



Bachelor's Third Year Project
Telecommunications and Informatics Engineering
Instituto Superior Técnico, Portugal

Python implementation of IGMPv3 protocol

by

Martim de Barros e Carvalhosa Tavares
96758

This Project was conducted under the supervision of:

Professor Rui Valadas
Eng. Pedro Oliveira

Index

1 – ABSTRACT	3
2 – INTRODUCTION	4
3 – BRIEF INTRODUCTION TO IGMPV3	6
3.1 – Message headers	7
3.2 – The Querier Election – state machine	9
3.3 – Messages – state machine	10
4 – THE CODE	12
4.1 – Packet	13
4.2 – Interface	16
4.3 – Router State	18
4.4 – Group State	20
5 – TESTS	21
5.1 – Running the main script	22
5.2 – Testing the interface’s sending and receiving capabilities	23
5.3 – Testing the Querier election	23
5.4 – Testing the: Reception of reports	24
6 – CONCLUSIONS	28
7 – BIBLIOGRAPHY	29

1 – Abstract

Multicast is a transmission method that is more efficient than other routing methods (e.g., unicast or broadcast) when the objective is to send information to groups of users on the Internet. In such situation, IGMP is the protocol whereby hosts signal their interest in participating in a certain multicast group, and version 3 of the protocol, allows that hosts signal their interest in specific sources of a multicast group. In this project, we implemented the server side of IGMPv3 in Python3 and tested it in Kathará, a network emulation environment. The tests demonstrated the correctness of the implementation.

2 – Introduction

Until a recent time, most of the Internet traffic's packets were transmitted via Unicast. This means that if a transmitter wanted to send data packets to multiple destinations, then it needed to send at least as many packets as the number of receivers. During many years, Unicast transmission seemed to be good enough. However, in the last few years, big companies like Zoom, Netflix, YouTube and Facebook, which have been stepping up in the market, have caught millions of clients with their real-time multimedia applications like video and streaming that required the transmission of huge amounts of packets to reach all of the users online, at the same time. Unicast became obsolete for such purpose, especially for long distance networks.

It was necessary to develop another method of data transmission that allowed sending packets to many different networks which would route them only to the interested hosts. This way, Multicast offers better performance in multimedia transmissions.

The first version of the IGMP – Internet Group Management Protocol – was first specified in 1989 [RFC 1112] at Stanford University, and had the goal to manage dynamic groups for IP multicast transmissions, whereby this management wouldn't run via the sending device itself, but via the integrated routers.

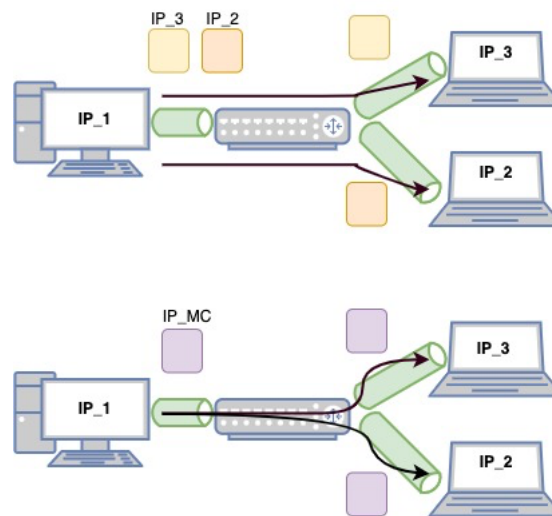


Fig. 1 – The author's representation of unicast problem for multi-destination packets

Multicast is the transmitting method that allows sending a data packet to more than one destination, at the same time. In MC – Multicast – the transmitter sends the data packets only once, and therefore it is up to the hosts to be able to catch the transmission for themselves. For that matter, hosts are organized in specific groups and routers only need to know to send the packets to the group and not to any specific unicast IP address.

This method reduces the traffic in many situations, e.g. when multiple clients are watching a football match, directly coming from an online server.

This project consisted in implementing the code for the internet protocol IGMPv3 – Internet Group Multicast Protocol version 3 – so that routers in a small network would be able to process IGMP messages that would arrive at their interfaces and so would know how to deal with each request.

The code was developed in Python programming language and its implementation was done focusing on an OOP – Object Oriented Programming – approach. Therefore, each part of the code was designed as a Class object. Every activity performed by the object was defined in its methods inside the class itself. In order to test the protocol developed in a network environment, it was also used an emulated network using Kathará, where it was configured a small network of two client computers; two routers; and one switch. The Python code was tested by running the code inside each router's Linux shell and a client-side script in the hosts' shells.

The overall system was designed according to the RFC3376 specifications and requirements for the right execution and performance of the protocol.

The structure of this report was previously defined in the index section. After this introduction, the paper starts by briefly explaining the protocol as it is defined in the RFC (**Brief introduction to IGMPv3**) where it is clarified the differences between the 3 versions of IGMP; the process of election of the Querier router; the actions performed by the router when facing an event which could be the reception of a message or an expired timer; the header format for all the messages exchanged. Moving on to **The Code** section, this part explains the methodology adopted to build the code implemented in Python and all its correlation between files. After that comes the **Testing** part where it is described the tests performed in order to assure the correct behavior of the protocol. Finally, and before the bibliography, we have the **Conclusions** section where it is mentioned the success of this project,

3 – Brief introduction to IGMPv3

The Internet Group Management Protocol (IGMP) is a Layer 3 protocol from the Internet used by hosts and adjacent routers on IPv4 networks to establish multicast group memberships. These multicast group memberships allow the network to support multiple requests with fewer traffic in it. It is possible as it sends the data from the sources of a multicast group only once instead of sending multiple copies of the data, one to each host that wants to receive from that group.

This way IGMP can be used for one-to-many networking applications such as online streaming video and gaming, and allows more efficient use of resources when supporting these types of applications.

IGMP is used on IPv4 networks. Multicast management on IPv6 networks is handled by Multicast Listener Discovery (MLD) which is a part of ICMPv6 in contrast to IGMP's bare IP encapsulation.

This protocol has had 3 versions so far. Initially, in version 1, there were two message types – Membership Queries and Reports. The queries were sent by the Querier router to the address 224.0.0.1 every 125 seconds in order to know if there still were hosts that wanted to receive from the multicast group. If at least one host was still interested in the group, then only one host should respond with a Report message.

In version 2, a few things diverge from the original protocol. To start with it, this version implements a mechanism to elect the router that is going to be the Querier. Another difference is that version 2 allows a group-specific query to only query membership in a single group instead of all groups. Although in version 1, hosts would only report if they still wanted to participate in the group, in this version they report either if they still want to receive from a certain group or if they no longer want to receive from such group. That is possible thanks to the enhancement of the Report message Type parameter in the header. In this version the Type parameter, in a Report, can take the values 0x16 or 0x17 (in hexadecimal) which represent the JOIN and the LEAVE group operations.

The latest version (v3) of the protocol, has a big difference from older versions which is that in version 3 hosts can specify the group sources from which they pretend or not to receive information. Version 3 also fixes many efficiency and network traffic problems. The first improvement and also a big difference from the older versions is the header format and also the message behavior. While older versions of IGMP had different message types for joining or leaving a group, version 3 supports both operations in the same message type. This means that the same header (0x22 – (version 3 membership report) can contain group records, each one with information regarding joining or leaving groups. It also supports multiple operation requests in the same message for example: if a host wants to join two groups, then it can state such desire in only one message as in older versions it would need to send two messages, one for each request.

3.1 – Message headers

This sub-section shows the structure of the IGMPv3 message headers. IGMPv3 only has 2 message headers: the Membership Query and the Membership Report. The first one being shown is the query header, in **Fig. 2**. The first line is almost common for both the query and the report header. It contains a “Type” field where it is specified if it is a query or a report; a “Max Response code” is also present to let the receiver know how much time it has to respond to the query and a checksum to assure the data integrity. Moving on in the query header, we have the “Group Address” which has the IP address of the MC group being queried (this option can be 0.0.0.0 if it is not a group-and-source-specific query); the “S” flag allows the receiver to know if it is supposed to suppress the group’s timer or not; the “QRV” – Querier’s Robustness Variable – is used to calculate timers; the “QQIC” – Querier’s Query Interval Code – specifies the query interval in which the Querier sends queries. All of the sources in the query are related to the group and are generally field if it is in a group-and-source-specific query.



Fig. 2 – Query’s header parameters and length

IGMPv3 Report has the structure presented in the following figure **Fig. 3**. Like it was mentioned in the last paragraph, the first line in the header is very similar to the query's first line but in this case the "Type" is filled with the value 0x22 (version 3 report). The report can contain as many group records as the host pretends. Each group record describes the host's intention on receiving or not data from a specific group. This way, if a host pretends to receive data from two specific groups e.g. 224.11.11.11 and 224.12.12.12 and does not pretend to receive from another one e.g. 224.13.13.13, then it can state a group record for each group specifying if it wants to keep in the group or not.

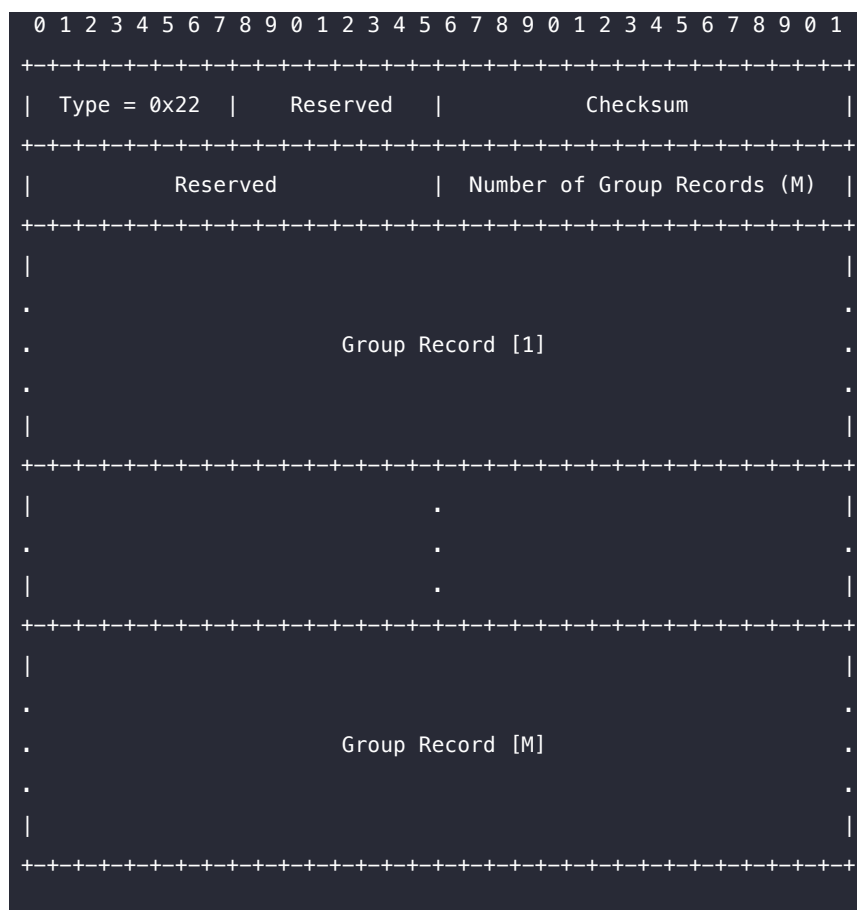


Fig. 3 – Report’s header parameters and length

The group record’s structure can be seen in the next figure (**Fig. 4**). These group records are a huge improvement from older versions of IGMP as clients can state the sources from which they want to receive data.



Fig. 4 – Group Record’s header parameters and length

3.2 – Querier Election – state machine

IGMP introduced the Querier election mechanism in version 2. In version 3 this idea was kept as it assures there is always one and only one router that is responsible to encourage hosts to give an update status to the routers on the groups and its sources states, as hosts respond to the queries sent by the Querier.

Just like version 2, in version 3 IGMP needs to have one multicast router per network to have a role in it. This specific role is designated as the **Querier** as mentioned above. The difference from this router to the others is that in a network it is crucial to have only one router who sends queries to be answered by hosts so that all routers can maintain state of the groups to know which routes should define for the packets received in their interfaces. The Querier is also responsible to verify the interest from the hosts on specific multicast groups.

If there were no Queriers, it would cause a problem. The problem would be the growth of message traffic in the network as every router would need to send queries to the network, in order to update and to know each host's desire to receive packets from multicast groups and from which sources.

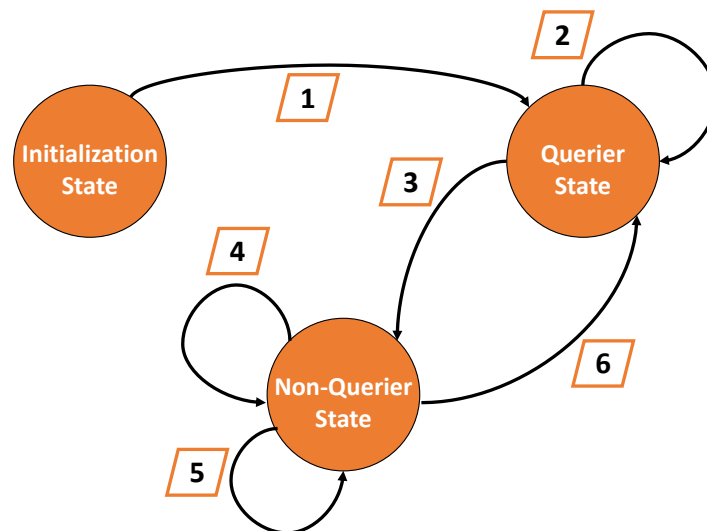


Fig. 5 – Router state machine during the Querier election

The figure above is a representation of this report's author interpretation on the state machine of a router when a Querier election begins. The numbers represent an event that causes a change in the router's state and are described in following lines.

- (1) Sends a query to all other MC routers in the network. Destination: 224.0.0.22
- (2) If the router receives a query from another router with a higher IP address, nothing happens to this router.
- (3) If it receives a query from a router with a lower IP address, then it records that router's IP as the new Querier and starts the "Other Querier Present" timer, according with the query's QQIC and the S flag.

- (4) If the router receives other queries from the Querier, then it starts the "Other Querier Present" timer again.
- (5) If it receives a query from a router with an IP address lower than the Querier's, then it records that router's IP as the new Querier and starts the "Other Querier Present" timer.
- (6) If the "Other Querier Present" timer expires and the router does not receive any query from the Querier, then it becomes the new Querier.

3.3 – Messages – state machine

Version 3 of IGMP also introduced a reduction on message header types. There are only two: the Membership Query and the Membership Report. The Membership Query is sent from the Querier with the purpose to get a response from one or more hosts in the network to whether or not they want to receive data from certain groups. It also sends this query for Group-Specific and Source-Specific intentions. The Membership Report can either be an answer to a query or not and it specifies the groups from which the host pretends to receive data and groups from which the host doesn't pretend to receive as well as each group's sources.

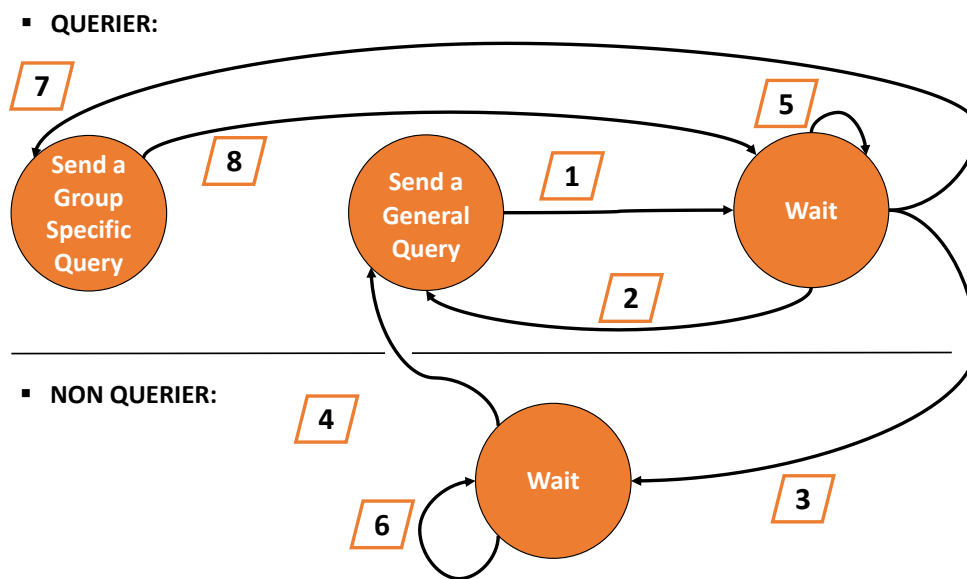


Fig. 6 – Reports and Queries events for the router's state machine

The figure above represents this report's author interpretation on the state machine of a router every time it either receives a query or a report or when a timer expires. The numbers represent an event that causes a change in the router's state and are described in following lines.

- (1) Sends a General Query to all other MC routers in the network, to announce himself as the Querier. Destination: 224.0.0.22. As the Querier, the router is supposed to

send these queries periodically, after the **Query Interval** defined in QQIC parameter and is 125 seconds by default.

- (2) The Querier sends a General Query after the **Query Interval** has passed.
- (3) The router receives a Query from a router with a lower IP address, and so becomes a non-Querier router, and therefore it starts the “**Other Querier Present Interval**” timer, which is the time before the router decides there is no Querier in the network.
- (4) If the “**Other Querier Present Interval**” timer expires, then the router decides there is no Querier in the network and becomes the Querier, and therefore it sends a General Query to all MC routers.
- (5) The MC router receives a report from a host. The MC router either creates a new state for the groups if none exists or refreshes timers for current state if the group already exists.
- (6) The MC router receives a report from a host. The MC router either creates a new state for the groups if none exists or refreshes timers for current state if the group already exists.
- (7) When the Querier receives a report from a host in which it states the desire to leave one or more groups it will send a group-specific-query for that/those groups.
- (8) The Querier starts the **Group membership interval** to decide there are no more hosts interested in the group and its specific sources. This happens if after this timer the router doesn’t receive any reports for that specific group. If there are no members the router updates this group’s state.

Regarding the Group Records, each has a “Type” parameter used to specify an instruction from the host on what it pretends to do with packets from the record’s sources. The following table is this report’s author interpretation, according to the RFC, on the actions performed by the routers when an incoming packet has a group record.

Operation Type	Mode	Querier	Actions performed
MODE_IS_INCLUDE	INCLUDE	Querier non-Querier	If the sources present in the report are not yet stored in the router, then they are added to the group's source list
	EXCLUDE	Querier non-Querier	If any source is present in sources list, then it is removed from it
MODE_IS_EXCLUDE	INCLUDE	Querier non-Querier	Will exclude the sources from the header that are not in the router's group state
	EXCLUDE	Querier non-Querier	If a source from the report is not in the router's group state, then it will be added
CHANGE_TO_INCLUDE_MODE	INCLUDE	Querier non-Querier	Does the same as a non-Querier but also sends a group and source specific query with the sources present in the group state but not in the received report. Adds the sources in the report that are not stored in the router's group state.
	EXCLUDE	Querier non-Querier	Does the same as a non-Querier. Also sends a group-specific query. Removes the sources from the group state that are also in the report.
CHANGE_TO_EXCLUDE_MODE	INCLUDE	Querier non-Querier	Does the same as a non-Querier. Also sends a group-and-source-specific query with the intersection of both source lists. Removes the intersection of both source lists from group state.
	EXCLUDE	Querier non-Querier	Does the same as a non-Querier. Also sends a query related to these new sources in the group state. Adds the sources in the report that are not in group state.
ALLOW_NEW_SOURCES	INCLUDE	Querier non-Querier	Adds the sources from the report that are not stored in the router to its group state
	EXCLUDE	Querier non-Querier	If the sources from the report are in the group state, then they are removed from it.
BLOCK_OLD_SOURCES	INCLUDE	Querier non-Querier	Sends a query with the sources common to the group state ones and the report's. Removes sources from its tables
	EXCLUDE	Querier non-Querier	Sends a query with the sources common to the group state ones and the report's. Removes sources from its tables

Table 1 – Router's actions on incoming Group Record with specific type

Other relevant information related to the functionalities of the protocol will be pointed out in the rest of this report.

4 – The Code

The Python code has essentially three main parts. The first one is the in the “packet” folder, where the message's header structure are defined. All of the files in that folder are shown in **Fig. 7** as well as the inheritance relations between the files. This section of the code was implemented always having in mind that the messages had to be encapsulated in different layers of a packet which would be processed after parsing the packet from bytes into the defined classes.

The second main part is related with the creation of a network interface and the third main part is related with the logic behind the protocol to update its groups and sources tables. The last two parts will be explained later in this report.

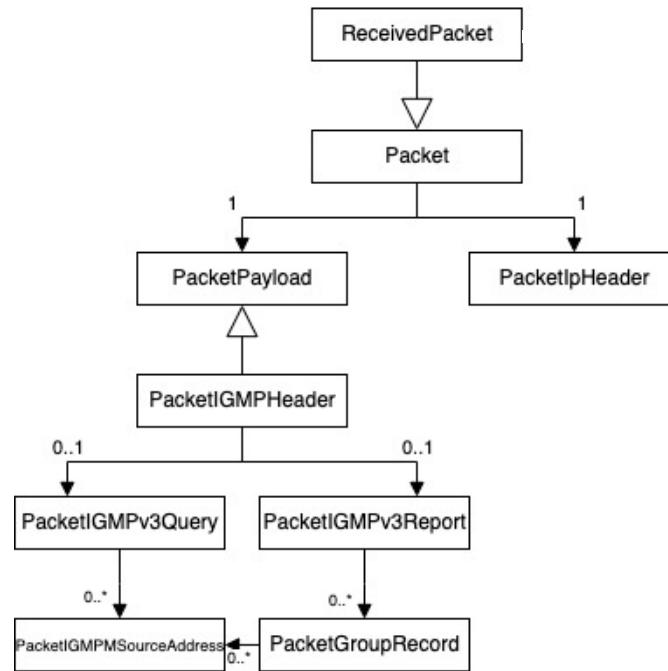


Fig. 7 – packet folder’s UML diagram representation

Regarding the location of the files and folders, the following figure can help understanding the file organization used.



Fig. 8 – Code files organization

4.1 – Packet

In this section of this report, it will be described the implementation of the messages in Python which were defined in the folder “/packet”.

A major difference from version 3 of the protocol to version 1 and version 2 is the message behavior and header format. The IGMP is considered to be part of the IP layer, as IGMP messages are transmitted in IP datagrams. Therefore, it was defined the “PacketIpHeader.py” with the PacketIpv4Header and the PacketIpv6Header classes, although in this project and on the real protocol it is only considered to run over IPv4.

Moving on to the message headers, the beginning of both headers has a similar look – [Type | Max Resp Code/Reserved | Checksum] – and therefore the “PacketIGMPHeader.py” class invokes another class which can either be the “PacketIGMPv3HeaderQuery.py” or the “PacketIGMPv3HeaderReport.py” depending on the type parameter defined at the top of the header. In IGMPv3 these two types of messages are represented with a type. For Membership Queries the type is **0x11** and for Membership Reports the type is **0x22**. The following code shows how the main class that represents an IGMPv3 header invokes the rest of the header. This way and depending on the type parameter, a PacketIGMPHeader object can also be a PacketIGMPHeaderQuery or a PacketIGMPHeaderReport.

```

104     igmp_payload = data[PacketIGMPHeader.IGMP_HDR_LEN:] #
105     if (type == 0x11 or type == 0x22): #
106         igmp_payload = PacketIGMPHeader.IGMP_MSG_TYPES #
107         [type].parse_bytes(igmp_payload) #
108     elif (v1_v2_type == 0x12 or v1_v2_type == 0x16 or #
109           v1_v2_type == 0x17): #
110         igmp_payload = PacketIGMPHeader.IGMP_MSG_TYPES #
111         ["older_versions"].parse_bytes(igmp_payload) #
112     igmp_payload.addType(v1_v2_type) #
113     return PacketIGMPHeader(igmp_payload) #

```

For the rest of the header, it was separated into two different classes as stated before. In figures **Fig. 2** and **Fig. 3**, it is presented the message header format for the Query and the Report, respectively. As we can see the headers have many more options and parameters than older versions of the protocol had.

The Query can have as many source addresses as the hosts’ INCLUDE/ EXCLUDE/ (...) operation requests it to be for the network. It also has the QQIC parameter and the S flag which are used to reference the mean time the Querier expects to send queries and the need to restart the “Other-Querier-Present” timer or not, respectively.

Following those header formats, it was defined the length of each parameter in the message as presented in the following code. In line 33 and 35 it’s defined the type of the message (in this case 0x11: Query) and the maximum time to respond to a query which is 10 by default. In line 41, it states the order of the information that is received in bytes. In this case in a sequence of the first 8 bytes from a query header, the first 4 bytes represent a set of 4 char (python type) for the IP of the group address; then there is a sequence of two bytes where the first one is to represent the parameters RESV, S and QRV, while the second one is to represent the QQIC. Finally, the “H” is there to indicate a sequence of two bytes (16 bits: hexadecimal) to represent the number of sources that are defined in the rest of the message. In line 42 it is calculated the length in bytes of these parameters.

Lines 44 and 45 have the same utility as 41 and 42 but here it defines a sequence order for each source address that is defined in the message.

```

33     IGMP_TYPE = 0x11                                     #
34     #Maximum response time is 10 by default              #
35     MAX_TIME = 10                                       #
36                                                         #
37     GROUP_ADDRESS = "0.0.0.0"                           #
38                                                         #
39     IGMP_VERSION = 3                                    #
40                                                         #
41     IGMP_HDR_Q = "! 4s BB H"                             #
42     IGMP_HDR_Q_LEN = struct.calcsize(IGMP_HDR_Q)        #
43                                                         #
44     SOURCE = "! 4s"                                       #
45     SOURCE_LEN = struct.calcsize(SOURCE)                 #

```

As it is possible to observe from the figure **Fig. 3**, the Report's header format can make explicit the group records (which contain the source addresses from the host's multicast groups) from which sources it desires to receive packets or not.

In terms of the code for this class, it used a similar implementation as the one used for the query class. Therefore, this part of the report will show the main methods implemented in the `PacketIGMPHeaderQuery` class as it is very similar to the `PacketIGMPHeaderReport`.

In each header file it was defined a few methods inherited from the `Packet.py` class, namely the `"bytes"` and the `"parse_bytes"`. The first method is responsible to turn the class's content into a bytes object while the other one converts a receiving bytes object into a class object.

The following code shows the implementation of the `"bytes"` in the query class. To convert the object to bytes it first needed to group the object's information to fit in the appropriate position of the "bytes array" and therefore in lines 103 to 105 it starts by creating a binary number where the first 4 bits of it is filled with the RESV parameter, the next bit is filled with the S flag and the last 3 bits are filled with the QRV parameter. It is in line 107 where I create the bytes object itself by filling the correspondending fields according to the sequence defined in `PacketIGMPv3HeaderQuery.IGMP_HDR_Q`. In lines 108 and 109, we go through every source address present in the query and we turn them into a bytes object and then we add each source (in bytes) to the complete bytes message object.

```

98     def bytes(self) -> bytes:                             #
99         """                                              #
100             Obtain packet in byte format                 #
101             """                                           #
102             #example of input: self.resv = 0b10101001    #
103             aux1 = self.resv << 4                       #
104             aux2 = self.s << 3                           #
105             resvSqrV = aux1 + aux2 + self.qrv            #
106                                                         #
107             msg = struct.pack(PacketIGMPv3HeaderQuery.IGMP_HDR_Q, #
108                               socket.inet_aton(self.group_address), resvSqrV, self.qqic, #
109                               len(self.source_addresses)) #
110             for source in self.source_addresses:          #
111                 msg += source.bytes()                    #
112                                                         #
113             return msg                                    #

```

The PacketIGMPHeader class has a particular code line different from the other classes in the “bytes” and “parse_bytes” methods which is the calculation of the checksum parameter both present in the query header and the report message. This line of code invokes a method from the “utils.py” to calculate the length of the whole IGMP packet for performance and integrity controls, as shown below.

```
74     igmp_checksum = checksum(msg_without_checksum) #
75     msg = msg_without_checksum[0:2] + struct.pack("! H", igmp_checksum) #
        + msg_without_checksum[4:] #
```

Figure **Fig. 7** shows the UML diagram of the classes present in the “packet” folder. It is also possible to see in the UML the “ReceivedPacket.py” file with the ReceivedPacket class which is the class that receives a packet object and decouples the IGMP message from the IPv4 header.

4.2 – Interface

In order to have these query and report packets moving around in the network the code needed to have implemented the methods that allow the code to access the router’s network interfaces as it is through these interfaces that the router can both send and receive data to be processed, in this case by the IGMP protocol. This way two object classes were added to the UML presented before. These two classes are defined in the files “Interface.py” and “InterfaceIGMP.py”.

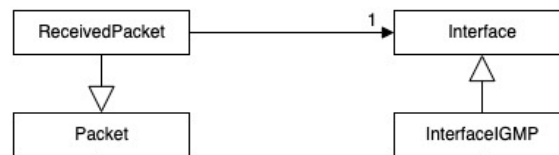


Fig. 8 –UML diagram of the interface

The Interface class is an abstract class where were defined the methods and attributes of the object while in InterfaceIGMP, it is described a specific logic for the IGMP protocol of how the interfaces should act upon each packet received or sent.

This part of the code was developed having in mind that once a router starts the IGMPv3 code on an interface, it should immediately create a thread to handle any incoming data through that interface. Therefore, in the following piece of code it is possible to observe that once the interface is initialized (in the constructor method) it stores the attributes for sockets of reception and shipping of data (lines 20 and 21). The method `enable` is triggered inside the constructor of the InterfaceIGMP class and it allows to create (line 31) and start (line 33) a thread to handle the method `receive` which is listening for any incoming data and forwards it to be processed if it is an IGMP packet and discards it if not.

```
1  def __init__(self, interface_name, recv_socket, send_socket, vif_index):#
    # (. . .) #
19     # set receive socket and send socket #
20     self._send_socket = send_socket #
```



```

21     self._recv_socket = recv_socket          #
22     self.interface_enabled = False           #
23                                             #
24 def enable(self):                           #
25     # (. . .)                               #
29     self.interface_enabled = True            #
30     # run receive method in background       #
31     receive_thread = threading.Thread(target=self.receive) #
32     receive_thread.daemon = True             #
33     receive_thread.start()                   #

```

In the InterfaceIGMP class it fills the attributes of the inherited class with the definition of the sockets such as with the family type which is, in our case, the AF_INET for internet communications (communication between more than one machine). It also defines the interface name and IP address based on a Python library called “netifaces” as well as creating a state for the router. The piece of code below shows how the interface will decide which type of packet it received based on the header so it can deploy the packet to its proper method/class that will process the information. First of all, once data arrives to the receiving socket, it will separate the Ethernet header from the rest of the packet (line 90) then it decouples the IP header from the IGMP header (line 91). After concluding that the packet came from a valid source (line 93 – “0.0.0.0” addresses from a source is, most probably, a DHCP protocol packet. It also cannot come from a multicast IP. Otherwise it is a valid IP for requesting an IGMP operation) it will send the data to one of three methods that will process the information (line 94) which can be `receive_version_3_membership_report`, `receive_version_3_membership_query` or `receive_unknown_type` (the last method is in case the packet is neither a query nor a report).

```

85 def _receive(self, raw_bytes, src_addr):    #
86     """                                    #
87     Interface received a new control packet #
88     """                                     #
89     if raw_bytes:                           #
90         raw_bytes = raw_bytes[14:]          #
91         packet = ReceivedPacket(raw_bytes, self) #
92         ip_src = packet.ip_header.ip_src     #
93         if not (ip_src == "0.0.0.0" or      #
94                 IPv4Address(ip_src).is_multicast): #
95             self.PKT_FUNCTIONS.get(packet.payload.getIgmpType(), #
96                                     InterfaceIGMP.receive_unknown_type)(self, packet)#

```

The two methods that process IGMP queries and reports in InterfaceIGMP also perform a check on the destination’s IP address to assure that the IP is valid within that specific message. A Membership Report is sent from a host and therefore the destination should always be the 224.0.0.22 so that only routers in the network receive it. Regarding an incoming query, the destination could be the 224.0.0.1 (all members of the network including the hosts) or it could be a multicast address (a group-specific query).

4.3 – Router State

In this next part of the code is described in the “RouterState.py” file. The router’s state is responsible to decide the actions performed by the router when an event occurs. Such event could be the reception of a packet or an expired timer.

The router’s state is initialized once the network interface turns on and according to the protocol it must assume to be the Querier and therefore the constructor method takes it into count. The following piece of code shows that the router state must be associated with the interface from which was created (line 32) and must have records of the multicast groups (line 40). According to the protocol the router becomes the Querier when it starts (line 37) and so must send a query to the other routers with destination “224.0.0.22” (lines 44 to 46). As it is the Querier it must start the timer that when expired will send another General-Query. As it is not a non-Querier router, at least at first, it does not need to set the Other-Querier-Present timer to check the existence of a Querier in the network (line 55).

```
32     self.interface = interface                                #
    (...)                                                    #
37     self.whoIsQuerier = self.interface.get_ip()              #
38     # state of each group                                    #
39     # Key: GroupIPAddress, Value: GroupState object          #
40     self.group_state = {}                                    #
    (...)                                                    #
43     # send general query to all the routers, where S=0 | qrv=2 | QQIC=125
44     packet = PacketIGMPv3HeaderQuery(0, 0, 2, 125, "0.0.0.0") #
45     igmp_pkt = PacketIGMPHeader(packet)                     #
46     self.interface.send(igmp_pkt.bytes(), "224.0.0.22")      #
    (...)                                                    #
50     timer = Timer(QUERY_INTERVAL, self.general_query_timeout) #
51     timer.start()                                            #
52     self.general_query_timer = timer                         #
    (...)                                                    #
55     self.other_querier_present_timer = None                  #
```

In this file it is also defined all the methods that control the timers and the events they must trigger when expired. The following code shows the use of these methods for the general query timer. There are 3 methods: one is to clear the timer and restart the countdown; one other is to clear the timer and stop it; and the last is to set the event once a timer reaches to 0. Lines 76 to 81 show the method that clears and stops the timer. This method is used e.g. in the `set_general_query_timer` method to clear the timer if not already and start the count (line 69 and 71). Line 70 sets the event that must be triggered once this timer expires which is the `general_query_timeout` method. Once triggered, this method is supposed to send a general query again (line 109) and restart the timer (line 110) but only if the router is still the Querier (line 108).

```
65     def set_general_query_timer(self):                        #
    (...)                                                    #
69     self.clear_general_query_timer()                          #
70     general_query_timer = Timer(QUERY_INTERVAL,              #
                                   self.general_query_timeout) #
71     general_query_timer.start()                               #
72     self.general_query_timer = general_query_timer           #
```

```
(...) #
76 def clear_general_query_timer(self): #
    (...) #
80 if self.general_query_timer is not None: #
81     self.general_query_timer.cancel() #
    (...) #
99 def general_query_timeout(self): #
    (...) #
104 #sends a new general query #
106 packet = PacketIGMPv3HeaderQuery(0, 0, 2, QUERY_INTERVAL, "0.0.0.0") #
107 igmp_pkt = PacketIGMPHeader(packet) #
108 if self.interface_state == "Querier": #
109     self.interface.send(igmp_pkt.bytes(), "224.0.0.1") #
110     self.set_general_query_timer() #
```

Facing the event of reception of a message of type query or report, the router must either discard the message, modify the group records state or check if there is another router that should become the Querier. When a router receives a Report packet it first extracts the information from the message related to multicast group (line 169); all sources from the group (line 170) and the type of operation the host wants to do with the sources. In line 173 it gets the object that is associated with that particular group and invokes the method to be executed on that group.

```
163 def receive_v3_membership_report(self, packet: ReceivedPacket): #
    (...) #
167 group_records = packet.payload.getPayload().group_records #
168 for group in group_records: #
169     mc_ip = group.getMulticastAddress() #
170     adds = group.source_addresses #list PacketIGMPMSourceAddress #
171     #e.g: TO_INCLUDE #
172     fnct = group.getRecordType() #
173     self.get_group_state(mc_ip).receive_v3_membership_report #
        (adds, fnct) #
```

These next lines of code show the event triggered by the reception of a query. The first “if” condition in line 179 describes a method that should be executed on the group object if the query has a multicast group IP which means it is a group-specific query or a group-and-source-specific-query. In this case, the router must start a group timer and it should receive reports from hosts reporting their desire to keep receiving or not from that specific group and its sources, until the timer expires.

The second condition assumes that the event it must trigger is a Querier election, as the query received was sent by a lower IP address router (line 184) and if so, the router becomes a non-Querier one. According to the RFC, the S flag parameter received in a Query (lines 191 and 192) indicates if it is supposed to suppress the restart of the timer (binary 1) or not (binary 0). In the majority of cases the S flag has the value 0.

```
175 def receive_query(self, packet: ReceivedPacket): #
    (...) #
179 if igmp_group != "0.0.0.0" and igmp_group in self.group_state: #
180     max_response_time = packet.payload.getIgmpMaxTime() #
181     self.get_group_state(igmp_group).receive_group_specific_query( #
        max_response_time, sources) #
182 # #
183 # querier election process #
```

```
184         if IPv4Address(ip_src) < IPv4Address(self.interface.get_ip()): #
185             #Becomes Non-Querier if not already #
186             self.change_interface_state(False) #
187             (...) #
191             if sFlag == 0: #
192                 self.set_other_querier_present_timer() #
```

4.4 – Group State

The last main configuration file that I wrote is the “GroupState.py”, in which it is described the actions to be performed on each group address and its sources after the occurrence of an event such as the reception of a report or query.

Starting with the event of reception of a group-specific query or a group-and-source-specific query, the `receive_group_specific_query` method is the one that deals with this event. If a router is receiving a query message, then it is not the Querier and, therefore, it must look at the multicast group address and its sources and initiate timers for each source and the group itself until the hosts send a report referring to these specific addresses. This method starts by checking if the query only has a MC address and no sources and therefore it is only needed to create a timer for the group (lines 220 to 224), otherwise it will check if the sources in the header are stored in the router’s groups and if so it will update its timer (lines 230 to 235) and if not it will start by storing the group and the sources and then it will start a timer for each source. This second case is important as if the router just plugged in and still has no information on the network’s groups then if it receives a query due to older reports sent by hosts to the other routers, it will know about these groups thanks to the receiving query.

```
217 def receive_group_specific_query(self, max_response_time, source_adds:#
    PacketIGMPMSourceAddress): #
218     self.set_group_timer() #
219     # Waits until it receives a report in which it is desired to #
    receive from this group #
220     if len(source_adds) == 0: #
221         self.clear_group_timer() #
222         group_timer = Timer( #
    MAX_RESPONSE_TIME_LAST_MEMBER_QUERY_INTERVAL, self.group_timeout) #
223         group_timer.start() #
224         self.group_timer = group_timer #
225     else: #
226         lst_ip = [] #
    (...) #
230     for s in lst_ip: #
231         if s in self.source_addresses: #
232             self.clear_source_timer(s) #
233             source_timer = Timer( #
    MAX_RESPONSE_TIME_LAST_MEMBER_QUERY_INTERVAL, self.source_timeout, [s]) #
234             source_timer.start() #
235             self.source_addresses[s] = source_timer #
```

The reception of report messages is what makes the router know exactly what groups, the hosts are still interested in. In the code, lines from 114 to 211 describe the ways to update the groups states when a report is received.

The first lines of the `receive_v3_membership_report` method clear the timers associated to the sources as a response from the hosts have arrived. In IGMPv3 each group is associated with a state which can either be INCLUDE or EXCLUDE. Based on this group's mode, a group state will be influenced by the type of operation that the host wants to be performed on the sources it sent to routers.

To understand the purpose of the method referred in last paragraph, you can observe **Table 1** in the brief introduction to IGMPv3 section of this report, which briefly specifies what should be done when a report arrives to the router and so is implemented in the code.

In each report message, the operation type field is set with a number from 1 to 6 that represents the operation itself. IGMPv3 routers decode from the number the operation type is being requested by the host. That association between the operation type and the corresponding number can be seen in the table below.

Operation Type	Corresponding number in the report
MODE_IS_INCLUDE	1
MODE_IS_EXCLUDE	2
CHANGE_TO_INCLUDE_MODE	3
CHANGE_TO_EXCLUDE_MODE	4
ALLOW_NEW_SOURCES	5
BLOCK_OLD_SOURCES	6

Table 2 – Operation type and corresponding number in the group record

5 – Tests

In this section, I will present the main tests executed to demonstrate the correct implementation of the IGMPv3 protocol in the routers.

I start by pointing out a problem with the network emulation system (Kathará) and Python sockets which were not able to let the ports send to the 224.0.0.22 address. As it was possible to send to the 224.0.0.1 address, I replaced in the code the first address mentioned to this second one.

The emulated network used during the tests is the one presented in the following image, and is composed by two routers, two hosts and a switch. The steps to configure the network in a virtual machine, before start using the main script, are shown in the next section of this report.

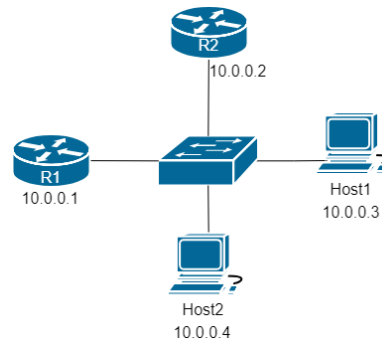


Fig. 9 – The emulated network topology used for testing

5.1 – Running the main script

In order to run the Python code that runs the router side's IGMPv3 protocol, it is needed to configure a few things.

- (1) The code needs a Python3 version to run. You can check the version installed in the OS by running the following command:

```
python --version
```

- (2) The emulation environment could be either GNS3 or Kathará or any other environment that emulates a network and is able to run code scripts. Kathará can be installed from its website: <https://www.kathara.org>

- (3) To start using the code you must start by running the emulated network. To do so, run the following commands:

```
cd IGMPv3/kathara_network/
bash script.sh
```

- (4) The next step is to copy the code's directory into the OS of each routing device. For that you should configure kathará to allow the sharing of files between the hosting OS (your own) and the emulated OS of the router. In alternative you can use these commands to access a shared directory with the files previously set in:

```
cd /shared/root/Router/
cd igmpv3/
```

- (5) Finally, run the command in the routers, to start the Python code:

```
python3 main.py
```

5.2 – Testing the interface’s sending and receiving capabilities

This first test had the goal to make sure that the network interfaces in each router were able send packets (namely queries and reports) and to listen to new incoming packets from their sockets and also the ability to know if the packet was either an IGMPv3 packet or some other type of packet that should be discarded by the protocol. Each red square and its corresponding number represent a part of the test which will now be described. To test it I used the “TestSendMessage.py” and “TestReceiveMessage.py” files.

In one terminal (Client 1) it was executed the code to send two messages: first a query and then a report. In the other router (Router 1) it ran the code to show it received packets. The first part of the test (1) the router on the left sent a query which content is represented in bytes format on the terminal. The router on the right shows it received a packet exactly like the one that was sent and was also able to get it was a query packet. On the second part of the test (2) the same was tested but this time with a report message.

This test had the only purpose to test the interface ability to send and receive and to interpret the content of each message received and therefore, the fact that a “client” device is sending both a report and a query does not test the protocol’s logic as a host can only send report messages and only the Querier router can only send queries.

[illegible]

Fig. 10 – Routers’ terminals showing the execution of this test

5.3 – Testing the Querier election

For the next test, I decided to test the protocol logic starting with the Querier election based on its state machine which was defined in section 3.2 of this report and previously presented in **Fig. 5**. The results are presented below. Each red square and its corresponding number represent a part of the test which will now be described. To test it I used the “TestQuerierElection.py” file.

For this test I used two emulated routers with two different IP addresses. Router 1 has a lower IP than Router 2. I started by running the code in Router 2 (1) and so it immediately assumed to be the Querier, until I run the code in Router 1 which sent a

query to the routers and Router 2 compared the IP addresses and concluded that there was another router with a lower IP address and so that router should become the Querier. Still in this test part I let the General-Query timer expire to guarantee that the Querier would send another query after this timer expired which happened and can still be observed in this first part of the test. After this I started the second part of the test (2) in which I shut down Router 1, the Querier, to observe Router 2's reaction. After the Other-Querier-Present timer expired, Router 2 concluded that there was no longer a Querier in the network and so it became the Querier itself. For the last part of the test (3) I simply turned Router 1 back on. The behavior was the expected as Router 1 sends a query once it is initialized and Router 2 concludes there is a router that is supposed to be the Querier, after receiving the query.

```

--- Startup Commands Log
++ ip addr add dev eth0 10.0.0.1/24
++ ip link set dev eth0 up
--- End Startup Commands Log

root@router1:~# cp /hoshome/Desktop/IST/Ano_3/ProjInt-IGMPv3/IGMPv3/igmpv3/ -f
root@router1:~# cd igmpv3/
root@router1:~/igmpv3# python3 TestQuerierElection.py
Starting interface
['lo', 'tunl0', 'ip6tnl0', 'eth0']
/* * * * * */
Current Querier IP: 10.0.0.1
Interface IP: 10.0.0.1
General Query Timer has expired.
General Query Timer has started.
^CTraceback (most recent call last):
  File "TestQuerierElection.py", line 17, in <module>
    i+=1
KeyboardInterrupt
^CException ignored in: <module 'threading' from '/usr/lib/python3.5/threading.py'>
root@router1:~/igmpv3# python3 TestQuerierElection.py
Starting interface
['lo', 'tunl0', 'ip6tnl0', 'eth0']
/* * * * * */
Current Querier IP: 10.0.0.1
Interface IP: 10.0.0.1

--- Startup Commands Log
++ ip addr add dev eth0 10.0.0.2/24
++ ip link set dev eth0 up
--- End Startup Commands Log

root@router2:~# cp /hoshome/Desktop/IST/Ano_3/ProjInt-IGMPv3/IGMPv3/igmpv3/ -f
root@router2:~# cd igmpv3/
root@router2:~/igmpv3# python3 TestQuerierElection.py
Starting interface
['lo', 'tunl0', 'ip6tnl0', 'eth0']
/* * * * * */
Current Querier IP: 10.0.0.2
Interface IP: 10.0.0.2
parseIGMPHdr: b'\x11\x0b\xef\x00\x00\x16\x02'\x00\x00'
Current Querier IP: 10.0.0.1
parseIGMPHdr: b'\x11\x0b\x05\x00\x00\x00\x02'\x00\x00'
Current Querier IP: 10.0.0.1
General Query Timer has started.
Current Querier IP: 10.0.0.2
General Query Timer has expired.
General Query Timer has started.
parseIGMPHdr: b'\x11\x0b\xef\x00\x00\x16\x02'\x00\x00'
Current Querier IP: 10.0.0.1

```

Fig. 11 – Routers' terminals showing the execution of this test

5.4 – Testing the Reception of reports

This next test had a few “sub-test” and it used another script called “client.py” ran in a client host device. This script was firstly designed to send version 3 reports but without specifying the sources from which hosts wanted to receive. This first “sub-test” tested the ability of understanding a report message and storing the group IP address in the router's tables. From the figure below, we can check that when a report arrived at the router's interface ports it was able to understand that it received a report message (1). In the router's test file, it was implemented a “while” cycle to show every 10 seconds all of the group records (group IP and its sources) which can be observed from the router's side terminal in the figure (2). As there are no sources being specified, the terminal shows 0 sources under “SOURCES:”.


```

root@client1:~/Client# python3 client.py
Indique a interface de captura:
1 - lo
2 - tun10
3 - ip6tn10
4 - eth0
Numero da interface: 4

root@router1:~/igmpv3# python3 TestGroupStateReceive.py
Starting interface
['lo', 'tun10', 'ip6tn10', 'eth0']
/* * * * * */
Current Querier IP: 10.0.0.1
Interface IP: 10.0.0.1
parseIGMPHdr: b"\x00\xed\x05\x00\x00\x01\x04\x00\x00\x00\x0c\x0c\x0c"
Got a REPORT
Method: group - report
parseIGMPHdr: b"\x00\xed\x05\x00\x00\x01\x04\x00\x00\x00\x0c\x0c\x0c"
Got a REPORT
Method: group - report
224.12.12.12
SOURCES:

```

Fig. 12 – Routers’ terminals showing the execution of this test

In a second “sub-test” we can also see the execution of two commands in the router’s terminal while executing the main program. With the first command we see the router’s groups table where it shows all IP addresses from the groups which the hosts are interested in (1). The second command shows a few properties from the selected group (2).

```

root@client1:~/Client# python3 client.py
Indique a interface de captura:
1 - lo
2 - tun10
3 - ip6tn10
4 - eth0
Numero da interface: 4

root@client2:~/Client# python3 client.py
Indique a interface de captura:
1 - lo
2 - tun10
3 - ip6tn10
4 - eth0
Numero da interface: 4

root@router1:~/igmpv3# python3 TestGroupStateReceive.py
/* You can insert a command at any time. */
/* The possible commands are these ones: */
/* You can check back the commands using */
/* the 'man' command in shell prompt. */
parseIGMPHdr: b"\x00\xed\x05\x00\x00\x01\x04\x00\x00\x00\x0c\x0c\x0c"
parseIGMPHdr: b"\x00\xed\x05\x00\x00\x01\x04\x00\x00\x00\x0c\x0c\x0c"
parseIGMPHdr: b"\x00\xed\x05\x00\x00\x01\x04\x00\x00\x00\x0c\x0c\x0c"
parseIGMPHdr: b"\x00\xed\x05\x00\x00\x01\x04\x00\x00\x00\x0c\x0c\x0c"
parseIGMPHdr: b"\x00\xed\x05\x00\x00\x01\x04\x00\x00\x00\x0c\x0c\x0c"
show mcgroups
/*
/* MULTICAST GROUPS' IP TABLE
/*
/* Group | 224.11.11.11
/* Group | 224.12.12.12
/* Group | 224.12.12.12
show groupstate 224.12.12.12
/*
/* GROUP IP | 224.12.12.12
/* GROUP MODE | INCLUDE

```

Fig. 13 – Routers’ terminals showing the execution of this test

Furthermore, in another “sub-test” I also set two hosts sending a report with a group address but no sources specified and then I shut them down. In the router side I programmed it to log when a report was received and the corresponding number of operation type (e.g. CHANGE_TO_INCLUDE). The results were the expected ones.

The next figure shows the test logs. I started by running the router with the “main.py” file and then I ran the “client.py” in the top left host. As the host is not specifying any sources then the operation should be CHANGE_TO_EXCLUDE {}. This means that the host wants to exclude none which also means it wants to include all sources. The corresponding number for that operation is the one that shows in the figure: “4” (1). This operation requires (according to the RFC) the Querier to send a group-specific query on that group. An answer was immediately verified in the logs with the number “2” – MODE_IS_EXCLUDE {}. This response means that at least one member wants to keep receiving from all sources of the group and therefore the router will keep it stored (2). These first two steps of the test were done twice as they were also applied to

the second host. After a while I shut the top left host down (^C) and so the host sent a report that stated it didn't want to be a member of the group any longer – “3” CHANGE_TO_INCLUDE_MODE {} (include none also means exclude all sources) (3). As there was still a host interested in the group it was expected that after the group-specific query was sent, the bottom left host would respond with a “2” (MODE_IS_EXCLUDE {}) which was verified in (4). After this log appeared I also shut down the other host which triggered the host to send a CHANGE_TO_INCLUDE_MODE {}. With this event the Querier sent a group-specific query to understand if there were still any members that wanted to remain with that group. Because there are no more routers interested in this group, no one responded to this query and therefore the group timer expired and the router decided to remove the group from its tables (5).

```

martimtavares -- root@client1: ~/Client - kathara - kathara connect -i client1 -- 80x24
root@client1:~/Client# python3 client.py
Indique a interface de captura:
1 - lo
2 - tunl0
3 - ip6tnl0
4 - eth0
Numero da interface: 4
^Croot@client1:~/Client#

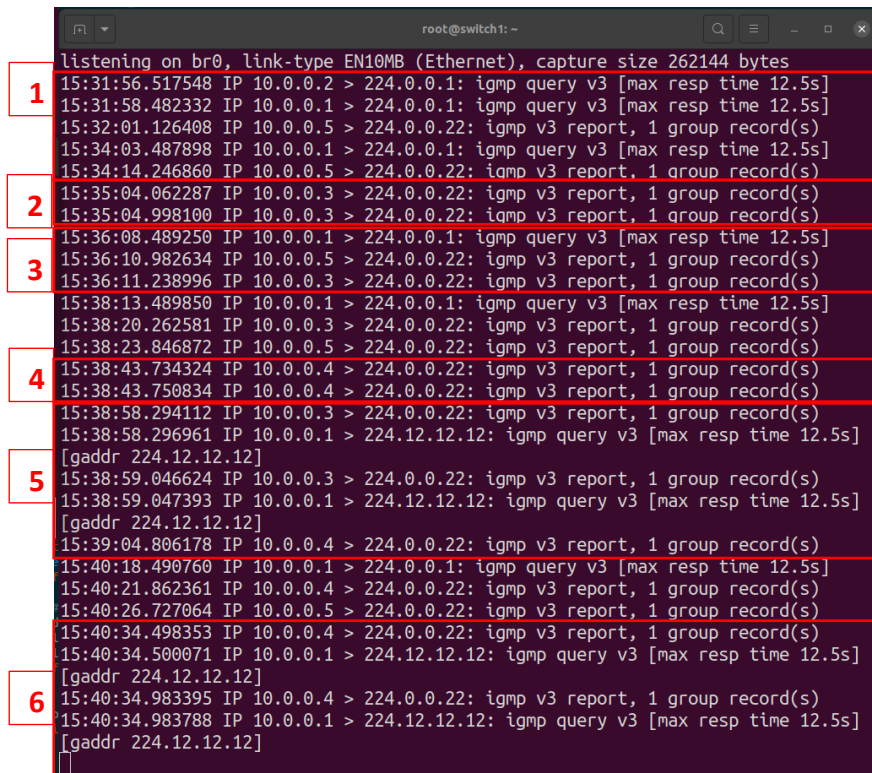
martimtavares -- root@router1: ~/igmpv3 - kathara - kathara connect -i router1 -- 80x47
parseIGMPHdr: b'\x00\xed\x05\x00\x00\x01\x04\x00\x00\x00\x0c\x0c\x0c'
RECEIVED A REPORT
4
INCLUDE
EXCLUDE
parseIGMPHdr: b'\x00\xed\x05\x00\x00\x01\x04\x00\x00\x00\x0c\x0c\x0c'
RECEIVED A REPORT
4
INCLUDE
EXCLUDE
parseIGMPHdr: b'\x00\xef\x05\x00\x00\x01\x02\x00\x00\x00\x0c\x0c\x0c'
RECEIVED A REPORT
2
parseIGMPHdr: b'\x00\xed\x05\x00\x00\x01\x04\x00\x00\x00\x0c\x0c\x0c'
RECEIVED A REPORT
4
EXCLUDE
parseIGMPHdr: b'\x00\xef\x05\x00\x00\x01\x02\x00\x00\x00\x0c\x0c\x0c'
RECEIVED A REPORT
2
parseIGMPHdr: b'\x00\xed\x05\x00\x00\x01\x04\x00\x00\x00\x0c\x0c\x0c'
RECEIVED A REPORT
4
EXCLUDE
parseIGMPHdr: b'\x00\xef\x05\x00\x00\x01\x02\x00\x00\x00\x0c\x0c\x0c'
RECEIVED A REPORT
2
parseIGMPHdr: b'\x00\xed\x05\x00\x00\x01\x04\x00\x00\x00\x0c\x0c\x0c'
RECEIVED A REPORT
4
EXCLUDE
parseIGMPHdr: b'\x00\xef\x05\x00\x00\x01\x02\x00\x00\x00\x0c\x0c\x0c'
RECEIVED A REPORT
2
parseIGMPHdr: b'\x00\xee\x05\x00\x00\x01\x03\x00\x00\x00\x0c\x0c\x0c'
RECEIVED A REPORT
3
parseIGMPHdr: b'\x00\xee\x05\x00\x00\x01\x03\x00\x00\x00\x0c\x0c\x0c'
RECEIVED A REPORT
3
parseIGMPHdr: b'\x00\xef\x05\x00\x00\x01\x02\x00\x00\x00\x0c\x0c\x0c'
RECEIVED A REPORT
2
parseIGMPHdr: b'\x00\xee\x05\x00\x00\x01\x03\x00\x00\x00\x0c\x0c\x0c'
RECEIVED A REPORT
3
parseIGMPHdr: b'\x00\xee\x05\x00\x00\x01\x03\x00\x00\x00\x0c\x0c\x0c'
RECEIVED A REPORT
3
Group timeout
[INFO]: Group removed.
  
```

Fig. 14 – Routers’ terminals showing the execution of this test

The last test for this section was based on another client test file which ran the version 3 protocol and also sent reports with group records with specific actions on specific source addresses. This was possible by using Python socket option (socket.sockopt) – IP_ADD_SOURCE_MEMBERSHIP. However, Python does not have this variable defined, I replaced the parameter option with the value that the variable is supposed to represent which is the value 39. This way I was able to make my client send IGMPv3 reports with sources associated to an operation. This test was performed using the help of the command “**tcpdump proto \igmp**” on the switch’s terminal to check all the IGMP packets that were being sent in the testing network.

The test results are presented in the figure below (Fig. 15). The first part of the test can be observed in box numbered with 1 (1). There we can see the Querier election process as mentioned already in this report. Unfortunately, in this test I had to replace the 224.0.0.22 address with the 224.0.0.1 as the first one was not working with the sockets and, therefore, the queries were always being sent to everyone in the network and so we can see the host 10.0.0.5 (the switch) sending a report on a group. After that I ran the

client process on host 10.0.0.3 (2) which sent two reports – the first one with a ALLOW_NEW_SOURCES operation on the source 193.5.4.3 and the second one with a INCLUDE 193.5.4.3. Following that, the general query timer expired and so the Querier had to send a general query to the 224.0.0.1 and all hosts in the network responded – the switch and client host (3). It is necessary to comment that the switch should not be reporting for any group, yet it did. The next step was running the client process in another host which had the address 10.0.0.4 with the same report as the first client already running in the network (4). At some point in the test, I pushed one of the hosts to leave a group and so its sources, by killing the process (5). As so, the Querier sent a group-and-source-specific query for the group which the host was leaving which was 224.12.12.12. As there was another host that pretended to keep in the group with that source, this other host responded with an INCLUDE report on that group and source. After a while I did the same as in (5) but with the other host in order to see the router's reaction which was expected. As there were no responses to a group-specific query, the router removed the multicast group from its routing tables (6).



```

root@switch1: ~
listening on br0, link-type EN10MB (Ethernet), capture size 262144 bytes
1 15:31:56.517548 IP 10.0.0.2 > 224.0.0.1: igmp query v3 [max resp time 12.5s]
15:31:58.482332 IP 10.0.0.1 > 224.0.0.1: igmp query v3 [max resp time 12.5s]
15:32:01.126408 IP 10.0.0.5 > 224.0.0.22: igmp v3 report, 1 group record(s)
15:34:03.487898 IP 10.0.0.1 > 224.0.0.1: igmp query v3 [max resp time 12.5s]
15:34:14.246860 IP 10.0.0.5 > 224.0.0.22: igmp v3 report, 1 group record(s)
2 15:35:04.062287 IP 10.0.0.3 > 224.0.0.22: igmp v3 report, 1 group record(s)
15:35:04.998100 IP 10.0.0.3 > 224.0.0.22: igmp v3 report, 1 group record(s)
3 15:36:08.489250 IP 10.0.0.1 > 224.0.0.1: igmp query v3 [max resp time 12.5s]
15:36:10.982634 IP 10.0.0.5 > 224.0.0.22: igmp v3 report, 1 group record(s)
15:36:11.238996 IP 10.0.0.3 > 224.0.0.22: igmp v3 report, 1 group record(s)
15:38:13.489850 IP 10.0.0.1 > 224.0.0.1: igmp query v3 [max resp time 12.5s]
15:38:20.262581 IP 10.0.0.3 > 224.0.0.22: igmp v3 report, 1 group record(s)
15:38:23.846872 IP 10.0.0.5 > 224.0.0.22: igmp v3 report, 1 group record(s)
4 15:38:43.734324 IP 10.0.0.4 > 224.0.0.22: igmp v3 report, 1 group record(s)
15:38:43.750834 IP 10.0.0.4 > 224.0.0.22: igmp v3 report, 1 group record(s)
15:38:58.294112 IP 10.0.0.3 > 224.0.0.22: igmp v3 report, 1 group record(s)
15:38:58.296961 IP 10.0.0.1 > 224.12.12.12: igmp query v3 [max resp time 12.5s]
5 [gaddr 224.12.12.12]
15:38:59.046624 IP 10.0.0.3 > 224.0.0.22: igmp v3 report, 1 group record(s)
15:38:59.047393 IP 10.0.0.1 > 224.12.12.12: igmp query v3 [max resp time 12.5s]
6 [gaddr 224.12.12.12]
15:39:04.806178 IP 10.0.0.4 > 224.0.0.22: igmp v3 report, 1 group record(s)
15:40:18.490760 IP 10.0.0.1 > 224.0.0.1: igmp query v3 [max resp time 12.5s]
15:40:21.862361 IP 10.0.0.4 > 224.0.0.22: igmp v3 report, 1 group record(s)
15:40:26.727064 IP 10.0.0.5 > 224.0.0.22: igmp v3 report, 1 group record(s)
15:40:34.498353 IP 10.0.0.4 > 224.0.0.22: igmp v3 report, 1 group record(s)
15:40:34.500071 IP 10.0.0.1 > 224.12.12.12: igmp query v3 [max resp time 12.5s]
6 [gaddr 224.12.12.12]
15:40:34.983395 IP 10.0.0.4 > 224.0.0.22: igmp v3 report, 1 group record(s)
15:40:34.983788 IP 10.0.0.1 > 224.12.12.12: igmp query v3 [max resp time 12.5s]
6 [gaddr 224.12.12.12]

```

Fig. 15 – Switch's terminal showing the IGMP packets

I include in this report a link to a video of me showing the performance and execution of the code by testing it in a virtual machine: <https://youtu.be/g86GWVsRAmc>.

6 – Conclusions

Many conclusions can be made from this project regarding multicast routing and IGMPv3 protocol as a solution for the unicast routing problems.

First of all, multicast routing is much more efficient than unicast when the objective is to send large amount of data to a large group of receivers, as in such situation unicast routing would overload the network, while multicast does not. IGMP is a protocol that brings life to the multicast routing option and protects the network performance from the point of view of latency, packet processing, and packet discarded due to router's overloaded data bus.

IGMP, as a multicast protocol, guarantees some privacy in client's metadata as the transmitting servers do not need to know the IP addresses of its clients, but instead it knows its own multicast group IP. If anyone wants to receive data from the server, only needs to catch the packets sent to that group.

From the point of view of version 3 of IGMP, this version improves even more the traffic latency problem and network traffic in the matter of IGMP exchanging messages, as a single packet can contain information related to many groups and its sources, when in older versions, a single report would contain only contain information related to one group. Version 3 of the protocol also allows the participants to choose specific sources from which they pretend to receive which is another advantage. However, the IGMPv3 queries and reports are now longer and therefore routers take more time to process each message. Another disadvantage from this version is the higher complexity of the protocol when compared with the other two versions.

All in all, IGMPv3 is currently the best approach to the problems referred in this paper and should keep being enhanced.

7 – Bibliography

Apart from Python tutorial websites, this project was developed according to all the information accessed from the following links.

IGMPv3 RFC3376 –

<https://datatracker.ietf.org/doc/html/rfc3376#page-7>

IGMPv2 project on GitHub by Pedro Oliveira –

<https://github.com/pedrofran12/igmp/tree/master>

Lecture 4 - IP Multicast Receiver Signaling with IGMPv3 –

<https://www.youtube.com/watch?v=Sgppv6HREFSs>

2016 Tier Up IEI - IGMP Multicast –

<https://www.youtube.com/watch?v=YbIf-WUjOGw>

Linux manual pages – IP_ADD_SOURCE_MEMBERSHIP –

<https://man7.org/linux/man-pages/man7/ip.7.html>

IGMP: What is behind the Internet Group Management Protocol –

<https://www.ionos.ca/digitalguide/server/know-how/igmp-internet-group-management-protocol/>

Instituto Politécnico da Guarda: Multicast –

<http://www.ipg.pt/user/~sduarte/rc/trabalhos/Multicast/Multicast.htm>