

Python implementation of IGMPv2, PIM-DM, and HPIM-DM

Pedro Francisco Carmelo de Oliveira
pedro.francisco.oliveira@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

March 24, 2019

This document describes what was implemented during the MSc Dissertation work. We start by describing the API offered by the Linux kernel in order to support multicast communication and routing and we finalize by referring the architectures and specifications of all implemented protocols (IGMPv2, PIM-DM, HPIM-DM).

- **Section 1** describes the multicast specific features offered by the Linux Kernel;
- **Section 2** describes the specification and corresponding implementation of IGMPv2, PIM-DM and HPIM-DM.

1 Multicast specific features of Linux Kernel

First it is important to understand what the kernel offers that is related to multicast.

As it was said in the previous section, Linux already offers strong APIs that are related to multicast. This enables a programmer to reuse code, avoiding reinventing the wheel.

This section is further divided in three subsections in order to better understand what is offered by the kernel and how a programmer can use those features.

1.1 Multicast Routing Table

Linux already implements a multicast routing table, allowing a user-level process to manipulate it. This process requires to implement a multicast routing protocol in order to dynamically manipulate the entries of that table, or do not implement any protocol and simply receive commands from the user to manipulate it statically.

The static solution is not the one we pretend and it is already implemented by SMCRout [1].

The dynamic solution is the one we pretend to use in order to react to changes of membership caused by directly connected hosts and routers. This is the solution used by implementations of multicast routing protocols in Linux such as mroute [2] (implementation of DVMRP) and pimd [3] (implementation of PIM-SM). The API offered by Linux, that is used by these processes and by the processes that were developed in this MSc Dissertation, is described in the man pages here [4] and will be further explained in this subsection. We only describe the details regarding the manipulation of IPv4 multicast routing entries. Manipulation of IPv6 multicast routing entries is not described since it was not used and because it is similar to the IPv4 manipulation.

The manipulation of the multicast routing table is accomplished in Linux by sending and receiving information from a special socket. This socket is called *mroute* socket and allows to add, remove and change entries from the multicast routing table. At most one process can open this socket, so it is not possible to have two processes manipulating the table at the same time, which means that at most one multicast routing protocol can be running at any time.

A *mroute* socket corresponds to an IGMP socket but with some enabled features. This socket must be opened with type `SOCK_RAW` and protocol number `IPPROTO_IGMP`. After opening this socket, we need to set some features.

The options that can be used to get or set information with the *mroute* socket are the following:

- **MRT_INIT** - activates the kernel mroute code, allowing the socket to set and get multicast routing related information;
- **MRT_DONE** - shutdowns the kernel mroute, i.e. removes all multicast routing related information that was set using the mroute socket;
- **MRT_ADD_VIF** - add a virtual interface;
- **MRT_DEL_VIF** - remove a virtual interface;
- **MRT_ADD_MFC** - add an (S,G) entry in the multicast routing table;
- **MRT_DEL_MFC** - remove an (S,G) entry from the multicast routing table;
- **MRT_ADD_MFC_PROXY** - add a (*,*) or (*,G) entry to the multicast routing table;
- **MRT_DEL_MFC_PROXY** - remove a (*,*) or (*,G) entry from the multicast routing table;
- **MRT_VERSION** - get the kernel multicast version;
- **MRT_ASSERT** - activate PIM assert mode;
- **MRT_PIM** - enable PIM code;
- **MRT_TABLE** - specify mroute table ID.

It is possible to set or get information using these options by calling the function *setsockopt* over the opened socket.

In order to manipulate multicast routing information using this socket, we first need to enable multi-cast forwarding in the kernel, which is accomplished by setting the **MRT_INIT** option as true.

After enabling multicast forwarding in the kernel, we need to specify which interfaces we want multi-cast forwarding to be enabled, i.e. interfaces that will receive or forward multicast data packets. This is accomplished by creating virtual interfaces, using the **MRT_ADD_VIF** option. The **MRT_DEL_VIF** can be used to remove previously created virtual interfaces. The structure used to create/remove virtual interfaces is presented in Listing 1. A virtual interface is identified by a number (*vifc_vifi*) and is created by specifying the IP address (*vifc_lcl_addr*) or the index (*vifc_lcl_ifindex*) of the physical interface. A virtual interfaces can also identify a logical interface like a tunnel interface or a register interface by specifying the corresponding flag (*vifc_flags*). Tunnel interfaces are used to create a tunnel and send data packets to a router that is not directly connected to a physical interface (with IP specified in *vifc_rmt_addr*). Register interfaces are used to get the received packet content, useful in PIM-SM to use in PIM Register messages. For a given virtual interface, it is also possible to specify a TTL threshold (*vifc_threshold*) in order to only accept multicast data packets that are received on that interface with a TTL greater than the one specified. The variable *vifc_rate_limit* is used to limit the bandwidth received by this interface.

```

1 struct vifctl {
2     vifi_t    vifc_vifi;           /* Index of VIF */
3     unsigned char vifc_flags;      /* VIFF_ flags */
4     unsigned char vifc_threshold;  /* ttl limit */
5     unsigned int  vifc_rate_limit; /* Rate limiter values (NI) */
6     union {
7         struct in_addr vifc_lcl_addr; /* Local interface address */
8         int             vifc_lcl_ifindex; /* Local interface index */
9     };
10    struct in_addr vifc_rmt_addr; /* IPIP tunnel addr */
11 };

```

Listing 1: Structure used to create/remove virtual interfaces using **MRT_ADD_VIF** and **MRT_DEL_VIF**.

After creating all virtual interfaces that will be used to forward multicast data traffic, one can start adding entries in the multicast routing table using the options **MRT_ADD_MFC** or **MRT_ADD_MFC_PROXY**. Previously created entries can be removed by using the options **MRT_DEL_MFC** or **MRT_DEL_MFC_PROXY**. The structure that is used to manipulate entries is presented in Listing 2. An entry is created by specifying the source IP (*mfcc_origin*) and group IP (*mfcc_mcastgrp*) addresses of the tree, virtual interface identifier of the root interface (*mfcc_parent*) and the OIL (*mfcc_ttls[MAXVIFS]*). Regarding the OIL, it is also possible to specify the minimum TTL that a packet must have to be forwarded through each non-root interface. If the minimum TTL of a given non-root interface is zero this means that packets will not be forwarded by that interface, otherwise the packet is only forwarded if the TTL of the data packet is greater than the defined TTL. The other options are not used for setting multicast routing entries and there is not much documentation referring when these should be used.

```

1 struct mfccctl {
2     struct in_addr mfcc_origin; /* Origin of mcast */

```

```

3      struct in_addr mfc_mcastgrp;           /* Group in question */
4      vifi_t mfc_parent;                     /* Where it arrived */
5      unsigned char mfc_ttls[MAXVIFS];       /* Where it is going */
6      unsigned int mfc_pkt_cnt;              /* pkt count for src-grp */
7      unsigned int mfc_byte_cnt;
8      unsigned int mfc_wrong_if;
9      int mfc_expire;
10 };

```

Listing 2: Structure used to create/remove entries from the multicast routing table.

If the option `MRT_ASSERT` is enabled, the socket can notify the user-level process when a non-root interface that was set to forward data packets for a given multicast routing entry starts receiving data packets regarding that same tree. By receiving this information, the process can act accordingly in order to elect which router is the one responsible for forwarding multicast data packets through the link that is connected by that interface. In the case of PIM, this would cause the exchange of PIM Assert messages in order to elect one Assert Winner.

Regarding option `MRT_PIM`, there is not much documentation referring it. By enabling this option we stopped receiving PIM control messages on sockets that would be used explicitly to exchange these control messages, suggesting that the kernel can handle PIM control messages by itself. Since this option has not the same meaning in all UNIX operating systems, such as FreeBSD (`MRT_PIM` is the same as `MRT_ASSERT` in this OS), it was not used.

By manually closing the `mroute` socket, all information on the multicast routing table is lost. Nevertheless, the option `MRT_DONE` should be set before closing the socket in order to explicitly remove all multicast information.

Until now a detailed description of all possible options that one can use to set or get information from the `mroute` socket was explained. Now it will be detailed how the kernel notifies the user-level process regarding tree information. In order to be notified by the kernel, we need to read all information that is sent to the `mroute` socket. This can be accomplished by creating a thread and having a while loop in which we interpret the received information and act accordingly.

The kernel can send to the user-level process the following types of messages:

- **IGMPMSG_NOCACHE** - a multicast data packet was received on an interface and there is no corresponding entry on the multicast routing table;
- **IGMPMSG_WRONGVIF** - a packet was received on a non-root interface that belongs to the OIL of a given tree;
- **IGMPMSG_WHOLEPKT** - content of a packet that was received on a Register interface.

When a multicast data packet is received by a router and there is no corresponding match in the multicast routing table, the kernel stores received packets in cache for a given amount of time and sends to the user-level process an `IGMPMSG_NOCACHE` message. This message has information regarding the interface that received those packets and the source and group addresses. By receiving this message, we should create an entry in the multicast routing table. After creating the corresponding entry, using the `MRT_ADD_MFC` or `MRT_ADD_MFC_PROXY` options, cached packets are forwarded through the OIL that was defined.

If the `MRT_ASSERT` option was enabled, the kernel can send `IGMPMSG_WRONGVIF` messages when a multicast data packet of a given tree is received on a non-root interface, if and only if that interface is part of the OIL. If a non-root interface is not part of the OIL, the kernel does not send any notification.

Regarding the `IGMPMSG_WHOLEPKT` message, this is sent when a packet was forwarded to a Register-type interface. This is accomplished by including a virtual interface identifier of a register interface into the OIL of a given tree. The kernel will send to the user-level process the content of received data packets. This can be useful for the PIM-SM protocol, where a router that is directly connected to the source will send data packets encapsulated on PIM Register messages to the Rendezvous Point (RP), while the (S,G) tree is not established between this router and the RP.

```

1  s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IGMP)
2  s.setsockopt(socket.IPPROTO_IP, MRT_INIT, true)
3  s.setsockopt(socket.IPPROTO_IP, MRT_ASSERT, true)
4

```

```

5  flags = 0x0
6  vif_index = 0
7  ip_interface = socket.inet_aton('10.0.0.11')
8  struct_mrt_add_vif = struct.pack('HBBI 4s 4s', vif_index, flags, 1, 0,
9                                     ip_interface, socket.inet_aton('0.0.0.0'))
10 s.setsockopt(socket.IPPROTO_IP, MRT_ADD_VIF, struct_mrt_add_vif)
11
12 vif_index = 2
13 ip_interface = socket.inet_aton('10.0.11.11')
14 struct_mrt_add_vif = struct.pack('HBBI 4s 4s', vif_index, flags, 1, 0,
15                                     ip_interface, socket.inet_aton('0.0.0.0'))
16 s.setsockopt(socket.IPPROTO_IP, MRT_ADD_VIF, struct_mrt_add_vif)
17
18 source_ip = socket.inet_aton('10.0.20.1')
19 group_ip = socket.inet_aton('224.1.1.2')
20
21 inbound_interface_index = 2
22 outbound_interfaces = [0]*32
23 outbound_interfaces[inbound_interface_index] = True
24 outbound_interfaces_and_other_parameters = outbound_interfaces + [0]*4
25
26 struct_mfcctl = struct.pack('4s 4s H ' + 'B'*MAXVIFS + ' IIIi', source_ip,
27                             group_ip, inbound_interface_index, *outbound_interfaces_and_other_parameters)
28 s.setsockopt(socket.IPPROTO_IP, MRT_ADD_MFC, struct_mfcctl)

```

Listing 3: Python example displaying creation of mroute socket.

Listing 3 shows how to manipulate the multicast routing table of Linux using Python. Lines 1-3 show the creation of the *mroute* socket and how to enable MRT_INIT and MRT_ASSERT options. Consider that a router has two interfaces, i0 and i2, that must participate in the multicast routing process with IP addresses 10.0.0.11 and 10.0.11.11 respectively. Lines 5-10 show the creation of the virtual interface corresponding to i0 and lines 12-16 show the same regarding interface i2. Regarding the creation of both virtual interfaces, flags are set to 0x0 because both virtual interfaces represent physical interfaces, the identifier of virtual interface of i0 is set 0 and to 2 on i2. Then in lines 21-28 it is presented the creation of an entry regarding tree (10.0.20.1, 224.1.1.2) with root interface set to virtual interface identifier 2 (interface i2) and Outgoing Interface List (OIL) including virtual interface index 0 (only i0 forwards multicast traffic). Regarding the functions used, socket.inet_ntoa allows to convert an IP address from a string format to a byte format, struct.pack allows to make conversions between Python values and C structs represented as Python strings (first parameter of this function).

1.2 Socket - send and receive multicast packets

Linux allows a socket to send and receive multicast data packets but it is required to enable a certain features using the *setsockopt* function. The most important options are:

- **IP_MULTICAST_IF** - allows to specify from which interface, multicast packets should be sent;
- **IP_MULTICAST_TTL** - allows to set the TTL of outgoing multicast packets originated from the socket. The default TTL for multicast packets is 1;
- **IP_MULTICAST_LOOP** - allows to set if a multicast packet should be looped back to the host, i.e. if a packet that was sent by a given socket should be received by other sockets from the same machine, that are interested in the group that is being targeted by the transmitted packet;
- **IP_ADD_MEMBERSHIP** - this allows to join a given multicast group. The socket will receive data packets that are destined to the previously joined group(s);
- **IP_DROP_MEMBERSHIP** - this allows to leave a multicast group that was previously joined (using the option above);
- **IP_ADD_SOURCE_MEMBERSHIP** - this allows to join a given source-specific multicast group. The socket will receive data packets originated from the specified source that are destined to the specified group;
- **IP_DROP_SOURCE_MEMBERSHIP** - this allows to leave the source-specific multicast group that was previously joined using the option above.

A more detailed description of each option and additional options can be found in the IP man pages [5].

Given the options above, if we want to receive multicast data packets from any source with the destination multicast group address “224.1.1.1” and destination UDP port “1234”, the example below shows how to accomplish this using Python:

```
1 # Create the datagram socket
2 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
3
4 # Bind server address (port number)
5 sock.bind(('', 1234))
6
7 # Tell the operating system to join multicast group on this socket
8 sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP,
9                 socket.inet_aton('224.1.1.1') + socket.inet_aton(ip_interface))
10
11 while is_running:
12     data, address = sock.recvfrom(1000)
13     print(data)
```

Listing 4: Python example displaying how to receive multicast data packets of group 224.1.1.1.

We start by creating a socket from family `AF_INET` (IPv4 socket) of type `SOCK_DGRAM` (because multicast only allows to send and receive datagrams). Then we specify the destination port of the received data packets that should be received by this socket, in this case destination port 1234. Then we use the function `setsockopt` to set the option `IP_ADD_MEMBERSHIP` in order to specify the group that the socket should join (224.1.1.1) and from which interface the joined group should receive multicast data packets (characterized by the IP address of the interface - `ip_interface`). The function `inet_aton` allows to convert the IP address from a string format to a byte format. Finally we use a while loop to receive multicast data packets and to print the received data.

If we want to send multicast data packets destined to group address “224.1.1.1” and destination UDP port “1234”, the example below shows how to accomplish this using Python:

```
1 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
2
3 # Set the time-to-live greater than 1 so they do go past the local network segment.
4 ttl = struct.pack('b', 12)
5 sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, ttl)
6
7 input_msg = 'hello world'
8 multicast_group = '224.1.1.1'
9 udp_port = 1234
10 sock.bind((ip_interface, udp_port))
11 sock.sendto(input_msg.encode('utf-8'), multicast_group)
```

Listing 5: Python example displaying how to send multicast data packets of group 224.1.1.1.

We create the same type of socket as the other example, then we define the TTL of the packets to a value greater than one (in order for the packet to be forwarded outside the link in which the interface is connected to - in the example the TTL is set to 12), we bind this socket to a given physical interface and to a given destination UDP port and then simply send the packet using the function `sendto`. This last function receives as argument the destination IP address (destination multicast group address).

1.3 IGMP - host part

Linux already implements part of the IGMP state machines, but only the host part. So if a programmer wants to receive multicast data packets from a given multicast group, this can be accomplished by opening a socket and specifying from which groups that socket should receive packets from, like Listing 4. The kernel automatically sends IGMP Report messages, answering to queries from routers, which alleviates the programmer from this task. Multicast packets that have a destination IP address that was not specifically joined by any socket are dropped and not processed.

The kernel can implement multiple versions of IGMP. The IGMP version can be set manually, otherwise the version will be selected automatically according to the received IGMP Query messages. For example, if the host has an implementation of IGMPv3, after joining a given group it will start sending IGMPv3 Report messages but if it hears an IGMPv2 Querier message, the host starts sending IGMPv2 Report messages.

Regarding IGMP router state machines, these are not implemented by the kernel, so in order for a router to get information regarding the membership status of directly connected hosts, IGMP needs to be implemented along with the multicast routing protocol.

2 Implemented protocols

Up until now we have described multicast-related features offered by the Linux kernel. In this section we will describe all protocols that were implemented during the MSc Dissertation.

We have implemented three protocols which are IGMP, PIM-DM and HPIM-DM. For each protocol we start by briefly describe the corresponding specification (state machines, timers, ...) and then we describe the corresponding implementation.

2.1 IGMP - router part

IGMP is a protocol independent from the multicast routing protocol, but it is essential in order for a router to understand which groups have hosts interested in receiving multicast data packets. As it was said in section 1.3, Linux already implements the IGMP host's state machines, but not the router's state machines. In order for a router to understand which groups have hosts interested in receiving data packets, it is required to implement the IGMP router's state machines. It was decided to implement IGMPv2 [6] since it is simpler compared to IGMPv3 [7] and because the additional features of IGMPv3 are not required for this work.

2.1.1 Protocol Specification

IGMPv2 defines three state machines for routers. One state machine is used to define if a router is the Querier of a given link, the other state machines are used to define if a given group has members interested in receiving data packets. The membership state machine depends whether the state of an interface is in a Querier or Non-Querier state. All state machines are specified in RFC2236 [6] and are shown in Figures 1, 2 and 3.

Regarding IGMPv2 Querier state machine, an interface of a router can be in one of two possible states, Querier or Non-Querier. This state machine defines if an interface of a router is responsible for querying hosts regarding membership information. Only one router per subnetwork should be in the Querier state, which is the router with the lowest IP address. All other routers connected to the same subnetwork must be in Non-Querier state, having the job of monitoring if the Querier is alive. Initially, after booting up, a router starts in the Querier state and if it hears a Membership Query message originated from a router with a lower IP address, it transitions to Non-Querier. This allows to have only one Querier per subnet.

For each interface, a router requires to have one of two timers, depending on its state, which are:

- **general query timer** - regulates the transmission of IGMP General Query messages. Only interfaces that are in a Querier state use this timer;
- **other querier present timer** - allows to monitor if the Querier is alive. This timer is only used by Non-Querier interfaces and is reset after hearing a Membership Query message. When this timer expires, the Non-Querier interface transitions to Querier in order to reelect the Querier of that subnetwork.

Regarding IGMPv2 membership state machines, these define, for a given interface, if a group address has members interested in receiving multicast data traffic. The membership state machine of an interface in a Querier state is more complex, since that interface beside reacting to the reception of messages, also requires query members, after one of them leaves a group. The membership state machine defines four states for interfaces that are in a Querier state and three states for interfaces that are in a Non-Querier state. All defined states are the following:

- **No Members Present** - the group has no members interested in receiving multicast data packets. All groups are initially in this state;

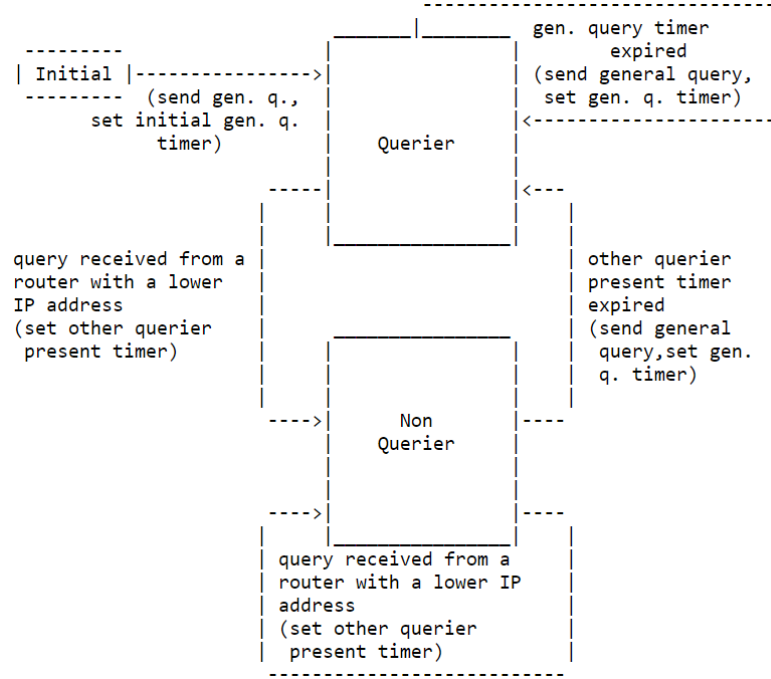


Figure 1: IGMPv2 Querier state machine.

- **Members Present** - the group has at least one member interested in receiving multicast data packets;
- **Version 1 Members Present** - this state is used for backward compatibility with IGMPv1 hosts. In this state, the group has at least one interested member that transmitted an IGMPv1 Membership Report message;
- **Checking Membership** - this defines a transient state. A router by hearing an IGMPv2 Leave message, waits a given amount before defining that the group has no members. If after a given amount of time the router does not hear an IGMP Membership Report message, the router transitions to “No Members Present”, otherwise the router transitions to “Version 1 Members Present” or “Members Present” depending on the version of the received Report message.

Membership information is only valid for a given amount of time, that is why periodically hosts need to retransmit Report messages. Regarding a given group, a router only monitors its state, it does not monitor which hosts are interested. For this reason, if a given group has interested members, that same group only stays in state “Members Present” or “Version 1 Members Present” as long as hosts keep refreshing the routers’ group state. A router, after a given amount of time, if it does not hear a Report message regarding a given group, it transitions to “No Members Present” since no one refreshed the router’s group state. This timeout was the way that IGMPv1 used to know when a given group stopped having members. IGMPv2 also uses the timeout mechanism but allows an host to accelerate this process by using the Leave message. If an host transmits an IGMP Leave message, the Querier transitions to Checking Membership (if there are no IGMPv1 hosts) and transmits an IGMP Group-Specific Query message. This message is used to give opportunity for other hosts still interested in this group, to inform all routers about their interest with a Report message, otherwise after a given amount of time without hearing a Report message, all routers transition this group to “No Members Present” state.

IGMPv2 membership state machine uses three timers to maintain a group state, which are:

- **timer** - timer used to maintain membership state. If this timer expires, it means that there are no longer hosts interested in this group. This timer is also used to verify membership interest, after hearing an IGMP Leave message;
- **retransmit timer** - when a router transitions to state “Checking Membership” it sends a Group-Specific Query and waits a given amount of time for interested hosts to send Report messages. This timer regulates the retransmissions of Group-Specific Query messages during the “Checking Membership” state;

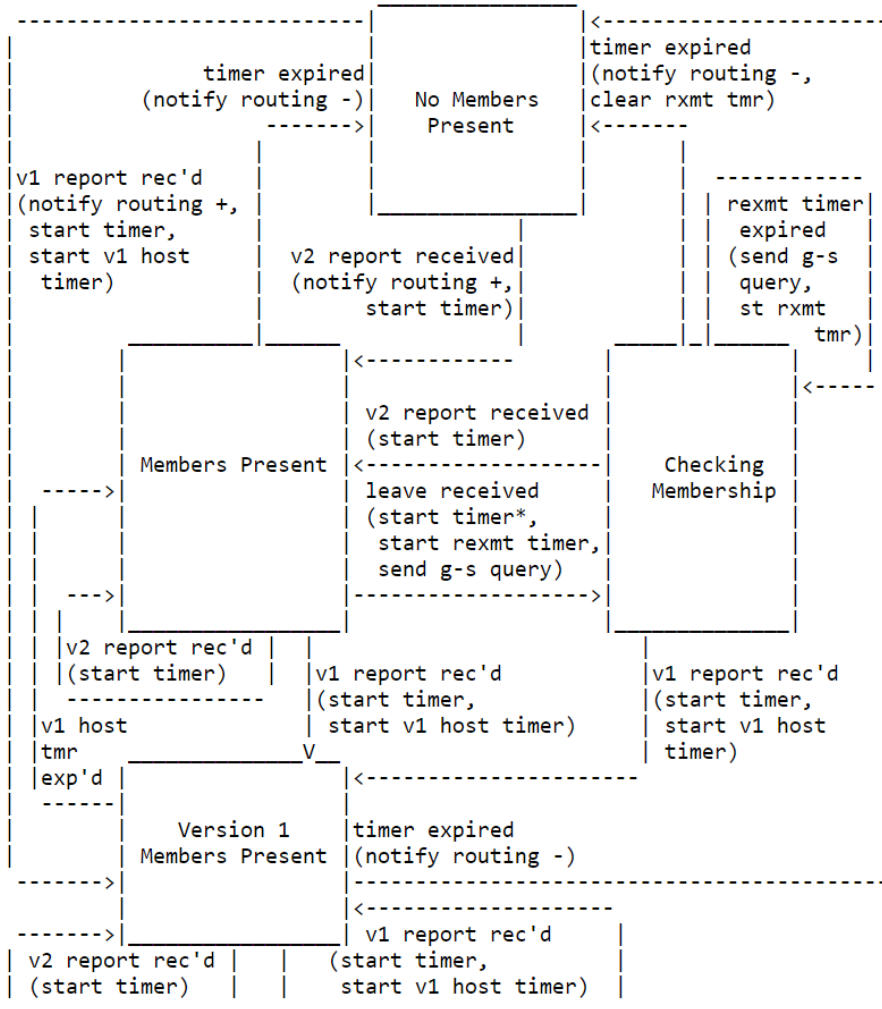


Figure 2: IGMPv2 Membership state machine of Querier routers.

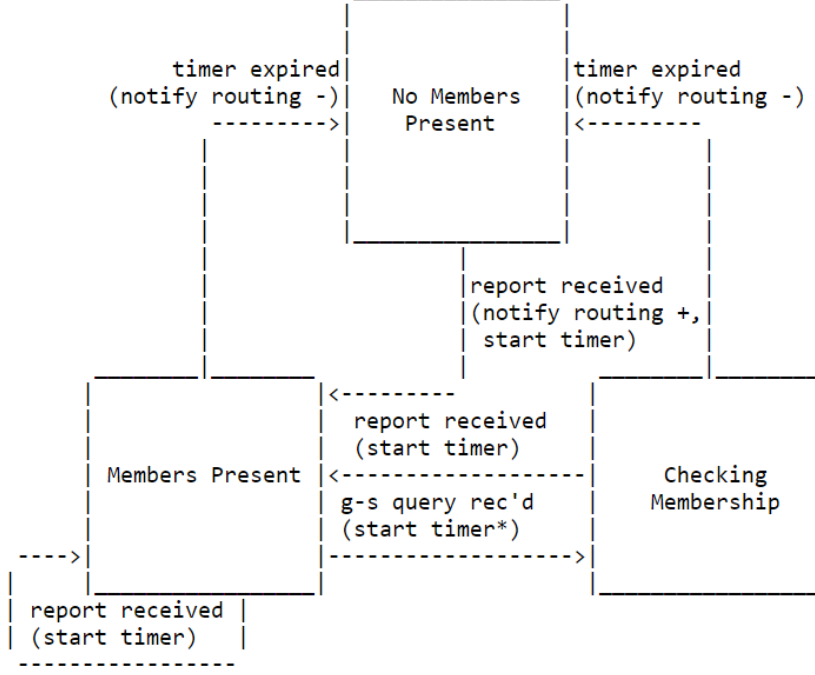


Figure 3: IGMPv2 Membership state machine of Non-Querier routers.

- **v1 host timer** - timer used to maintain membership state of IGMPv1 hosts. When this timer expires it means that there are no IGMPv1 hosts interested in this group.

Regarding the messages that hosts and routers can send/receive, there are four types of IGMP messages:

- **Membership Query** - this message is sent by routers via interfaces that are in a “Querier” state. Beside this message being used by routers to elect the Querier, it is also used to query hosts’ interest. There are two sub-types of Query messages, one used by the router to query about all groups, being this message called General Membership Query, the other message is called Group-Specific Query and is used by the Querier to query a specific group;
- **Version 2 Membership Report** - this message is sent by hosts (that implement IGMPv2) and has the goal of informing all routers, connected to the same link, about their interest in a given group;
- **Version 1 Membership Report** - same as above, but is only sent by hosts that implement IGMPv1;
- **Leave Group** - this message is sent by hosts that are no longer interested in a group that have previously joined. This message is only sent by hosts that implement IGMPv2.

IGMP messages are encapsulated inside IP datagrams, being identified by having 2 as the IP protocol number (at the IPv4 header).

2.1.2 Protocol Implementation

This implementation is stored in our GitHub repositories. Since the IGMP implementation was not intended to be used alone, this is integrated in the implementations of PIM-DM and HPIM-DM. These can be accessed in our GitHub repositories here [8, 9].

Regarding our implementation, the following diagram (Figure 4) represents the used software architecture. This is an object oriented implementation and rectangles represent the used classes, lines with a closed arrowhead represent associations and lines with an open arrowhead represent inheritance.

Below there is an explanation of each class:

- **InterfaceIGMP** - this represents a physical interface that has IGMP enabled. It simply sends and receives IGMP control messages via a raw socket. An InterfaceIGMP will have a thread running in

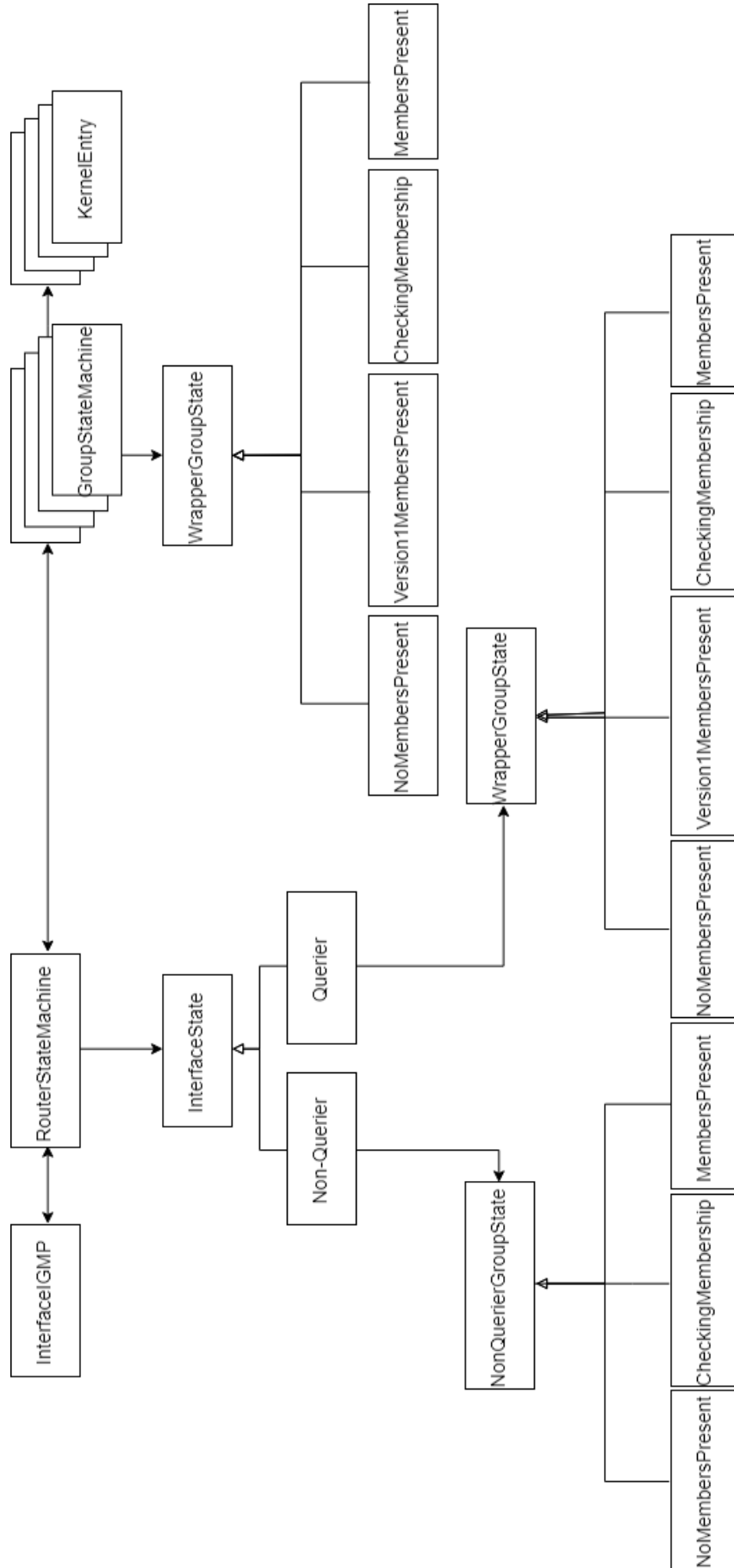


Figure 4: IGMPv2 state machine implementation diagram.

the background, waiting for these messages to arrive. Each `InterfaceIGMP` is associated with one `RouterStateMachine` which will process received messages by the raw socket;

- **RouterStateMachine** - this object beside being associated with one `InterfaceIGMP` (in order to send IGMP control messages), it is also associated with one `InterfaceState` and multiple `GroupStateMachines`. All received IGMP messages will be processed by this object and propagated to other state machines according to the messages' type. This object can create information regarding new groups that were reported by hosts, being represented by `GroupStateMachines`. This object also uses the two timers that were referred in section 2.1.1, which are the "general query timer" and "other querier present timer", in order to monitor when to send Queries and if the Querier is alive, respectively;
- **InterfaceState** - abstract class representing the state of an interface, which can be Querier or Non-Querier;
- **Querier** - implements the Querier state of the Querier state machine. The protocol specification is represented in Figure 1. This object implements the transitions for each possible event of the Querier state. This object also references the four possible states in which a group can be at (`NoMembersPresent`, `CheckingMembership`, `MembersPresent` and `Version1MembersPresent`), which are represented in Figure 2;
- **NonQuerier** - implements the Non-Querier state of the Querier state machine. The protocol specification is also represented in Figure 1. This object implements the transitions for each possible event of the Non-Querier state. This object also references the three possible states in which a group can be at (`NoMembersPresent`, `CheckingMembership` and `MembersPresent`), which are represented in Figure 3;
- **GroupStateMachine** - this object aggregates information regarding a given group. This object besides having information regarding the group, it stores the three timers referred in section 2.1.1 and it is also associated with a given group state (one of the subclasses of `WrapperGroupState`) and a list of `TreeInterfaces`. The subclasses of `WrapperGroupState` define the state of this group and the `TreeInterfaces` represent interfaces of trees maintained by the multicast routing protocol that have the same group address. The list of `TreeInterfaces` will be used to notify them about membership changes. If the multicast routing protocol maintains trees (`S1,G1`) and (`S2,G1`), `GroupStateMachine` of group `G1` would store the reference of both entries in order to notify them about changes of membership. In the protocol specification, transitions of membership are represented by actions "notify routing +" and "notify routing -" in Figures 2 and 3;
- **WrapperGroupState** - "interface" for the membership states. The implemented states of `WrapperGroupState` are "wrappers" in the sense that they do not implement directly the membership state, instead they act as an indirection layer. By having this indirection, we access the real implemented state via `RouterStateMachine` since the membership state depends whether the interface is in a Querier or Non-Querier state;
- **TreeInterface** - represents an interface (root or non-root) of a given (`S,G`) tree that is being maintained by the multicast routing protocol. The explanation of this object will be detailed in the next sections, related with the multicast routing protocols.

All `GroupStateMachines` are stored by the corresponding `RouterStateMachine` via a dictionary. This allows to access the state of a given group by the group address (key of the dictionary), which allows to accelerate the lookup ($O(1)$). The `TreeInterfaces` are stored in `GroupStateMachine` using a list, since an interface of a given tree has no advantage in being directly indexed by a key, like the `GroupStateMachines`.

The reason for having the membership state machines implemented using different classes, by three interfaces/abstract classes is for efficiency purposes. If the `GroupStateMachine` directly referenced the implementation of the state of a group and if the interface transitioned from Querier to Non-Querier or vice-versa, this would require to iterate through every `GroupStateMachine` and change its state to the implementation of that same state according to the Querier state machine. A better way to avoid fixing the states each time there is a change in the Querier state machine is to simply set the state to a wrapper/proxy of that same state. All events of a group are sent to the proxy, which simply invokes that same event over the implemented state that is referenced by `RouterStateMachine` in the Querier or Non-Querier state. A simple code snippet illustrates how this implementation handles the reception of an IGMPv1 Membership Report at an interface in a Querier state:

```

1 class GroupState(object):
2     def __init__(self, router_state, group_ip: str):
3         self.router_state = router_state

```

```

4         self.state = NoMembersPresent # Wrapper state of NoMembersPresent
5         ...
6
7     def get_interface_group_state(self):
8         return self.state.get_state(self.router_state)
9
10    def receive_v1_membership_report(self):
11        self.get_interface_group_state().receive_v1_membership_report(self)
12
13    ...
14 File NoMembersPresent #represents a static class
15    def get_state(router_state):
16        return router_state.interface_state.get_no_members_present_state()
17
18 class RouterState(object):
19    def __init__(self, interface):
20        # state of the router (Querier/NonQuerier)
21        self.interface_state = Querier
22        ...
23
24 class Querier:
25    @staticmethod
26    def get_no_members_present_state():
27        return NoMembersPresent #Implementation of NoMembersState
28                               #state of Querier interface
29    ...

```

Listing 6: Code illustration on how to get access to implemented membership state.

When an interface receives an IGMPv1 Membership Report regarding group G via InterfaceIGMP, it sends that message to RouterStateMachine. The latter will search for the GroupStateMachine referenced by the received messages and will invoke method `recv_v1_membership_report` of GroupStateMachine (line 10-11). This method first obtains the real state implementation of this group by invoking `get_interface_group_state()` method (declared in lines 7-8). This method invokes over the wrapper state (in this case `NoMembersPresent` - declared in the constructor - line 4) the `get_state` method (implemented in lines 15-16). The `get_state` method will go to the RouterState and obtain the real implementation of `NoMembersPresent` state (implemented in lines 26-27).

The IGMP implementation uses threads in the following classes:

- **InterfaceIGMP** - a thread to handle the reception of messages. This allows to run in the background a method that has a while loop waiting for messages over the socket. When a message is received, it invokes the appropriate method according to the type of the received IGMP message.
- **RouterStateMachine and GroupStateMachine** - the timers that are used to maintain the protocol state correspond to threads. These timers are implemented by the Python Timer object¹, which is a subclass of Python's Thread. These timers are simply threads that execute a method after a given amount of time, useful for running events related to the expiration of timers.

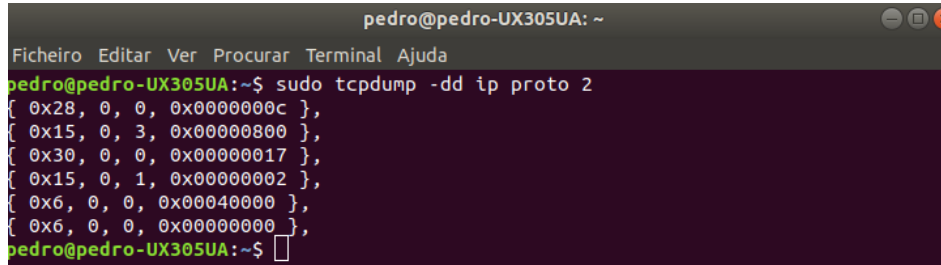
Regarding the reception of IGMP control messages, by the InterfaceIGMP object, the socket used for receiving these messages does not use the API referred in section 1.2, for the following reasons:

- IGMP control messages can be sent to any group address belonging to 224.0.0.0/4, which would require to explicitly join all possible group addresses in order to properly receive those messages. A socket can only join up to 20 groups using the option `IP_ADD_MEMBERSHIP`, so this would require to have multiple sockets to receive control messages;
- Even if we explicitly joined all possible groups, Linux would send Report messages regarding all joined groups. This would cause other routers connected to the same link to not know when hosts are interested in receiving data packets since all routers would Report all possible groups in order to process IGMP control messages;

¹<https://docs.python.org/2/library/threading.html#timer-objects>

- If we do not explicitly join all possible groups, these packets would be dropped at a lower layer in the network stack, which would not allow a router to know which groups have members.

In order to handle this problem, it was used a Berkeley Packet Filter (BPF) [10] for a socket to receive IGMP packets. This is accomplished by defining a filter to a given socket. The syntax of the filter was “ip proto 2”, which allows the socket to receive all IPv4 packets that are received with IP protocol number equal to 2. In order to set this filter in a given socket it is required to know the corresponding bytecode, which can be discovered by running the command “tcpdump -dd ip proto 2” in the terminal:



```

pedro@pedro-UX305UA: ~
Ficheiro Editar Ver Procurar Terminal Ajuda
pedro@pedro-UX305UA:~$ sudo tcpdump -dd ip proto 2
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 3, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 0, 1, 0x00000002 },
{ 0x6, 0, 0, 0x00040000 },
{ 0x6, 0, 0, 0x00000000 },
pedro@pedro-UX305UA:~$

```

Figure 5: Get BPF bytecode.

Now to use this filter in the socket, we set the socket’s `SO_ATTACH_FILTER` option with the function `setsockopt` and from that point on, all IGMP messages are received by this socket. The following code snippet illustrates how to set this option in Python:

```

1 FILTER_IGMP = [
2     struct.pack('HBBI', 0x28, 0, 0, 0x0000000c),
3     struct.pack('HBBI', 0x15, 0, 3, 0x00000800),
4     struct.pack('HBBI', 0x30, 0, 0, 0x00000017),
5     struct.pack('HBBI', 0x15, 0, 1, 0x00000002),
6     struct.pack('HBBI', 0x6, 0, 0, 0x00040000),
7     struct.pack('HBBI', 0x6, 0, 0, 0x00000000),
8 ]
9
10 # RECEIVE SOCKET
11 rcv_s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
12                       socket.htons(InterfaceIGMP.ETH_P_IP))
13
14 # receive only IGMP packets by setting a BPF filter
15 bpf_filter = b''.join(InterfaceIGMP.FILTER_IGMP)
16 b = create_string_buffer(bpf_filter)
17 mem_addr_of_filters = addressof(b)
18 fprog = struct.pack('HL', len(InterfaceIGMP.FILTER_IGMP), mem_addr_of_filters)
19 rcv_s.setsockopt(socket.SOL_SOCKET, InterfaceIGMP.SO_ATTACH_FILTER, fprog)

```

Listing 7: Python example on how to set BPF filter regarding IGMP control packets.

Lines 1-8 define the filter obtained by Figure 5. Lines 10-12 create the socket that will be used to receive IGMP control messages. In order to set the BPF filter, the socket needs to be of family `AF_PACKET`, type `SOCK_RAW` and protocol number `ETH_P_IP` (frames that contain IPv4 packets). Lines 14-18 convert `FILTER_IGMP` from a list of bytes to a reference of the variable that stores those bytes. `fprog` will store the required structure of the filter. Line 19 sets the option `SO_ATTACH_FILTER` of the socket.

2.2 PIM-DM

We searched for PIM-DM implementations and we have only found one [11], but according to its release notes, the latest change was made in December of 1998. Since this implementation was no longer maintained and it was outdated, we opted to make an implementation of PIM-DM ourselves, in Python, that was according to the latest protocol specification (RFC3973 [12]).

2.2.1 Protocol Specification

PIM-DM defines five state machines in order for a router to know when it should forward multicast traffic through a given interface, when it should notify the upstream router regarding changes of membership interest and to decide which router is the Assert Winner of a given link. All state machines are specified in RFC3973 [12]. Figures 6, 7, 8 and 9 show a simplistic version of four of these state machines. The other state machine is simply a “boolean” and is directly dependent on the state of IGMP. These state machines are defined for each (S,G) tree and have the following goal:

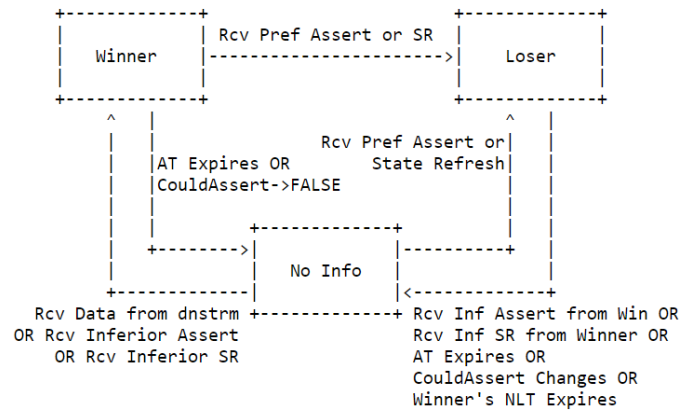


Figure 6: Simplistic PIM-DM Assert state machine.

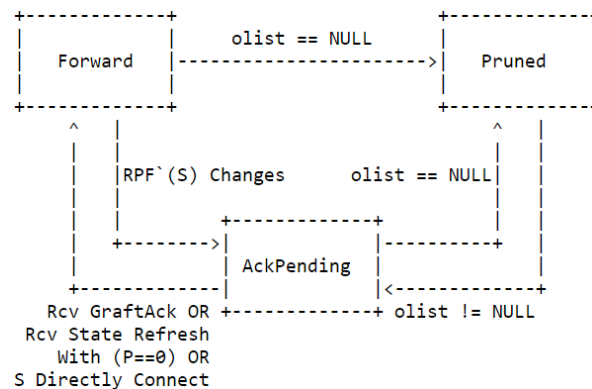


Figure 7: Simplistic PIM-DM Upstream state machine.

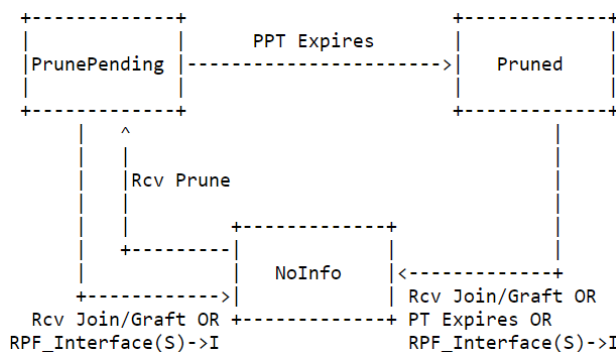


Figure 8: Simplistic PIM-DM Downstream state machine.

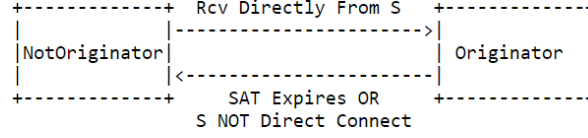


Figure 9: Simplistic PIM-DM State Refresh state machine.

- **Upstream Interface** - only the root interface, of a given (S,G) tree, is responsible for executing the events and respective actions of this state machine. The Upstream Interface state machine depends directly on the state of all non-root interfaces and it simply defines when this router is interested in receiving multicast data traffic regarding a certain tree. This state machine defines three states, which are the Forward, Pruned and AckPending and they are used to know when that interface should send Join, Prune and Graft messages. A root interface is in a Forward state when the router has at least one non-root interface responsible for forwarding multicast traffic and there is interest in receiving data traffic through the link that connects that interface. A root interface is in a Pruned state when the router is not interested in receiving multicast data traffic because no non-root interface has hosts nor routers interested in receiving traffic or because those interfaces are not responsible for forwarding multicast traffic. The AckPending is a transient state used to guarantee reliability of Graft messages, which are messages that are used to inform the upstream router that the router transitioned from a state in which there was no interest in receiving multicast data traffic (Pruned state) to a state in which the router has interest in receiving multicast data traffic regarding that same tree (wants to transition to Forward but it stays in AckPending while it does not hear a Graft-Ack in response to a Graft).

This state machine requires three timers in order to maintain the state of a root interface, which are the GraftRetry Timer, Override Timer and Prune Limit Timer. The GraftRetry timer regulates the retransmission of Graft messages, i.e. if this timer expires and the router did not hear a Graft-Ack message, it retransmits a Graft message and resets this timer. The Override timer has the goal of regulating the transmission of Join messages after hearing a Prune message, i.e. the timer is used to control the Prune Override mechanism. The Prune Limit timer is used to limit the transmission rate of Prune messages that are sent by the interface, i.e. the root interface can only send Prune messages from time to time.

- **Downstream Interface** - only non-root interfaces, of a given (S,G) tree, are responsible for executing the events and respective actions of this state machine. The states of this state machine define, for a given non-root interface, if there is downstream interest in a (S,G) tree. This downstream interest is determined by the reception of Prune, Join and Graft messages. This state machine defines three states, which are the NoInfo, PrunePending and Pruned. The NoInfo state defines that a non-root interface has at least one downstream router interested in receiving multicast data traffic or simply there was no router that pruned the link. The Pruned state defines that there is no interest, i.e. after hearing a Prune, no downstream router overrode this message with a Join. The PrunePending state corresponds to a transient state caused by hearing a Prune message. The router will stay in PrunePending state for a given amount of time, in order to give opportunity for downstream routers to override the Prune.

This state machine requires two timers to maintain the state of non-root interfaces, which are the PrunePending Timer and Prune Timer. The PrunePending timer is set when a non-root interface hears a Prune message and its goal is to give time to downstream routers to override the Prune by sending a Join message, i.e. the router will only place its non-root interface in a Pruned state if this timer expires due to not hearing a Join message. The Prune timer controls the amount of time that a non-root interface will stay in a Prune state (around 3 minutes) and is set when the PrunePending timer expires. If the Prune timer expires, the non-root interface transitions to a NoInfo state, causing the network to be flooded again with data packets.

- **Assert** - the Assert state machine is defined for all types of interfaces. For root interfaces, this state machine is used to understand which upstream router is the Assert Winner. For non-root interfaces is used to elect the Assert Winner. This state machine is characterized by three states, which are the Winner, Loser and NoInfo. The Winner state can only be achieved by non-root interfaces and it simply means that this interface has won the Assert and that it has been elected as the Assert Winner of a given link. The Loser state can be achieved by root and non-root interfaces. For root interfaces the Loser state means that the router knows which router is the Assert Winner by

hearing Assert messages. For non-root interfaces the Loser state means that the interface has lost the Assert and that there is other router with a better metric or equal metric and greater IP. The NoInfo state can be achieved by both root and non-root interfaces and it simply means that there is no information regarding who is the Assert Winner. For root interfaces, Assert Winner information will be obtained by using the unicast routing table. For non-root interfaces that are in a NoInfo state, they consider themselves as Assert Winners, because there was no election yet (maybe there is only one upstream router connected to the link or Assert messages were lost). Even if Assert messages are lost, if a non-root interface that is in a NoInfo or Winner state receives multicast data packets, this causes the exchange of Assert messages that triggers a reelection of a single Assert Winner.

This state machine requires one timer to maintain the state of interfaces, which is the Assert Timer, used to control the amount of time that the Assert state should be maintained. When this timer expires, the Assert state is removed, which causes a transition to the NoInfo state.

- **State Refresh** - this state machine is used by a router to know when and if it should send State Refresh messages. This state machine defines two states, which are the Originator and NotOriginator. Only routers that are directly connected to active sources of multicast traffic are placed in Originator state, otherwise they are placed in NotOriginator state. In the Originator state, the router will periodically send State Refresh messages through non-root interfaces. Routers in the NotOriginator state simply do not generate State Refresh messages.

State Refresh messages will be flooded through the network, causing the refresh of the Pruned state in non-root interfaces (in the Downstream Interface state machine) and also the reelection of the Assert Winner (in the Assert state machine).

This state machine requires two timers to maintain the state of a tree, which are the State Refresh Timer and Source Active Timer. The State Refresh timer is only used by routers that are in a Originator state (directly connected to active sources of multicast traffic) and controls when State Refresh messages should be generated. The Source Active Timer is used by the router directly connected to the source, to understand when the source of multicast traffic is no longer active. When the Source Active Timer expires, the router directly connected to the source, in Originator state, will transition to NotOriginator state and stop generating State Refresh messages.

- **Local Membership** - this state machine defines if a given interface has members interested in receiving multicast data traffic, being directly dependent on the IGMP state machine. The defined states are NoInfo and Include. The NoInfo state defines that there is no interest. The Include state defines that there are directly connected hosts interested. The NoInfo state means that the group (in case of IGMPv2) is in “No Members Present” state. The Include state means that the group is in one of the other three states (“Members Present” or “Version 1 Members Present” or “Checking Membership”) of IGMPv2.

The discovery of neighbors does not have any specific state machine. RFC3973 [12] only specifies the transmission frequency of Hello messages and when a known neighbor should be declared to have failed. While it does not have any state machine regarding this subject, some events about neighbor failures are used by the other state machines, that were described above, like the Assert state machine (in order to react to the Assert Winner’s failure).

Regarding the forwarding of multicast data packets, the RFC points that the outgoing interface list (OIL) is calculated the following way:

$$olist(S, G) = immediate_olist(S, G) \ (-) \ RPF_interface(S)$$

with $immediate_olist(S, G) = pim_nbrs \ (-) \ prunes(S, G) \ (+) \ (pim_include(*, G) \ (-) \ pim_exclude(S, G))$
 $(+) \ pim_include(S, G) \ (-) \ lost_assert(S, G) \ (-) \ boundary(G);$

The “macros” that are defined above determine which interfaces should forward multicast data packets regarding a given (S,G) tree. The operations (+) and (-) are used to perform calculations over sets of interfaces:

- **A (+) B** - this operator performs the union between two sets, i.e. the resulting set corresponds to all elements that are in A and B;
- **A (-) B** - this operator returns all elements that are in A and not in B.

The RFC's defined "macros" have the following purpose:

- **RPF_interface(S)** - returns the root interface for source S. PIM-DM uses the term RPF interface instead of root interface, but they mean the same thing;
- **pim_nbrs** - returns all interfaces that have at least one active neighbor;
- **prunes(S,G)** - returns all interfaces that are in a Pruned state;
- **pim_include(*,G)** - returns all interfaces that directly connect members of group G (determined by IGMP);
- **pim_include(S,G)** - returns all interfaces that directly connect members of group G that want traffic from source S (determined by IGMP);
- **pim_exclude(S,G)** - returns all interfaces that directly connect members of group G that do not seek to receive traffic from source S;
- **lost_assert(S,G)** - returns all interfaces that have lost the Assert, i.e. interfaces that are in a Loser state in the Assert state machine;
- **boundary(G)** - returns all interfaces with an administratively scoped boundary of group G. This is used to define boundaries in order to not forward multicast traffic of group G through all interfaces.

So the olist is defined by all interfaces that are returned by the macro `immediate_olist(S,G)`, excluding the `RPF_interface(S)`. This is obvious since the root interface (interface used to receive traffic) must not forward data packets received by it.

The macro `immediate_olist(S,G)` returns all interfaces that have at least one active neighbor, have members interested in receiving multicast traffic regarding this tree (only interested in group G or interested in both S and G) or are not in a Pruned state, did not lost the Assert mechanism and are not scoped regarding group G. This is one of the most difficult parts of the protocol, since the RFC defines the olist using macros that operate over sets, instead of specifying which conditions must be guaranteed in order for a non-root interface to forward data packets.

2.2.2 Protocol Implementation

The implementation of PIM-DM is stored in our GitHub repository and can be accessed here [8].

The following diagram (Figure 10) represents the used software architecture. Like IGMP, this is an object oriented implementation and rectangles represent the used classes, lines with a closed arrowhead represent associations and lines with an open arrowhead represent inheritance.

Below there is an explanation of each class:

- **Run** - this module is used to start/stop the protocol process and can be used to interact with it, in order to add/remove PIM/IGMP interfaces, list neighbors, list entries and state of all multicast trees. When the user wants to start the protocol process, this module creates a daemon process with an opened socket. This allows to have the protocol running in the background, and by having an opened socket, the user can interact with it, to send commands and retrieve information. The socket that is opened is of AF_UNIX family and SOCK_STREAM type. This family and type of socket is used for inter-process communication and simply allows processes to communicate with each other by writing and reading to a file. The daemon process that is running, simply has a while loop waiting for the arrival of commands on this socket, and acts accordingly to the command given by the user. Other process simply sends commands of the user, to the same file used for the inter-process communication. The daemon process has a reference of Main, which is a service that can interact with the protocol;
- **Main** - is a service used to interact with the protocol. All commands received by Run are invoked over Main and sent to Kernel, which is the core of this application. All commands that are used to retrieve information, like list neighbors and list entries, are processed by the Main. This service will retrieve information from the Kernel module, process that same information and send it back to Run in a human readable format (table format).
- **Interface** - abstract class that has common code related to InterfaceIGMP and InterfacePIM, such as a thread that is running in the background for receiving control messages, and common methods to obtain the IP address of the physical interface and to close the socket (when the interface is removed);
- **InterfaceIGMP** - corresponds to a simplistic view of the implementation of IGMPv2, that has been described in section 2.1.2. Before creating an InterfaceIGMP, it is required to create a virtual interface associated with the physical interface (in Kernel module);

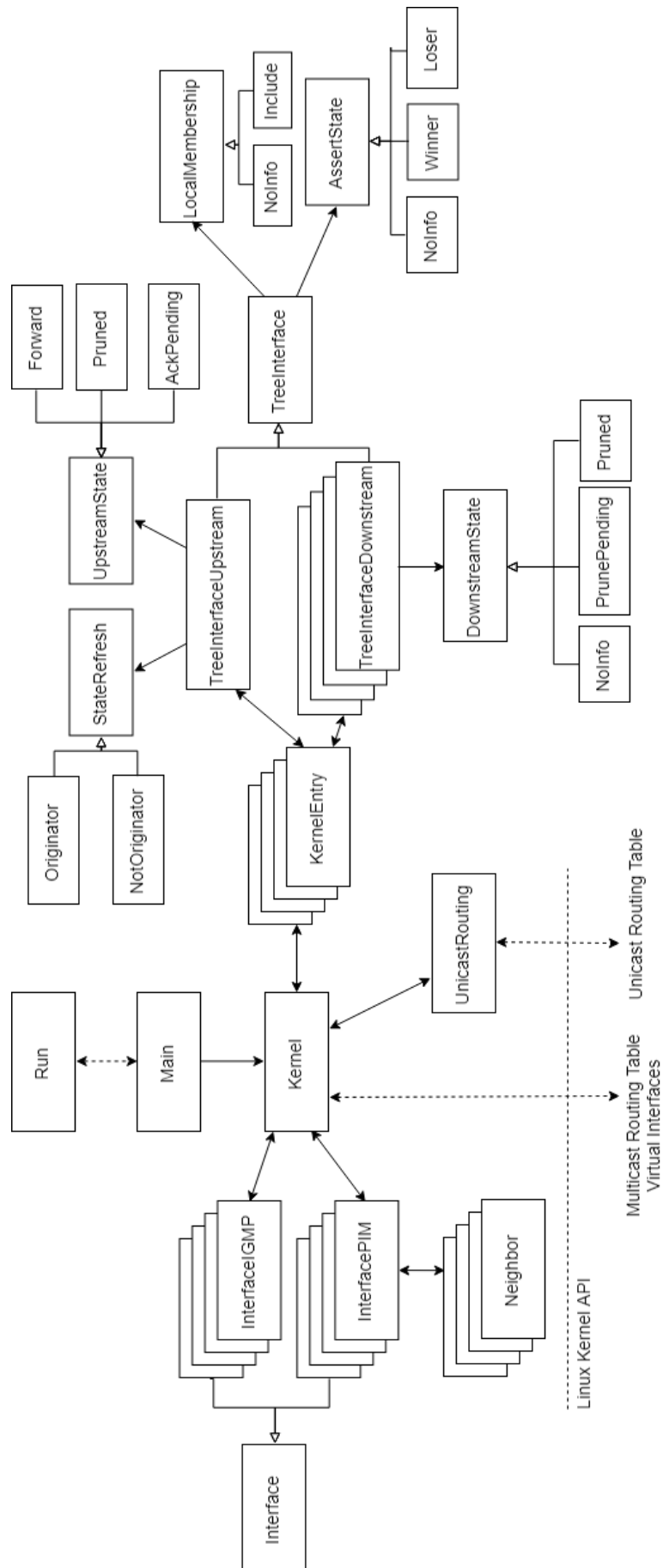


Figure 10: PIM implementation diagram.

- **InterfacePIM** - this represents a physical interface that has PIM enabled. It simply sends and receives PIM control messages via a raw socket. This socket will receive all packets that have 103 as the Protocol number at the IP Layer (PIM control messages). This class is responsible for monitoring the neighborhood relationships with other PIM enabled routers and will periodically send Hello messages. Before creating an InterfacePIM, it is required to create a virtual interface associated with the physical interface (in Kernel module). Each InterfacePIM is associated with multiple Neighbors that represent the known neighbors detected via Hello messages;
- **Neighbor** - this object represents a neighbor detected by a given InterfacePIM. This object will monitor the neighborhood relationship, in order to detect when it has failed. This object has a reference of InterfacePIM and a reference of all entries that require to monitor the liveness of this neighbor (trees that consider this neighbor to be Assert Winner). The reference to those trees are useful in order to quickly notify them, when this neighbor, Assert Winner of those trees, fails;
- **Kernel** - this class is intended to be an abstraction regarding all interactions between this application and the Linux kernel. The API offered by the Linux kernel has been discussed in section 1.1. This class has an opened *mroute* socket, which allows to interact with the kernel in order to set/remove/change entries in the multicast routing table and also to create/remove virtual interfaces. Beside the abstraction related with the kernel, it also stores KernelEntries which are an abstraction of entries in the multicast routing table. The Kernel class also stores all interfaces that were previously added by the user (InterfaceIGMP and InterfacePIM) and is responsible for creating/removing virtual interfaces in the kernel, when a physical interface is added/removed respectively;
- **KernelEntry** - corresponds to a (S,G) tree that is present in the multicast routing table. This class has information about the state of each interface that has been previously enabled. The interfaces that are referenced by this class are of type TreeInterfaceUpstream and TreeInterfaceDownstream. One KernelEntry must have one TreeInterfaceUpstream, which corresponds to the root interface, and can have multiple TreeInterfaceDownstreams, which correspond to the non-root interfaces. These interfaces store all states of an interface regarding a given tree;
- **TreeInterface** - super-class of TreeInterfaceUpstream and TreeInterfaceDownstream. It references state machines that are shared by all types of interfaces, such as Local Membership and Assert. It has also common methods used by all types of interfaces;
- **TreeInterfaceUpstream** - sub-class of TreeInterface, representing an abstraction of a root interface of a given tree. Besides the state of the super-class, it also stores state regarding the Upstream Interface state machine (UpstreamState class) and the State Refresh state machine (StateRefresh class);
- **TreeInterfaceDownstream** - sub-class of TreeInterface, representing an abstraction of a non-root interface of a given tree. Besides the state of the super-class, it also stores state regarding the Downstream Interface state machine (DownstreamState class);
- **DownstreamState, NoInfo, PrunePending and Pruned** - implementation of the Downstream Interface state machine;
- **UpstreamState, Forward, Pruned and AckPending** - implementation of the Upstream Interface state machine;
- **StateRefresh, Originator and NotOriginator** - implementation of the State Refresh state machine;
- **AssertState, NoInfo, Winner and Loser** - implementation of the Assert state machine;
- **LocalMembership, NoInfo and Include** - implementation of the Local Membership state machine. This is similar to a boolean and it is only useful for knowing if there are directly connected hosts interested in receiving traffic regarding this tree (this state is maintained by IGMP - InterfaceIGMP);
- **UnicastRouting** - abstraction to obtain information from the unicast routing table, for RPF checks and also to be notified when there are changes in the unicast routing table (changes to the root interface and also to the RPC).

Most of the protocol was implemented, except for some minor things, that were simply not implemented or not implemented according to the RFC:

- **Some Hello options** - regarding Hello messages, these can include a variable number of options. The RFC specifies 4 options, which are the Hello Hold Time, Generation ID, State Refresh and LAN Prune Delay. Other options are related with PIM-SM and are not defined in RFC3973 [12]. Only the first three referred options were implemented, used respectively, to control the neighbor liveness, a random number used to detect neighbor reboots and if the neighbor supports State

Refresh. The LAN Prune Delay was not implemented since it only used to negotiate, between all neighbors, some values of timers regarding the Prune Override mechanism.

- **Prune Hold Time of “0xFFFF”** - according to RFC3973 [12], if a Prune message is received with a Prune Hold Time of 0xFFFF, this means that the Prune state must not be removed. This feature was not implemented, and if this Hold Time is included in a Prune message, the router would only maintain the Prune state for 0xFFFF seconds.
- **Variable number of trees in Prune/Join/Graft/Graft-Ack** - these messages can include a variable number of trees, in order to notify a router about a change, using the least amount of messages possible. In this implementation, we only include one tree per message. Also, we did not discover any implementation of PIM-DM/PIM-SM that included more than one tree per message.
- **Assert reelection** - according to the specification, an Assert reelection should occur whenever a router that considers itself to be the Assert Winner of a given link hears a multicast data packet in a non-root interface. This should occur, when the non-root interface is in a NoInfo or Winner state, in the Assert state machine. In order to use the native API of Linux, the kernel only notifies the user-level process regarding the reception of data packets in a non-root interface, if that interface is included in the Outgoing Interface List (OIL). For this reason, if two routers consider themselves as Assert Winners, but one of them considers that there is downstream interest (NoInfo state in Downstream Interface state machine), while the other does not consider that there is downstream interest (Pruned state in Downstream Interface state machine), this would not cause a reelection because the interface of the last router would not be part of the OIL. Only if both non-root interfaces forward packets (depending on the Assert, Downstream Interface and Local Membership state machines), this reelection occurs.

In order for the user to interact with the protocol process, we have created some commands in the Run class, in order for the user to verify in which state all trees were at and also to add and remove interfaces from the protocol process. We will describe each implemented command:

- **Start** - this command starts the protocol process if it was not previously running;
- **Stop** - this command stops the protocol process if it was previously running. It will clear all stored state and stop the daemon process;
- **Restart** - this command simply performs the action of Stop followed by Start;
- **List Interfaces** - this command returns to the user a table with information regarding all physical interfaces of the machine, illustrating which interfaces were enabled for IGMP and PIM-DM;
- **List Neighbors** - this command returns to the user a table with information regarding all neighbors that are being monitored by the protocol process (only in interface that enabled PIM-DM). For each neighbor, it shows some options that were exchanged in Hello messages, such as Hello Hold Time and Generation ID. It also shows the “uptime” of this neighbor, i.e. the time that has passed since the last Hello message from it was received;
- **List State** - this command returns to the user a table with information regarding the state maintained by both IGMP and PIM-DM.

Information maintained by IGMP corresponds to all groups that are being maintained and in which state those group are at (in each interface with IGMP enabled).

Information maintained by PIM-DM corresponds to all trees (KernelEntries). Information regarding the state of each interface is also returned, such as the state of all state machines (Downstream Interface, Upstream Interface, Assert and LocalMembership).

- **Multicast Routes** - this command simply returns the output of the command “ip mroute show”. This command returns the Multicast Routing Table of the Linux Kernel;
- **Add Interface** - this command allows the user to enable the PIM-DM protocol in a given interface. By adding an interface with this command, that interface can exchange PIM-DM control messages. This command adds an interface with the StateRefresh option disabled;
- **Add Interface StateRefresh** - the same as above but allows to add an interface having the StateRefresh option enabled;
- **Add Interface IGMP** - this command allows the user to enable the IGMP protocol in a given interface. By adding an interface with this command, the interface can exchange IGMP control messages and monitor membership state regarding all groups;
- **Remove Interface** - this command disables the PIM-DM protocol in a given interface. This interface will no longer exchange PIM-DM control messages;

- **Remove Interface IGMP** - same as above, but for IGMP;
- **Verbose** - return to the user a bunch of logs that are happening in real time, such as state transitions, message reception, ...;
- **Test** - this command was implemented only for testing purposes. By enabling this option, the process will send all logs, in real time, to a server. The server process can process those logs in order to verify if some state transitions are happening.

Now we will describe the most important data structures that were used to maintain state/information regarding PIM-DM.

Regarding the storage of neighbor routers, in the InterfacePIM class, it was used a dictionary, having as key the IP address of the known neighbor and as value a reference to the corresponding Neighbor object. This allows to accelerate the search of a neighbor router ($O(1)$).

The structure used to store KernelEntries, in the Kernel class, corresponds to a dictionary inside of another dictionary. The first dictionary uses as key the source IP address of known trees, which stores a second dictionary. The second dictionary uses as key the group IP address and as a value the corresponding (S,G) KernelEntry. This way of storing entries was chosen in order to accelerate the process of informing entries about RPC changes. When a given subnet suffers an RPC change, it is required to notify all (S,G) entries, that have source S belonging to the suffered subnet. This RPC change can cause an alteration of the interfaces' roles (from root to non-root or vice-versa). By storing entries this way, we do not need to iterate over all (S,G) trees in order to notify all KernelEntries that are part of this change.

Each KernelEntry stores all interfaces associated with the corresponding (S,G) tree in a dictionary. This dictionary has the virtual interface index of a physical interface as its key and the reference to the corresponding TreeInterface as its value. This is used to notify the interface object associated with the (S,G) tree about the reception of control messages. So when an InterfacePIM receives a control message, regarding (S,G) tree, it passes that message to the Kernel, that obtains the KernelEntry responsible for that tree. By having the KernelEntry, the message is transmitted to the corresponding TreeInterface, determined by the virtual interface index of the physical interface. The TreeInterface will act accordingly to the received message type, by executing the events associated with the reception of that message in all state machines.

The PIM-DM implementation uses some threads in order to have some jobs running in the background. Timers are also used and these correspond to threads that run a specific method after a given amount of time (implemented by Python's Timer object). The PIM-DM implementation uses threads in the following classes:

- **Run** - uses a thread to handle the reception of commands from the user;
- **InterfaceIGMP** - described in the implementation of IGMP (section 2.1.2);
- **InterfacePIM** - uses a thread to handle the reception of messages. This allows to run in the background a method that has a while loop waiting for messages over the socket. When a messages is received, it invokes the appropriate method according to the type of the received PIM message.

Also, a timer used to regulate the transmission of Hello messages is used.

- **Neighbor** - timer used for determining the failure of a given PIM-DM neighbor;
- **TreeInterface** - timer used to maintain protocol state associated with any type of interface (root and non-root), like the Assert timer;
- **TreeInterfaceUpstream** - timers used to maintain protocol state associated with root interfaces, such as the Graft Retry timer, Override timer, Prune Limit timer, State Refresh timer and Source Active timer. Also, a thread used for receiving (S,G) data packets, by a socket, is used in order to determine when a directly connected source becomes inactive (used in the StateRefresh state machine);
- **TreeInterfaceDownstream** - timers used to maintain protocol state associated with non-root interfaces, such as the Prune Pending timer and the Prune timer.

After creating an entry in the multicast routing table, associated with a given (S,G) tree, originator routers require to monitor the transmission of data packets by the corresponding source. This is required for the StateRefresh state machine. In order to accomplish this, if the TreeInterfaceUpstream detects that it is directly connected to the source, it will monitor those data packets. This monitoring is accomplished by a socket that only receives (S,G) data packets. This socket is set with a BPF filter, like it was explained in the IGMP implementation. Instead of monitoring all IGMP control messages, this socket will monitor all packets that have S as the source IP address, G as the group IP address and that are not IGMP

control messages (protocol number at the IP layer is different than 2). It is required to explicitly ignore IGMP control messages because those messages can be transmitted having S as the source IP address and G as the group IP address, if source S is reporting or leaving group G. For this reason, we define this restriction in the BPF filter.

In order to implement this protocol, some libraries were used, which will be described now.

In order to obtain information from the unicast routing table, in the `UnicastRouting` class, it was used a library called *PyRoute2* for this purpose. Information regarding the maintenance of Linux's unicast routing table was not detailed in this document, since it is out of scope of this MSc Dissertation. But basically what *PyRoute2* does is communicating with the Linux kernel, interpret messages from it and passing them to the application that is using this library. From those messages, it is possible to understand when there is a change in the unicast routing table, which is being manipulated statically or by a unicast routing protocol.

For IP address calculations, such as determining if a given IP address belongs to a given subnetwork, it was used the *ipaddress* library. This is useful in order to determine which (S,G) entries suffered a RPC change.

Also, for obtaining the IP address associated with a given interface name, it was used *netifaces* library. This is used for all sockets that require to be created, when the user pretends to enable this protocol in a new interface. The user simply specifies the name of a given interface and the implementation gets its corresponding IP address.

In order to display the current state of the multicast routing protocol, it was used *PrettyTable* library. This was used to output all information, of list commands, in a table format.

Regarding the daemon process, this was possible by using the code from here².

This implementation has some known issues/limitations:

- It does not react to IP modification of an interface - This is because the library used to interact with the unicast routing table has some issues regarding this matter. *PyRoute2* basically caches all information from the unicast routing table and then changes its internal state, according to changes in the unicast routing table. This allows to not manually iterate through all entries of the real unicast routing table, instead all verifications are performed by analyzing the cached information of *PyRoute2*. When the IP address of an interface changes (and corresponding subnet), in some circumstances *PyRoute2* still stores the previously subnet as being directly connected and the new subnet is not stored, causing the selection of the root interface to not be performed correctly. While in the GitHub issues of this library it is referred that this issue was already fixed, we had some issues when an interface changed its IP address multiple times.
- Does not react to the shutdown of an interface that is being monitored by the protocol - for the same reason above.

2.3 HPIM-DM Protocol

2.3.1 Protocol Specification

HPIM-DM defines 8 state machines in order to create and maintain neighborhood relationships, maintain neighbor state regarding each (S,G) tree, determine when and which control messages must be transmitted and also to decide through which interfaces data packets must be forwarded.

HPIM-DM state machines are defined in [13]. Briefly, this protocol defines the following state machines:

- **Neighbor/Synchronization state machine** - each interface must maintain neighborhood relationships with other routers that run the same protocol. This requires to synchronize state with new neighbors that are detected on a given link. Each neighbor can be in one of four states, according to the synchronization process, which are the UNKNOWN, MASTER, SLAVE and SYNCED. In the UNKNOWN state, as the name suggests, the neighbor is unknown, meaning that it was not discovered yet. All neighbors start initially in the UNKNOWN state. In the MASTER state, an ongoing synchronization is occurring and the neighbor is considered to be the Master of that process. In the SLAVE state, an ongoing synchronization is occurring and the neighbor is considered to be the

²A simple unix/linux daemon in Python - http://web.archive.org/web/20131017130434/http://www.jejik.com/articles/2007/02/a_simple_unix_linux_daemon_in_python/

Slave of that process. In the SYNCED state, the neighbor has finished the synchronization process correctly.

For this state machine, two timers are required, being called Sync timer and Neighbor Liveness timer. The Sync timer is used to regulate retransmissions of Sync messages in case the neighbor does not respond in due time. The Neighbor Liveness timer is used to detect failures of the neighbor router. The Neighbor Liveness timer is reset whenever an Hello message, from the neighbor router, is received.

- **Maintenance of Neighbor state regarding (S,G) Tree** - a router requires to monitor two types of neighbor state for each (S,G) tree, which are the upstream and interest states. Upstream state is used to determine if a neighbor can logically connect to the source of multicast traffic, while the interest state is used to determine if a neighbor is interested in receiving (S,G) data packets.

This state machine determines when a router should set a neighbor state, by hearing control messages from it.

- **(S,G) Tree maintenance state machine** - this state machine defines the state of a given (S,G) tree, depending on the state of neighbors connected to any interface and also depending whether the source is considered to be active for that same tree.

This state machine is defined differently for originator and non-originator routers, i.e. for routers that are or are not directly connected to the source of multicast traffic. A tree can be in one of three states: ACTIVE, UNSURE and INACTIVE. In the ACTIVE state, there is a logical connection between the root interface of a router and the source of multicast traffic, by Upstream neighbors or by being directly connected to an active source. In case there are Upstream neighbors connected to the root interface, these must respect the feasibility condition in order to avoid maintaining a tree indefinitely due to loops. In the UNSURE state, there is still a logical connection with the source of multicast traffic, via Upstream neighbors, but there is the possibility of a loop being formed or there are only Upstream neighbors connected to non-root interfaces. The UNSURE state is a transient state, that a tree can be at, when it is initially being constructed or when it is being removed or due to changes at the broadcast tree. In the INACTIVE state, there is no logical connection between the router and the source of multicast traffic.

For originator routers, the tree will be in ACTIVE state as long as the source keeps transmitting data packets. For these types of routers, the tree will be in UNSURE state when the source does not transmit data packets for a given amount of time but the router connects to an Upstream neighbor via one of its non-root interfaces. In the INACTIVE state, the router does not hear data packets for a given amount of time, neither it is connected to Upstream neighbors in any of its non-root interfaces. Originator routers require a timer called Source Active timer, used to determine when the source of multicast traffic is no longer considered to be active.

For non-originator routers, the tree state depends only on the existence of Upstream neighbors connected to the router's interfaces. In the ACTIVE state, the router is connected to an Upstream neighbor via its root interface and the feasibility condition holds. In the UNSURE state, the router does not connect to Upstream neighbors via its root interface but connects to Upstream neighbors via one of its non-root interfaces or simply the feasibility condition does not hold due to a loop being formed. In the INACTIVE state, the router does not connect to Upstream neighbors in any of its interfaces.

- **(S,G) Assert state machine** - this state machine defines if a given non-root interface is responsible for forwarding data packets to the link that it is connected to. This state machine defines two states, which are the "ASSERT WINNER" (AW) and "ASSERT LOSER" (AL). In the AW state, the interface is responsible for forwarding data packets. In the AL the interface is not responsible for forwarding data packets. The state of a given interface depends on the existence of Upstream neighbors connected to it and also on the RPC offered by the own router and by the other Upstream neighbors connected to it. The AW is the one that offers the lowest RPC. In case of a tie, the IP address is used to break the tie, being the router with the greatest IP the winner.
- **Forward state machine** - this state machine defines if a given non-root interface should forward multicast data packets. This state machine defines two possible states, which are the FORWARDING and PRUNED. In the FORWARDING state, the non-root interface must forward multicast

data packets. In the PRUNED state, the non-root interface must not forward multicast data packets. This state depends on the Assert state, of the same interface, and also on the interest of directly connected neighbors and hosts. If a non-root interface is placed in a AW state and has one neighbor or host interested in receiving data packets, the interface is placed in a FORWARDING state, otherwise it is placed in a PRUNED state.

- **Router/Root interface interest state machine** - this state machine defines if a router has interest in receiving data packets. There are two possible states in which a router can be at: INTERESTED and NOT INTERESTED. In the INTERESTED, the router is interested in receiving data packets, because it must forward those data packets to a given non-root interface. In the NOT INTERESTED, the router is not interested in receiving data packets.

A router will be in INTERESTED state, if there is at least one non-root interface in a FORWARDING state, otherwise it will be in NOT INTERESTED state.

- **Control message transmission state machine** - this state machine defines when and which types of control messages each interface must transmit, depending on the state transitions of the other state machines and also depending on the internal changes at the unicast routing table. These control messages will be used by the neighbors that receive them, to set state regarding the router that is transmitting them.

The events and corresponding actions depend on the type of the interface (root or non-root) and also whether the interface is or is not directly connected to the source of multicast traffic, i.e. different types of interfaces react differently to the same events.

- **Control message reliability** - this state machine deals with the reliability of transmitted control messages. This state machine deals differently to messages transmitted with destination IP address of type unicast and multicast. In the case of unicast, only one destination must ACK the transmitted message. In the case of multicast, all known neighbors must ACK the transmitted message. If one of those neighbors did not acknowledge, a retransmission mechanism is used, until there is the confirmation from all neighbors.

For this state machine, a timer is required for dealing with the retransmission of control messages, named RetransmissionTimer.

Also, every interface must periodically transmit Hello messages in order to maintain neighborhood relationships. This is achieved by a timer, called Hello timer, used to regulate the transmission periodicity.

Regarding the multicast routing table, the OIL must include all non-root interfaces that are in a FORWARDING state in the Forward state machine. Interfaces that are of type root or that are non-root but in a Pruned state, are not part of the OIL.

2.3.2 Protocol Implementation

The implementation of HPIM-DM is stored in our GitHub repository and can be accessed here [9].

The following diagram (Figure 11) represents the used software architecture. Like IGMP and PIM-DM, this is an object oriented implementation and rectangles represent the used classes, lines with a closed arrowhead represent associations and lines with an open arrowhead represent inheritance.

The same structure of PIM-DM was used in HPIM-DM, only differing the classes that were used to model the new state machines.

Regarding the following classes, these are the same that were used in the implementation of PIM-DM:

- **Run**
- **Main**
- **Interface**
- **InterfaceIGMP**
- **Kernel** - only differs the code used to set information regarding received control messages, to a given KernelEntry, due to changes of API in the used classes of HPIM-DM.
- **LocalMembership, NoInfo and Include**
- **UnicastRouting**

The followings classes are different or new, compared to the implementation of PIM-DM, thus further detailed here:

- **InterfaceProtocol** - similar to InterfacePIM that was used in the implementation of PIM-DM. Just like InterfacePIM, this represents a physical interface that has enabled the routing protocol. It simply sends and receives control messages via a raw socket. This protocol by still not being a standard, it has no Protocol Number associated with it, so it was opted to use the PIM Protocol Number (103) in this implementation. The used socket will receive all packets that have 103 as the Protocol number at the IP Layer. This class will only process control messages and store received state in the corresponding neighbor's structure. This object is also associated with multiple ReliableTransmission objects that are responsible for guaranteeing reliable transmission of control messages. Besides that, it will also periodically transmit Hello messages in order to form and maintain neighborhood relationships with other routers connected to the same link.

Like InterfacePIM, before creating an InterfaceProtocol it is required to create a virtual interface associated with the physical interface (in the Kernel module);

- **ReliableTransmission** - responsible for guaranteeing the reliable transmission of control messages. Each (S,G) tree will have its own ReliableTransmission object, having information regarding current control messages that are being transmitted. Each object will monitor the received acknowledges. For control messages destined to all neighbors (destination IP is multicast), this object will only consider the control message to have been reliably transmitted when all known neighbors acknowledge that message. For control messages destined to a single neighbor (destination IP is unicast), this object will only consider the message to have been reliably transmitted when its destination acknowledges it. This object will also implement a retransmission mechanism, in case the message is not acknowledged by at least one neighbor that is supposed to;
- **Neighbor** - this object represents a known neighbor detected by a given InterfaceProtocol. This object will be responsible for monitoring the corresponding neighbor in order to detect when it has failed/rebooted. Also, associated with each neighbor, a state regarding the synchronization is stored (UNKNOWN, MASTER, SLAVE or SYNCED).

Beside monitoring the liveness of a neighbor, this object will also store all its corresponding state received in control messages, such as if it is UPSTREAM or NOT UPSTREAM and INTERESTED or NOT INTERESTED for each known (S,G) tree. It will also be responsible for storing the last received sequence number, regarding each tree, in order to not interpret messages that arrive out of order (stored state is fresher than the received one);

- **State, Unknown, Master, Slave and Synced** - implementation of the synchronization state machines;
- **KernelEntry, KernelEntryOriginator, KernelEntryNonOriginator** - represent an entry in the multicast routing table. It is required to distinguish between entries in which the router is considered to be originator and non-originator, since the calculation of a tree state is different for both cases.

Each KernelEntry will be associated with a TreeState, representing the state in which a tree is at, and associated with multiple TreeInterfaces. One of those interfaces must be selected to be the root interface, which is represented by objects TreeInterfaceRootOriginator and TreeInterfaceRootNonOriginator. The remaining interfaces will be non-root interfaces, represented by TreeInterfaceNonRoot object. It is required to distinguish between those types of interfaces because the reaction to the same events is different for root and non-root interfaces;

- **TreeState, Active, UNSURE and INACTIVE** - represent the states in which a given tree can be at. A tree will be in a state according to the existence of Upstream neighbors, reception of data packets and if the router is originator or non-originator;
- **TreeInterfaceRootOriginator** - represents an interface considered to be root that is directly connected to the source of multicast traffic. This interface will monitor the reception of multicast data packets regarding that tree, in order to define in which state a tree will be at.
- **TreeInterfaceRootNonOriginator** - represents an interface considered to be root that is not directly connected to the source of multicast traffic. This class implements the state machine of a root interface of a non-originator router in order to determine when control messages should be transmitted by this interface;

- **TreeInterfaceNonRoot** - represents an interface considered to be non-root. This class implements the state machine of a non-root interface in order to determine when control messages should be transmitted by it;
- **AssertState, Winner and Loser** - like PIM-DM, HPIM-DM also has an Assert state machine. The only difference corresponds to the number of states, events and corresponding actions to be taken by HPIM-DM;
- **DownstreamInterestState, DI and NDI** - define for a given interface if there is downstream interest (DOWNSTREAM INTERESTED or NOT DOWNSTREAM INTERESTED).

Just like the PIM-DM implementation, the Run class also allows the user to interact with HPIM-DM by using commands. We will describe each implemented command:

- **Start** - same as PIM-DM implementation;
- **Stop** - same as PIM-DM implementation;
- **Restart** - same as PIM-DM implementation;
- **List Interfaces** - same as PIM-DM implementation;
- **List Neighbors** - similar to PIM-DM implementation. The only difference corresponds to the additional information that is presented for each known neighbor. A neighbor listed with this command will include information regarding its state (according to the Neighbor state machine), its Hello Hold Time, the “uptime” and the sequence numbers exchanged during the synchronization process (neighbor’s BootTime and SnapshotSN);
- **List State** - similar to PIM-DM implementation. The only difference corresponds to the presented information regarding each tree, due to the different state machines of both protocols;
- **List Neighbor State** - this command returns all states, regarding Upstream and Interest of each known neighbor, for each tree. For neighbors that are considered to be UPSTREAM, it is also shown their RPC to the source subnet. For neighbors that are considered to be NOT UPSTREAM it is only shown information regarding their interest (INTERESTED or NOT INTERESTED);
- **List Sequence Numbers** - since sequence numbers are very important in this protocol, we have implemented a command that retrieves all stored sequence numbers. This includes the sequence number of all interfaces (BootTime and last transmitted sequence number). For each neighbor, it is shown the BootTime, SnapshotSN, CheckpointSN and also all sequence numbers stored for each tree (last received control message regarding each tree);
- **Multicast Routes** - same as PIM-DM implementation;
- **Flood Initial Data Packets** - this command allows the user to control the default initial interest information regarding all NOT UPSTREAM neighbors. This can control if initial data packets should be forwarded even if no neighbor has transmitted an Interest message yet, which results in a flooding behavior for the first data packets. If enabled, this allows first data packets to not be lost, otherwise they would be lost. This option is enabled by omission;
- **Add Interface** - same as PIM-DM implementation, but enables HPIM-DM in a given interface;
- **Add Interface IGMP** - same as PIM-DM implementation;
- **Remove Interface** - same as PIM-DM implementation, but disables HPIM-DM in a given interface;
- **Remove Interface IGMP** - same as PIM-DM implementation;
- **Verbose** - same as PIM-DM implementation;
- **Test** - same as PIM-DM implementation.

Now we will briefly describe all data structures used to store information in each class.

Like PIM-DM implementation, all entries (KernelEntry/KernelEntryOriginator/KernelEntryNonOriginator) are stored in the Kernel class the same way, i.e. by using a dictionary inside of another dictionary. The key of the first dictionary corresponds to the source IP address, being the value a reference to a second dictionary. This last dictionary uses the group IP address as key and the corresponding KernelEntry as value. Also, the Kernel class stores all interfaces using a dictionary, having the virtual interface index as key and the corresponding Interface object as value.

Also, like PIM-DM, each KernelEntry/KernelEntryOriginator/KernelEntryNonOriginator stores information regarding TreeInterfaces in a dictionary, being the virtual interface index used as key and the corresponding reference to that object used as value. Also, each KernelEntry stores the interest state of each interface (if some neighbor is interested) and also which UPSTREAM neighbor offers the best RPC, to the source subnet, of each interface. This information is obtained by performing a calculation, based

on the stored state of all neighbors connected to each interface. Both states, are stored in dictionaries, having as key the virtual interface index and as value the corresponding state.

Regarding the storage of neighbor routers, each `InterfaceProtocol` stores known neighbors in a dictionary. The neighbor IP address is used as key of that dictionary, being the value a reference to the corresponding `Neighbor` object.

Each `Neighbor` object stores everything related with that neighbor:

- It stores the last received sequence number from this neighbor, of each (S,G) tree, in a dictionary. The tuple (source IP address, group IP address) is used as key and the last received sequence number is used as value.
- It stores the upstream state of this neighbor, for each (S,G) tree, in a dictionary. The tuple (source IP address, group IP address) is used as key and the RPC state is used as value. This RPC state is obtained by the reception of `IamUpstream` messages, which includes the metric preference and metric of the neighbor router to the source of multicast traffic. If the value associated with a given tree is not “null”/“None”, this means that the neighbor router considers itself as UPSTREAM, otherwise the neighbor router is NOT UPSTREAM, for that same tree.
- It stores the interest state of this neighbor, for each (S,G) tree, in a dictionary. The tuple (source IP address, group IP address) is used as key and a boolean indicating the neighbor’s interest is used as value. This state is obtained by the reception of `Interest` and `NoInterest` control messages. To set the interest of a neighbor router, this means that the neighbor is NOT UPSTREAM, so the interest and upstream dictionaries must be manipulated carefully.
- It also stores a list, used for synchronization purposes, that contains the router’s own snapshot of all (S,G) entries that must be included on `Sync` messages destined to the neighbor router.

Each `InterfaceProtocol`, besides storing the neighbors, it also stores information regarding messages that are currently being reliably transmitted by the interface. This information is stored in a dictionary, being the tuple (source IP address, group IP address) used as key and the reference to the corresponding `ReliableTransmission` object used as value.

This implementation uses some threads in order to have some jobs running in the background. Some of those threads have already been referred in the PIM-DM implementation (section 2.2.2). Besides some of those threads, this implementation also uses threads in the following classes:

- **InterfaceProtocol** - timer used for the periodic transmission of Hello messages.
- **Neighbor** - timer used for determining a failure of a given neighbor and also timer used for the retransmission of `Sync` messages.
- **ReliableTransmission** - timer used to control the retransmission of control messages that have not been acknowledged yet by at least one neighbor that was supposed to, in due time.
- **TreeInterfaceRootOriginator** - thread used to receive data packets from a directly connected source. Also, it uses a timer to determine when the source becomes inactive. Both are be used to determine the tree state of an originator router.

The libraries that were used and described in the implementation of PIM-DM (section 2.2.2) are the same ones used in the implementation of this protocol.

This implementation is also affected by the issues referred in the PIM-DM implementation.

In order to perform tests, a Wireshark dissector was implemented to fully dissect exchanged control messages.

References

- [1] Joachim Nilsson. SMCRoute - A static multicast routing daemon. <https://github.com/troglobit/smcroute>, 2018. Online; accessed 20 July 2018.
- [2] Joachim Nilsson. An implementation of DVMRP for Linux & BSD. <https://github.com/troglobit/mrouted>, 2017. Online; accessed 20 July 2018.
- [3] Joachim Nilsson. PIM-SM/SSM Multicast Routing for UNIX. <https://github.com/troglobit/pimd>, 2018. Online; accessed 20 July 2018.

- [4] Pavlin Radoslavov. *multicast – Multicast Routing*, September 2003.
- [5] Michael Kerrisk. *IP(7) Linux Programmer’s Manual*, February 2018.
- [6] William C. Fenner. Internet group management protocol, version 2. RFC 2236, RFC Editor, November 1997.
- [7] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. Internet group management protocol, version 3. RFC 3376, RFC Editor, October 2002.
- [8] Pedro Oliveira. PIM-DM implementation. https://github.com/pedrofran12/pim_dm, 2018. Online; accessed 16 September 2018.
- [9] Pedro Oliveira. HPIM-DM implementation. https://github.com/pedrofran12/hpim_dm, 2018. Online; accessed 16 September 2018.
- [10] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. Linux Socket Filtering aka Berkeley Packet Filter (BPF). Online; accessed 07 October 2018.
- [11] Kurt Windisch. Protocol Independent Multicast - Dense Mode (PIM-DM): Routing Daemon Software. Online; accessed 1 November 2017.
- [12] A. Adams, J. Nicholas, and W. Siadak. Protocol Independent Multicast - Dense Mode (PIM-DM). RFC 3973, RFC Editor, January 2005.
- [13] Pedro Oliveira. HPIM-DM state machines. Internal Report, Instituto Superior Técnico, November 2018. https://github.com/pedrofran12/hpim_dm/tree/master/docs/HPIMStateMachines.pdf.