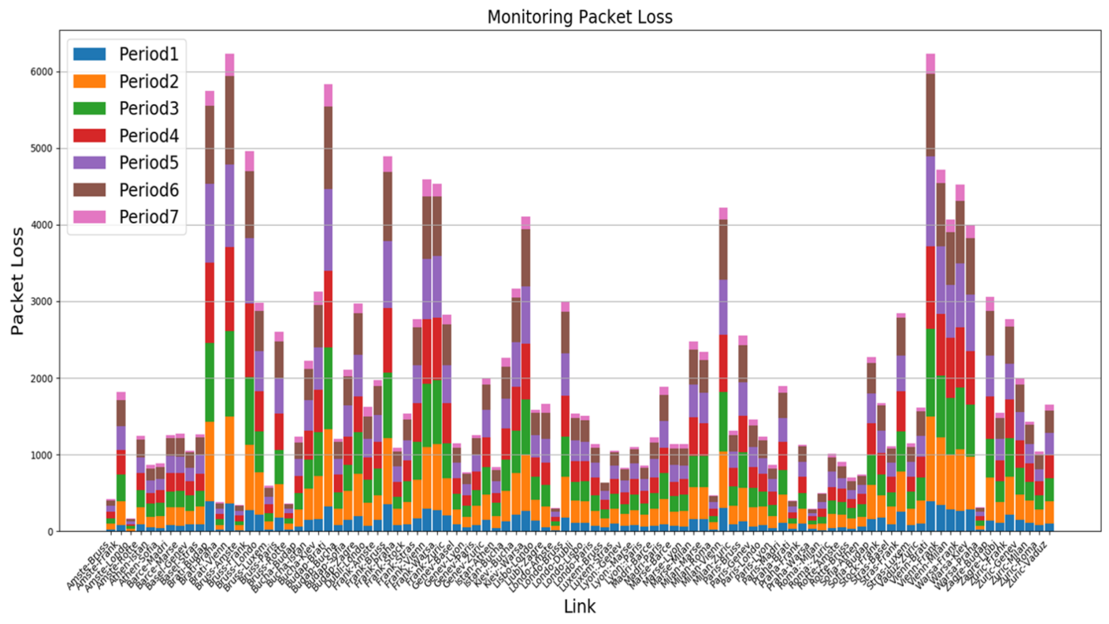


Figure 5.8. Packet Loss, BICS, L=20s, monitored traffic=N:N.

Total packets	Total packets counter
Total packets sent	5.280.000
Total packets recv	5.092.393
Total packets lost	187.607

5.2 GÉANT topology

5.2.1 Test with loss=0%

The first table shows the results of the testing with loss=0% and L=120s. The counters values are the same both in results.json and in summary.dat. This means that the controller measurements are correct.

Total packets	Total packets counter
Total packets sent	7.800.000
Total packets recv	7.800.000
Total packets lost	0

The second table shows the results of the testing with loss=0% and L=60s. The counters values are the same both in results.json and in summary.dat. This means that the controller measurements are correct.

Total packets	Total packets counter
Total packets sent	7.800.000
Total packets recv	7.800.000
Total packets lost	0

The third table shows the results of the testing with loss=0% and L=30s. The counters values are the same both in results.json and in summary.dat. This means that the controller measurements are correct.

Total packets	Total packets counter
Total packets sent	7.800.000
Total packets recv	7.800.000
Total packets lost	0

The last table shows the results of the testing with loss=0% and L=20s. The counters values are the same both in results.json and in summary.dat. This means that the controller measurements are correct.

Total packets	Total packets counter
Total packets sent	7.800.000
Total packets recv	7.800.000
Total packets lost	0

5.2.2 Test with loss=1% and monitored traffic type=1:N

Fig. 5.9 shows the results of the testing with L=120s, loss=1% and filter on IT node with ip_src=10.0.0.10. The filter type is 1 to N. This means that the traffic generation is always all-to-all but the monitored traffic is one-to-all. Only from the Milan node to all the other nodes. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

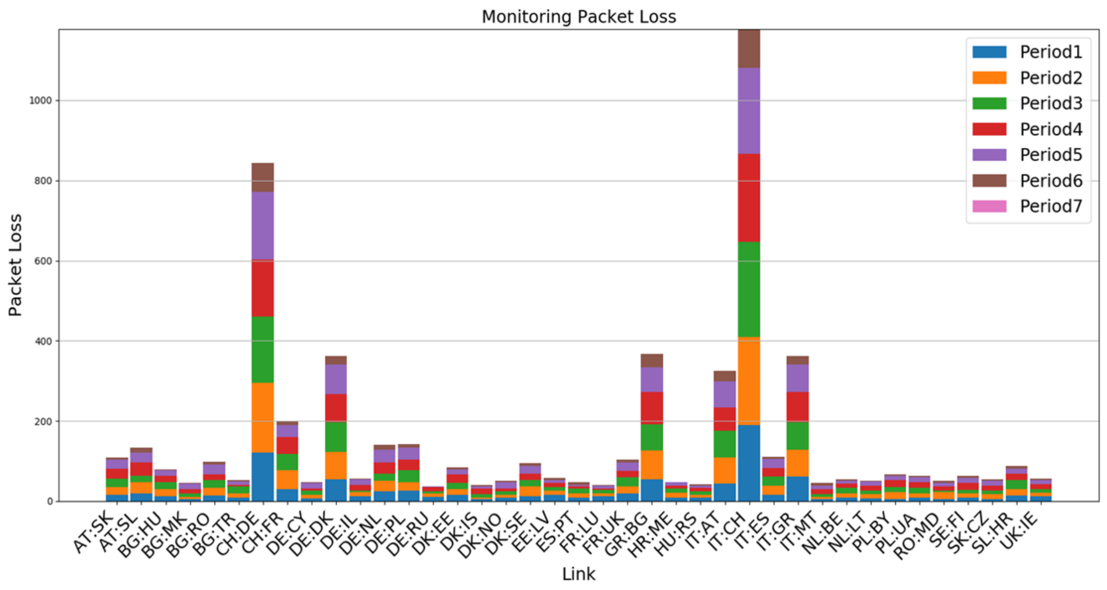


Figure 5.9. Packet Loss, GÉANT, L=120s, monitored traffic=1:N.

Total packets	Total packets counter
Total packets sent	7.800.000
Total packets recv	7.529.240
Total packets lost	270.760
Total packets lost with filter	5.891

Fig. 5.10 shows the results of the testing with L=60s, loss=1% and filter on IT node with ip_src=10.0.0.10. The filter type is 1 to N. This means that the traffic generation is always all-to-all but the monitored traffic is one-to-all. Only from the Milan node to all the other nodes. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

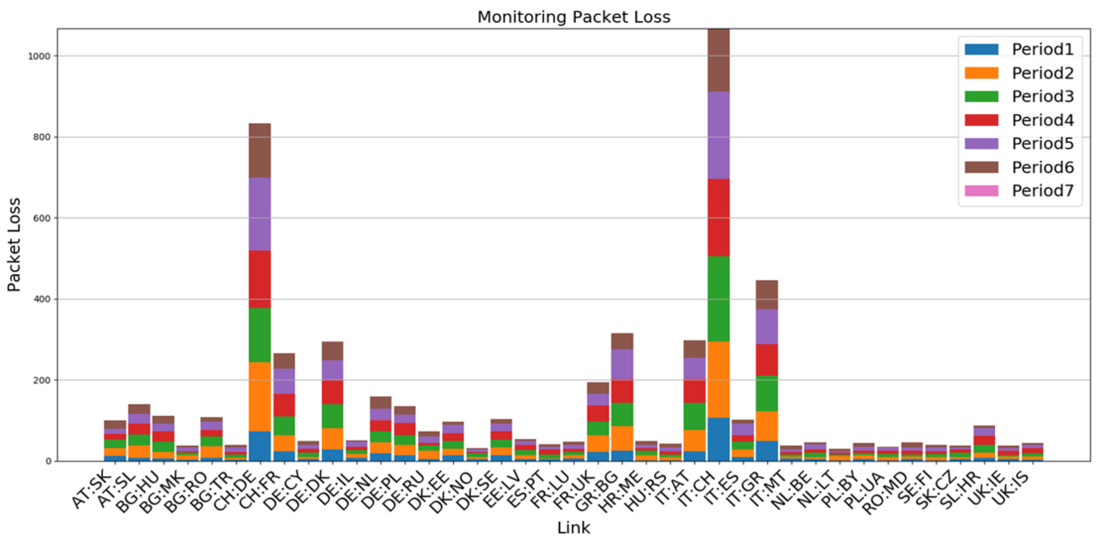


Figure 5.10. Packet Loss, GÉANT, L=60s, monitored traffic=1:N.

Total packets	Total packets counter
Total packets sent	7.800.000
Total packets recv	7.528.598
Total packets lost	271.402
Total packets lost with filter	5.779

Fig. 5.11 shows the results of the testing with L=30s, loss=1% and filter on IT node with ip_src=10.0.0.10. The filter type is 1 to N. This means that the traffic generation is always all-to-all but the monitored traffic is one-to-all. Only from the Milan node to all the other nodes. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology.

It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

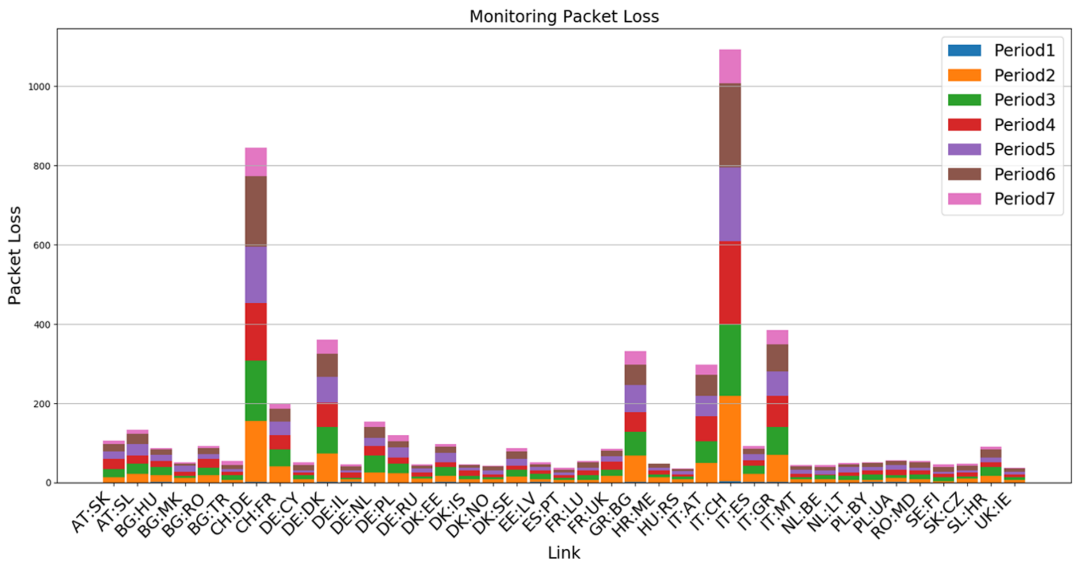


Figure 5.11. Packet Loss, GÉANT, L=30s, monitored traffic=1:N.

Total packets	Total packets counter
Total packets sent	7.800.000
Total packets recv	7.518.236
Total packets lost	281.764
Total packets lost with filter	5.653

Fig. 5.12 shows the results of the testing with L=20s, loss=1% and filter on IT node with ip_src=10.0.0.10. The filter type is 1 to N. This means that the traffic generation is always all-to-all but the monitored traffic is one-to-all. Only from the Milan node to all the other nodes. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

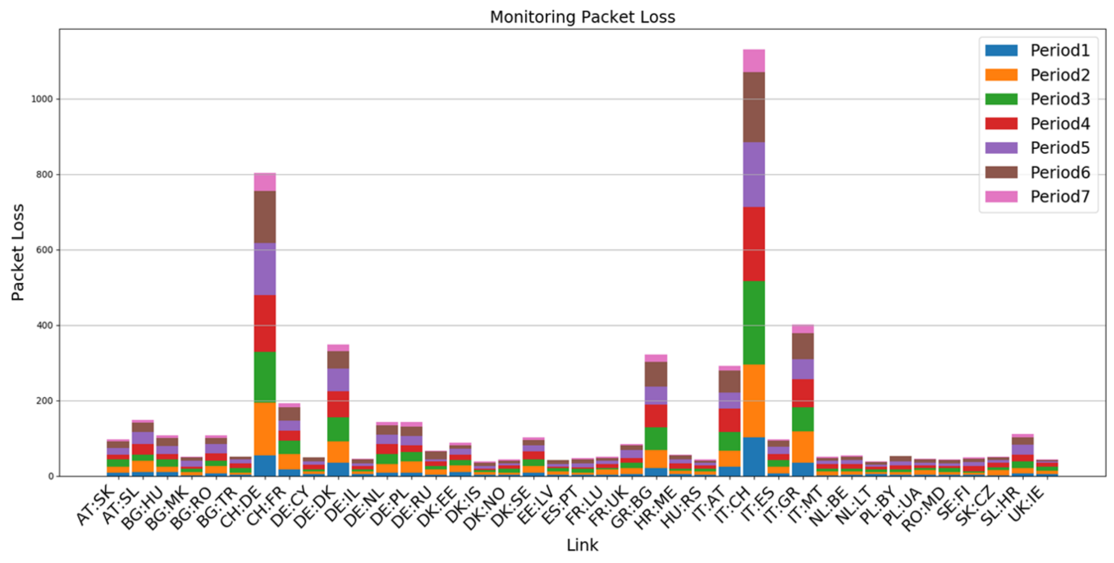


Figure 5.12. Packet Loss, GÉANT, L=20s, monitored traffic=1:N.

Total packets	Total packets counter
Total packets sent	7.800.000
Total packets recv	7.528.123
Total packets lost	271.877
Total packets lost with filter	5.744

5.2.3 Test with loss=1% and monitored traffic type=N:N

Fig. 5.13 shows the results of the testing with L=120s, loss=1%. There is no filter. This means that the traffic generation is always all-to-all and the monitored traffic is all-to-all. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

Total packets	Total packets counter
Total packets sent	7.800.000
Total packets recv	7.524.620
Total packets lost	275.380

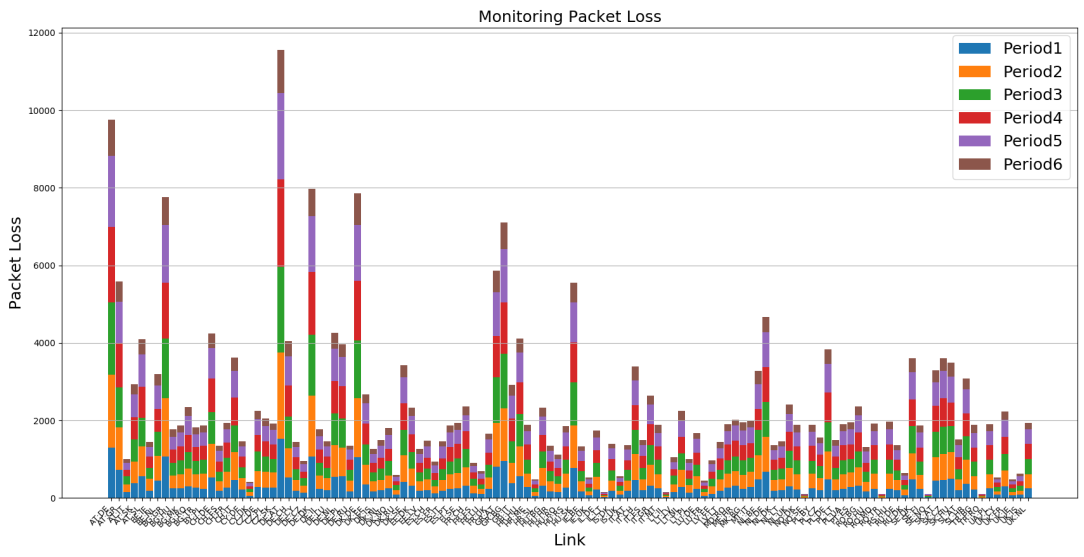


Figure 5.13. Packet Loss, GÉANT, L=120s, monitored traffic=N:N.

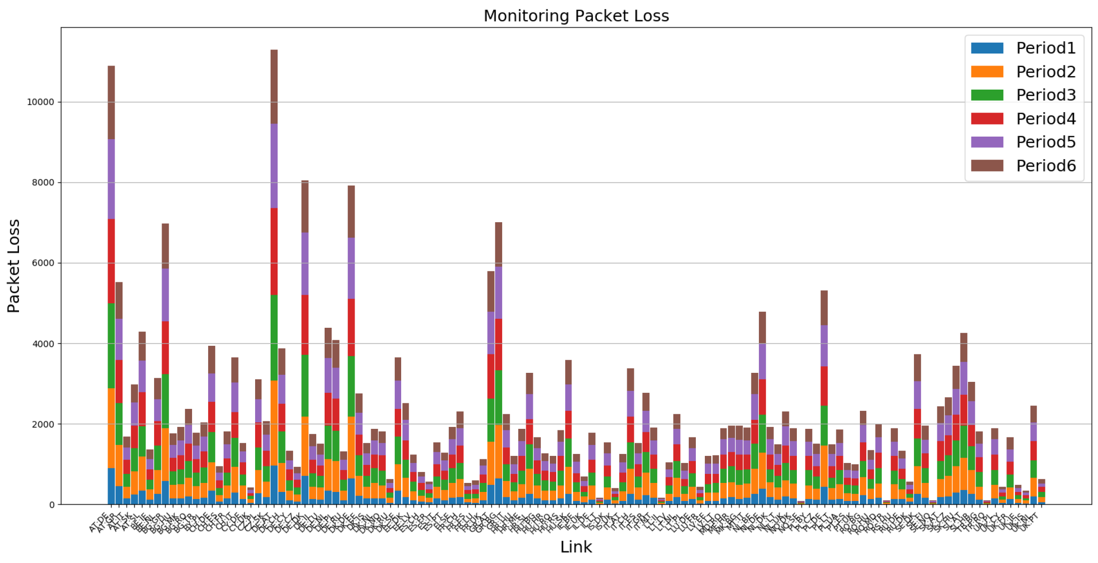


Figure 5.14. Packet Loss, GÉANT, L=60s, monitored traffic=N:N.

Fig. 5.14 shows the results of the testing with L=60s, loss=1%. There is no filter. This means that the traffic generation is always all-to-all and the monitored traffic is all-to-all. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

Total packets	Total packets counter
Total packets sent	7.800.000
Total packets recv	7.527.370
Total packets lost	272.630

Fig. 5.15 shows the results of the testing with L=30s, loss=1%. There is no filter. This means that the traffic generation is always all-to-all and the monitored traffic is all-to-all. The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

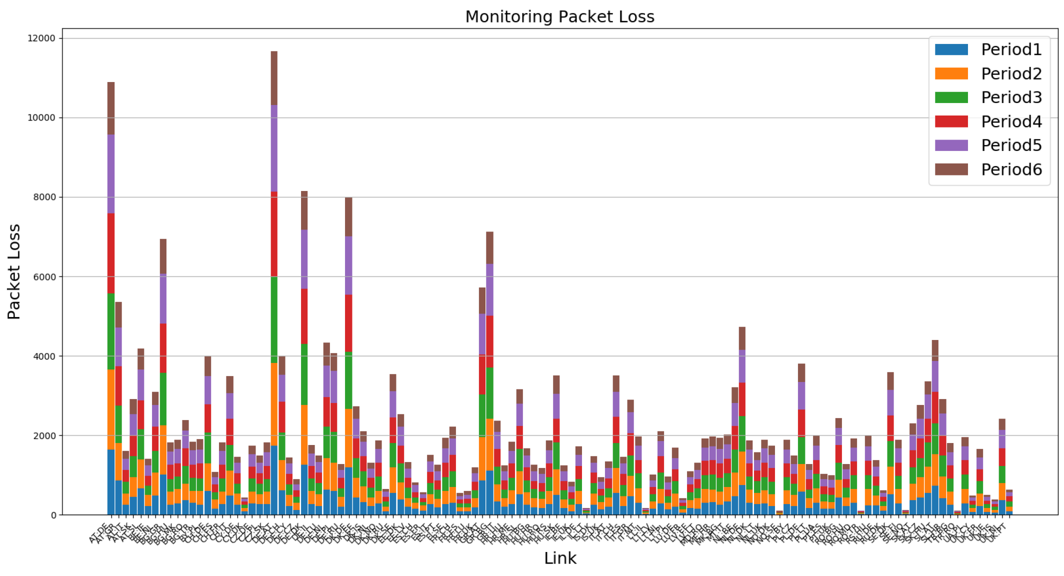


Figure 5.15. Packet Loss, GÉANT, L=30s, monitored traffic=N:N.

Total packets	Total packets counter
Total packets sent	7.800.000
Total packets recv	7.528.716
Total packets lost	271.284

Fig. 5.16 shows the results of testing with L=20s and loss=1%. There is no filter. This means that the traffic generation is always all-to-all and the monitored traffic is all-to-all too.
The histogram presents on the y-axis the total number of lost packets and on the x-axis the links in the topology. It shows the total number of packets lost for each link in the topology, in each period. The counters values are the same both in results.json and in summary.dat.

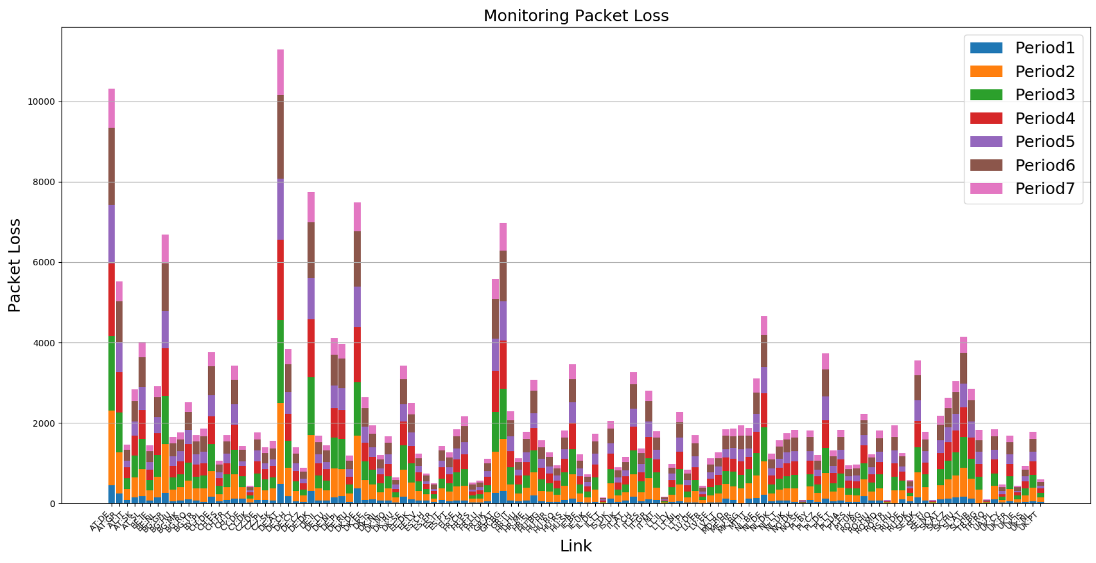


Figure 5.16. Packet Loss, GÉANT, L=20s, monitored traffic=N:N.

Total packets	Total packets counter
Total packets sent	7.800.000
Total packets recv	7.528.499
Total packets lost	271.501

5.3 The relation between physical and extended graph

To test the clustering algorithms on real topologies, two scripts have been implemented: iterative_clustering.py and recursive_clustering.py. Both scripts take the name of the .graphml file, which contains the information about the topology, from the command line and save the resulted clusters in a .json file.
In the first (the recursive one), the code takes in input the adjacency matrix, that is the matrix that for each link (h1, h2) of the graph, has a 1 in the position (1,2), as we have seen before. The second implementation, instead takes in input the list of links in both the input and output directions (h1, h2) and (h2, h1).
The clusters are created in three steps. First of all, it is emulated the physical topology in Mininet reading the information from the .graphml file. Then it is created the graph G, the extended graph. This is produced starting from the chosen topology and creating two links for each physical link and internal links to the routers that connect each interface to the other, as we can see in the Fig. 3.5.
At the end, from the exteded graph G it is created the monitored graph $\bar{G}$ and then from the monitored graph it is created a cluster $\tilde{G}$ as a subnetwork of the graph $\bar{G}$.
Both the scipts provide the cluster list as output and save it in the clusters.json file.
Applying the clustering algorithm, both iterative and recursive, to all the network topologies present in the dataset of The Internet Topology Zoo, we noticed that the interface-level graph is much larger than the physical graph. It was clear in all the plotted graphs in which the relation is close to linear. In the transition from the physical graph to the extended graph the dimension of the network changes a lot. So has been implemented a script that takes in input, for each topology, the number of physical and extended nodes and the number of physical and extended edges and creates a scattered graph which clearly shows this relation.
In the Fig. 5.17 is showed the relation between nodes in physical and logical topology. On the x-axis there is the number of the physical nodes for each topology in the dataset and on the y-axis there is the number of the extended nodes for each topology in the dataset.
Otherwise, in the Fig. 5.18 is showed the relation between edges in physical and logical topology. On the x-axis there is the number of the physical edges and on the y-axis there is the number of the extended edges for each logical topology. Each logical topology has a number of extended nodes and extended edges greater than the physical one.

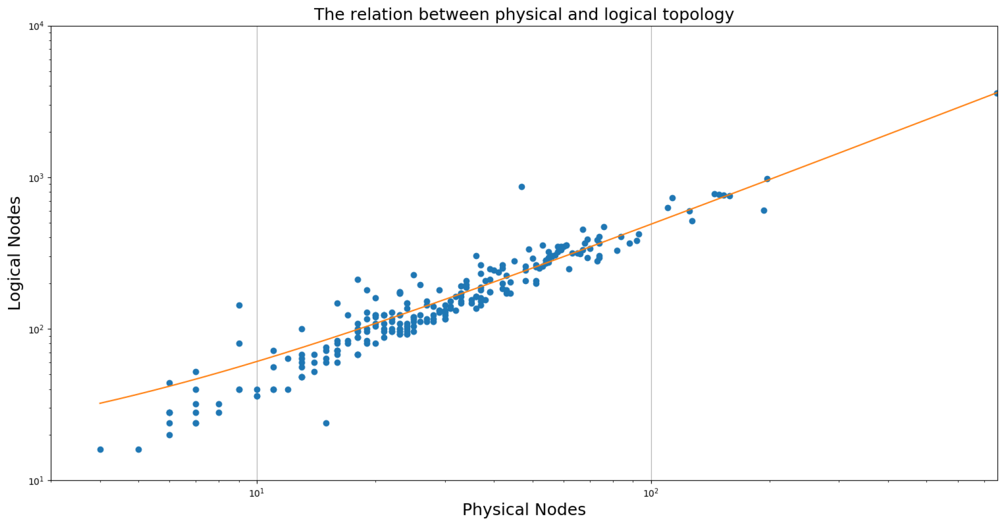


Figure 5.17. The relation between physical and logical nodes. The vector of coefficients p returned from the polyfit function is : [4.76989943, 13.20133961].

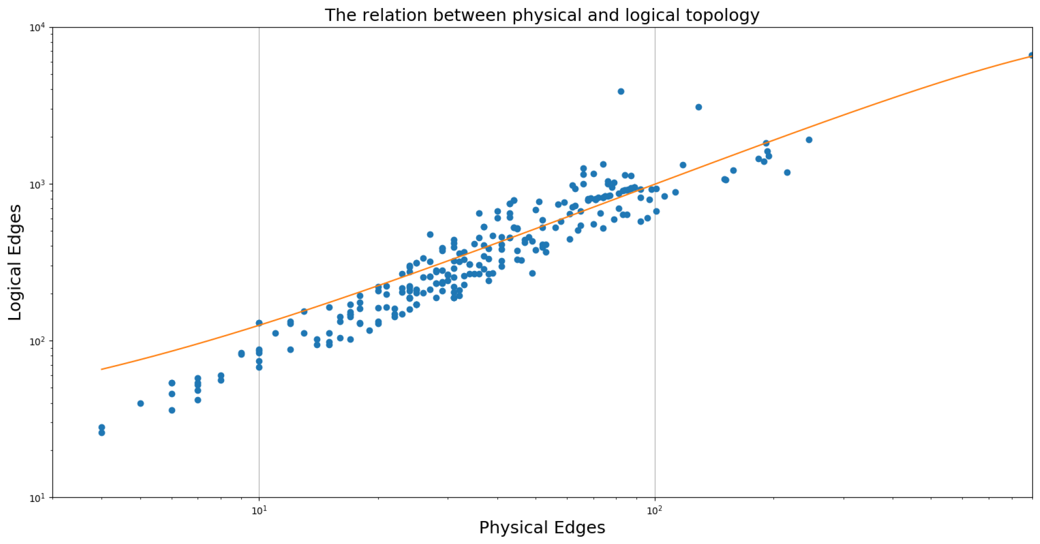


Figure 5.18. The relation between physical and logical edges. The vector of coefficients p returned from the polyfit function is : [7.87518742 110.29895886].

5.4 Clusters features

This section concerns the study of the clusters' features in a real situation. The goal is to examine the features of clusters in three real network topologies. One of small size and the other ones of medium size. The first topology is Géant with 40 physical nodes and 61 physical edges. The second one is Colt Telecom with 153 physical nodes and 191 physical edges and the last one is Cogent Communications with 197 physical nodes and 245 physical edges. In order to describe the features of clusters it was used a box plot. In statistics, the box plot is a graphical representation used in order to describe the distribution of a sample through simple dispersion and position indices. It is represented through a box divided into two parts, from which two segments emerge. The box itself is delimited by the first and third quartiles, q1/4 and q3/4, and it is divided inside by the median, q1/2. The segments (the "whiskers") are delimited by the minimum and maximum values. In this way the four equally populated intervals delimited by the quartiles are graphically represented. The quartiles are those values that divide the population (the set of elements that are the object of study) in four parts of equal size. Instead the median is defined as the value or set of values that are in the middle of the distribution.

5.4.1 Géant topology

The first testing scenario concerns Géant topology.

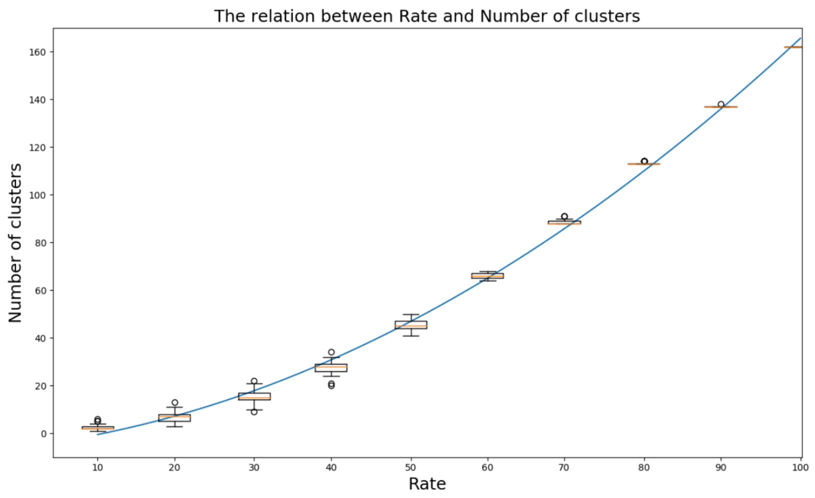


Figure 5.19. Géant: The relation between rate and numbers of clusters with polyfit.

The first box plot has on the x axis the random percentage of the selected monitored nodes and on the y axis the total number of clusters. As we can see in the Fig. 5.19, the number of clusters increases when increasing the percentage of the selected nodes and there is a little variability, in fact the values are all quite close to the median. This behavior is evident in the graph Fig. 5.19 in which the curve polynomial fitting is also plotted, in addition to the box plot, using the function polyfit provided by the numpy library: **numpy.polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False)**. This function returns the coefficients of a polynomial of degree deg that is the least squares fit to the data values y given at points x. If y is 1-D the returned coefficients will also be 1-D. If y is 2-D multiple fits are done, one for each column of y, and the resulting coefficients are stored in the corresponding columns of a 2-D return. So given a polynomial

$$p(x) = p_0 x^{deg} + ... + p_{deg} \tag{5.1}$$

of degree deg to points (x, y). This function returns a vector of coefficients p that minimises the squared error. In the graph Fig. 5.19 the coefficients p are: [2.22421590e-03 1.58886653e-01 -5.61153775e+00]. The second box plot, instead, has on the x axis the random percentage of the selected monitored nodes and on the y axis the value of the cluster's diameter calculated on the extended graph. The diameter d of a graph is the greatest distance between any pair of vertices. To find the diameter of a generic graph, first we find the shortest path between each pair of vertices, then the greatest length of any of these paths is the diameter of the graph.

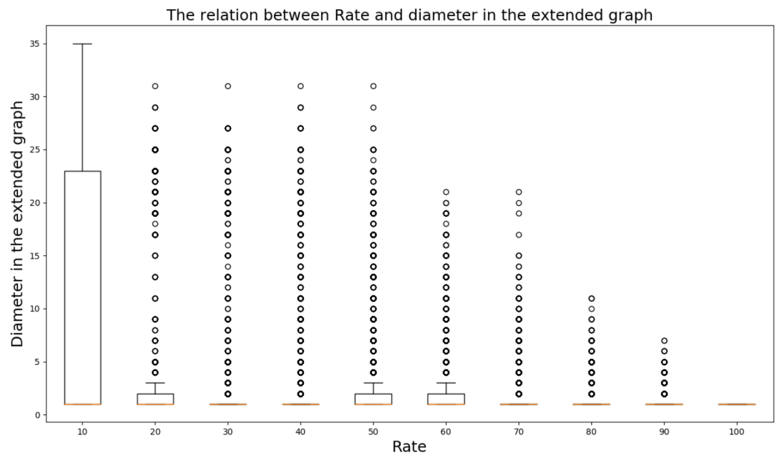


Figure 5.20. Géant: The relation between rate and extended diameter.

As we can see in the Fig. 5.20, the value of the median in all the boxes is the same, it is equal to 1, and only with the rate equals to 10 we have diameters with higher values, in fact the maximum value is 35. Basically the sense is to understand what range of variation there is and which are the most common values.
In this graph the median is 1 because 1 is the most present value (it means that in most cases there are all 1 in the sequence), and there are relatively few cases different from 1 that are the outliers. These have high values in tha range between 10 to 50 and decrease in the range between 50 and 100. An outlier is an observation that is numerically distant from the rest of the data. It is defined as a data point located outside the fences of the boxplot.
The box plot, Fig. 5.21, has on the x axis always the rate and on the y axis the value of the cluster's diameter calculated on the monitored graph. The value of the median in all the boxes is equal to 1. The outliers, instead, reach the maximun value with the rate equals to 50 and decrease in the range between 50 and 100.

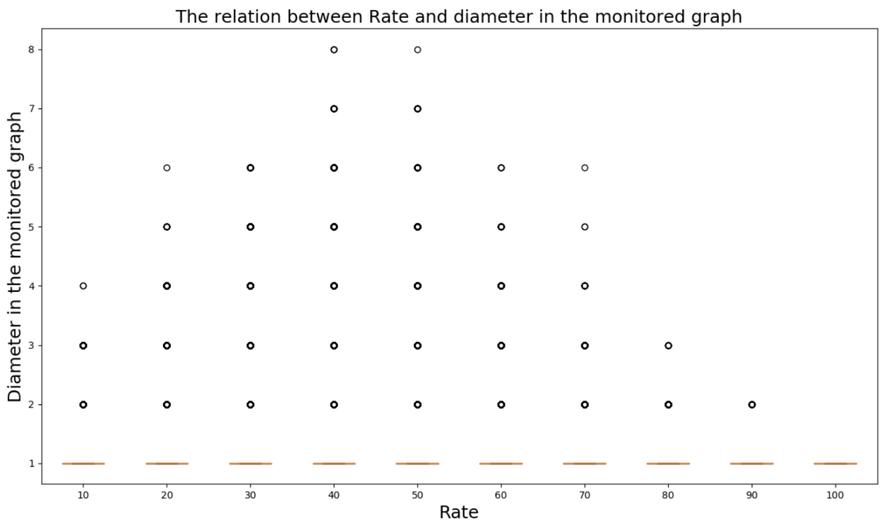


Figure 5.21. Géant: The relation between rate and monitored diameter.

The box plot, in the Fig. 5.22, has on the x axis the random percentage of the selected nodes monitored (the rate) and on the y axis the number of the cluster's nodes in the exdended graph. The value of the median in all the boxes is the same, it is equal to 1 and only with the rate equals to 10 we have higher values. In fact the maximum value, in this box, is equal to 220. There are relatively few cases different from 1 that are the outliers. These decrease in the range between 50 and 100.

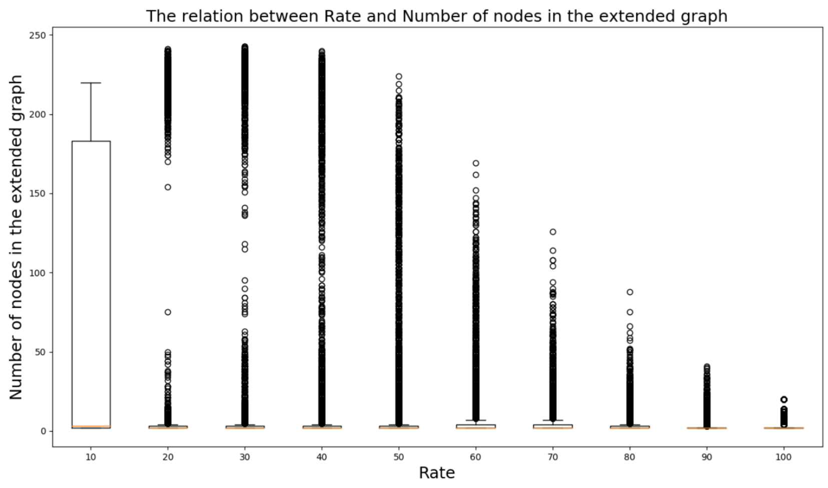


Figure 5.22. Géant: The relation between rate and extended nodes.

The following box plot, in the Fig. 5.23, has on the x axis the random percentage of the selected monitored nodes and on the y axis the number of cluster's nodes in the monitored graph. The value of the median in all the boxes is the same, it is equal to 1 and only with the rate equals to 10 we have higher values. In fact the maximum value is equal to 23. There are relatively few cases different from 1 that are the outliers. These increase in the range between 10 and 50 and decrease in the range between 50 and 100.

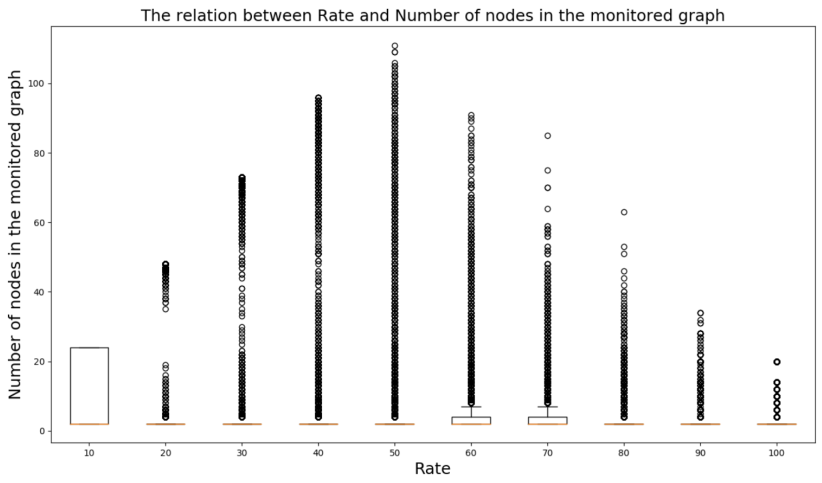


Figure 5.23. Géant: The relation between rate and monitored nodes.

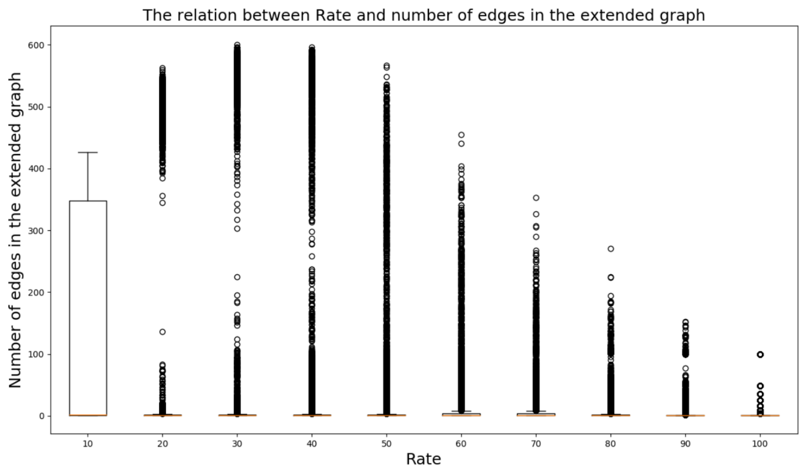


Figure 5.24. Géant: The relation between rate and extended edges.

The box plot, in the Fig. 5.24, has on the x axis the random percentage of the selected monitored nodes and on the y axis the number of the cluster's edges in the extended graph. The value of the median in all the boxes is equal to 1. There are relatively few cases different from 1 that are the outliers. These decrease in the range between 50 and 100.

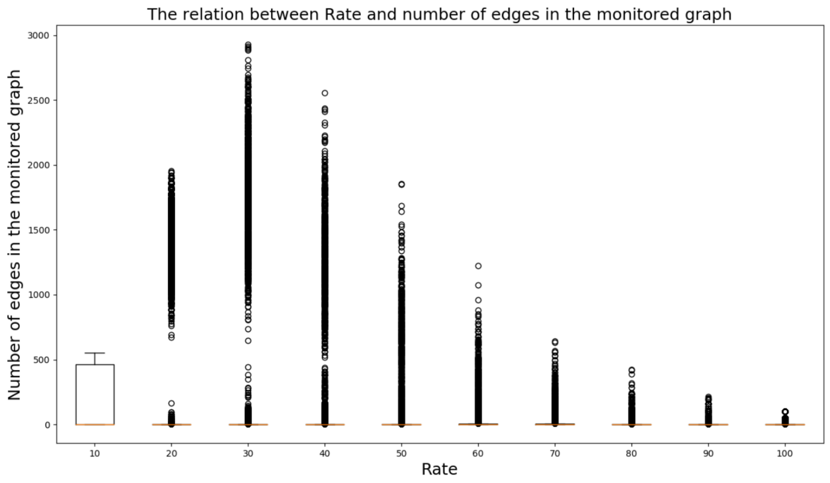


Figure 5.25. Géant: The relation between rate and monitored edges.

In the last box plot, Fig. 5.25, the value of the median in all the boxes is equal to 1. There are relatively few cases different from 1 that are the outliers that decrease in the range between 40 and 100.

5.4.2 Colt Telecom topology

The second testing scenario concerns Colt Telecom topology. The first box plot has on the x axis always the rate and on the y axis the total number of clusters. As we can see in the Fig. 5.26, the number of clusters increases when increasing the percentage of selected nodes and the values are all quite close to the median. In the graph the curve polynomial fitting is plotted using the polyfit function provided by the numpy library. The degree is equal to 2 and the vector of coefficients p is : [5.68029116e-04 3.18698189e-01 -3.09312305e+01].

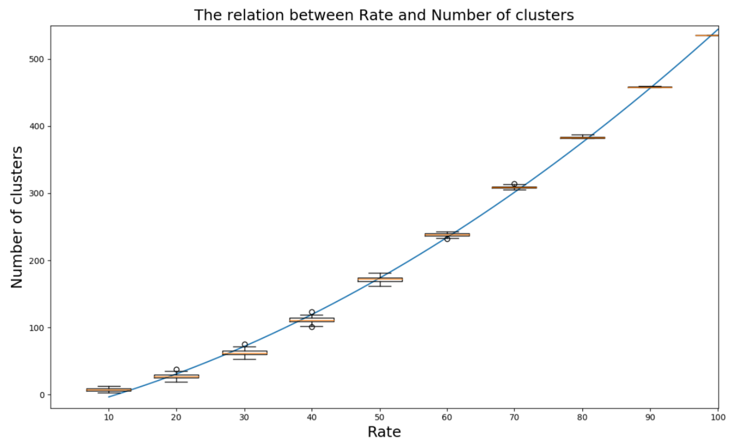


Figure 5.26. Colt Telecom: The relation between rate and numbers of clusters with polyfit.

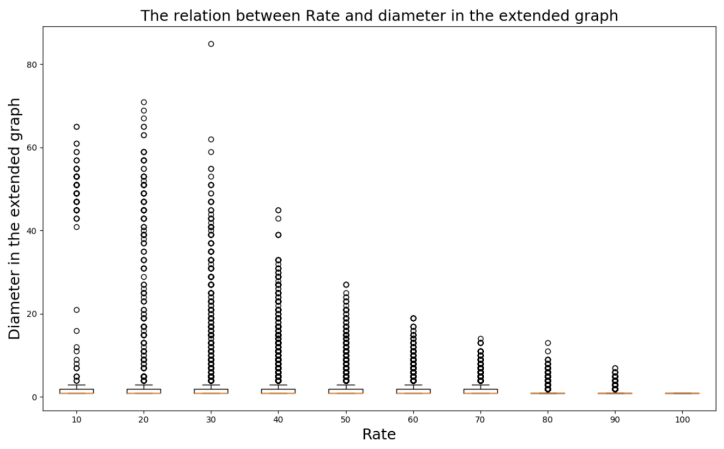


Figure 5.27. Colt Telecom: The relation between rate and extended diameter.

The second box plot, Fig. 5.27, has on the x axis the rate and on the y axis the value of the cluster's diameter calculated on the extended graph. The value of the median in all the boxes is equal to 1 and the outliers decrease the value with the increasing of the rate.

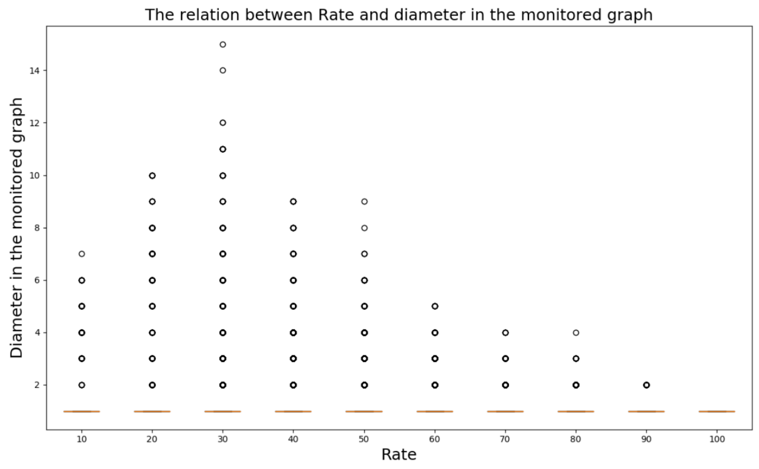


Figure 5.28. Colt Telecom: The relation between rate and monitored diameter.

The box plot, Fig. 5.28, has on the x axis always the rate and on the y axis the value of the cluster's diameter calculated on the monitored graph. The value of the median in all the boxes is equal to 1 and the outliers, instead, decrease the value with the increasing of the rate.

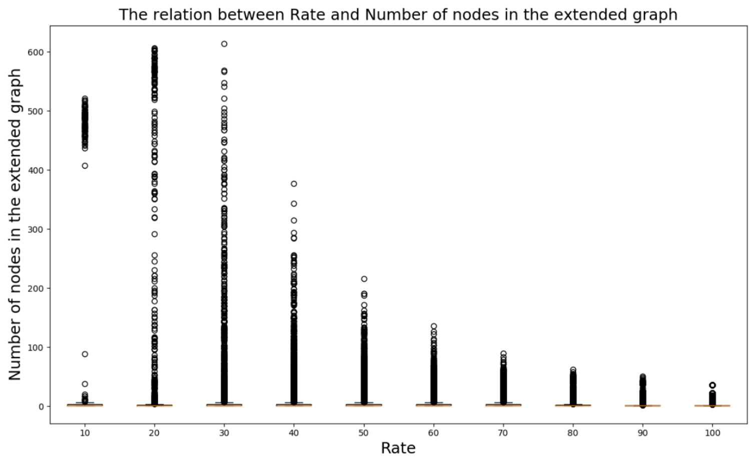


Figure 5.29. Colt Telecom: The relation between rate and extended nodes.

The box plot in the Fig. 5.29 has on the x axis always the rate and on the y axis the number of the cluster's nodes in the exdended graph. The value of the median in all the boxes is equal to 1 and the outliers, instead, decrease the value with the increasing of the rate.

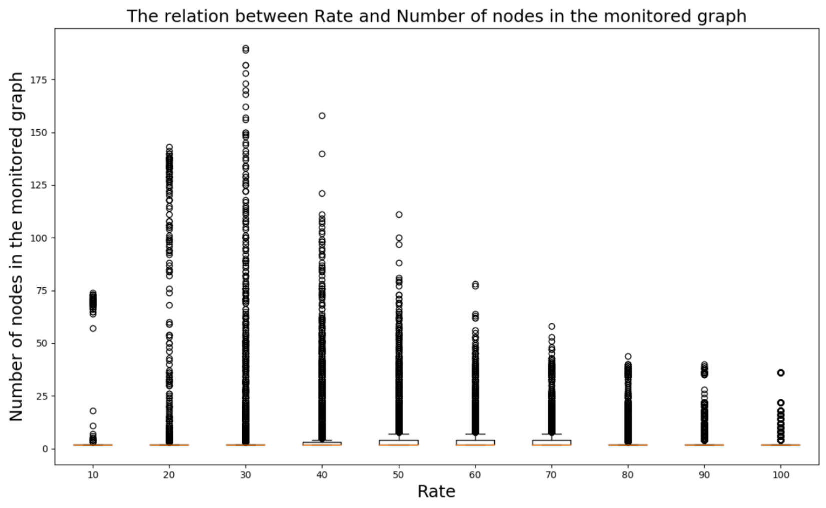


Figure 5.30. Colt Telecom: The relation between rate and monitored nodes.

In the box plot, Fig. 5.30, the value of the median in all the boxes is equal to 1 and the potential outliers, instead, decrease the value with the increasing of the rate.

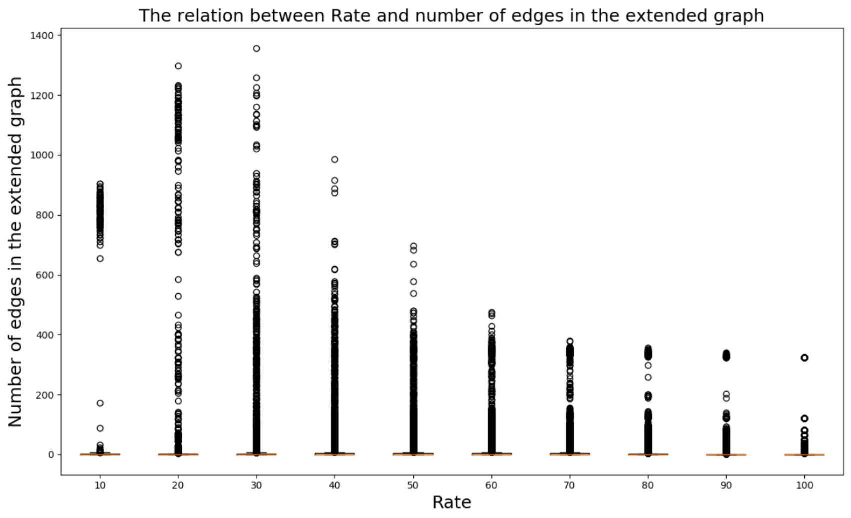


Figure 5.31. Colt Telecom: The relation between rate and extended edges.

The box plot in the Fig. 5.31 has on the x axis always the rate and on the y axis the number of clusters edges in the extended graph. The outliers decrease the value with the increasing of the rate.

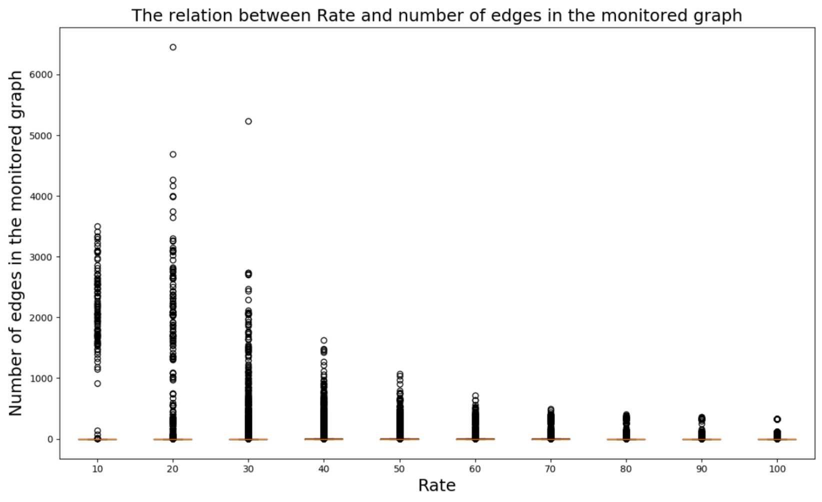


Figure 5.32. Colt Telecom: The relation between rate and monitored edges.

The box plot in the Fig. 5.32, instead, has on the x axis the rate and on the y axis the number of clusters edges in the monitored graph. The value of the median in all the boxes is equal to 1 and the potential outliers, instead, decrease the value with the increasing of the rate.

5.4.3 Cogent Communications topology

The last testing scenario concerns Cogent Communications topology. The first box plot has on the x axis always the rate and on the y axis the total number of clusters. As we can see in the Fig. 5.33, the number of clusters increases when increasing the percentage of selected nodes and the values are all quite close to the median. In the graph below the curve polynomial fitting is plotted using the polyfit function provided by the numpy library. This function returns a vector of coefficients p equlas to: [4.66357925e-04 3.01312539e-01 -4.25581667e+01].

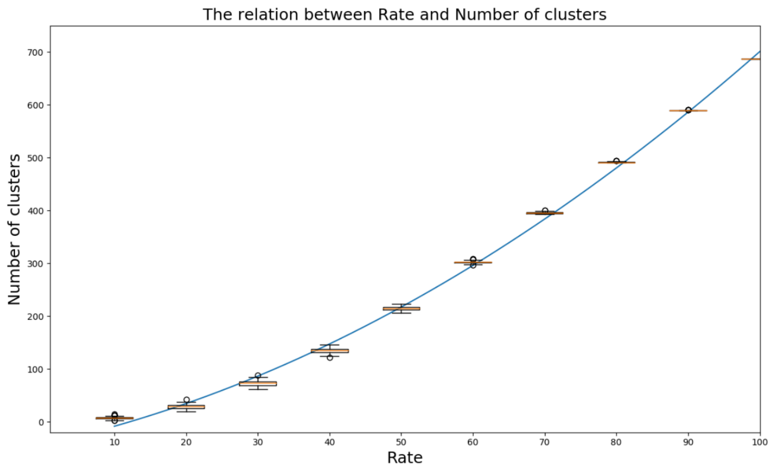


Figure 5.33. Cogentco: The relation between rate and numbers of clusters with polyfit.

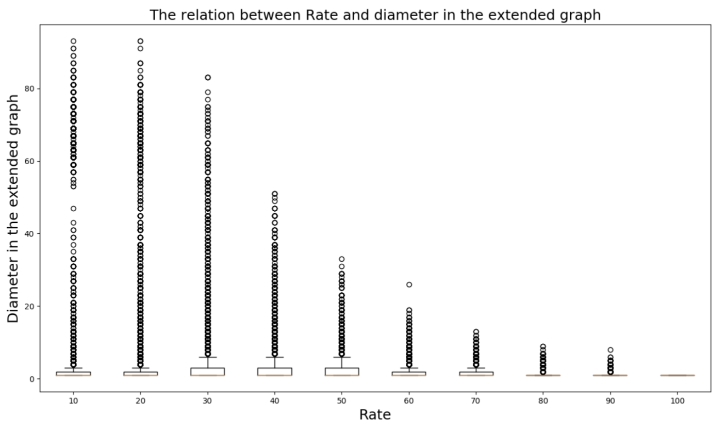


Figure 5.34. Cogentco: The relation between rate and extended diameter.

The second box plot, Fig. 5.34, has on the x axis the rate and on the y axis the value of the cluster's diameter calculated on the extended graph. The value of the median in all the boxes is the same, it is equal to 1, and the potential outliers decrease the value with the increasing of the rate.

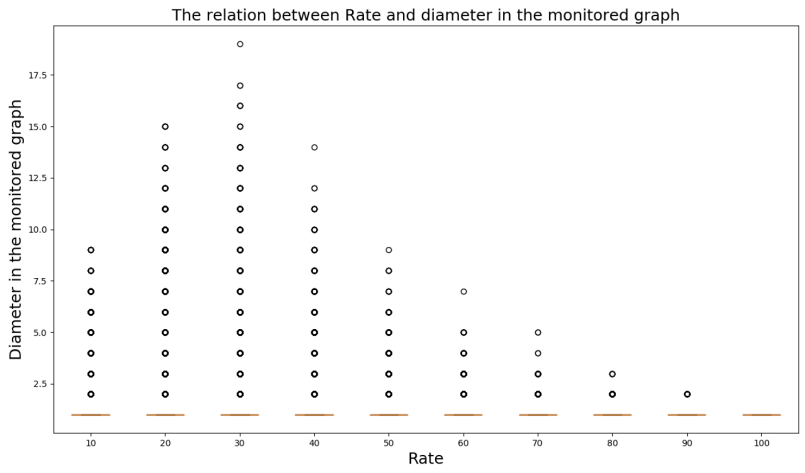


Figure 5.35. Cogentco: The relation between rate and monitored diameter.

The box plot, Fig. 5.35, has on the x axis the rate and on the y axis the value of the diameter of the cluster calculated on the monitored graph. The value of the median in all the boxes is equal to 1. The potential outliers, instead, decrease the value with the increasing of the rate.

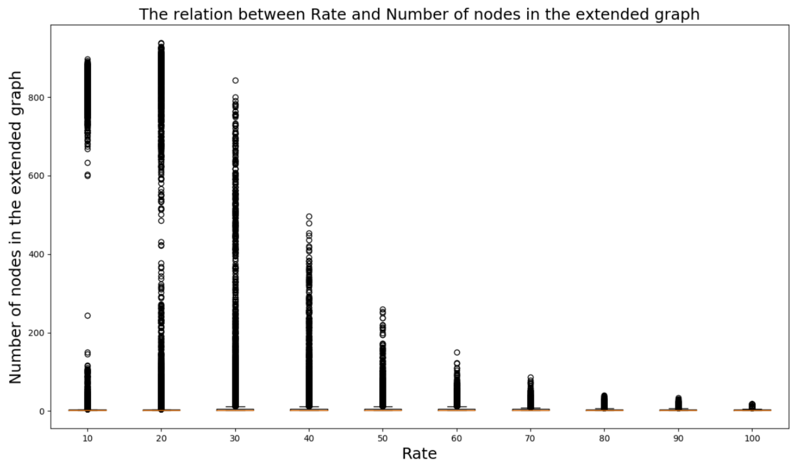


Figure 5.36. Cogentco: The relation between rate and extended nodes.

The box plot in the Fig. 5.36 has on the x axis always the rate and on the y axis the number of the cluster's nodes in the exdended graph. The value of the median in all the boxes is equal to 1. The potential outliers, instead, decrease the value with the increasing of the rate.

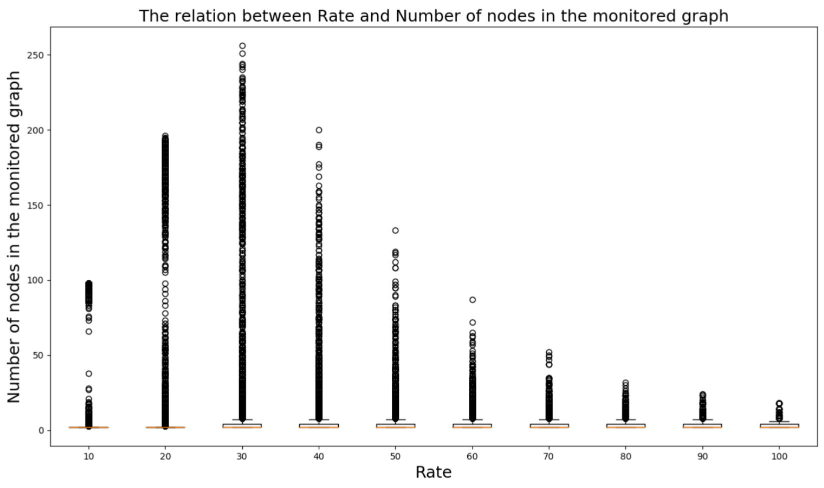


Figure 5.37. Cogentco: The relation between rate and monitored nodes.

The box plot in the Fig. 5.37 has on the x axis always the rate and on the y axis the number of the cluster's nodes in the monitored graph. As we can see in the Fig. 5.37, the value of the median in all the boxes is equal to 1 and the potential outliers, instead, decrease the value with the increasing of the rate.

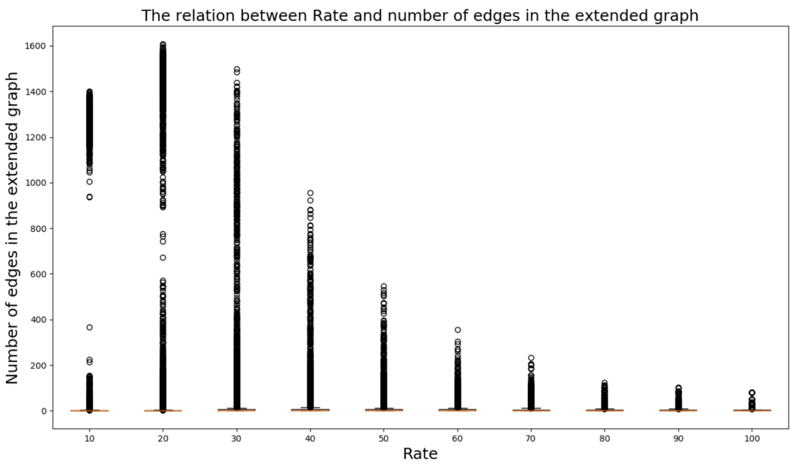


Figure 5.38. Cogentco: The relation between rate and extended edges.

The box plot in the Fig. 5.38 has on the x axis always the rate and on the y axis the number of clusters edges in the extended graph. The value of the median in all the boxes is equal to 1 and the outliers, instead, decrease the value with the increasing of the rate.

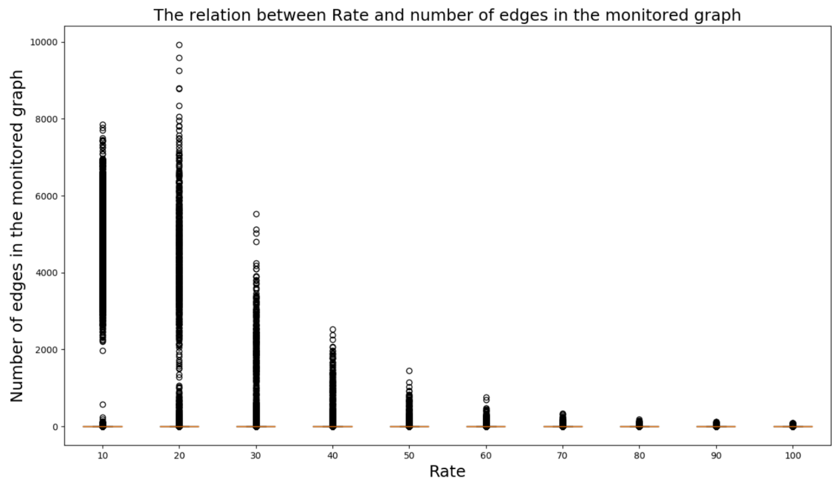


Figure 5.39. Cogentco: The relation between rate and monitored edges.

The box plot in the Fig. 5.39, instead, has on the x axis the random percentage of the selected monitored nodes and on the y axis the number of clusters edges in the monitored graph. The value of the median in all the boxes is the same, it is equal to 1 and the potential outliers, instead, decrease the value with the increasing of the rate.

5.4.4 The relation between nodes and edges

In the following box plots we have on the x-axis the number of selected nodes in the monitored graph and on the y-axis the number of edges in the monitored graph. These three plots represent how y varies as a function of x in three different topologies: Geant, Colt Telecom and Cogent Communications. Fig. 5.40 shows how the number of edges in the monitored graph varies as a function of x, in the Geant topology. There are 10 boxes, one for each rate. As we can see, the values are all quite close to the median therefore there is a little variability.

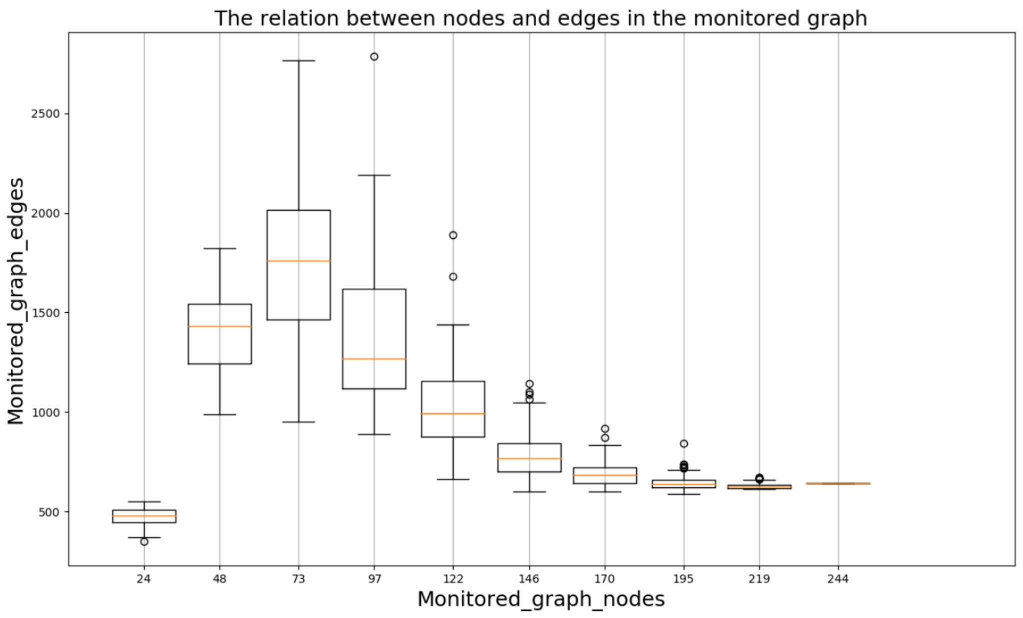


Figure 5.40. The relation between nodes and edges for Geant.

Fig. 5.41 and Fig. 5.42 show how the number of edges in the monitored graph varies as a function of x, in the Colt Telecom and Cogent Communications topology. There are always 10 boxes, one for each rate (the percentage of the selected monitored nodes). There is a little variability in both the plots, in fact the values are all quite close to the median. The value of y, the monitored edges, decreases as the nodes increase. So if you select a small percentage of nodes, for example 10, 20 or 30 percent, the number of edges increases. Otherwise, if a higher percentage of nodes is selected, more than 50 percent, for example 60, 70, 80 percent, the number of monitored edges decreases.

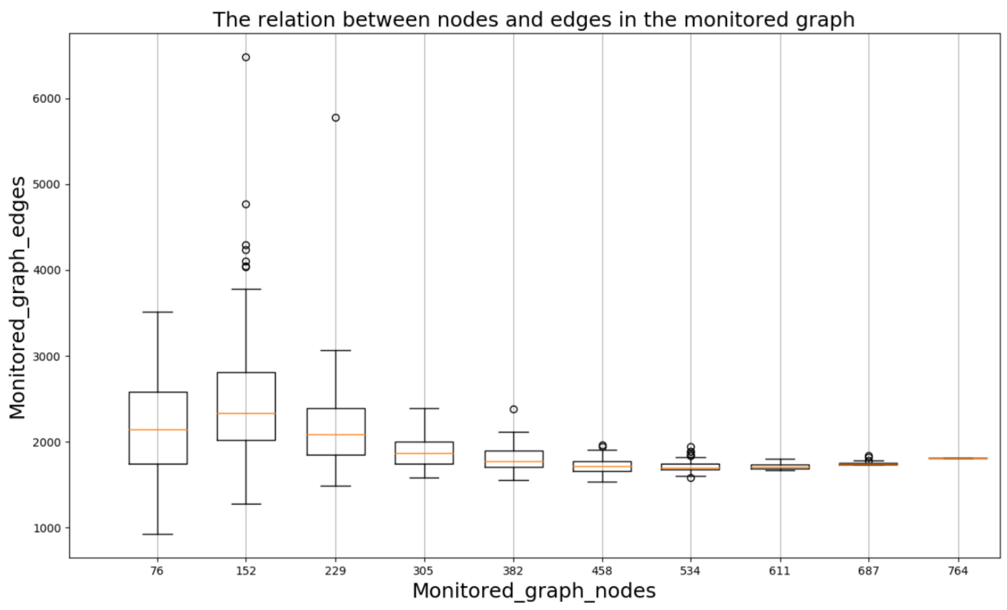


Figure 5.41. The relation between nodes and edges for Colt Telecom.

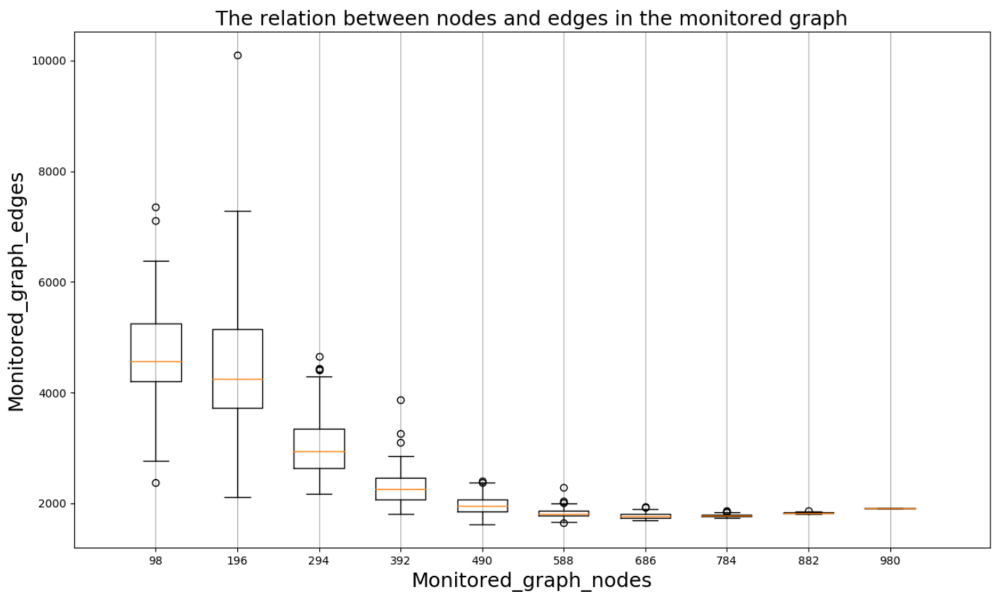
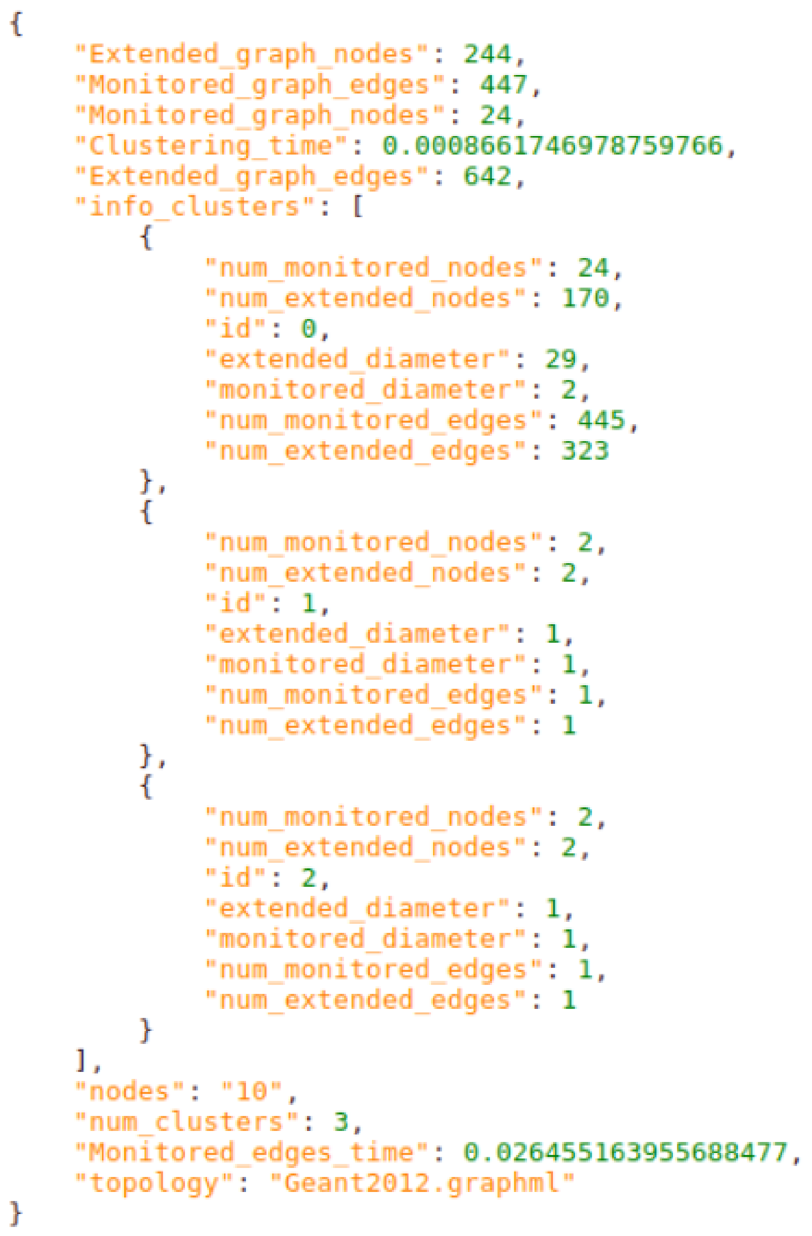


Figure 5.42. The relation between nodes and edges for Cogentco.

5.4.5 Execution time

Another test concerns the execution time. In particular, two times are considered. We measure the time to calculate the edges in the monitored graph (monitored_edges_time), and the time to calculate the clusters, (clustering_time).
Several graphs were represented but the most significant graph is in the following. The graph represents the monitored_edges_time number that depends on the value of monitored_graph_nodes number. It has on the x-axis the number of nodes in the monitored graph and on the y-axis the time needed to calculate the edges in the monitored graph itself. The network topologies considered are three respectively: Geant, Colt Telecom and Cogentco (Cogent Communications). The first topology has 40 nodes and 61 edges, the second 153 nodea and 191 edges and the third 197 nodes and 245 edges.
The experiment consists to starts the CompleteIterativeClustering_v5.py and the CompleteRecursiveClustering_v5.py scripts 100 times. These scripts allow to calculate the clusters given a specific topology. The first is the implementation of the iterative clustering algorithm instead the second is the implementation of the recursive one. Both require, as arguments from the command line, the name of the graphml file, in which the network topology is contained, and a value that can be either an integer or a string. The rate is the integer value. A number value ranging from 0 to 100 that represents the random percentage of the selected nodes. The string, instead, is a list of specific interfaces on which the clustering script is applied. So, in the output, it is returned a json file with the results of the 100 iterations. For each iteration there is a report that contains different information. An example of report is in the Fig. 5.43.
As we can see, we have different information for each iteration. These are respectively: the number of nodes and edges in the extended graph, the number of nodes and edges in the monitored graph, the clustering time, the monitored edges time, the selected rate, the name of chosen topology, the number of clusters and a list, info_clusters, with the information about each cluster.
This set of information is: the id of cluster, the number of nodes and edges in the monitored graph, the number of nodes and edges in the extended graph and the diameter calculated both in the extended and in the monitored graph. This information is useful to plot the graph, in particular the used fields are: the Monitored_edges_time, the Clustering_time and the Monitored_graph_nodes.



```
{
    "Extended_graph_nodes": 244,
    "Monitored_graph_edges": 447,
    "Monitored_graph_nodes": 24,
    "Clustering_time": 0.0008661746978759766,
    "Extended_graph_edges": 642,
    "info_clusters": [
        {
            "num_monitored_nodes": 24,
            "num_extended_nodes": 170,
            "id": 0,
            "extended_diameter": 29,
            "monitored_diameter": 2,
            "num_monitored_edges": 445,
            "num_extended_edges": 323
        },
        {
            "num_monitored_nodes": 2,
            "num_extended_nodes": 2,
            "id": 1,
            "extended_diameter": 1,
            "monitored_diameter": 1,
            "num_monitored_edges": 1,
            "num_extended_edges": 1
        },
        {
            "num_monitored_nodes": 2,
            "num_extended_nodes": 2,
            "id": 2,
            "extended_diameter": 1,
            "monitored_diameter": 1,
            "num_monitored_edges": 1,
            "num_extended_edges": 1
        }
    ],
    "nodes": "10",
    "num_clusters": 3,
    "Monitored_edges_time": 0.026455163955688477,
    "topology": "Geant2012.graphml"
}
```

Figure 5.43. Output report from one iteration.

In the first subsection is showed the relation between the monitored edges time and the nodes. In the second subsection, instead, is showed the relation between the clustering time and the nodes both with the application of the iterative and the recursive clustering algorithm. All the plots represent how y varies as a function of x in three different topologies: Geant, Colt Telecom and Cogent Communications.

Monitored edges time

Fig. 5.44 shows how the time varies as a function of x, in the Geant topology. Where x is the number of nodes in the monitored graph and y is the time to calculate the edges. There is a little variability, the values are all quite close to the median except for rare cases. The time increases as the selected nodes increase.

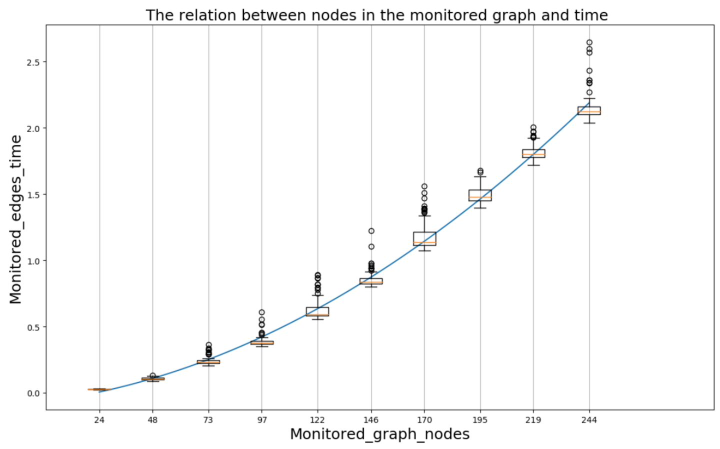


Figure 5.44. Géant: The relation between nodes and monitored edges time with polyfit.

The polyfit function returns a vector of coefficients p equlas to: [2.87552995e-05 2.21897830e-03 -6.45052168e-02].

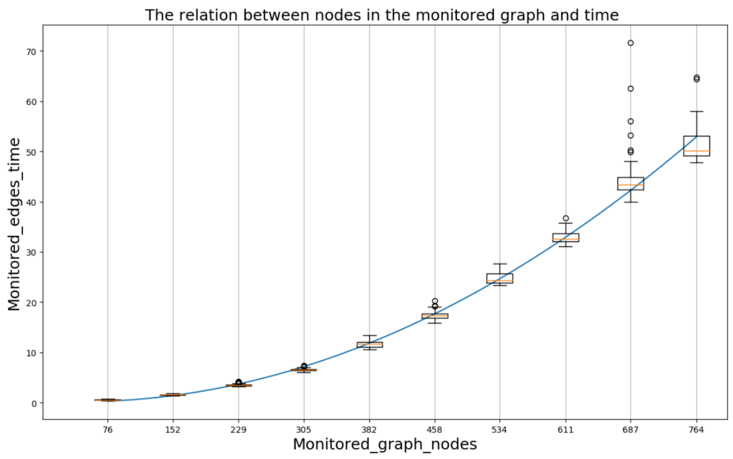


Figure 5.45. Colt Telecom: The relation between nodes and monitored edges time with polyfit.

Fig. 5.45 shows how the time varies as a function of x, in the Colt Telecom topology. There are always 10 boxes, one for each rate. There is a little variability in fact the values are all quite close to the median. As the selected nodes increase, the time to calculate the edges in the monitored graph increases. In fact, for low percentages of selected nodes, the value of time on the y-axis is lower, instead of for high percentages of selected nodes the time is greater. The vector of coefficients p in this case is: [1.01292329e-04 -8.60993588e-03 3.41452651e-01].
The last graph in the Fig. 5.46 shows how the time varies as a function of x, in the Cogent Communication topology. There are always 10 boxes, one for each rate. There is a little variability in fact the values are all quite close to the median. As we can see from the figures below, as the selected nodes increase, the time to calculate the edges in the monitored graph increases. In fact, for low percentages of selected nodes, the value of time on the y-axis is lower, instead of for high percentages of selected nodes the time is greater.

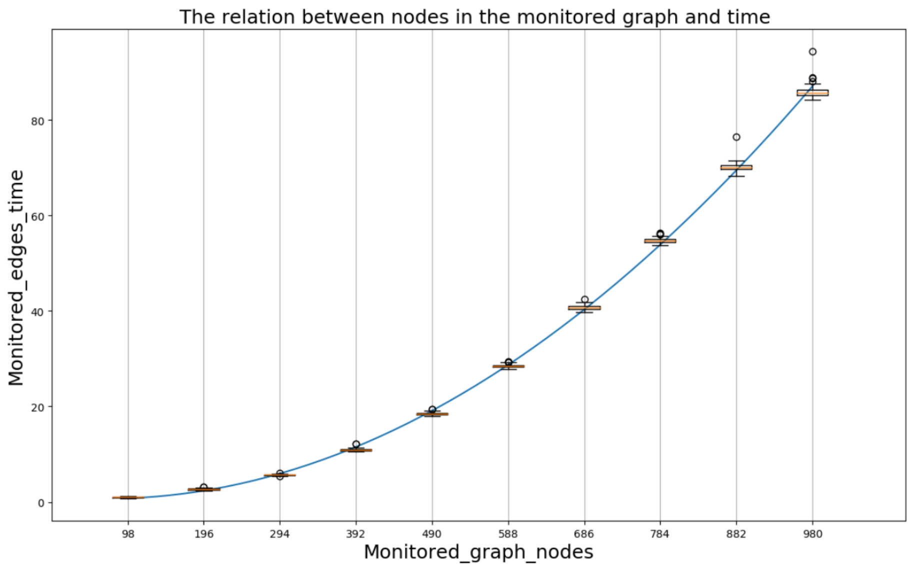


Figure 5.46. Cogentco: The relation between nodes and monitored edges time with polyfit.

In the graph above the curve polynomial fitting is plotted using the polyfit function provided by the numpy library. The degree is always an int value equals to 2. The polyfit function returns a vector of coefficients p equals to: [1.04078310e-04 -1.44115449e-02 1.17258769e+00].

Clustering time

In the graphs below are showed the relation between the time needed to calculate the clusters and the number of nodes in the monitored graph. These values are calculated using resepectively the recursive and the iterative clustering algoritms. Both scripts take the name of the .graphml file, which contains the information about the topology, from the command line, and save the resulted clusters in a .json file. In the first (the recursive one), the code takes in input the adjacency matrix, that is the matrix that for each link (h1, h2) of the graph, has a 1 in position (1,2). The second implementation (the iterative one), instead, takes in input the list of links in both the input and output directions (h1, h2) and (h2, h1).

Iterative clustering algorithm In the graphs below the values plotted are calculated using the iterative clustering algorithm.

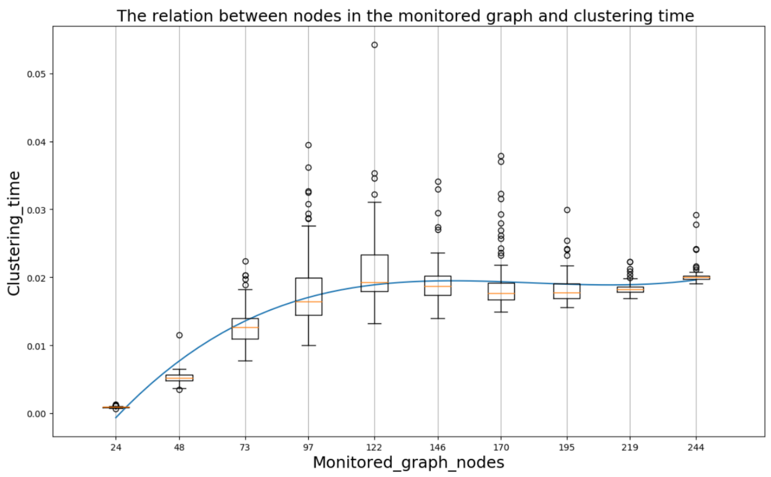


Figure 5.47. Géant: The relation between nodes and clustering time using the iterative algorithm.

As we can see in the Fig. 5.47 the graph shows the relation between the clustering time and the nodes. There is a little variability in fact the values are all quite close to the median except for some values, the outliers, that are located outside the fences of the boxplot. The time to calculate the clusters increases with the increasing of the nodes in the monitored graph. In the graph the curve polynomial fitting is plotted using the polyfit function provided by the numpy library. This function returns a vector of coefficients p that minimises the squared error. In the graph above the coefficients p are: [5.69032886e-09 -3.09880546e-06 5.47197919e-04 -1.20748912e-02].

The second graph concerns the Colt Telecom topology. The values are all quite close to the median except for rare cases. The time to calculate the clusters increases with the increasing of the nodes in the monitored graph. The polyfit function returns a vector of coefficients p equals to: [1.09814027e-09 -1.62664435e-06 9.02571475e-04 -4.60743963e-02].

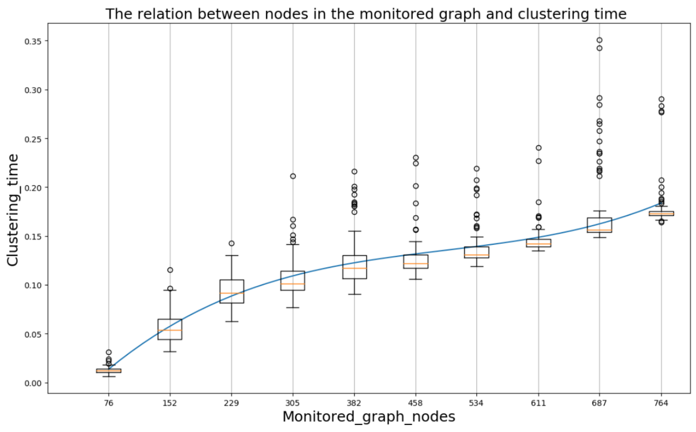


Figure 5.48. Colt Telecom: The relation between nodes and clustering time using the iterative algorithm.

The last graph concerns the Cogent Communications topology. The time to calculate the clusters increases with the increasing of the nodes in the monitored graph. The polyfit function returns a vector of coefficients p equals to: [1.31625791e-09 -2.29833378e-06 1.27315515e-03 -5.90698015e-02].

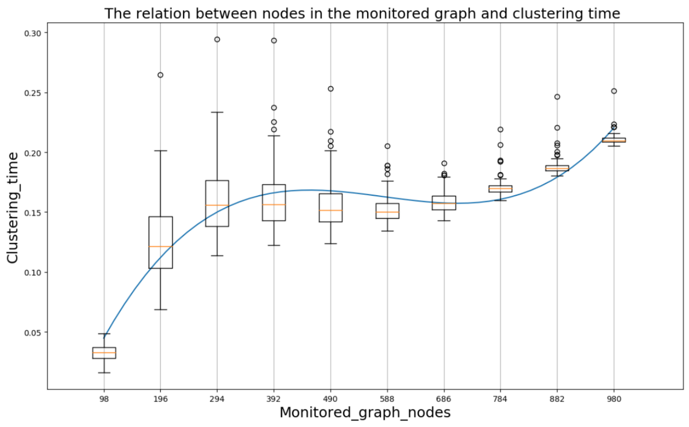


Figure 5.49. Cogentco: The relation between nodes and clustering time using the iterative algorithm.

Recursive clustering algorithm In the graphs below the values plotted are calculated using the recursive clustering algorithm. The first graph concerns the Geant topology.

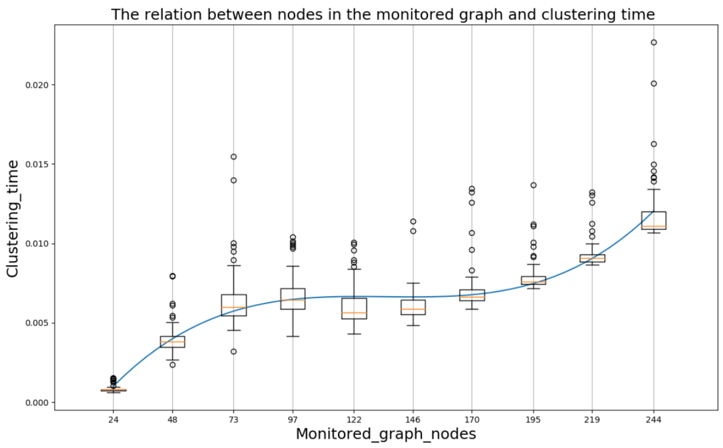


Figure 5.50. Géant: The relation between nodes and clustering time using the recursive algorithm.

Fig. 5.50 shows the relation between the clustering time and the nodes in the monitored graph. The values are all quite close to the median except for rare cases. The time to calculate the clusters increases with the increasing of the nodes in the monitored graph. The polyfit function returns a vector of coefficients p equals to: [4.28600359e-09 -1.73384402e-06 2.32032079e-04 -3.62816207e-03].

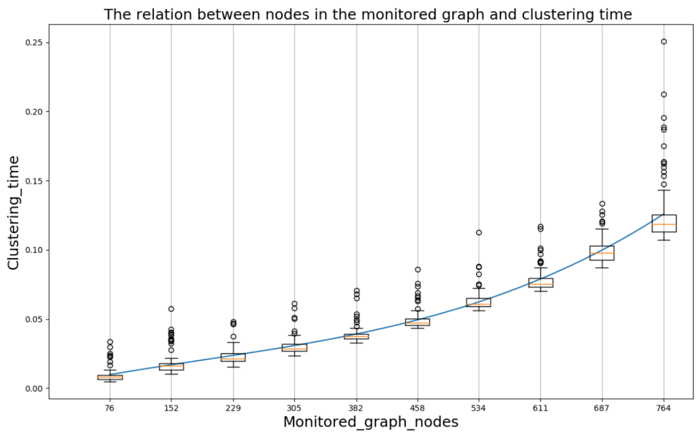


Figure 5.51. Colt: The relation between nodes and clustering time using the iterative algorithm.

The second graph concerns the Colt Telecom topology. The time to calculate the clusters increases with the increasing of the nodes in the monitored graph. The polyfit function returns a vector of coefficients p equals to: [2.93717507e-10 -1.68248245e-07 1.19741865e-04 1.61866303e-03].

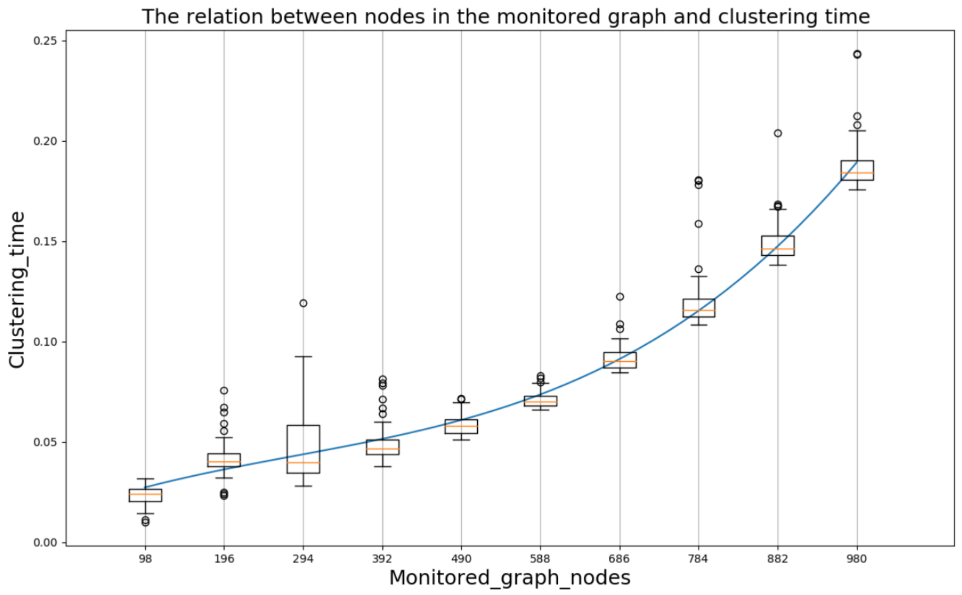


Figure 5.52. Cogentco: The relation between nodes and clustering time using the iterative algorithm.

The last graph concerns the Cogent Communications topology. The time to calculate the clusters increases with the increasing of the nodes in the monitored graph. The polyfit function returns a vector of coefficients p equals to: [2.79954603e-10 -2.39065580e-07 1.42942540e-04 1.52339895e-02].

Chapter 6

Conclusions and future work

The thesis turned out to be more interesting and challenging of what had been envisioned at the beginning, because of the many problems and choices that came out during this work. The limitations due to the technology and tools used and the impossibility to solve easily some uses cases, forced us to think to more various solutions to keep the project as coherent as possible to the initial idea, that is to say, the implementation and the evaluation of the multipoint alternate marking technique. The first problem encountered concerns a feature present in the open vSwitches used in the emulation of the real network topology with Mininet: The flow caching mechanism. This problem has been highlighted by considering the input and the output counters in two open vSwitches s1 and s2 connected by a point-to-point link in the examined topology. After generating traffic, the output counters of s1 and the input counters of s2 didn't match. The issue concerns the ovs's flow caching feature. In ovs switches there are two paths: the fast-path and the slow-path. The rules "active", that are more often used by traffic, are put in the fast path. The rules, instead, that match more rarely remain in the slow path. In this way, most of the traffic doesn't pass through the slow path. The ovs will take the openFlow flows and generate datapath flows in the fastpath. Occasionally, it will pull the stats from the datapath flows and update the appropriate openFlow rule stats. The problem, so, is that when the traffic is marked to 0, only the rules that mark and count the packets to 0 are active and so they are in the datapath. These two rules are merged into a single rule in the datapath. When the marking rule has changed, the 2 rules that are active change. Therefore the single rule in the datapath is changed. During the update of the single rule in the datapath, it may happen that the part of the marking is updated, but the part of the counting is not updated, so the packets are counted in the wrong way.

This happens bacause when you change the active flows, probably the datapath flow is not modified and then some of traffic that had the old tos value is then being associated with the currently active openFlow rules. So, the solution adopted is to adding other open vSwitches, switches T, in the topology between the switches S and the hosts and distributing the marking rules and the counting rules in different switches. The marking rules are installed in switches T and the counting rules are installed in switches S. In this way it is avoided the merging of these two rules in a single rule and then the problem of updating the merged rule. Because we have two different rules in two different switches. The second problem encountered concerns the reading of counters, in partcular the reading of the output counters. Because openflow v1.0 doesn't provide counters on the output port. In particular openflow doesn't provide a field match on the output port. So the solution adopted is to use the tos field of the ip header as a temporary metadata in which is written the output port number. Some bits of the tos field are used for the output port and another bit is used for the marking. Before forwarding the packet to the output port the tos field is resetted, deleting the port number and keeping the bit marking. After the implementation of the multipoint alternate marking technique (The controller program), we performed a lot of tests on real network topologies of small, medium and big size, to check the accuracy of the controller measurements. Two different tests are performed. The first one without emulating a certain percentage of losses in the network. The second one, emulating a percentage of losses equals to 1% in all links between switches S in the network. In both cases we started the tests with L=120s, where L is the bit marking interval, then we progressively reduced the duration of this interval to 60s, 30s and 20s. The traffic generation was always all-to-all but the monitored traffic changed. It will be one-to-all, for example from a specific node to all the other nodes, or all-to-all. So we have some graphs that show the total number of packets lost for each link in the topology, in each monitoring period. In all the plotted graphs we can notice that the counters values are the same both in results.json and in summary.dat. This means that the controller measurements are correct. We noticed that with values smaller than L=20s we have errors. The error is that the counters in the controller and those in the "summary.dat" didn't match. The problem is due to the implementation of the controller program. The insertion of the rules in all the switches in the topology is not done in parallel, so if you set a low marking period L for example L=5s, the flowrules are not installed correctly in all the switches and then I read wrong values in the counters. The time needed to insert the rules in all the switches is longer than the period L. The possible solution, not implemented, is to execute all the functions used to install the flow rules in all the switches in different threads.

This is a possible solution. In the first part of the thesis we have seen how to determine the number of packets lost globally in the monitored network, exploiting only the data provided by the counters in the input and output nodes. But we can also exploit the data provided by the other counters in the network to converge on the smallest subnetworks, that we call clusters, where packet losses occur.
So the second part of the thesis is about the creation of the clusters, given a network topology. The algorithms used are two. One recursive and the other one iterative. The clustering algorithms, both iterative and recursive, were applied to all the network topologies present in the dataset of The Internet Topology Zoo. It has been noticed that in the transition from the physical graph to the extended graph, the dimension of the network changes a lot. We noticed, from the plotted graphs, that the dimension of the extended graph is greater than the physical one. The interface-level graph is much larger than the physical graph. This was clear in the plotted graphs in which the relation is close to linear. To test their efficiency and scalability, both clustering algorithms have been applied to different real network topologies of various sizes. The experiment consists in launching the clustering algorithm 100 times for each topology and for each percentage of the selected nodes to be monitored. We have for each topology and for each selected rate 100 different executions, with a total of 1000 executions for a single topology. This experiment aims at analyzing the behavior of the clustering algorithm and to evaluate clusters' features. To describe the clusters' features it was used a box plot. In statistics, the box plot is a graphical representation used to describe the distribution of a sample through simple dispersion and position indices. The median, instead, is defined as the value/set of values that are in the middle of the distribution. We noticed that the number of clusters increases when increasing the percentage of the selected monitored nodes. We also noticed that the values are all quite close to the median, therefore there is a little variability. Another important examined cluster's feature is the diameter. The diameter d of a graph is the greatest distance between any pair of vertices. To find the diameter of a generic graph, first we find the shortest path between each pair of vertices, then the greatest length of any of these paths is the diameter of the graph. We noticed that the cluster's diameter both in the extended and in monitored graph is most often equal to 1. This means that the dimension of a cluster is small and this is important because this means that it is simple to learn where packet losses occur. Another feature examined is the relation between monitored nodes and edges in three different topologies: Geant, Colt Telecom and Cogent Communications. We have some graphs in which on the x-axis there is the number of monitored nodes for a given topology and on the y-axis there is the number of monitored edges for the same topology.

We noticed, from the plotted graphs, that the values are all quite close to the median therefore there is a little variability. The number of monitored edges decreases as the selected nodes increase. Another important feature examined is the execution time. We measure the time to calculate the clusters (clustering_time) and the time to calculate the edges in the monitored graph (monitored_edges_time). We measure the execution time for both the Geant (a topology with 40 nodes and 61 edges) and Cogent Communications (a topology with 197 nodes and 245 edges). We noticed that the values are all quite close to the median except for rare cases therefore there is a little variability in all the plotted graphs. The clustering time increases with a larger number of nodes in the monitored graph. This happens both using the iterative and the recursive algorithm.

Bibliography

[1] *TCP congestion control.* URL: https://en.wikipedia.org/wiki/TCP_congestion_control.

[2] *Y.1731 : OAM functions and mechanisms for Ethernet based networks.* URL: https://www.itu.int/rec/T-REC-Y.1731-200605-S/en.

[3] *OAM Messages.* URL: https://www.juniper.net/documentation/en_US/junose13.3/topics/concept/ethernet-oam-lfm-messages.html.

[4] G. Fioccola et al. *Alternate-Marking Method for Passive and Hybrid Performance Monitoring.* URL: https://tools.ietf.org/html/rfc8321.

[5] A. Sapio et al. *Passive monitoring in multipoint packet networks.* Tech. rep. Politecnico di Torino, 2018.

[6] B. Claise, B. Trammell, and P. Aitken. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information.* URL: https://tools.ietf.org/html/rfc7011.

[7] G. Fioccola et al. *Multipoint Alternate Marking method for passive and hybrid performance monitoring.* 2018. URL: https://datatracker.ietf.org/doc/draft-fioccola-ippm-multipoint-alt-mark/.

[8] N. Feamster, J. Rexford, and E. Zegura. *"The road to SDN." Queue 11, no. 12 (2013): 20.* URL: https://www.cs.princeton.edu/~jrex/papers/queue14.pdf.

[9] F. Xenofon, K. M. Marina, and K. Kontovasilis. *Software Defined Networking Concepts. Software Defined Mobile Networks (SDMN): Beyond LTE Network Architecture.* Tech. rep. (2015): 21-44.

[10] N McKeown et al. *OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review 38, no. 2 (2008): 69-74.* Tech. rep.

[11] *SDN architecture, Openflow*. URL: https://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-59/161-sdn.html.

[12] *Flowgrammable, Openflow*. URL: http://flowgrammable.org/sdn/openflow/.

[13] B. Pfaff et al. *The Design and Implementation of Open vSwitch. In NSDI (pp. 117-130)*.

[14] *Accelerating Open vSwitch to "Ludicrous Speed"*. URL: https://networkheresy.com/tag/megaflows/.

[15] *The Mininet Emulator*. URL: http://mininet.org/overview/.

[16] *The Internet Topology Zoo*. URL: http://www.topology-zoo.org/dataset.html.

[17] *GÉANT*. URL: https://www.geant.org/Networks.

[18] *BICS*. URL: https://bics.com/.