Robust Multicast Routing Protocol

Pedro Francisco Carmelo de Oliveira

Thesis to obtain the Master of Science Degree in

Telecommunications and Informatics Engineering

Supervisor: Prof. Rui Jorge Morais Tomaz Valadas

Examination Committee

Chairperson: Prof. Ricardo Jorge Fernandes Chaves
Supervisor: Prof. Rui Jorge Morais Tomaz Valadas
Member of the Committee: Prof. Amaro Fernandes de Sousa

October 2018

Acknowledgments

I would like to start by thanking my supervisor, Prof. Rui Valadas, for proposing me this Dissertation theme, which I truly consider to be very interesting and in which it was possible to put in practice my technical knowledge and creativity. Also, I would like to thank for his valuable support, motivation and availability throughout the development of this work. Additionally, a special thanks to Alexandre Silva also for his important support and contribution to this work.

I would also like to thank Internet Engineering Task Force (IETF) for their effort in standardizing multicast routing protocols, allowing this technology to be open and available to everyone. Also, to Instituto de Telecomunicações (IT) for their research in the area of telecommunications.

A special thanks to Instituto Superior Técnico (IST) for providing me a solid academic background, allowing me to accomplish this work successfully.

Last, but not least, a special thanks to my family for their support, motivation and encouragement during my whole academic life, which were essential to make this possible.

Resumo

Protocolos de encaminhamento multicast são uma solução eficiente para comunicações em grupo, ao fornecer uma ligação lógica entre fontes e recetores multicast, através de uma ou mais árvores de distribuição, que minimizam o consumo de largura de banda e evitam o envio de tráfego multicast para zonas da rede onde não existe interesse em receber este tráfego. Vários protocolos de encaminhamento multicast foram propostos para encaminhamento intra-domínio, mas os protocolos PIM (PIM-DM e PIM-SM) são atualmente os preferidos, estando implementados na maioria dos equipamentos de rede.

Os protocolos PIM são protocolos do tipo soft-state que apresentam várias limitações que usualmente são corrigidas através da reconstrução periódica das árvores de distribuição multicast. Estas limitações prejudicam o uso destes protocolos em redes com elevada largura de banda.

Este trabalho propõe uma versão hard-state do PIM-DM. Ao contrário do PIM-DM, mensagens de controlo são entregues de forma fiável e sequenciadas de acordo com a sua ordem de transmissão, e as árvores de distribuição multicast são formadas exclusivamente através de mensagens de controlo. Além disso, cada router monitoriza todos os seus vizinhos que podem potencialmente ser seus pais na árvore (i.e. que possam encaminhar dados multicast), uma característica que permite manter a árvore correta mesmo na presença de reconfigurações e falhas na rede. Adicionalmente, um processo de sincronização assegura que novos routers que se liguem à rede recebam de imediato o estado necessário para se conectarem a árvores já existentes e começar a receber dados multicast. Finalmente, várias otimizações foram realizadas de maneira a reduzir a quantidade de estado armazenado e várias características foram incluídas de maneira a evitar ataques de segurança.

A correção do protocolo foi assegurada usando argumentação lógica e model checking, através da linguagem Promela e a ferramenta SPIN. O protocolo foi implementado em Python e foi extensivamente testado num ambiente de rede emulado.

Palavras-chave: protocolo de encaminhamento multicast, árvores de distribuição multicast, Dense Mode, hard-state, HPIM-DM, PIM-DM.

Abstract

Multicast routing protocols are an efficient solution for group communications, by providing logical connections between multicast sources and receivers, usually through one or more distribution trees, that minimize the bandwidth consumption and avoid delivering multicast traffic to network zones not interested in receiving it. Several multicast routing protocols were proposed for intra-domain routing, but the PIM protocols (PIM-DM and PIM-SM) are currently the preferred ones, being implemented by most vendors.

PIM protocols are soft-state protocols that have several limitations, usually overcome through the periodic reconstruction of the multicast distribution trees. These limitations impair the use of these protocols in high-speed networks.

This work proposes a hard-state version of PIM-DM. Unlike PIM-DM, the control messages are reliably delivered and sequenced according to their transmission order, and the construction of the multicast distribution tree is exclusively performed through control messages. Moreover, each router keeps track of all neighbors that can potentially become parents on the tree (i.e. to act as feeders of multicast data), a novel feature that allows keeping the tree correct even in the presence of network reconfigurations and failures. In addition, a synchronization process ensures that a new router attaching the network receives immediately the state information required to connect to existing trees and start receiving multicast data. Finally, several optimizations were performed to minimize the amount of stored state and several features were included to avoid security attacks.

The correctness of the protocol was assessed using logical reasoning and model checking, through the Promela language and the SPIN tool. The protocol was implemented in Python and was extensively tested in a network emulated environment.

Keywords: multicast routing protocol, multicast distribution trees, Dense Mode, hard-state, HPIM-DM, PIM-DM.

Contents

List of Tables

List of Figures

Nomenclature

AL Assert Loser

AW Assert Winner

BPF Berkeley Packet Filter

DVMRP Distance Vector Multicast Routing Protocol

HPIM-DM Hard-state Protocol Independent Multicast - Dense Mode

IETF Internet Engineering Task Force

IGMP Internet Group Management Protocol

IP Internet Protocol

MLD Multicast Listener Discovery

MOSPF Multicast Open Shortest Path First

OIL Outgoing Interface List

OSPF Open Shortest Path First

PIM Protocol Independent Multicast

PIM-DM Protocol Independent Multicast - Dense Mode

PIM-SM Protocol Independent Multicast - Sparse Mode

RFC Request For Comments

RIP Routing Information Protocol

RP Rendezvous Point

RPC Root Path Cost

RPF Reverse Path Forwarding

TTL Time To Live

UDP User Datagram Protocol

Chapter 1

Introduction

IP Multicast is a communication technique that allows a host to communicate with a group of hosts. This group can be on the subnetwork of the source or can span different subnetworks, being identified by a multicast IP address.

In IP multicast routing, routers create a logical structure, usually one or more distribution trees, for forwarding traffic from the multicast source to all interested receivers. For this to be possible, routers run a multicast routing protocol to create/remove trees and to trim existing trees, to remove redundant paths, and to attach new interested receivers and sources or remove no longer interested receivers and inactive sources.

Several IP multicast routing protocols have been proposed over the years. These protocols can be categorized as belonging to one of three categories, based on their working principles: Dense Mode, Sparse Mode and Link-State [1]. In Dense Mode protocols, such as PIM-DM [2] and DVMRP [3], routers first build a broadcast tree connecting the source to all routers, which is then pruned to remove redundant paths and paths that do not lead to interested receivers. In Sparse Mode protocols, such as PIM-SM [4], routers build a tree that is shared by the sources that transmit data to the same group, connecting all interested receivers. In Link-State protocols, such as MOSPF [5], a router has a global view of the location of sources and interested receivers.

PIM protocols (PIM-DM and PIM-SM) are currently the preferred protocols for intra-domain multicast routing, and are implemented by most vendors. These protocols keep being updated by the PIM working group of IETF [6]. However, they suffer from several limitations that compromise their performance. In this work, we have concentrated on PIM-DM, and specified and developed an hard-state version of this protocol that overcomes its limitations, called Hard-state PIM-DM (HPIM-DM). The correctness of the protocol was verified through logical reasoning and model checking, using the Promela [7] language and the SPIN tool [8]. The resulting specification was then implemented in Python.

Additionally, we have also implemented two other protocols in Python: IGMPv2 [9] and PIM-DM. IGMPv2 allows multicast receivers to signal their interest in multicast traffic. PIM-DM was implemented in order to better understand how this protocol works, and also because there was no open implementation of its most recent specification. By implementing PIM-DM we also contributed to the open source and

scientific communities.

All implemented protocols were tested in a network emulated environment, Netkit-NG [10], in order to verify if they were according to the specifications.

1.1 Objectives

The goal of this work is to specify, implement and test a new multicast routing protocol, named HPIM-DM, that overcomes the limitations of the PIM-DM protocol.

1.2 Contributions

- Implementation of IGMPv2 and PIM-DM in Python;
- Specification, implementation and testing of a new multicast routing protocol named HPIM-DM.

1.3 Document Structure

This document is divided into the following chapters:

1. **Introduction** - Motivation and objectives of this work;
2. **Overview** - Description of multicast routing principles and protocols. DVMRP, PIM-DM, PIM-SM and MOSPF are analyzed in this section. Additionally we list some identified problems in Dense Mode protocols, that motivated the development of HPIM-DM;
3. **HPIM-DM** - Specification of our protocol, with a detailed description of how it works. Additionally we include some security considerations;
4. **HPIM-DM correctness** - Proof of HPIM-DM correctness;
5. **Implementation** - Description of all implementations performed in this MSc Dissertation;
6. **Tests** - Description of tests performed to our implementations;
7. **Conclusions** - Final remarks about this work.

In appendix we include:

- **Appendix A** - Details about an alternative solution for the maintenance of interest state in HPIM-DM;
- **Appendix B** - HPIM-DM state machines;
- **Appendix C** - Detailed description of the models, written in Promela, used to verify the correctness of our protocol specification;
- **Appendix D** - Detailed description of all implementations performed in this work (IGMPv2, PIM-DM and HPIM-DM), along with a brief description regarding the specification of these protocols. Also, information about the Linux API is included;
- **Appendix E** - Detailed description of all tests to our implementations and corresponding results, performed in a network emulated environment (Netkit-NG [10]);

Chapter 2

Overview of multicast routing

In this chapter we start by describing the principles of multicast routing protocols. Then we describe the principles of each category of protocols, being Dense Mode, Sparse Mode and Link-State. For each category we also describe some existing protocols.

Regarding the Dense Mode protocols, we describe the issues present in those protocols that motivated the development of HPIM-DM.

2.1 Multicast Routing Principles

Multicast traffic is characterized by a source that sends packets destined to a group of receivers. This group is identified by its multicast IP address, also called group address, and all its receivers are called members of this group. This address consists on a usual IPv4/IPv6 address but with reserved prefix (224.0.0.0/4 in IPv4 [11] and FF00::/8 in IPv6 [12]). In order to have receivers of multicast traffic scattered by different subnets, it is required a multicast routing protocol. In the literature and in this document, for a certain multicast traffic, the notation S is used to refer its source IP address and G is used to refer its group address.

2.1.1 Multicast Distribution Trees

When a host wants to send a multicast packet, it simply sends it (with the destination IP address set to the group's address). For each multicast traffic, from a given source and destined to a given group, a multicast routing protocol will define a logical structure representing the paths through which multicast traffic should be forwarded. This structure is called multicast distribution tree and should only include paths that can reach its members. A multicast distribution tree has to adapt dynamically to the entrance and exit of its members and to possible failures of routers and links. This means that routers and links can be added or removed from the tree in order to deal with the problems previously mentioned.

The routers directly connected to the source are called *originator* routers. Note that there can be more than one originator for a given tree. The *parent* of a router is the router immediately above it in the tree from which multicast data must be received. The *child* of a router is a router immediately bellow it

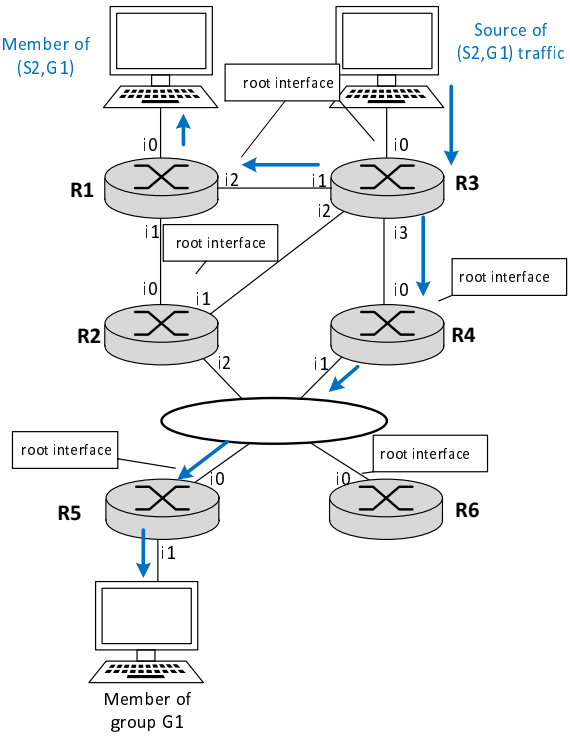

Figure 2.1: Topology with (S2,G1) tree.

in the tree to which multicast data must be sent. Originators have no parent. The *leaf* routers of the tree are the routers that have no children.

Figure 2.1 illustrates a tree with its source being the host that is at the top right. This tree has R3 as the originator. This router is parent of R1 and R4, which is the same as saying that these routers are children of R3. Router R5 is a leaf router by not being connected to any routers below it (has no children).

All multicast routing protocols perform an incoming interface check as the primary mechanism to determine whether to forward or drop an incoming multicast packet. This verification is called Reverse Path Forward (RPF) [13, 14], since the forwarding is based whether the packet reached the interface that is on the "reverse path" back to the root of the tree. How a router defines which interface is on the "reverse path" is protocol dependent.

The interface of a router that receives multicast traffic from the parent router is called the *root* interface; the remaining interfaces are called *non-root* interfaces. The corresponding cost of a router to the root of the tree is called *Root Path Cost* (RPC), being this through the root interface of a router. Some non-root interfaces may not be allowed to transmit multicast traffic; this will happen whenever they have no downstream routers or receivers interested in receiving multicast traffic. Thus, a non-root interface can be in one of two states: FORWARDING, when allowed to transmit multicast data, or PRUNED, when not allowed to transmit multicast data. Under stable conditions, a router receives multicast data on its root interface, and retransmits it through its FORWARDING non-root interfaces.

For a certain tree, all non-root interfaces that are in a FORWARDING state, will be part of the Outgoing Interface List (OIL). Multicast packets entering through the root interface, will be forwarded through

the OIL.

An interface is placed in a PRUNED state, when that interface does not reach members of a certain group or to simply remove redundant paths from the tree. The state of an interface can be changed dynamically, for example with the entrance/exit of members.

Regarding Figure 2.1, consider that routers select the root interface of a tree by having in consideration the number of hops between the own router and the source of multicast traffic. This way, R3 considers i0 as its root interface by being directly connected to the source, R4 considers i0 as its root interface by being the one nearest to the source (one hop of distance). The selection at other routers is performed in a similar way. These interfaces are the ones that must receive data packets in order for the router to forward them through non-root interfaces.

For the operation of layer-3 multicast routing protocols the links between routers can be abstracted in two types: point-to-point and shared links. *Point-to-point* links connect two and only two routers. *Shared links* can attach two or more routers, and can abstract relatively complex layer-2 networks. Based on the type of a link, the protocol must deal with the existence of redundant paths, i.e. multiple routers able to forward data packets to the same link, and also deal with the interest of multiple routers, i.e. not all routers connected to a link may be interested in receiving data packets. Having this in consideration, protocols must be able to determine when a non-root interface should forward data packets, being placed in FORWARDING or PRUNED state.

A multicast routing protocol will store information about each tree in the router's multicast routing table. Each entry/tree will have information regarding the root interface and the OIL. Every change in the multicast routing table will reflect the current state of the tree. By storing this information, routers can speedup the routing process, since this information is cached. The tables below, show as an example, the content of the multicast routing tables of routers R1 and R3 of Figure 2.1.

(S,G)	Root Interface	OIL
(S2, G1)	i2	i0

Table 2.1: Multicast routing table of R1

(S,G)	Root Interface	OIL
(S2, G1)	i0	i1, i3

Table 2.2: Multicast routing table of R3

Multicast distribution trees can be of two types: source trees and shared trees. The trees that routers form are protocol dependent.

- In source trees, the tree is rooted at the source of the multicast traffic (source S) and spans all members of group G, which belong to the leafs of the tree. This tree is built according to the "location" of source S. Notice that each tree is characterized by source S and group G, which means that the tree is not shared for traffic sent from a different source and destined to a different group. The notation (S,G) is used to refer this type of trees. Figure 2.1 shows a source tree, since the represented tree (dashed blue arrows) has at its root the source of the traffic. Source trees are usually formed by Dense Mode and Link-State protocols, which will be explained in the following sections. Protocols like DVMRP, MOSPF and PIM-DM form source trees.

- In shared trees, the tree is not rooted at the source of the multicast traffic, like source trees, but

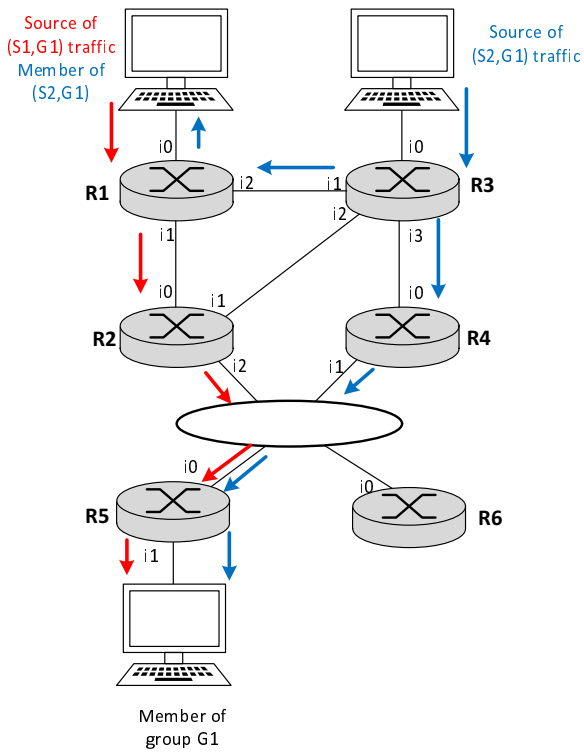

Figure 2.2: Topology with two trees.

instead is rooted at some router at the core of the network, and spans all members of group G. The router that is at the root of the tree is often called rendezvous point (RP). In this type of structure, multicast traffic is forwarded from its source to the RP, and this router forwards multicast traffic down the shared tree to all its members. Routers that belong to the shared tree forward multicast traffic irrespective of its source. The notation (*,G) is used to refer this type of trees. Protocols that form shared trees are called Sparse Mode protocols, which will be explained in following sections. PIM-SM is an example of a protocol that forms shared trees.

Note that both types of trees should only include paths that connect interested receivers.

Multiple trees can coexist in the same network. Consider Figure 2.2, with red arrows representing the paths of (S1,G1) data packets, and blue arrows representing the paths of (S2,G1) data packets. In this case, each tree has its own root interface, meaning that different trees do not share the same types of interfaces. Also, the state of non-root interfaces is not shared for different trees. A multicast routing protocol has the goal of maintaining these independent trees, in order to forward data packets to interested receivers only.

2.1.2 Membership Information

Hosts can express their interest in receiving multicast traffic to their directly connected router(s) by using a protocol independent from the multicast routing protocol. The Internet Group Management Protocol (IGMP) is used for IPv4 networks and Multicast Listener Discovery (MLD) is used for IPv6 networks. These protocols run between the multicast receivers and the routers to which they are directly

connected to. There are two possibilities: IGMPv2 [9] and MLDv1 [15] are only capable of signaling interest on a multicast group; IGMPv3 [16] and MLDv2 [17] can signal interest on a specific source from a multicast group.

By using these protocols, each router will know if it is connected to interested receivers. This information needs to be propagated to other routers, in order to inform them, where are members of multicast traffic.

2.2 Dense Mode multicast routing protocols

2.2.1 Principles

Dense mode protocols build source trees, delivering traffic from each active multicast source to a set of interested receivers, belonging to some multicast group. These trees are only maintained as long as the multicast source is active, i.e. keeps sending multicast data. This contrasts with unicast routing protocols, where the routing structure is maintained independently of the presence of unicast data.

In Dense Mode multicast routing protocols the tree is defined such that the router interfaces that belong to the tree (i.e. the root interface and the non-root interfaces in FORWARDING state) are the ones in the shortest paths from the multicast receivers to the multicast source. This type of tree is sometimes called *reverse shortest path* tree, because the tree is defined in a direction opposite to the flow of data. The RPC at a router corresponds to the cost of this shortest path (from its root interface).

Dense Mode multicast routing protocols start by building a broadcast tree, i.e. a tree that spans all routers. This is usually performed by flooding data packets throughout the whole network using RPF.

A critical issue in Dense Mode multicast routing protocols is to ensure that there are no redundant transmissions of multicast data, i.e. that data is not transmitted on a link without need. There are two cases: On point-to-point links, data must not be transmitted through a non-root interface if the interface on the other side of the link is also a non-root interface. On shared links, data must be transmitted by only one of the non-root interfaces attached to the link. The protocol that selects the non-root forwarding interface of shared links is called the *Assert* protocol; this interface is called the *Assert Winner* (AW), and the remaining non-root interfaces are called *Assert Losers* (AL).

In Figure 2.1 there is a point-to-point link that connects R1 to R2. Since the interfaces of both routers are non-root type, these must be pruned from the tree, i.e. both interfaces must be placed in a PRUNED state. Also in this figure, R2 and R4 are connected to each other by non-root interfaces in a shared link. In this case, one of these routers must be selected as the AW. Consider that R4 is the one elected as AW, this means that R2 is AL. For this reason interface i2 of R2 is placed in a PRUNED state.

After removing redundant paths, we have a broadcast tree that still spans all routers. However, not all routers may be interested in receiving multicast traffic. This happens when a router has no multicast receivers nor downstream routers interested in receiving multicast data. A router that loses interest in a tree sends through its root interface a Prune message, which places the receiving non-root interface in PRUNED state, if no other router connected to the same link is interested. The message is further

transmitted upwards the tree through the shortest path until it finds an interested router or an originator, at which point it is discarded. To renew interest in a tree, a router sends a Graft message through its root interface, which places the receiving non-root interface in FORWARDING state and, as in the pruning case, the message is transmitted upwards the tree through the shortest path until it finds an interested router.

Figure 2.1 represents the paths that multicast data traverses, as blue arrows. In this figure the tree was already pruned, since not all links forward multicast packets. In this case, R2 has already notified R3 that is not interested in receiving data, through a Prune message. In case this router becomes interested in receiving data packets, it would transmit a Graft message to R3, in order to be part of the tree again. The exchange of these messages will be illustrated in the following sections.

There are two main protocols implementing Dense Mode multicast routing: PIM-DM and DVMRP. We will describe them in the next two sections.

2.2.2 PIM-DM

PIM-DM [2] relies on a unicast routing protocol to determine the shortest paths from the routers to the multicast source. This means that the Reverse Path Forwarding technique takes into consideration the unicast routing table to select the root interface.

The construction of the tree is triggered by the first multicast data packet transmitted by the source. The packet is then transmitted using the Reverse Path Forwarding (RPF) technique. This technique allows the *controlled* broadcast of packets from one router to all other routers, i.e. without indefinite circulation. Under this technique, an originator router transmits packets through all its non-root interfaces. Other routers retransmit a packet received on a root interface through all their non-root interfaces, and discard packets received on non-root interfaces. This behavior causes an initial flood of multicast data packets, allowing all routers to know about the existence of the tree. This behavior is illustrated in Figure 2.3 for (S2,G1) tree.

Initially we have all non-root interfaces forwarding data packets. The broadcast tree must be pruned in order to remove redundant paths and also to remove paths that do not lead to interested receivers.

To eliminate redundant non-root interfaces (building, in this way, a broadcast distribution tree), there are specific techniques for point-to-point and shared links. On point-to-point links, when a non-root interface receives a multicast data packet (which it should not), it replies with a Prune message to the sending interface, which places that interface in PRUNED state. On shared links, when a non-root interface receives a multicast data packet, it replies with an Assert message, which triggers the election of an AW. Assert messages carry the RPC of the sending router; the interface declaring the lowest RPC becomes the AW, and all remaining interfaces become ALs. Interfaces in AL state are placed in a PRUNED state.

In Figure 2.3 we have redundant paths. The link that connects R1 to R2 is a point-to-point link and both routers are connected to it by non-root interfaces. The transmission of data packets from both routers triggers the transmission of Prune messages, causing both routers to place their interfaces in a

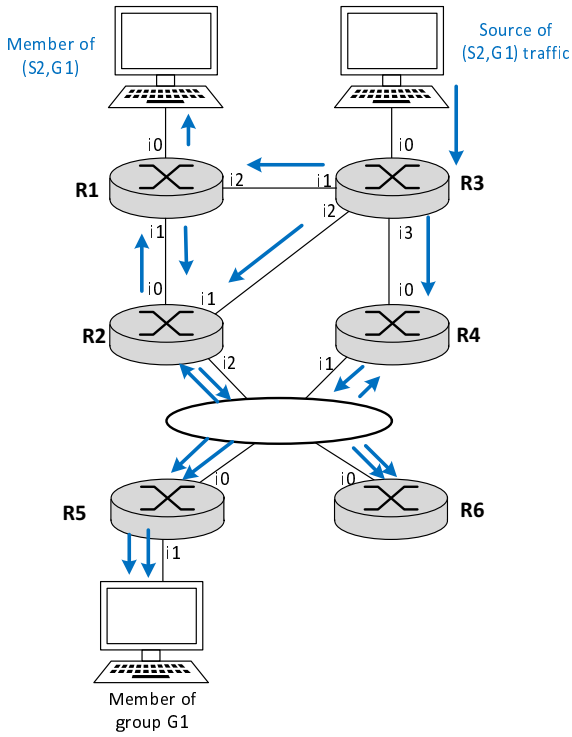

Figure 2.3: Initial flood of multicast data packets.

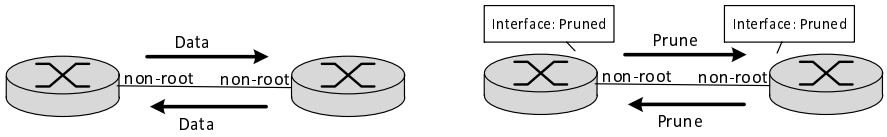

Figure 2.4: Remove redundant paths in point-to-point links.

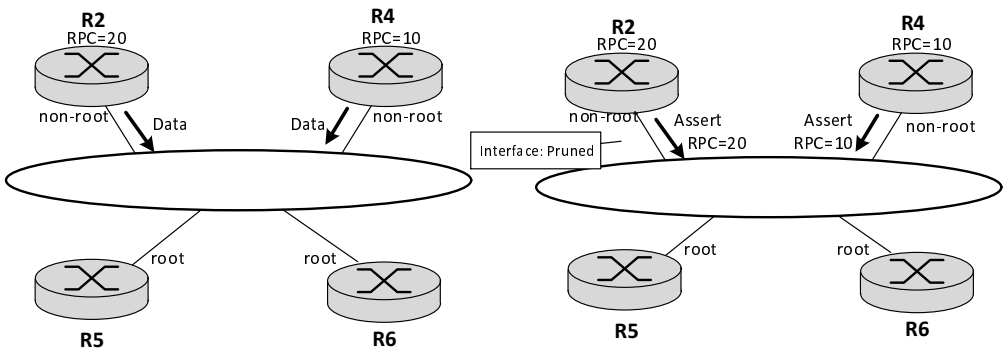

Figure 2.5: Remove redundant paths in shared links.

PRUNED state. This is illustrated in Figure 2.4.

In Figure 2.3, the shared link that connects R2, R4, R5 and R6, by having two routers connected by non-root interfaces to this link, R2 and R4, they initially forward data packets. By having each non-root receiving multicast data packets originated from its neighbor, both routers multicast Assert messages. R4 by transmitting an Assert message with a better RPC, is elected as AW, while R2 considers itself as AL. This is illustrated in Figure 2.5.

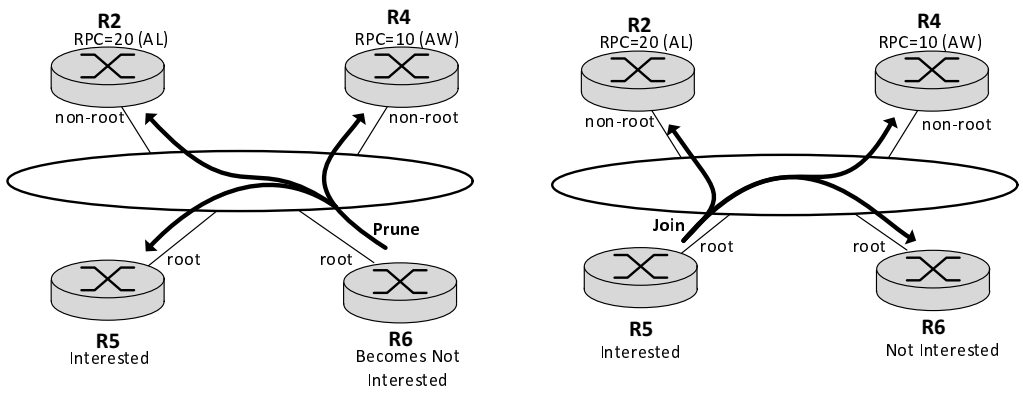

Figure 2.6: PIM Prune Override mechanism.

After removing all redundant non-root interfaces we have still a broadcast tree connecting all routers. Since not all routers may be interested in receiving data packets, these must be removed from it. This transforms the initial broadcast tree into a multicast distribution tree, that only spans interested receivers.

In order for a router to manifest its disinterest in a given tree, it transmits a Prune message as soon as it hears data packets from it. This message is rate-limited in order to not transmit it every time data packets arrive, since there may be other routers interested in receiving data packets. On point-to-point links, the reception of this message causes the interface to be placed in a PRUNED state. On shared links, the Prune Override mechanism is used, in order to control the forwarding behavior at the non-root interface of the AW. A Prune message is multicasted, so that all routers hear this message. If an interested router hears this message, it multicasts a Join message that has the goal of overriding the Prune. The AW by hearing a Prune message, waits some time in order to give opportunity to other routers to override the Prune with a Join. If the AW does not hear a Join, it places its interface in a PRUNED state, otherwise it places its interface in a FORWARDING state. Figure 2.6 illustrates the Prune Override mechanism.

In Figure 2.6, we have the same shared link that connects R2, R4, R5 and R6, that has already elected R4 as the AW. The first data packets by reaching a non interested router, R6, causes it to multicast a Prune message. This message is received by all routers, being one of them an interested router. Router R5 by still being interested in receiving data packets, multicasts a Join message. The Join message by reaching R4, after the Prune message, causes the AW to still forward multicast data packets.

In the same example, now consider that both R5 and R6 are not interested in receiving data packets. In this case, no router would override the Prune message. This means that R4 would never receive a Join, thus placing its non-root interface in a PRUNED state.

Links that were pruned from the distribution tree may later connect interested routers. In order for a router to connect again to a tree, it unicasts a Graft message destined to the router responsible for forwarding multicast data packets. In the case of point-to-point links, this is simply its neighbor, while on shared links this the AW. This message is ACK-protected, offering reliability guarantees, unlike the Join message. The AW by hearing a Graft, places its interface in a FORWARDING state and answers

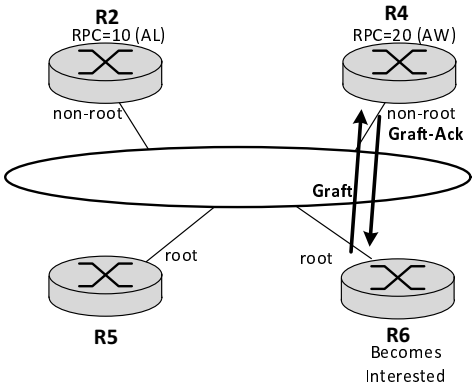

Figure 2.7: PIM Graft.

back with a Graft-Ack message. If the router does not hear a Graft-Ack, it retransmits the Graft until this message is acknowledged. In Figure 2.7, R6 was previously pruned from the tree and by becoming interested in receiving data packets, it unicasts a Graft message to R4 (AW). This router by receiving a Graft, places its interface in a FORWARDING state and answers back a Graft-Ack message to R6.

In PIM-DM, PRUNED state has a finite lifetime. This means that a non-root interface placed in a PRUNED state will eventually timeout, transitioning to a FORWARDING state. This allows to deal with tree reconfigurations and possible inconsistent state at routers (being described in the next sections). This lifetime is of around 3 minutes, meaning that every 3 minutes the network floods multicast data packets, causing the reconstruction of the tree. Later was introduced a mechanism, called State Refresh, that avoids this periodic flood of multicast data packets. This consists in having the originator router periodically transmitting State Refresh messages, as long as the source remains active. These messages are forwarded down the tree by the AW at shared links and by non-root interfaces at point-to-point links, regardless of whether the interface is in a PRUNED or a FORWARDING state. A router by receiving this message, refreshes all its PRUNED interfaces. Besides this, State Refresh messages also include RPC information, allowing to reelect the AW. In case this message is transmitted by a router that offers a worse RPC, the new AW multicasts an Assert message, in order for all routers to elect it as the new AW.

2.2.2.1 PIM-DM issues

We have identified some issues in the PIM protocol, which will be analyzed in this section.

In general, PIM-DM has issues that the protocol only handles by the periodic rebuilding of the broadcast tree or the use of State Refresh messages. This can take as long as 3 minutes, if State Refresh is not enabled, or 1 minute otherwise. PIM-SM [4] inherits some of these issues. Problems can be caused by the absence of reliability in some control messages, which means that some PIM control messages can be lost in the network and the sender of those messages has no clue if they were or were not delivered to a certain router.

Lack of reliability in Join and Prune messages In shared links, PIM protocols use the Prune Override

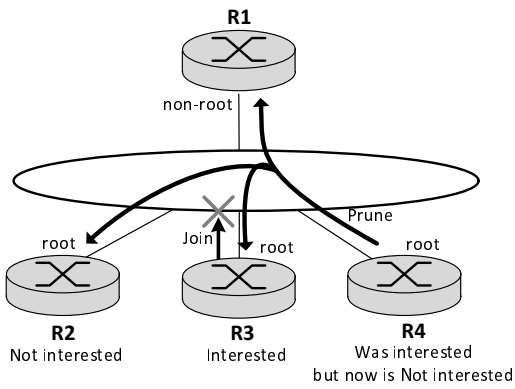

Figure 2.8: PIM Join message is not reliably protected, causing problems on LAN segments.

mechanism to understand when the Assert Winner of this link should stop forwarding (S,G) traffic. There is a problem with this mechanism since Join messages are not reliably protected. If an (S,G) interested downstream router hears a Prune, it sends a Join, but if this Join is lost, the upstream router will cease (S,G) traffic if it does not hear a Join after a certain amount of time (timer that regulates the Prune Override times-out). Figure 2.8 shows this problem. In this figure, R3 sends a Join message after hearing the Prune message from R4. If the Join is lost (for example at the buffer of a switch), R1 by not hearing the Join, would cease (S,G) traffic to this shared link.

In order to mitigate this issue, whenever the Assert Winner prunes its non-root interface, it sends a Prune message, also known as Prune Echo message. This Prune Echo message will serve as a confirmation that the link was pruned and that downstream routers still interested in receiving data packets should override it with a Join. This mechanism simply reduces the probability of a router pruning a link that connects to interested downstream routers, it does not totally solve it, since Prune messages are also not reliably protected.

If a link is pruned without being detected by an interested router, this would only be solved by the periodic reconfiguration of the tree, which can take as long as 3 minutes without State Refresh enabled or 1 minute otherwise.

Delay of Prune Override Also in shared links, an upstream router only prunes a non-root interface when its (S,G) Prune Override timer times-out. This delay can be configured by the network administrator and has a default value of 3 seconds. This way of pruning is not efficient if there is a chain of routers connected to each other by multi-access networks. Consider Figure 2.9. In this figure, the last router sends a Prune because it does not connect to members of group G. The penultimate router only prunes its interface after 3 seconds, when its Prune Override timer times-out. Since the penultimate router has all its non-root interfaces in a PRUNED state, it sends a Prune message through its root interface. The same 3 seconds pause has to be done in all other routers. In this figure a total of 9 seconds would be required to prune traffic destined to group G, causing multicast traffic to be forwarded during this time interval even with no multicast router being interested on it. If there was no State Refresh, multicast traffic would be flooded every time the PRUNED state times-out, causing the Prune process to take place again, which would be completed again after 9 seconds.

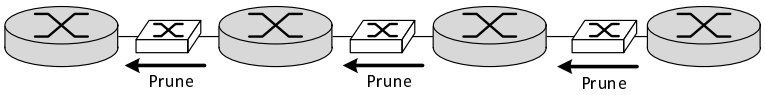

Figure 2.9: PIM Prune problem in a chain configuration.

New routers are unaware of previously formed trees Other problem regarding PIM-DM is that when a PIM-DM router boots up, it is not informed about which trees are currently active in the network. In PIM-DM, a router learns about the existence of trees by receiving data packets or State Refresh messages. Consider Figure 2.10. In this figure, R1 is connected to the link by a non-root interface, being considered the AW, and R2 and R3 are connected to it through root interfaces. Before R2 booted up, the (S,G) source tree was already formed, and this LAN was removed from the tree, since there was no downstream router interested in group G (R3 sent a Prune message regarding (S,G) tree). Since no other router overrode the Prune, R1 pruned its non-root interface from the (S,G) tree. When R2 boots up, discovers via IGMP that it has a directly connected host interested in receiving traffic regarding that tree. The new router will not be able to know about the existence of the (S,G) through the use of data packets, because R1 already pruned its non-root interface. PIM-DM allows to solve this issue by using the State Refresh mechanism, which would forward State Refresh messages down the broadcast tree, refreshing the state of the (S,G) in each router. This mechanism is optional, which may or may not be implemented by all PIM-DM routers.

If State Refresh is disabled, then the new booted up router would only learn and receive (S,G) multicast traffic if:

- Other downstream router sends a Graft message. This message can not be sent by R2 since it does not know about the existence of this tree.
- R1 PRUNED state times-out (around 3 minutes).

In the case of using the State Refresh capability of PIM-DM, this has some issues too. According to section 4.3.4 of RFC3973 [2], “If the neighbor is downstream, the router MAY replay the last State Refresh message for any (S,G) pairs for which it is the Assert Winner indicating Prune and Assert status to the downstream router”. State Refresh messages are forwarded down the tree every 60 seconds, even through non-root interfaces that were pruned from the tree, as long as the source keeps transmitting data packets. If R1 did not receive any State Refresh message yet or this message was lost upwards in the tree, R2 would be prevented from receiving (S,G) traffic until the first State Refresh message reaches R1. Since State Refresh messages are not reliably protected, even if R1 replays this message, it is possible that R2 would not receive it.

Slow tree reconfigurations In PIM, there is no tree reconfiguration in case there is an RPC change. This may cause the tree to have a sub-optimal configuration, since multicast data packets may not be forwarded through the paths that offer the lowest RPC. This issue is due to the Assert state machine not react to RPC changes.

So if the current elected Assert Winner changes its RPC, to a lower or higher value, it does not

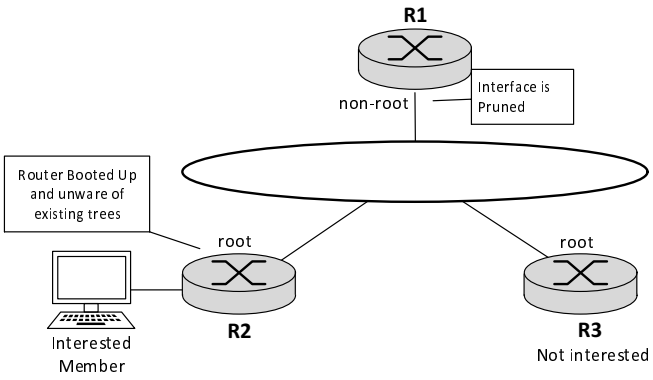

Figure 2.10: PIM Router booted up/restarted and does not know about the already formed source trees.

notify its neighbors. This causes all neighbors to think that the current Assert Winner remains with the last notified RPC configuration. A new reelection would occur, reconfiguring the tree, after the Assert Winner state times out (state is maintained around 3 minutes) or when the next State Refresh message is transmitted (every 1 minute).

This issue causes the tree to react slowly to network changes.

AW unaware of interested neighbors In shared links, the Assert Winner does not know which downstream routers are interested. If the only interested router fails, the Assert Winner would still forward data packets to a link. This would be solved by the other downstream routers that would eventually transmit a Prune message, by receiving data packets. The Prune transmission is rate-limited causing the other downstream routers to not prune instantly the link.

This causes all links between the failed router and the source to forward data packets, reacting slowly to the failure of interested downstream routers.

Lack of message ordering guarantees PIM control messages are not sequenced. For this reason there are no ordering guarantees regarding the reception of messages. So if a router transmits two messages, the underlying network may change its order, not being detected at the receiver.

This can be problematic if a downstream router transmits a Prune and then a Join/Graft. If the underlying network changes its order, the last received message at the Assert Winner would be a Prune, which would indicate that its source is no longer interested in receiving multicast data packets. This can cause a router to prune prematurely a link while there are still interested downstream receivers.

In case the downstream router transmits a Graft/Join and then a Prune and the underlying network changes its order, this would be less problematic, since the Assert Winner would forward data packets to the link. Nevertheless, if there are no interested downstream routers, all routers between the Assert Winner and the source of multicast traffic would forward unnecessarily multicast data packets.

These problems are aggravated by the first identified issue, since the Prune Override mechanism, used to control the interest of downstream routers in a link, does not use any reliability protection.

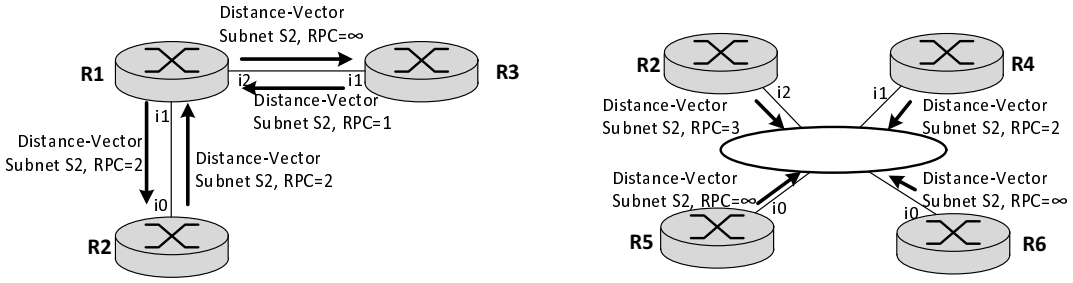

Figure 2.11: DVMRP Distance-Vector exchange.

2.2.3 DVMRP

DVMRP [3] is a distance-vector protocol based on the unicast routing protocol RIP [18], that exchanges distance-vector messages to configure multicast trees. This information allows routers to determine which of their interfaces is of type root and non-root, for each possible (S,G) tree.

The protocol transmits periodically distance-vector messages with information regarding each subnet existing on the network. Through the received messages, a router can decide which of its interfaces is the one that is closer to the source, i.e. root interface. This protocol uses a Poison Reverse process, consisting of echoing back to the router that transmitted the best distance-vector message, a distance-vector with a path cost of infinity. This allows to avoid the count to infinity issue, besides allowing a non-root interface to determine which of its neighbors depend on it to receive multicast traffic originated from a given subnetwork. This means that by periodically exchanging distance-vector messages, routers in a link can determine which non-root interface will be the one responsible for forwarding multicast traffic (AW), besides knowing which neighbors depend on it (which of them are connected to the link by a root interface).

The initial and periodic exchange of distance-vector messages, allows routers to form a truncated broadcast tree, meaning a tree that connects all routers but without any redundant paths and without having any data packets involved.

Figure 2.11 shows the exchange of distance-vector messages regarding the subnet of source S2 (of Figure 2.1). Consider that these messages have already the final RPC, meaning that the protocol has already stabilized and determined the least amount of hops to subnet of source S2. Figure 2.11 at the left, illustrates the exchange of these messages in two point-to-point links. In the point-to-point link that connects interface i2 of R1 to interface i1 of R3, R1 transmits messages with an infinite RPC by reaching the least amount of hops through this interface, while R3 transmits an RPC of 1 (one hop of distance). By exchanging these messages, R3 knows that its neighbor R1 is connected to this link by a root interface. In the other point-to-point link, that connects R1 to R2, they both exchange non infinite distance-vector messages, meaning that both routers are connected through non-root interfaces. This information allows each router to determine the type of interface connected to each link.

Regarding Figure 2.11 at the right, we have a shared link exchanging distance-vector messages regarding the same subnet. Both R2 and R4 transmit distance-vector messages with a non infinite RPC, while R5 and R6 transmit messages with an infinite RPC. The exchange of these messages allows

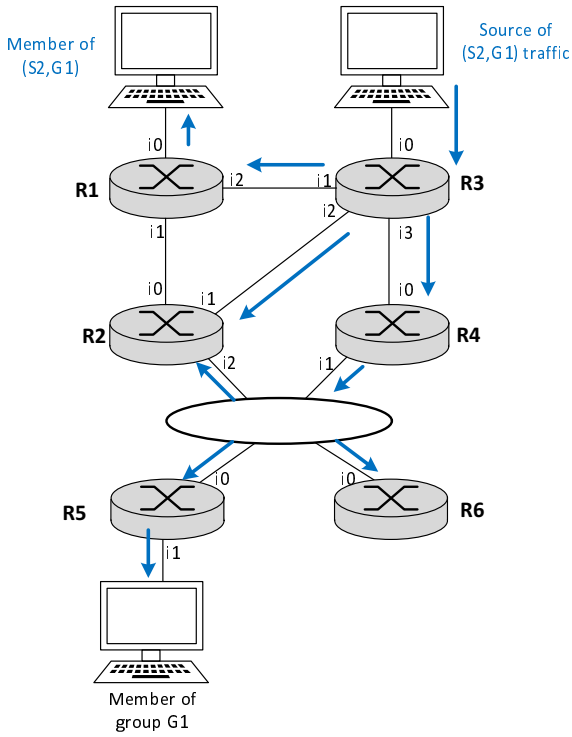

Figure 2.12: DVMRP truncated broadcast tree.

to determine which router offers the lowest RPC, which in this case is R4. This means that R4 will be considered as AW for all multicast data packets originated from the subnet of S2. This router also determined that two of its neighbors are connected by root interfaces to this link, by having transmitted messages with infinite RPC.

As soon as multicast data packets start being received, they are forwarded through all non-root interfaces that offer the lowest RPC (AW) and that have at least one neighbor connected by a root interface to the same link. Figure 2.12 illustrates the initial forwarding of multicast data packets. As we can see the previously exchanged distance-vector messages already pruned redundant paths from the tree, thus the name truncated broadcast tree.

In DVMRP, the AW maintains the interest state of all its neighbors, unlike PIM-DM that only maintains the global interest of a link based on the last received control message. As soon as a router hears a data packet, in which it has no interest, it unicasts a Prune message to the AW. If later it wants to receive data packets again regarding this tree, it unicasts a Graft message to the AW. This Graft message is ACK-protected, requiring the AW to answer back a Graft-Ack message. Both messages were already already illustrated in the PIM protocol. Regarding Prune messages, these are always unicasted to a single neighbor (AW) even in shared links, unlike Prune messages in PIM-DM that are multicasted.

When a non-root interface does not offer the best RPC or it has no dependent downstream routers, i.e. no one transmitted to it a distance-vector with a path cost of infinity, or all dependent downstream routers have transmitted a Prune, the interface is placed in a PRUNED state, otherwise the interface is placed in a FORWARDING state. By having all non-root interfaces in a PRUNED state, the router transmits a Prune message through its root interface in order to prune the tree.

Like PIM-DM, the PRUNED state at non-root interfaces has a finite lifetime. In this protocol, PRUNED state only lasts 2 minutes, meaning that periodically pruned interfaces are placed in a FORWARDING state, causing data packets to be flooded throughout the whole network. Unlike PIM-DM, this protocol has no mechanism to prevent this.

2.2.3.1 DVMRP issues

DVMRP protocol was the first specified source-based multicast routing protocol. This protocol by being a distance-vector protocol based on RIP, it inherits some of its issues.

Slow convergence Like RIP, DVMRP offers slow convergence. This causes multicast distribution trees to not be reshaped instantaneously in reaction to changes at the network.

Each router transmits route updates every 60 seconds. This causes the network to be overwhelmed by these control messages, besides reacting slowly to changes.

Tree must have a diameter lower than 32 between the source and interested receivers DVMRP uses a Poison Reverse process in order to avoid network loops and to determine which downstream routers depend on a given upstream router to receive multicast data packets. By hearing an infinite distance-vector message, a router determines that its neighbor is connected to the link by a root interface. The problem is that a distance-vector greater than 31 is considered to be infinite. This implies that the distance between the source of multicast data packets and its interested receivers can be composed at most by a maximum of 31 hops.

DVMRP uses hop-count as metric This implies that all trees are constructed in order to forward data packets through the interface that is closer, in terms of number of hops, to the source. Current unicast routing protocols no longer use number of hops as metric, since these selected paths may not be the best ones. For example, the path that traverses less routers may be the one that offers less bandwidth. This may cause the constructed tree to be sub-optimal.

Periodic flood of multicast traffic The router responsible for forwarding multicast data packets, to a given link, only maintains PRUNED state, about its neighbors, for a given amount of time. If those neighbors do not refresh their PRUNED state in this router, this will eventually timeout, causing the router to consider its neighbors to be interested again in receiving data packets. If the source is still active, this causes a periodic re-flood of multicast data packets, even when there are no interested receivers.

2.3 Sparse Mode multicast routing protocols

2.3.1 Principles

Sparse Mode protocols build (*,G) shared trees rooted at a central router (RP) and spanning all members of group G. These trees connect all interested receivers of a given group to all multicast

sources that are transmitting data packets to the same group. These trees are maintained as long as there are interested receivers.

Sparse Mode protocols build a shared tree as soon as a router discovers an interested receiver. In this case, the router transmits Join messages towards the RP. This message travels router by router until it reaches the RP or a router that was already part of the shared tree. A router by receiving this message, simply adds the non-root interface that received this message to the OIL of the shared tree.

If later a router no longer connects to interested receivers or routers, it transmits a Prune message towards the RP. Like the Join message, the Prune also travels router by router until it reaches the RP or a router that still wants to be part of the tree. A router by receiving a Prune message needs to be cautious, since it can only PRUNE the interface that received the packet, if all downstream routers connected to the same link do not want to be part of the tree.

In shared-links only one non-root interface must forward these data packets. Like Dense Mode protocols, this is accomplished by the Assert protocol that elects a single non-root interface as AW, being this interface responsible for forwarding all data packets to the link.

In order to have multicast data packets to reach all its members of the (*,G) shared tree, these packets need to be forwarded from its source to the RP, and from the RP to members of group G.

A Sparse Mode protocol requires to inform the RP when sources start transmitting data packets. If the RP already formed a (*,G) shared tree, and the source is transmitting to group G, a (S,G) source tree that connects the RP to source S needs to be created. Packets from S destined to G will be forwarded from the source to the RP using the (S,G) source tree, and then forwarded from the RP down the (*,G) shared tree, in order to reach its members.

Also, it is required that the RP dis-attaches from the (S,G) source tree when all routers do not want to be part of the (*,G) shared tree.

2.3.2 PIM-SM

Like PIM-DM, PIM-SM also relies on a unicast routing protocol to determine the shortest paths from a router to the root of the tree. PIM-SM requires that all routers know the identity (IP) of the RP. This can be accomplished by either manually configuring each router or by using a Bootstrap mechanism to dynamically learn this information.

In order to have a router to be part of the (*,G) shared tree, because it is now connected to a member of group G, this router sends through its root interface a Join message. The message by traveling to the RP allows to attach this router to the (*,G) shared tree. For a router to no longer be part of a (*,G) shared tree, it sends a Prune message through its root interface.

In case a Prune message is received by an interface connected to a point-to-point link, the interface is placed in a Pruned state.

In case a Prune message is received by an interface connected to a shared link, a non-root interface can only be placed in a PRUNED state if it does not connect to interested receivers or routers of the (*,G) tree. In order to deal with this case, the same Prune Override mechanism described in PIM-DM

(section 2.2.2) is used.

Up to this point, it was described the creation and maintenance of (*,G) shared trees. One advantage of PIM-SM, compared to other Sparse Mode protocols, is that it is not limited to the creation of only shared trees. One router can dissociate from the (*,G) shared tree and join a (S,G) source tree rooted at the source of the multicast traffic. By joining the (S,G) tree, multicast data packets do not require to go through the RP, allowing to reduce latency and possible congestion at the RP. This mechanism is called Shortest Path Tree Switchover and works as follows:

- A router that is directly connected to receivers that has previously joined a (*,G) shared tree (being G a group that has interested receivers), when it starts receiving multicast traffic destined to group G originated by a given source S, it learns about the existence of (S,G) and sends an explicit Join through its root interface that is on the reverse path to source S. This message is propagated hop-by-hop until it reaches the router that is directly connected to source S. All routers between the source and the interested receiver join the (S,G) source tree.
- Routers that joined the (*,G) shared tree and (S,G) source tree, now receive duplicate data packets of source S destined to group G by the two trees. In order to prevent this situation, these routers send Prune messages regarding the (S,G) source tree towards the RP. This prevents multicast data packets to flow through paths of the shared tree that have routers that already joined the source tree (that is rooted at the source of multicast traffic).

The mechanism described above, that allows a router to join a source tree, is usually accomplished by specifying a bandwidth threshold, i.e. only multicast traffic that exceeds this threshold causes a router to connect to the (S,G) source tree.

Up to this point, it was described the creation and maintenance of (*,G) shared trees and (S,G) source trees, which are triggered by routers that directly connect to interested receivers. Now it will be described how sources send data packets to the RP.

When a host (source S) wants to send multicast traffic destined to group G, it simply sends it. This is received by its directly connected router (also known as originator). The originator requires to inform the RP about the existence of this source. For each (S,G) multicast packet received by that router, it sends a unicast packet with the multicast packet encapsulated on it, destined to the RP. This message is called PIM Register message and has the purpose of informing the RP about the existence of source S, that is sending packets destined to group G. When the PIM Register message is received by the RP, this router de-encapsulates the multicast packet from it, and:

- In case the RP has already formed the (*,G) tree, the multicast packet is forwarded down the shared tree, in order to reach members of group G. The RP then sends a Join message towards the source, in order to form a (S,G) source tree from the originator to itself. This source tree will be used to send “natively” multicast packets from source to the RP, instead of them being encapsulated on PIM Register messages.
- In case the RP does not know about the existence of members of group G, the multicast packet is discarded. In order to stop receiving (S,G) multicast packets encapsulated on PIM Register messages, the RP sends via unicast, a PIM Register-Stop message destined to the originator.

If the RP formed a source tree with the originator router, when it starts receiving (S,G) multicast packets from it, it sends a PIM Register-Stop. If the originator receives a PIM Register-Stop message, it stops sending encapsulated (S,G) multicast packets. When the RP does not have routers interested in the (*,G) tree (there are no members of group G), it sends a Prune towards the originator of source S, in order to cease multicast traffic, from the source towards the RP.

In case there are multiple non-root interfaces connected to a shared link and they transmit data packets, the same Assert mechanism described in PIM-DM (section 2.2.2) is used, in order to elect a single Assert Winner.

As in each multicast routing protocol analyzed so far, each entry in the multicast routing table has a lifetime. In this protocol, non-root interfaces in a FORWARDING state must be refreshed in order to remain in this state, otherwise they are placed in a PRUNED state (expires after 3 minutes). In order to prevent the "expiration" of this state, interested routers send Join messages towards the RP every 1 minute.

2.4 Link-State multicast routing protocols

2.4.1 Principles

Link-state protocols build (S,G) source trees, like Dense Mode protocols, but do not rely on a initial broadcast tree, instead, control messages are flooded throughout the network, with information about the location of members of multicast traffic. This information, along with the topology of the network is stored in a database in order to determine a shortest path tree rooted at the source of multicast traffic and spanning all its members. MOSPF is a protocol that builds multicast distribution trees using this approach.

2.4.2 MOSPF

OSPF stands for Open Shortest Path First and it is a unicast routing protocol. MOSPF is an extension to support multicast routing through the already existent OSPF. MOSPF is defined in RFC1584 [5] and it was supported on OSPFv2 [19] but was deprecated on OSPFv3 [20].

The unicast routing protocol, OSPF, is based on a Link-State approach, i.e. routers exchange information between them (called LSA - Link State Advertisements) about the prefixes assigned to routers and links, so that all OSPF routers know about the topology of the network, also known as network graph. These LSAs are stored in the LSDB (Link State DataBase). Routers by knowing the whole network topology can make forwarding decisions based on the path that offers the lowest cost to some router by using Dijkstra algorithm [21]. The MOSPF extension introduces a new type of LSA, called group-membership-LSA, that is used to disseminate information about where in the network are interested receivers of a certain group G. This LSA is created by the router that is directly connected to the receiver, and this type of information is flooded throughout the whole network in order to inform all MOSPF routers about

the existence of members of a certain group G, in specific subnets. On a subnet that connects multiple routers and hosts, only one router (the designated router) floods the group-membership-LSA.

This multicast routing protocol follows a Link-State approach by taking advantage of the network graph that is built by OSPF and extending it with the new group-membership-LSAs. The interfaces used to forward (S,G) multicast traffic are determined on demand, upon arrival of the first (S,G) multicast packet. The construction of trees is performed by the same algorithm of OSPF, calculating the shortest-paths that connect the source of multicast traffic to all interested receivers stored in the LSDB. By knowing if the router is on the path between the source and a receiver, it forwards data packets through the corresponding non-root interface.

Chapter 3

Hard-state Protocol Independent Multicast - Dense Mode (HPIM-DM)

The Hard-state Protocol Independent Multicast - Dense Mode (HPIM-DM) is a multicast routing protocol designed to overcome the limitations of PIM-DM and react promptly to events susceptible of changing the multicast distribution tree, such that the multicast tree is maintained correct at all times. It is a hard-state protocol: except for Hello messages (aimed at maintaining neighborhood relationships) there is no periodic circulation of control messages. As in PIM-DM, the protocol relies on a unicast routing protocol to define the multicast tree, and the control messages are transmitted using the RPF principle. Unlike PIM-DM this protocol is control-driven, i.e. maintaining the multicast tree is only performed through control messages.

The explanation of the protocol is divided in several parts. Section 3.1 addresses the Hello protocol, which is used to maintain neighborhood relationships. In section 3.2 we discuss how a tree that spans all routers, the broadcast tree, is built and maintained, and section 3.3 explains how this tree is pruned according to the interest of routers, leading to the multicast tree. Section 3.4 explains how data is flooded initially, while the multicast tree is being formed. Section 3.5 discusses how control messages are sequenced. Section 3.6 explains the synchronization process, which allows new routers to obtain the information needed to connect to existing multicast trees. Section 3.7 discusses the reliability of transmitted control messages. Section 3.8 presents the format of the control messages. Finally, section 3.9 addresses the protocol mechanisms that avoid security attacks.

3.1 Hello protocol

A mechanism is needed to maintain neighborhood relationships. Routers need to know which are their active neighbors that want to participate in the multicast routing process. This mechanism is used in many other routing protocols, e.g. OSPF and EIGRP, and is known as the Hello protocol.

There are three phases in neighborhood relationships: detection of neighbor, establishment of relationship, and maintenance of relationship. In our protocol, the detection of a neighbor can be performed

by any control message, and not just by Hello messages. This contrasts with other protocols, but aims at speeding up the protocol convergence when a new neighbor is found. In our protocol, the establishment of a relationship (i.e. when a neighbor is considered active) requires that the two neighbors synchronize with each other; the synchronization process is discussed in section 3.6. Finally, the maintenance of a relationship (i.e. knowing if the neighbor is still alive) is carried out by the Hello protocol, through the periodic transmission of Hello messages.

The Hello protocol works as follows:

- A router keeps sending periodically Hello messages on each of its links, to all its neighbors. On shared links, the transmission of Hello messages is multicasted to all neighbors, such that only a single message needs to be transmitted.
- After establishing a relationship with a neighbor (which requires synchronization), a router keeps monitoring the Hello messages transmitted by the neighbor. A neighbor is declared inactive when a predefined time period (typically four times the periodicity of Hello transmissions) has passed without having received any Hello from it.

Thus, the Hello protocol requires one timer to regulate the transmission of Hello messages, called Hello Timer (HT), and one timer per neighbor to monitor the Hello messages received from the neighbor, called Neighbor Liveness Timer (NLT).

In other protocols, such as OSPF and IS-IS, there is a clear separation between the establishment of neighborhood relationships and the synchronization of information (in case of OSPF, the synchronization is the exchange of link state databases). In these protocols, the establishment of neighborhood relationships is done exclusively by the Hello protocol. Hello messages transmitted by a router carry the addresses of all its neighbors on the link, and a neighbor is considered active when bidirectional communication with it has been established. Specifically, a router considers a neighbor active when it sees its address listed in the messages transmitted by the neighbor. This solution is prone to replay attacks: an attacker may record the initial Hello message transmitted by a router, where no neighbor is declared, and then use this message to destroy neighborhood relationships. Our protocol avoids this problem by declaring a neighbor active only when a successful synchronization with it was completed. Thus, the exchange of Hello messages is not sufficient to declare a neighbor active.

Figure 3.1 illustrates the replay attack that OSPF is susceptible to. Router R2 considers R1 active when it receives an Hello from R1 with its own address; R1 considers R2 active in the same way. If later an attacker replays the first Hello sent by R1, with no neighbors listed, R2 ceases from considering R1 active. In our protocol, Hello messages do not carry the addresses of known neighbors, which allows to not be vulnerable to this attack.

3.2 Broadcast tree maintenance

The broadcast tree allows multicast data to be transmitted from an originator (a router directly attached to the multicast source) to all other network routers. This tree may be pruned from routers not

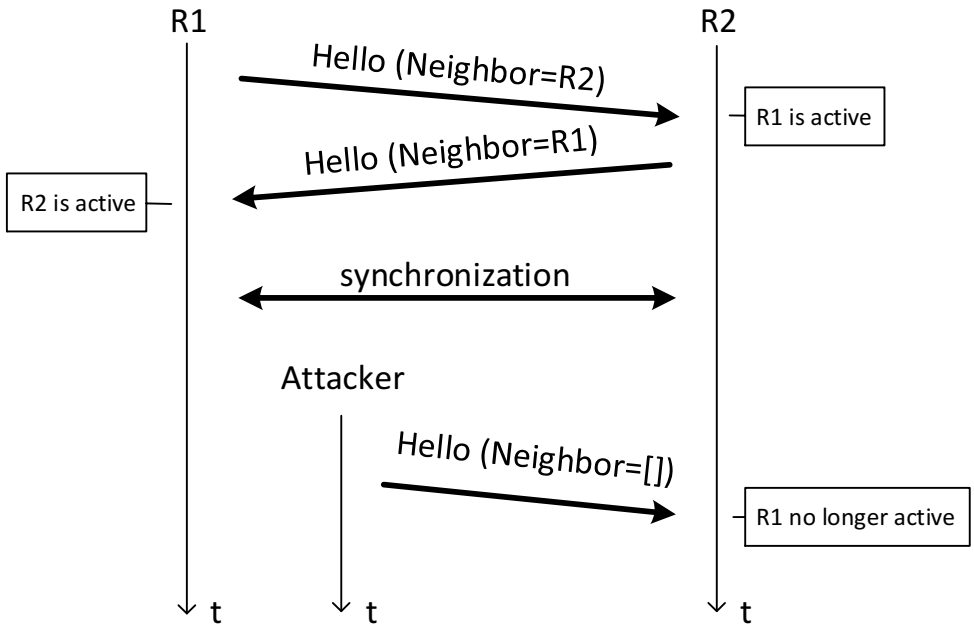

Figure 3.1: Hello replay attack.

interest in receiving multicast data, an issue that will be discussed in section 3.3. In our protocol, a router always keeps information on the broadcast tree (i.e. who its parent in the tree is), irrespective of its interest in receiving multicast data. Thus, the broadcast tree is maintained by all routers, as long as the corresponding source remains active. In PIM-DM, the broadcast tree is built by the multicast data, and the tree is maintained at a router as long as data or State Refresh messages keep arriving. In our protocol, the broadcast tree is maintained at a router as long as it has a parent on the tree. This is performed without the need for the periodic circulation of control messages or data. Thus, PIM-DM is a soft-state protocol whereas our protocol is an hard-state protocol.

Similarly to PIM-DM, the broadcast tree maintenance relies on an unicast routing protocol and on the RPF flooding mechanism. The unicast routing protocol allows determining, at a router, the root interface (the one that should receive multicast data) and the non-root interfaces (the ones that should transmit multicast data) for a given tree. The RPF mechanism allows forwarding control messages downwards the tree.

3.2.1 Basic maintenance mechanisms

Control messages Two control messages are used in the tree maintenance procedure: IamUpstream and IamNoLongerUpstream; we will refer to these two messages as the upstream messages. The upstream messages are transmitted downwards the tree using the RPF technique and its transmission is ACK-protected. The IamUpstream message indicates to a receiving router that the sending router can act as a feeder of multicast data, and the IamNoLongerUpstream message indicates the opposite. The IamUpstream message includes the RPC of the sending router to elect the AW at non-root interfaces and to eliminate routing loops.

Upstream state of neighbors Each router stores information on whether a neighbor is upstream or not. Thus, at a given router, a neighbor can be in one of two states: UPSTREAM or NOT UPSTREAM. A neighbor is UPSTREAM at a router, if the last message received from the neighbor was an IamUpstream message. An UPSTREAM neighbor is a router from which multicast data can be received. Storing information on all UPSTREAM neighbors, and not just on the parent router, is a distinguishing feature of this protocol that allows fast convergence when the tree changes. The parent router is selected among the UPSTREAM routers.

Parent router The parent of a router in the broadcast tree is the UPSTREAM neighbor reachable through the root interface that provides the lowest RPC. Moreover, an UPSTREAM neighbor is only considered parent if it provides an RPC lower than the router itself (i.e. if it is closer to the source). This condition is used in other routing protocols, to prevent the existence of routing loops; it is called feasibility condition [22–25]. In our case, it is used to ensure that routers do not maintain trees indefinitely in the presence of routing loops. This will be illustrated in section 3.2.5.

A router will only transmit an IamUpstream message downwards the tree if it has a parent on the tree. Thus, when a router receives an IamUpstream message on a root interface, it can be certain that all routers between itself and the source form a tree structure that can forward multicast data.

Installing and removing a tree When a source becomes active, the originator routers initiate the construction of the broadcast tree by transmitting IamUpstream messages through all their non-root interfaces; as mentioned above, these messages are broadcast to all other routers using the RPF technique. Thus, IamUpstream messages are forwarded downwards the tree, i.e. through non-root interfaces, if received on a root interface from its parent on the tree, and are discarded otherwise. Likewise, when a source becomes inactive, the originator routers initiate the broadcast of IamNoLongerUpstream messages to signal all other routers of this event, so that routers can remove their state relative to the tree. The initiative of transmitting IamUpstream and IamNoLongerUpstream messages may not belong to the originators, in which case the messages may not target all network routers.

Assert protocol The AW is elected in the same way in point-to-point and shared links. The election is a simple process, since IamUpstream messages carry the RPC of the sending router, and routers keep information on who their UPSTREAM neighbors are and what are their RPCs. When a source becomes active, all non-root interfaces attached to shared links transmit IamUpstream messages. Based on this information, each interface (root or non-root) elects the AW as the non-root interface that advertised the lowest RPC. If only one non-root interface transmits an IamUpstream message, that interface will become AW.

3.2.2 Tree states

A router is considered to be in one of three states regarding a tree: ACTIVE, INACTIVE, or UNKNOWN. In the ACTIVE state, the source is considered to be active and, in UNKNOWN state, it is considered to be inactive. The INACTIVE state is a transient state, where the router keeps some information about the tree but is unsure on whether the source is active or not; this state was introduced

for protocol correctness and to speed-up convergence. The definitions of the tree states are slightly different in originator and non-originator routers. The definition is the following for non-originator routers:

- ACTIVE - A router is in ACTIVE state if it has at least one UPSTREAM neighbor connected to its root interface, and at least one of these neighbors offers an RPC lower than its own RPC.
- INACTIVE - A router is in INACTIVE state if it has no UPSTREAM neighbor connected to its root interface but the router has at least one UPSTREAM neighbor on some non-root interface or if all UPSTREAM neighbors connected to the root interface offer an RPC greater or equal than its own RPC.
- UNKNOWN - A router is in UNKNOWN state if it has no UPSTREAM neighbor connected to any interface.

For originator routers, a router is considered ACTIVE if it is receiving multicast data, INACTIVE if it is not receiving multicast data but is connected to some UPSTREAM neighbor, and UNKNOWN if it is not receiving multicast data and is not connected to any UPSTREAM neighbor.

Importance of the INACTIVE state The INACTIVE state is required both for correctness and to speed-up convergence, and is a distinguishing feature of this protocol. In fact, since the protocol is hard-state, a router can only remove information regarding a tree when it is completely sure that the tree is no longer active. Thus, a router must keep maintaining state about a tree as long as it has UPSTREAM neighbors for that tree. UPSTREAM neighbors are surely in ACTIVE state and, therefore, believe that the tree is active. In INACTIVE state, either a router loses all its UPSTREAM neighbors, in which case the tree will become UNKNOWN, or the router reconnects to an UPSTREAM neighbor on its root interface (e.g. due to a change in its root interface or the appearance of a new UPSTREAM neighbor at its root interface), in which case it will become ACTIVE. In the latter case, the election of the AW is fast since the router already has information on the UPSTREAM neighbors connected to its non-root interfaces.

Consider a solution where the state of a tree is completely removed in INACTIVE state. In this case, the router would lose information regarding its UPSTREAM routers, and would not be able to reconnect to the tree.

3.2.3 AW definition

A router by storing information regarding all UPSTREAM neighbors and their RPCs, can calculate which is the UPSTREAM neighbor that offers the lowest RPC to the source of multicast traffic. This neighbor will be called BestUpstreamNeighbor. At a root interface, the BestUpstreamNeighbor is considered the one responsible for forwarding multicast traffic. At a non-root interface, the BestUpstream-Neighbor and its RPC is used, together with the interface RPC, to determine if the interface is AW or AL.

The AW is defined not only for the ACTIVE state, but also for the INACTIVE and UNKNOWN states. This is motivated by the data flooding issue, which will be discussed in section 3.4. Specifically, a non-root interface of a given tree is considered AW in the following conditions:

- If the tree is ACTIVE and the RPC offered by the router is lower than the RPC offered by the BestUpstreamNeighbor. In case of tie, the IP address is used to break the tie, and the highest IP address wins.
- If the tree is INACTIVE and no neighbor is considered UPSTREAM;
- If the tree is UNKNOWN.

If a tree is ACTIVE at a router, a non-root interface becomes AW if it provides the lowest RPC, considering all UPSTREAM neighbors connected to the interface. Recall that the RPC is included in all IamUpstream messages to enable the AW election. If a tree is UNKNOWN at a router, all its non-root interfaces become AWs. This allows the initial flooding of data, when the tree is still not formed. If a tree is INACTIVE at a router, all its non-root interfaces become AWs, except in interfaces where there is at least one UPSTREAM neighbor. This minimizes the loss of data during transient periods.

3.2.4 Reacting to changes in the broadcast tree

Once a broadcast tree is formed, it may change due to the following events: (i) change of unicast interface cost, (ii) router failure, (iii) insertion of new router. When this happens, a transition from an old tree to a new tree may need to be performed and, during this period, routers may be temporarily disconnected from the source (i.e. placed in the INACTIVE state). It may also be the case that a router loses completely its (unicast) connectivity with the source.

Message transmissions triggered by state transitions State transitions may trigger the transmission of IamUpstream or IamNoLongerUpstream messages downwards. Specifically, when a router changes from ACTIVE to INACTIVE or UNKNOWN states it sends an IamNoLongerUpstream message through all its non-root interfaces, to signal downstream routers that it is no longer connected to the tree. Contrarily, when a router changes from INACTIVE or UNKNOWN states to ACTIVE state it sends an IamUpstream message through all its non-root interfaces, to signal downstream routers that it became connected to the source. Finally, no message is transmitted when a router stays in the same state or when it changes from the INACTIVE to UNKNOWN state or vice-versa.

Events that trigger state changes Several events may trigger state changes: (i) reception of IamUpstream or IamNoLongerUpstream messages, (ii) reception of multicast data, (iii) change of RPC, (iv) loss of root interface, (v) change of interface role, (vi) connection to new neighbor, and (vii) neighbor failure.

Reaction to sensing of multicast data on root interface of originator router When the first multicast data packet arrives at the root interface of an originator router, the router becomes ACTIVE (and sends IamUpstream messages through all its non-root interfaces) and sets a timer to control the liveness of the source; this timer is called Source Active Timer (SAT). The timer is reset whenever a data packet arrives. When the timer expires, this means that the source ceased to transmit data, and the router changes to INACTIVE or UNKNOWN (and sends IamNoLongerUpstream messages through all its non-root interfaces), depending on whether it still has UPSTREAM neighbors.

Reaction to messages received on root interface of non-originator router The reception of IamUpstream or IamNoLongerUpstream messages at the root interface of a non-originator router may trigger state transitions. The behavior depends on the state of the receiving router and the RPC of the received message. Specifically, the reception of these messages may trigger a change in the parent: another router may become parent, the router may cease having a parent, the router may start having a parent, or the router may keep not having a parent. Recall that the parent is the UPSTREAM neighbor that provides the lowest RPC, subject to the restriction that the router RPC must be higher than the parent RPC.

When an IamUpstream message is received, if the router is ACTIVE and keeps having a parent, the router stays ACTIVE (and no message is transmitted downwards); otherwise, if the router ceases from having a parent, the router changes to INACTIVE or UNKNOWN (and sends an IamNoLongerUpstream message through all its non-root interfaces). Likewise, if the router is INACTIVE or UNKNOWN and starts having a parent, the router changes to ACTIVE (and sends an IamUpstream message through all its non-root interfaces). Finally, if the router is INACTIVE or UNKNOWN and keeps not having a parent, the router stays in the same state (and no message is transmitted).

When an IamNoLongerUpstream message is received, if the router is ACTIVE and ceases having a parent, the router changes to INACTIVE or UNKNOWN (and sends an IamNoLongerUpstream message through all its non-root interfaces). If the router is INACTIVE it may stay INACTIVE or change to UNKNOWN, and if it is UNKNOWN it stays UNKNOWN (in both cases no message is transmitted).

When a router has to change to INACTIVE or UNKNOWN states, the decision depends on the number of UPSTREAM routers that are left: if none is left the transition is to UNKNOWN, and if at least one is left the transition is to INACTIVE.

Reaction to messages received on non-root interface Messages received on non-root interfaces (of originator or non-originator routers) set the state of a neighbor, as being an UPSTREAM neighbor or not, and may change the interface state from AW to AL or vice-versa. If the router is UNKNOWN and receives an IamUpstream message, it changes to INACTIVE, since it just discovered an UPSTREAM neighbor on a non-root interface. Likewise, if the router is INACTIVE and receives an IamNoLongerUpstream message from the last known UPSTREAM neighbor, it changes to UNKNOWN. If the router is ACTIVE, it remains in the same state irrespective of the reception of IamUpstream or IamNoLongerUpstream messages in non-root interfaces.

Reaction to RPC changes (without interface role change) of non-originator routers When a router is ACTIVE and its RPC changes, if the router remains ACTIVE it sends an IamUpstream message through all non-root interfaces, to inform its downstream routers of its new RPC value. This message may trigger the reelection of the AW at one or more non-root interfaces of the router. However, the router may change to INACTIVE if its RPC becomes lower or equal than the RPC of all its UPSTREAM neighbors connected to the root interface. If the router is INACTIVE and its RPC changes, it changes to ACTIVE if the RPC becomes higher than the RPC of at least one UPSTREAM neighbor connected to the root interface.

Reaction to loss of root interface A router may lose its root interface if the interface fails or loses the unicast route to the source. In this case, all interfaces will be considered non-root. Moreover, if the interface fails all state relative to the neighbors on this interface must be removed. If the router was ACTIVE, it changes to INACTIVE or UNKNOWN, depending on whether it has UPSTREAM neighbors (and sends an IamNoLongerUpstream message through all its previous non-root interfaces). If the router was INACTIVE there are two cases: (i) if the interface failed the router may remain INACTIVE or change to UNKNOWN, depending on whether it has UPSTREAM neighbors; (ii) if the unicast route was lost, the router remains INACTIVE.

Reaction to interface role change of the originator It may happen that the root interface of the originator gets disconnected from the source. In this case, the router becomes non-originator and follows the procedure described above relative to the loss of root interface.

Reaction to interface role changes of non-originator router When a router is ACTIVE and its root interface changes, if the router remains ACTIVE it sends an IamNoLongerUpstream through the new root interface and an IamUpstream message through the new non-root interface. These messages notify the neighbors of the router in these interfaces of its new role (as being UPSTREAM or NOT UPSTREAM). In the case of non-root interfaces that kept its role, an IamUpstream message needs only to be transmitted if there was, simultaneously, a change in the router RPC. If a router is ACTIVE and there is no parent in the new root interface, it changes to INACTIVE (and sends an IamNoLongerUpstream message through all its previous non-root interfaces). If a router is INACTIVE and a parent is found in the new root interface, it changes to ACTIVE (and sends an IamUpstream message through all its new non-root interfaces).

Reaction to change from non-originator to originator When a router enables an interface directly connected to the source, it changes its role from non-originator to originator. If the router was ACTIVE, it remains ACTIVE, sets the Source Active Timer, and transmits an IamUpstream message through all its non-root interfaces. If the router was INACTIVE it remains in this state until it receives multicast data, and no message transmission is required.

Reaction to failure of neighbor When a neighbor fails (as signaled by the Hello protocol), all state information relative to this neighbor must be removed. This may trigger a change of state, if the failed neighbor was an UPSTREAM neighbor. The behavior in this case is the same as if the router received an IamNoLongerUpstream message from the neighbor.

Reaction to detection of new neighbor When a router connects to a new neighbor, the two routers must synchronize with each other. During this process, the neighbor provides information on if it can serve as an UPSTREAM neighbor to the router. The neighbor can do so if it is ACTIVE and the synchronization is being made through one of its non-root interfaces. Thus, the behavior is equivalent to receiving an IamUpstream message. The synchronization process will be explained in section 3.6.

Reaction to reboot or synchronization attempt of known neighbor If a known neighbor reboots or initiates a new synchronization (e.g. because it was unable to communicate with the router for a while),

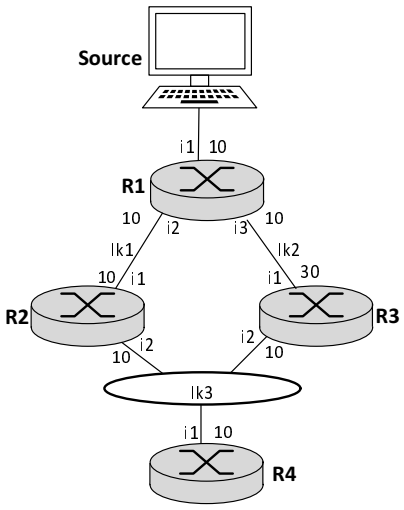

Figure 3.2: Broadcast tree maintenance example.

the tree information stored for that neighbor may no longer be valid. In this case, all information regarding this neighbor must be removed, and a new synchronization with the neighbor must be performed. This will be addressed in section 3.6.

3.2.5 Examples

Consider the network of Figure 3.2, comprising 4 routers and three links; lk1 and lk2 are point-to-point links, and lk3 is a shared link. R1 is connected to a multicast source and, therefore, it is an originator router. The Figure indicates the interface costs of the underlying unicast routing protocol. According to these costs the root interfaces are i1-R1 with RPC=10, i1-R2 with RPC=20, i2-R3 with RPC=30, and i1-R4 with RPC=30.

Initial formation of the tree When the source becomes active, R1 also becomes ACTIVE and sends IamUpstream messages through its non-root interfaces, i.e. to routers R2 and R3. When these messages are received, both R2 and R3 set R1 as an UPSTREAM neighbor. However, since the message was received at R3 through a non-root interface, it will not trigger further message transmissions. Moreover, R3 becomes INACTIVE since, at this point, it has no UPSTREAM neighbor on its root interface. Contrarily, since R2 received the message on its root interface, it transmits an IamUpstream message through its non-root interface (i.e. on the shared link) containing its own RPC, which is 20. Moreover, R2 becomes ACTIVE since it has an UPSTREAM neighbor on its root interface. The message multicasted by R2 is received by R3 and R4, and both routers set R2 as an UPSTREAM neighbor, and both become ACTIVE since R2 is reachable through their root interfaces. Finally, R3 transmits an IamUpstream message in its non-root interface. This message will not trigger further message transmissions at R1 since it was received in a non-root interface. The AW at the shared link is R2, since only R2 sent an IamUpstream message on this link. By the same reason, R1 is the AW at link lk1. At link lk2, both routers transmitted IamUpstream messages, and R1 becomes the AW since it offered a lower RPC.

Change of interface role Suppose now that the interface cost of i1-R3 is changed to 5. Then, there

will be a change in interface roles: i1-R3 becomes root and and i2-R3 becomes non-root. Thus, R3 sends IamUpstream through its new non-root interface (i2-R3) and IamNoLongerUpstream through its new root interface (i1-R3). When R2 and R4 receive the IamUpstream (with RPC=15), they set R3 as an UPSTREAM neighbor and elect R3 as the new AW. When R1 receives the IamNoLongerUpstream, it sets R3 as a NOT UPSTREAM neighbor, and R1 keeps being AW.

AW failure Suppose now that the AW (currently R3) fails. In this case, R2 and R4 remove the state information relative to R3, and elect R2 as the new AW of the shared link, based only on the stored information. No control messages circulate in this case. Note that R4 has information that R2 is an UPSTREAM neighbor offering an RPC of 20; R2 considers itself AW since it has no UPSTREAM neighbor on the shared link.

Originator failure In the scenario with the previous interface costs (cost of i1-R3 equal to 5), suppose that R1, the originator router, fails. When R2 and R3 detect the failure, they become INACTIVE and transmit an IamNoLongerUpstream message at the shared link. When R2 receives the message from R3 it becomes UNKNOWN, and when R3 receives the message from R2 it also becomes UNKNOWN. R4 remains ACTIVE when it receives the first message, since it still has one UPSTREAM neighbor with an RPC lower than the RPC of R4, but changes to UNKNOWN when it receives the second message.

Failure of link lk1 Consider again the scenario with the initial interface costs (cost of i1-R3 equal to 30), and suppose that link lk1 fails. When R2 detects the failure it becomes UNKNOWN and transmits an IamNoLongerUpstream message at the shared link. Suppose that R3 receives this message before the unicast routing protocol finds the new path to the source (which implies a change in the role of its interfaces). In this case, R3 becomes INACTIVE, since it still has one UPSTREAM neighbor, but no longer in its root interface (which is R1). It will also send an IamNoLongerUpstream message through its non-root interface, to R1. At this point, R4 becomes UNKNOWN, since it lost its only UPSTREAM neighbor (which was R2). When finally the interface role changes, R3 becomes ACTIVE again and sends an IamUpstream message in its new non-root interface (i2-R3). At this point, R4 returns to ACTIVE state, since a new UPSTREAM neighbor appeared at its root interface. R2 also changes to ACTIVE if in the meantime its interface i2-R2 became root; otherwise, it changes to INACTIVE, moving only to ACTIVE when i2-R2 changes to root (due to the unicast routing protocol).

It is also possible that in R3 the unicast routing protocol triggers the interface role change before the arrival of the IamNoLongerUpstream message from R2. In this case, R3 keeps being ACTIVE but sends an IamNoLongerUpstream message through the previous non-root interface (i.e. to R1) and an IamUpstream message through the new non-root interface (i.e. to the shared link). The behavior of R4 depends on the order by which the messages sent by R2 and R3 are received: it becomes temporarily UNKNOWN if the message from R2 is received first, or keeps being ACTIVE if the message received from R3 is received first.

Importance of feasibility condition We illustrate now the importance of the feasibility condition. Consider the network of Figure 3.3. When the source is active the root interfaces are i1-R1, i1-R2 and i1-R3. Moreover, due to the initial transmission of IamUpstream messages, R3 considers R1 as UPSTREAM

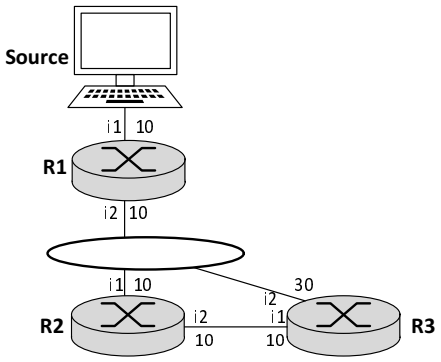

Figure 3.3: Importance of feasibility condition.

at interface i2-R3 and R2 as UPSTREAM at interface i1-R3, R2 considers R1 and R3 as UPSTREAM at interface i1-R2, and R1 considers R3 as UPSTREAM at interface i2-R1. Now suppose that the source stops transmitting multicast data. Consider first that the feasibility condition is ignored, in which case the ACTIVE state is only defined by having UPSTREAM neighbors at the root interface. In this case, R1 changes to INACTIVE and transmits IamNoLongerUpstream at the shared link. When R2 receives this message, it remains ACTIVE since it still has an UPSTREAM neighbor at its root interface, which is R3. Thus, it transmits no message to R3. When R3 receives the message, it removes R1 as UPSTREAM at interface i2-R3, but it remains ACTIVE since it has an UPSTREAM neighbor at its root interface, which is R2. Thus, both R2 and R3 remain ACTIVE, despite the source being no longer active. The problem is that R2 believes that it is possible to reach the source via R3, and R3 believes it is possible via R2: a routing loop is formed!

Consider now that the feasibility condition is included. When the IamNoLongerUpstream message arrives at R2, R3 is still considered UPSTREAM. However, R3 offers an RPC higher than R2: R3 offers 30 and R2 offers 20. Thus, R3 cannot be considered a parent of R2, and R2 becomes INACTIVE and sends IamNoLongerUpstream through i2-R2. When R3 receives this message it becomes UNKNOWN and sends IamNoLongerUpstream through i2-R3. Finally, when R1 and R2 receive this message they both remove R3 as UPSTREAM neighbor and become UNKNOWN. Thus, all routers understand that the source stopped transmitting multicast data.

3.3 Interest maintenance

The broadcast tree must be pruned from routers not interested in receiving multicast data, to save resources. In this section, we explain the protocol behavior that allows maintaining a multicast tree free of not interested routers.

3.3.1 Interest state of routers and interfaces

Interest state of routers and interfaces Regarding the interest on an (S,G) tree, a router can be in one of two states: INTERESTED, if the router is interested in receiving multicast data, or NOT INTERESTED,

if the router is not interested. The interest of a router depends on the assert state and on having downstream devices (routers or multicast receivers) interested in receiving multicast data on its non-root interfaces. We say that a non-root interface with downstream devices interested in receiving multicast data is in DOWNSTREAM INTERESTED state, and is in NOT DOWNSTREAM INTERESTED otherwise.

Definition of router interest according to interface forwarding state The interest state of a router is defined in terms of the forwarding state of its non-root interfaces: a router is INTERESTED if it has at least one non-root interface in FORWARDING state, and is NOT INTERESTED otherwise. Moreover, a non-root interface is FORWARDING if it is both AW and DOWNSTREAM INTERESTED.

Signaling interest per group or per source and group The interest of multicast receivers is signaled through the IGMP or MLD protocols, in case of IPv4 and IPv6, respectively. These protocols run between the multicast receivers and the routers to which they are directly connected to. In case of per-group interest (IGMPv2 and MLDv1), expressing interest on a group may impact all trees of that group; in case of per-source and per-group interest (IGMPv3 and MLDv2), expressing interest on a source and group may impact only the corresponding tree.

What interfaces store interest information Routers are required to store the interest state of their neighbors. However, this is not required at root interfaces, since these interfaces are not allowed to transmit multicast data and, therefore, do not need to know if their neighbors are interested or not. Moreover, regarding non-root interfaces, all interfaces store interest information, but only the AW is required to have it updated.

What interfaces are interested in receiving multicast data Non-root interfaces are not interested in receiving multicast data, since according to the RPF operation only root interfaces should receive it. Root interfaces may or may not be interested.

Interest according to tree state We consider that the router interest state is only meaningful in the ACTIVE state. Thus, a router ignores interest information received in INACTIVE or UNKNOWN states.

3.3.2 Interest messages

Message types The interest information is signaled through Interest and NoInterest messages. These messages are transmitted (i) through root interfaces to signal upstream neighbors about a router's interest in receiving multicast data, and (ii) through non-root interfaces to signal other interfaces attached to the same link about their lack of interest in receiving multicast data. Recall that non-root interfaces are not interested in receiving multicast data.

Building a tree according to interest When a tree is activated, the IamUpstream messages start flowing downstream and routers start discovering Upstream routers. When the first IamUpstream message is transmitted on a link, the interfaces attached to the link react by sending their interest information. Interest information must also be sent when the interest of a router changes, and triggered by other events.

Which neighbors should receive interest information As noted in section 3.3.1, only the AW is required to have the most recent information regarding the interest of the routers attached to a link. Thus, when its neighbors have some indication that their interest information might not be correct at the AW, they are required to send their interest information to the AW.

Why interest messages are unicasted to the AW Since only the AW is required to have the most recent interest information, the interest information is unicasted to the AW, and not multicasted (as in the case of upstream messages). Since interest messages are transmitted reliably through ACK-protection, unicasting saves in the number of transmitted ACKs.

Why interest information needs to be stored at all non-root interfaces The interest information must be stored at all non-root interfaces, despite the fact that only the AW is required to have the most recent information. This is needed since the AW election may not be simultaneous at all routers and, during a transient period where the AW is changing, different routers may have different views on who the AW is. Thus, it may happen that the new AW receives interest information while still believing it is AL; the router must store the interest information, since it will only be sent once by its neighbors. This shows that the arrival of interest information at an AW may precede its change to this state. This issue is illustrated through an example in section 3.3.6.

Can a router control the interest state of neighbors When a router sends an Interest or a NoInterest message to a neighbor, there is no guarantee that the corresponding information is stored at the neighbor, since a neighbor stores interest information according to its tree state. For example, when a router sends interest information to an INACTIVE or UNKNOWN neighbor the information is not stored. This contrasts with IamUpstream and IamNoLongerUpstream messages. When a router sends these messages to a neighbor, the neighbor stores the corresponding state for sure, independently of the tree state. Thus, while routers have full control over the upstream information stored at their neighbors, the same is not true for interest information. However, a router can detect when its interest is incorrectly stored at the AW, in which case it sends its interest information to the AW.

Double meaning of IamUpstream messages IamUpstream messages are only transmitted by non-root interfaces and, as noted above, non-root interfaces are not interested in receiving multicast data. Thus, receiving an IamUpstream message from a neighbor indicates that the neighbor is not interested in receiving multicast data. In our protocol, we allow the interest state of a neighbor to be set by the reception of IamUpstream messages.

3.3.3 Triggering events

The interest information can either be sent by root interfaces or non-root interfaces.

Root interfaces Root interfaces need to send interest information whenever (i) the interest of the router changes, (ii) the AW changes, (iii) the root interface changes, and (iv) an IamUpstream message is received from the current AW, which remains AW. The reason for the last event has to do with the

possibility that the AW changes state and loses its interest information without other routers noticing. This case is illustrated through an example in section 3.3.6.

Non-root interfaces Non-root interfaces are not interested in receiving multicast data, since according to the RPF operation only root interfaces should receive it. This fact has to be notified to the AW, if the interface itself is not the AW. There are two cases regarding the events that trigger the transmission of no interest information.

The first case is when the neighbors of the router consider it to be NOT UPSTREAM (because the interface never transmitted an IamUpstream message or because the last transmitted upstream message was IamNoLongerUpstream). In this case, a NoInterest message needs to be transmitted when: (i) an IamUpstream message from AW that remains AW is received, (ii) AW changes, or (iii) the interface changes from root to non-root.

The second case is when the neighbors of the router consider it to be UPSTREAM (because the router transmitted an IamUpstream message). In this case, the router can trust that its neighbors keep the correct interest information, even when the above events occur. Thus, there is no need to react to these events.

3.3.4 Configuring the initial downstream interest of routers

We allow that the network manager configures initially if each non-root interface has downstream routers (and only routers) interested in receiving multicast traffic. This, together with the IGMP or MLD protocols, determines if the interface is initially (i.e. before the circulation of any control messages) in DOWNSTREAM INTERESTED or NOT DOWNSTREAM INTERESTED state and, therefore, if the router is initially INTERESTED or NOT INTERESTED. The initial configuration is immediately superseded by the reception of interest messages from neighbors. The motivation for introducing this flexibility on configuring initially the interest of downstream routers has to do with the flooding of multicast data, which will be discussed in section 3.4.

3.3.5 Dealing with the possibility of losing data in transient state

When the AW changes, neighbors may need to send interest information to the AW, and this may introduce a delay in determining the correct forwarding state of the new AW. Thus, there may be temporary loss of data, if the new AW is initially NOT DOWNSTREAM INTERESTED (e.g. because it has outdated interest information) and takes some time to understand that there are downstream routers interested in receiving multicast data. This problem can be minimized by introducing some hysteresis in the FORWARDING state of the previous AW. Specifically, when a non-root interface changes from AW to AL state, it will be allowed to transmit data for an additional period of time.

Possibility of multicasting interest messages There are other alternatives to minimize this problem. One of them is the multicast of interest messages. This increases the probability that, upon being elected, the AW has the correct downstream state. However, as noted above, this requires more

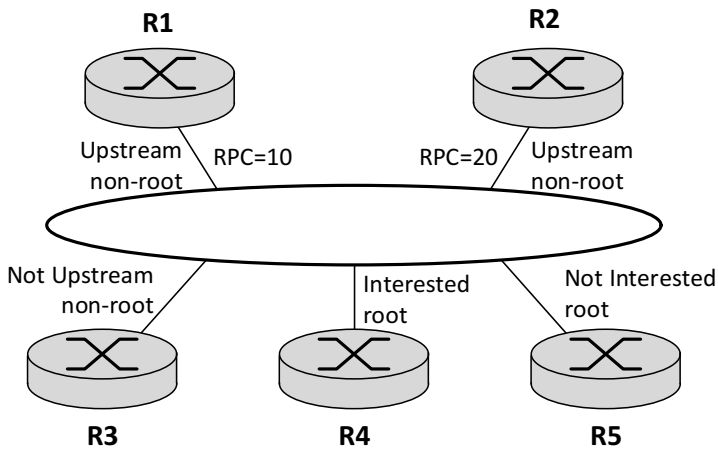

Figure 3.4: Interest example.

message transmissions since interest messages are ACK-protected. We have selected the unicasting alternative, since changes in the interest of routers occur more frequently than broadcast tree reconfigurations (e.g. changes of AW). Changes of interest are triggered by the interest of multicast receivers, and this is a common and legitimate event in multicast networks; broadcast tree reconfigurations are triggered by changes in the network itself (e.g. router failures, router additions, or changes in unicast costs), which are less frequent and less desirable.

Ensuring that the downstream interest is correct at all times The multicast alternative described above per se does not ensure that the downstream interest is correct at all times. For example, a router that changes from ACTIVE to INACTIVE and then again to ACTIVE would lose the interest state of its NOT UPSTREAM neighbors. Ensuring that the interest information stored at neighbors remains correct at all times requires multicasting interest messages and introducing additional complexity in the protocol. This solution is described in Appendix A.

3.3.6 Examples

Consider the shared link represented in Figure 3.4, which is part of a larger network. Consider that R1, R2, and R3 attach to the link through non-root interfaces, R4 and R5 through root interfaces, and there are no multicast receivers attached to the link. Moreover, R1 and R2 are UPSTREAM neighbors, R3, R4 and R5 are NOT UPSTREAM, R4 is INTERESTED and R5 is NOT INTERESTED. Furthermore, consider that R1 provides an RPC of 10 and R2 an RPC of 20.

Initial expression of interest When the source is switched on, suppose that R2 is the first to send an IamUpstream message on the link. In this case R2 is considered AW, and all routers have to express their interest. R5 and R3 unicast a NoInterest message to R2, and R4 unicasts an Interest message to R2. The behavior of R1 depends on its tree state when it receives the IamUpstream message of R2: if it is UNKNOWN or INACTIVE, it sends a NoInterest message; if is ACTIVE, it does nothing since it already sent its own IamUpstream message on the link.

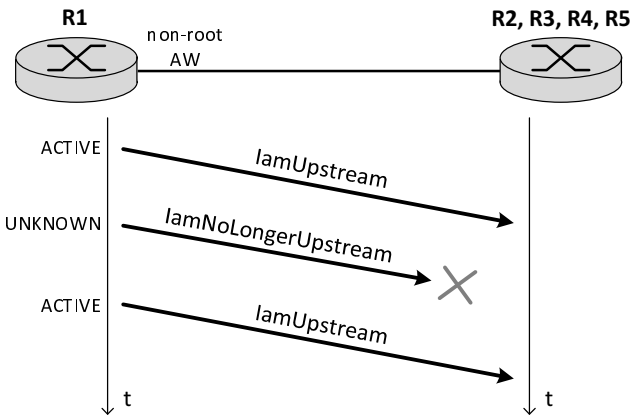

Figure 3.5: AW loses interest state without other routers noticing.

In any case, R1 will have to transmit an IamUpstream message on the link, which includes its RPC. When other routers receive this message, they learn that R1 is the new AW. In this case, R3, R4, and R5, restate their interest information by retransmitting the same interest messages, but now to R1. R2 sends nothing, since it is sure that R1 has the correct interest information about itself. Recall that a router is always sure that its neighbors store the correct upstream information about itself. In this case, R1 knowns that R2 is UPSTREAM and, because of the double meaning of this state, it also knows that R2 is not interested.

R4 becomes not interested Suppose now that R4 becomes NOT INTERESTED, leaving no one interested to receive multicast data at the link. In this case, R4 unicasts a NoInterest message to R1 (the AW). Then, the non-root interface of R1 becomes NOT DOWNSTREAM INTERESTED and, therefore, PRUNED. If R1 has no other non-root interface interested in receiving multicast data, it will change to being NOT INTERESTED and sends a NoInterest message through its root interface, to signal its parent on the tree that it no longer needs to receive multicast data.

R1 interface becomes root In relation to the initial scenario, suppose now that the interface of R1 becomes root. In this case, R1 sends an IamNoLongerUpstream message on the link, which forces the AW to change from R1 to R2. Thus, all other neighbors, including R1, have to express their interest to R2: R3 and R5 unicast a NoInterest message to R2, R4 unicasts an Interest message, and the message unicasted by R1 can be either Interest or NoInterest depending on the interest state of the router.

R1 retransmits IamUpstream on the link Again in relation to the initial scenario, suppose that R1 becomes INACTIVE (e.g. because it is no longer connected to the tree), sends an IamNoLongerUpstream message which is lost, and then changes back to ACTIVE and sends an IamUpstream message. This is illustrated in Figure 3.5. In this case, R1 remains AW but all neighbors, except R2, have to retransmit their interest. In fact, since R1 changed to INACTIVE its downstream interest information could have been removed, and that is why R3, R4, and R5 need to retransmit their interest. R2 does not need since it is sure that its upstream state is stored at R1: R1 knows that R2 is not interested (since it is an UPSTREAM neighbor).

Why interest information needs to be stored at all non-root interfaces Consider again the initial

scenario, and the sequence of events depicted in Figure 3.6. Suppose that R1 gets disconnected from the tree and, therefore, changes from ACTIVE to INACTIVE and transmits an IamNoLongerUpstream message. Moreover, suppose that this messages is received first by R3, R4, and R5, and only later by R2. Upon receiving the IamNoLongerUpstream message, the first routers elect R2 as the new AW and send immediately to R2 their interest messages. Suppose that when R2 receives these interest messages, it still did not receive the IamNoLongerUpstream message sent by R1 and, therefore, it still considers itself AL. Despite that, R2 will have to store the interest information, otherwise it will not have it when it finally receives the IamNoLongerUpstream message and realizes it is the new AW.

Role of interest in the forwarding state of point-to-point interfaces Consider the network of Figure 3.7, with three routers connected by point-to-point links. R1 is the originator and is the parent of both R2 and R3. In the point-to-point link connecting R2 to R3, each router considers the other as UPSTREAM and, therefore, as NOT INTERESTED. For this reason, interfaces i2-R2 and i2-R3 are NOT DOWNSTREAM INTERESTED, and are placed in PRUNED state. Interfaces i3-R2 and i3-R3 by being connected to interested receivers are placed in FORWARDING state. This causes R2 and R3 to transmit Interest messages through their root interfaces to the respective AW (i.e. to i2-R1 and to i3-R1). After this exchange of messages, R1 forwards traffic directly to R2 and to R3.

In case Receiver1 becomes not interested in receiving multicast data, detected by IGMP or MLD protocols, i3-R2 is placed in PRUNED state. This causes R2 to transmit a NoInterest message through its root interface to the link's AW, i.e. i2-R1. This interface becomes NOT DOWNSTREAM INTERESTED and is placed in PRUNED state.

Consider again the initial scenario (both receivers interested), but suppose that the interface cost of i1-R2 is set to 30. This causes R3 to become parent of R2. At lk1, each router considers its neighbor as UPSTREAM and NOT INTERESTED, and both interfaces are placed in PRUNED state. At lk3, i2-R3 is considered the AW, and i2-R2 is a root interface. Router R2 transmits an Interest message to i2-R3 and the interface is placed in FORWARDING state. Router R3 also transmits an Interest message through its root interface to the link's AW, which is i3-R1, and this interface is placed in FORWARDING state. The forwarding of traffic from R1 to Receiver1 is now via R3 and R2.

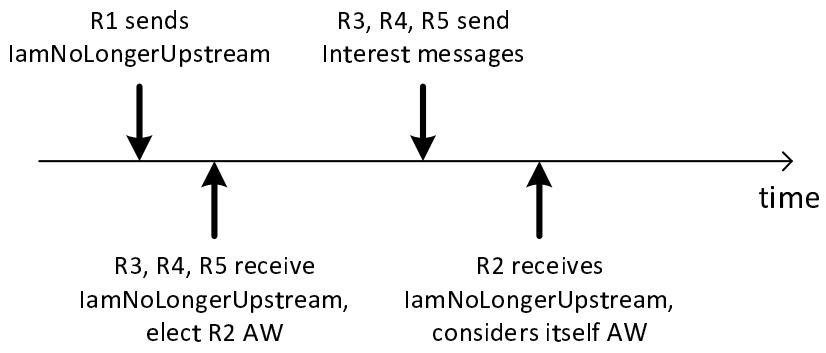

Figure 3.6: Sequence of events when new AW receives interest messages before realizing it is the AW.

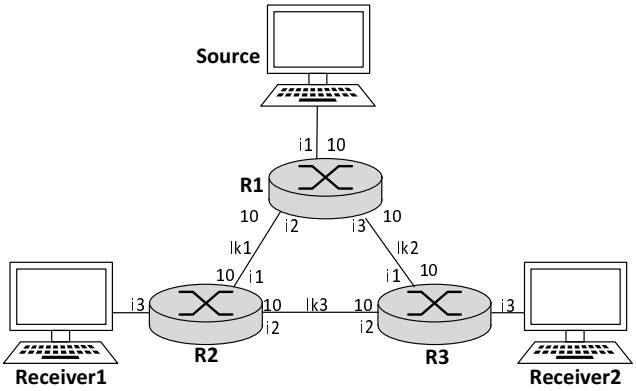

Figure 3.7: Role of interest in the forwarding state of point-to-point interfaces.

3.4 Data flooding

Data packets are forwarded down the multicast tree using the RPF technique. Packets should be received at the root interface of a router and transmitted on all its non-root interfaces in FORWARDING state. A data packet received on a non-root interface should always be discarded. In a well formed multicast tree, data packets are transmitted only once in each link. However, this may not be true during transient states.

The construction of the multicast tree is made using only control messages, and building the tree takes time. To avoid losing the initial data packets, the flooding of data must be allowed even if the tree is still not built.

Recall that, in a stable tree, multicast data is received at a root interface and transmitted through the non-root interfaces that are in FORWARDING state. Thus, if initial data is not to be lost, all non-root interfaces must be initially in FORWARDING state.

In section 3.3 the FORWARDING state of a non-root interface was defined in terms of its assert state and its downstream interest state. Specifically, an interface is FORWARDING if it is both AW and DOWNSTREAM INTERESTED.

As discussed in section 3.2.3, the AW state was defined for INACTIVE and UNKNOWN states, in addition to the ACTIVE sate. If a tree is UNKNOWN, all non-root interfaces are placed in AW state. If a tree is INACTIVE, non-root interfaces with no UPSTREAM neighbors are placed in AW state. In these cases, if there is an UPSTREAM neighbor, that neighbor is in ACTIVE state and, therefore, is in better conditions to become AW (since it is connected to the tree).

Regarding the downstream interest, its initial state can be partially configurable by the network manager (see section 3.3.4), and this provides a tool for defining whether or not data flooding is allowed before the tree is being built. A network manager can configure the initial downstream interest of routers connected to each non-root interface. If it declares the routers as interested, the interface becomes DOWNSTREAM INTERESTED. If it declares the routers not interested, the interface becomes NOT DOWNSTREAM INTERESTED, if there are also no downstream receivers interested, as signaled by the IGMP/MLD protocols.

Thus, to allow the initial flooding of data, the network manager must configure initially all non-root interfaces as DOWNSTREAM INTERESTED. Since these interfaces are initially in AW state (because the tree is initially UNKNOWN), they will be initially in FORWARDING state, and data will be flooded. In fact, this is similar to the initial data flooding of PIM-DM. During this period, data packets will not be lost, but may be transmitted more than once on links, which penalizes the link utilization.

The network manager may also prevent the initial data flooding (if there are no interested receivers), by configuring the interest of its downstream routers as not interested. In this case, data packets may be lost, but the link utilization will not be penalized.

3.5 Message sequencing

Control messages need to be processed at the intended receivers according the order they were transmitted, otherwise the protocol may behave incorrectly. For example, if an originator transmits an IamNoLongerUpstream message (because its source was switched off), and shortly after transmits an IamUpstream message (because its source was switched on again), the latter must be the last one to be processed at all other routers, otherwise the routers will keep believing that the source remains inactive. The underlying network does not guarantee the preservation of the transmission order.

In this section we explain how the message sequencing problem is addressed in our protocol.

3.5.1 Sequence Numbers

One way to solve the message sequencing problem is to use Sequence Numbers (SNs), i.e. to number the messages sequentially, such that whenever a new message needs to be transmitted, conveying more recent information, its SN is incremented by one. We assume the use of a linear space of SNs, starting at SN=1 and ending at a final value that depends on the number of bits used for SN encoding. Moreover, we assume that the protocol is designed such that outdated information is no longer useful, in which case only the highest SN received from a neighbor needs to be stored. Using a linear space of SNs poses the problem of what to do when the final SN value is reached. This event will be called SN overflow, and the way to deal with it is addressed in section 3.5.3.

SNs per message type and per tree The SNs are per message type, and not per individual message. The relevant message types are two: upstream messages (IamUpstream and IamNoLongerUpstream) and interest messages (Interest and NoInterest). A receiver needs to know if a received IamUpstream is more recent than a received IamNoLongerUpstream, and the same is true for Interest and NoInterest messages. Thus, routers must use the same SN space for IamUpstream and IamNoLongerUpstream messages, and the same SN space for Interest and NoInterest messages. These SN spaces can be independent but, as it will be discussed in section 3.5.2, it is possible that messages of different types share the same SN space.

Note that, in principle, there must be different SNs spaces for messages of different trees. Again, as it will be discussed in section 3.5.2, it is possible that messages of different trees share the same SN

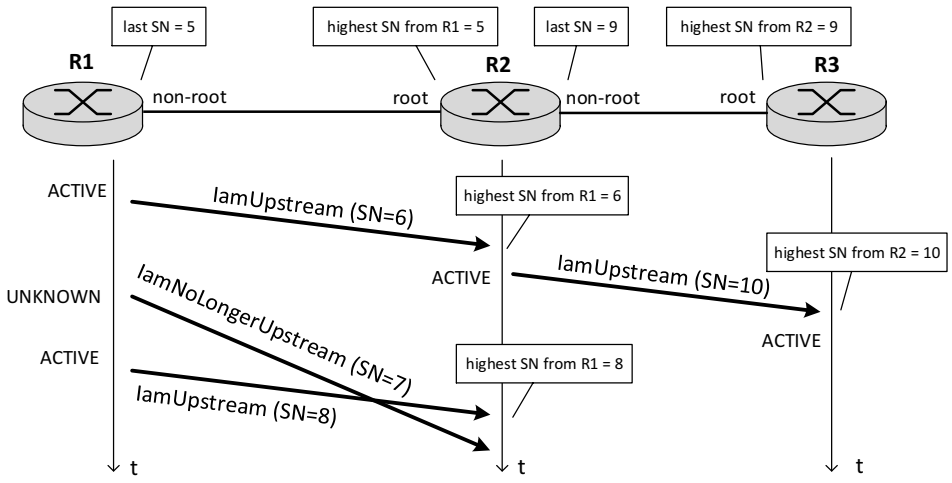

Figure 3.8: Local sequence numbers.

space.

Local message sequencing Message sequencing is implemented locally between neighbors. Each router stores the highest SN received from each neighbor and, in this way, can determine if a received message contains more recent information. Moreover, the SNs of the transmitted messages need not be correlated with the received ones. Thus, the SNs are local to each link.

Local ordering implies end-to-end ordering The local sequencing of messages ensures that messages transmitted between any two routers in the network (and not just between neighbors) are processed in the correct order. This follows immediately from the fact that sequencing is ensured locally.

Example To illustrate the use of SNs, consider the example of Figure 3.8. R1 is directly connected to a multicast source, and all routers are initially UNKNOWN regarding the corresponding tree. In this example, the source is switched on, then switched off, and finally switched on again. Suppose that, initially, the last SN used by upstream messages, was SN=5 at router R1, and SN=9 at router R2. When the source is switched on, R1 becomes ACTIVE and sends an IamUpstream message to R2 with SN=6. Upon receiving this message, R2 sets the highest SN received from R1 to SN=6, since the stored SN is lower (it is SN=5). Moreover, R2 becomes ACTIVE and sends an IamUpstream message to R3 with SN=10. Upon receiving this message, R3 sets the highest SN received from R2 to SN=10.

When the source is switched off, R1 sends an IamNoLongerUpstream message with SN=7 and, when it is switched on again, it sends an IamUpstream message with SN=8. Suppose that the order of arrival of these two messages is reversed, i.e. the IamUpstream arrives first at R2. At this point, R2 sets the highest SN received from R1 to SN=8, since the stored SN is lower (it is SN=6). The received message maintains the router in ACTIVE state and, therefore, no message is transmitted to R3. When the delayed IamNoLongerUpstream message is finally received at R2, it is ignored since its SN is lower than the highest one received so far (SN=8).

3.5.2 Enlarging the scope of SN spaces

Having to store SNs per neighbor, per message type, and per tree, requires considerable storing space. In our protocol, we devised a solution where (i) the messages transmitted by a router use a single SN space per interface, irrespective of message type and tree, and (ii) each router stores the highest SN received from each neighbor and the corresponding tree, irrespective of message type. If the received sequencing information is only stored per neighbor (and ignoring tree information), there is the danger that the information from one or more trees is missed. Thus, transmitted SNs are per interface and the received SNs are stored per neighbor and tree. In this way, the amount of sequencing information that needs to be stored is significantly reduced.

Example Figure 3.9 illustrates these optimizations, and motivates the need for storing the SNs of received messages per neighbor and per tree. R1 maintains information regarding two trees: (S,G1) and (S,G2). Suppose that R1 becomes ACTIVE for (S,G1), then becomes ACTIVE for (S,G2), and finally becomes UNKNOWN for (S,G1). Thus, it transmits first an IamUpstream message for (S,G1) with SN=1, then an IamUpstream message for (S,G2) with SN=2, and finally transmits an IamNoLongerUpstream message for (S,G1) with SN=3 and a NoInterest message for (S,G1) with SN=4. Since the transmitted SNs are per interface, R1 increments the SN by one whenever a new message is transmitted, irrespective of message type and tree. R2 has to store sequencing information per neighbor and per tree. Thus, when the first message arrives it stores information about the neighbor, the tree, and the SN, i.e. [R1, (S,G1), SN=1]. As before, the stored SN is the highest SN received so far. Suppose that the second and third messages are received in reverse order. When the (S,G1) IamNoLongerUpstream message is received, R2 updates the sequencing information to [R1, (S,G1), SN=3], and when the (S,G2) IamUpstream is received, it updates the sequencing information to [R1, (S,G2), SN=2]. Since these messages belong to different trees, its sequencing information is stored independently, as if the messages were received without order reversal.

However, suppose for a moment that the sequencing information is not stored per tree, but only per neighbor. In this case, R2 would store [R1, SN=1] when the (S,G1) IamUpstream message is received, [R1, SN=3] when the (S,G1) IamNoLongerUpstream message is received, and the (S,G2) IamUpstream message would be ignored since it has a lower SN than the stored one. Thus, the upstream information regarding (S,G2) and R1 will become incorrect at R2. This example illustrates the need for storing received SNs per neighbor and per tree.

Problem introduced by merging the SN space of message types Merging the SN space of message types can be problematic. There are circumstances where a router needs to transmit two messages of different types, to notify its neighbors about its upstream and interest state. By having a single SN space, in case the two messages are received in reverse order, the message with the lower SN would never be processed, since the message with higher SN already arrived. For example, if a router transmits an IamNoLongerUpstream message followed by a NoInterest message and the second message arrives first at the receiver, the router would not process the IamNoLongerUpstream message, and the corresponding upstream state will become incorrect.

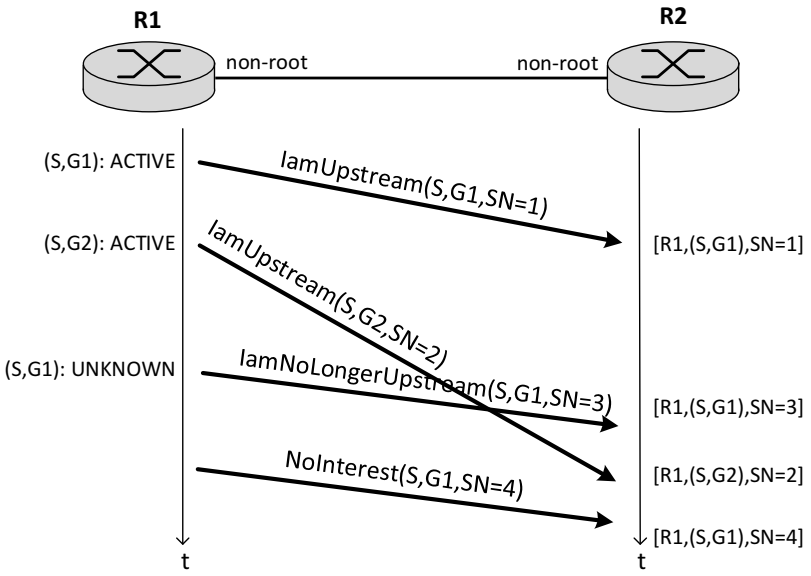

Figure 3.9: Transmitted SNs per interface and received SNs stored per neighbor and tree.

Increasing the semantic scope of control messages This problem can be solved by increasing the semantic scope of some control messages. Specifically, (i) IamUpstream messages are also interpreted as NoInterest, and (ii) Interest or NoInterest are also interpreted as IamNoLongerUpstream.

The first case was already introduced in section 3.3. The double meaning is possible since IamUpstream messages are only transmitted by non-root interfaces, and non-root interfaces are not interest in receiving multicast data. Moreover, due to this double meaning, the transmission of NoInterest messages is suppressed whenever an interface transmits IamUpstream messages.

The double meaning of the second case is made possible by the first case: since an UPSTREAM interface never transmits interest messages, receiving an interest message is an indication that the sending interface is NOT UPSTREAM. However, unlike the first case, we do not suppress the transmission of IamNoLongerUpstream messages. This is because interest messages are unicasted to the AW and, therefore, cannot convey upstream state to all neighbors. Thus, the double meaning is only used to ensure that the correct upstream state is set, in case of order reversal: an IamNoLongerUpstream message is always transmitted before the Interest or NoInterest messages but, if it arrives after and gets discarded (because it has lower sequence number), the arrival of the interest message already set the upstream state of the sending router as NOT UPSTREAM.

Example Consider the example of Figure 3.10. R1 and R2 connect to the link through a non-root interface, and R3 through a root interface; R1 offers a lower RPC and, therefore, it is AW. Suppose that the interface of R1 becomes root, but the router becomes interested in receiving multicast data. In this case, R1 determines that R2 is the new AW. R1 will then send an IamNoLongerUpstream message followed by an Interest message, with the Interest message having a higher SN. The IamNoLongerUpstream message is multicasted to all neighbors but the Interest message is only unicasted to the AW. Now suppose that the messages are received at R2 in reverse order. In this case, the IamNoLongerUpstream message is ignored at R2 since, when it arrives, a message with a higher SN has already been processed.

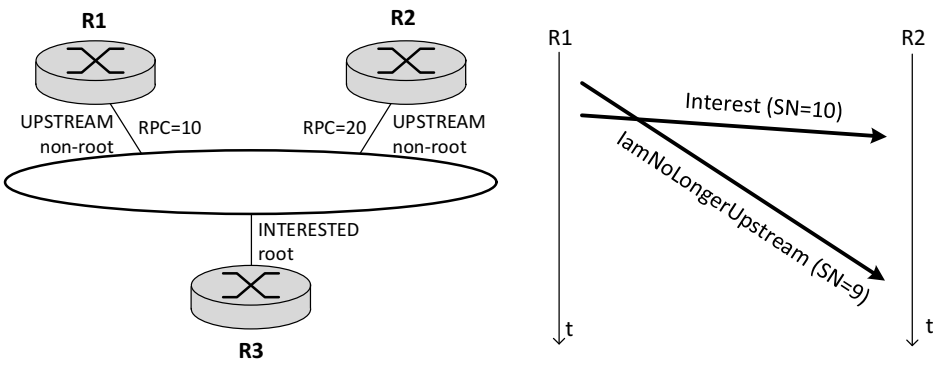

Figure 3.10: Merging the semantic scope of upstream and interest messages.

However, due to the double meaning of the Interest message, the correct upstream state regarding R1 is set at R2. When R2 receives the Interest message it sets the state of R1 as being NOT UPSTREAM and INTERESTED (and realizes it is the new AW). R3 does not receive the Interest message (nor it needs, since it is not AW). When it receives the IamNoLongerUpstream message it elects R2 as the new AW and sends it an Interest message.

If the Interest message had not a double meaning, the upstream state of R2 regarding R1 would become incorrect: R2 will keep believing that R1 is UPSTREAM and, therefore, the AW.

3.5.3 SN restart

Routers are only required to store the highest SN received so far. In this setting, a message received with an SN lower or equal than the stored one is ignored, since it is considered to include outdated information. However, there are situations where the SN needs to be restarted from the initial value, i.e. SN=1. In this case, messages with a lower SN can indeed include fresher information.

SN restart scenarios There are two situations where the SN has to be restarted:

- Reboot of router (due to configuration action or failure) - When a router reboots it loses all its information, and message sequencing has to be started from the initial SN value.
- SN overflow - When the SN reaches its final value, the next SN is the initial SN value.

BootTime When there is an SN restart, messages with lower SN may be ignored despite including fresher information. One solution to this problem is to include in transmitted messages an additional sequence number that is incremented when an SN restart event occurs (reboot or SN overflow). This is similar to the Extended Sequence Numbers used by OSPFv2 [26] and IS-IS [27] to prevent replay attacks. This sequence number is called BootTime.

BootTime implementation alternatives The BootTime sequencing must be preserved, even in the case of reboots. There are two ways of implementing it. One solution is to use the router's clock, and make the BootTime equal to the clock value when an SN restart event occurs. This solution is suggested in [27]. Another solution is to rely on the router's non-volatile memory to store the last BootTime value.

In this case, the BootTime is a sequence number that is incremented by one when an SN restart event occurs. Moreover, a new BootTime must replace the previous one stored in the router's non-volatile memory. This solution is suggested in [26, 27]. In our implementation we have used the first solution.

BootTime precedence over SN The BootTime is transmitted in all control messages together with the SN. The first takes precedence over the former in deciding which message is fresher. Thus, a message with a higher BootTime is always considered fresher. When two messages have the same BootTime, the one with highest SN is considered fresher.

Storage of received BootTime The receivers only have to store one BootTime per neighbor, unlike SNs which have to be stored per neighbor and per tree.

BootTime per interface When a router receives a control message from a neighbor with a BootTime higher than the previous one, it must resynchronize with the neighbor. The synchronization process is described in section 3.6. The BootTime is per interface (and not per router), to avoid synchronizing all interfaces in case of SN overflow.

Example The example of Figure 3.11 illustrates the use of the BootTime (referred as BT). In the Figure, R1 becomes UNKNOWN and transmits an IamNoLongerUpstream message with SN=1000 followed by a NoInterest message with SN=1001. Both messages have BT=1 and therefore the second message is considered fresher. Based on these messages, R2 sets the state of R1 as being NOT UPSTREAM and NOT INTERESTED. Now suppose that R1 reboots and changes to ACTIVE. It will then send an IamUpstream message with the initial SN value (SN=1) but with a BootTime incremented by one (BT=2). The previous SN was lost due to the reboot, but the BootTime was not since it was stored in R1's non-volatile memory. The IamUpstream message is considered fresher than the previous one since it has a higher BootTime. Thus, router R2 sets correctly the upstream state of R1 as being UPSTREAM. If the BootTime was not used, the last message would be discarded, and R2 would keep believing, incorrectly, that R1 is NOT UPSTREAM.

3.5.4 Removing sequencing information

As discussed in section 3.5.2, a router is required to maintain SN information per neighbor and per tree, which can occupy significant memory resources. Given that multicast sources (and the corresponding trees) can be switched on and off fairly frequently, an important question is for how long the SNs need to be stored at routers.

Removing SNs when there is a network delay bound SNs need to be stored as long as there is the possibility of an older message being received. Suppose that there is an upper bound on the delay between the transmission of a message and its reception by all routers; we denote this bound by T. In this case, the SN of a message can be safely removed T time units after its reception. This is illustrated in Figure 3.12. R1 transmits an IamNoLongerUpstream message with SN=1 followed by an IamUpstream message with SN=2, and they are received in reverse order. When the IamUpstream message arrives at R2, R2 stores its SN. This SN needs only be stored for T time units since it is guaranteed that after

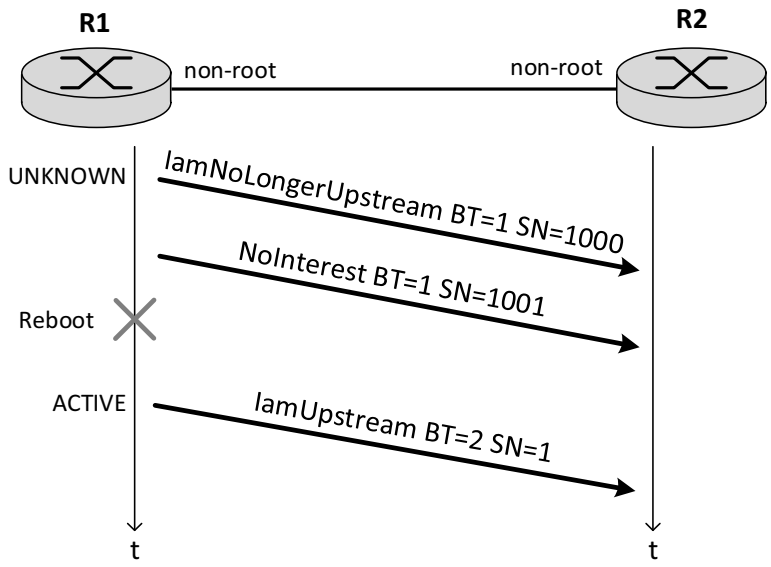

Figure 3.11: Use of BootTime.

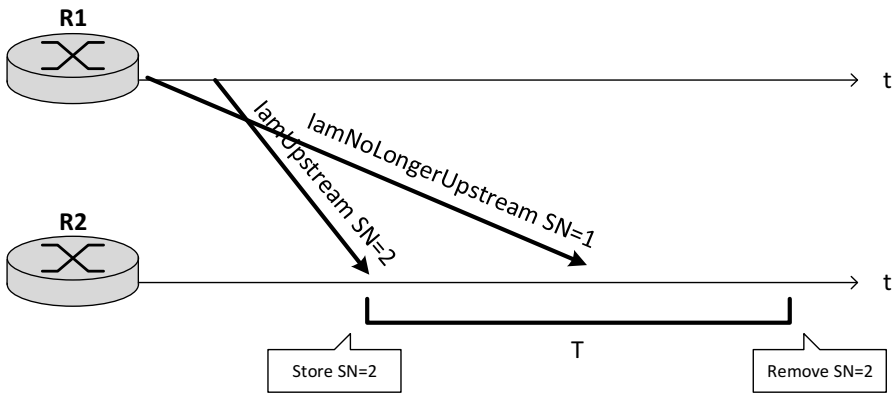

Figure 3.12: Removing SNs when there is a network delay bound.

this period no message with a lower SN will be received. Thus, the first message, despite arriving after the second one, arrives surely during this period, and is ignored since it has a lower SN.

The possibility of replay attacks However, the network may be subject to replay attacks. An attacker may capture a message transmitted between routers, and retransmit it later. In this case, if the message is retransmitted to a router after the SN being removed, the router will have to accept it, and this can cause malfunction. Unfortunately, there is no delay bound for a replay attack and, to avoid these attacks, SNs must be stored indefinitely.

The CheckpointSN We devised a solution to minimize the amount of stored sequencing information. Recall that transmitters perform message sequencing per interface (irrespective of tree) but receivers are required to store SNs per transmitter and per tree. In our solution, a router sends periodically to each neighbor information on the highest SN that has been acknowledged so far, such that all SNs lower than it have also been acknowledged. This SN will be called the CheckpointSN. Note that a message with a higher SN can be acknowledged before a message with a lower one; the CheckpointSN is such that all

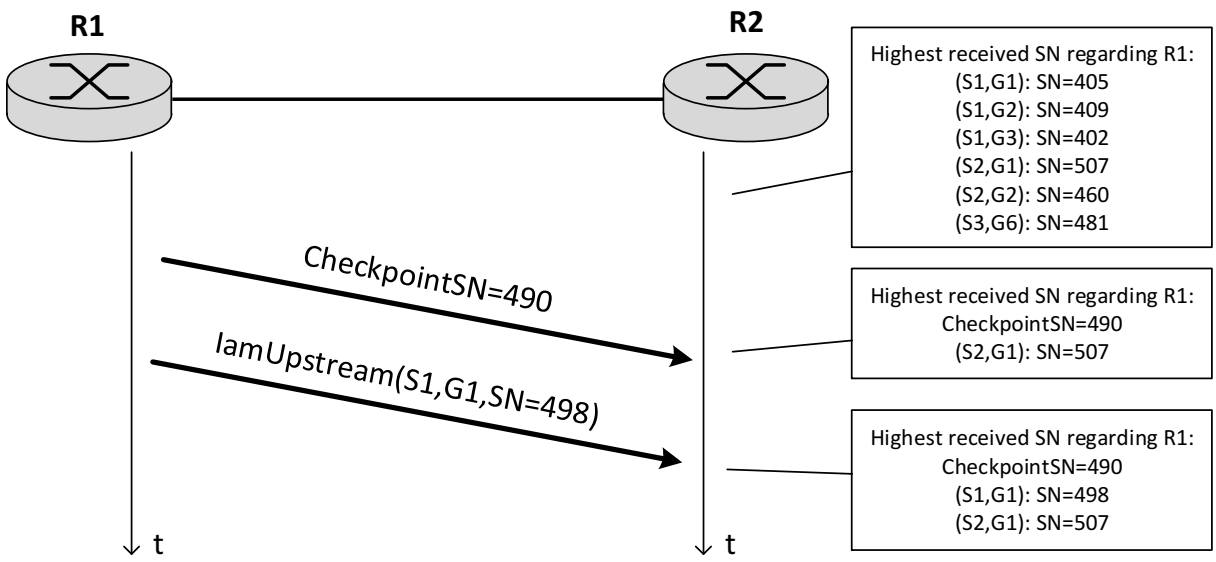

Figure 3.13: Use of CheckpointSN.

SNs lower than it have already been acknowledged. When a router receives the CheckpointSN, it can be sure that the transmitter will not send any message, for any tree, with a lower SN. Thus, it can remove all sequencing information of trees that have an SN lower than the CheckpointSN and, from then on, assume that the stored SN for any other tree (for which it might receive information in the future) equals the CheckpointSN.

Example Figure 3.13 gives an example. Initially R2 stores SNs of 7 trees regarding R1. Assume that the CheckpointSN is 490. This means that all messages transmitted by R1 with a SN equal or lower than 490 have already been acknowledged. When R2 receives the CheckpointSN, it can remove the sequencing information of 6 trees, the ones for which the highest received SN is lower than 490. Moreover, the CheckpointSN is stored and becomes the SN of all unknown trees. If later R1 sends a message for (S1,G1) with SN=498, R2 will store again the sequencing information of this tree, since the corresponding SN is higher than the CheckpointSN.

Control messages used for the CheckpointSN transmission The CheckpointSN can be transmitted using a specific control message type. This message needs not be ACK-protected since failure of receiving it would not compromise protocol correctness. Another option is to include the CheckpointSN in Hello messages. However, since the frequency of Hello transmissions is much higher than the one required for CheckpointSN transmissions, the CheckpointSN should only be included in some Hello messages. In our implementation we have adopted the second solution.

3.6 Synchronization of tree information

A router attaching the network must obtain information from its neighbors, to start receiving multicast data as soon as possible. In particular, it must learn which neighbors are UPSTREAM neighbors for

each tree currently active in the network, so that it can connect to these trees. The synchronization is also required in other situations.

3.6.1 Triggering events

The synchronization is triggered by the following events: (i) new neighbor detected, (ii) known neighbor rebooted or its SN overflowed, (iii) bidirectional communication with known neighbor temporarily interrupted.

The first case occurs when a control message sent by an unknown router is received. Note that any control message, and not only Hello messages, can be used to detect a new neighbor. The second case occurs when the neighbor sends a control message with a BootTime higher than the one currently stored for that neighbor. Note that all control messages include the BootTime. The third case occurs if one router, say R1, considers that a neighbor, say R2, is disconnected, but the opposite is not true (e.g. because of temporary loss of Hello messages in only one direction). In this case, R1 will trigger a synchronization as soon as it resumes receiving control messages from R2.

3.6.2 Sync messages

When two neighbors synchronize, they send to each other the list of trees for which they consider being UPSTREAM in relation to the neighbor, and the corresponding RPCs. A router considers itself as UPSTREAM regarding a tree in relation to a neighbor if (i) it is ACTIVE for that tree, and (ii) it is connected to the neighbor through a non-root interface.

This information is exchanged in Sync messages. More than one Sync message may need to be transmitted to a neighbor, depending on the number of trees for which the router considers itself as being UPSTREAM regarding that neighbor. Moreover, the transmission of Sync messages must be protected. These features are similar to the initial exchange of link state information that occurs in OSPF [28]. Thus, we use a procedure similar to the so-called database description process of OSPF. This process will be detailed in section 3.6.5.

3.6.3 Consistency of the synchronization process

While a router is synchronizing with a neighbor, upstream or interest messages susceptible of changing the state of one or more trees being communicated to the neighbor may arrive at the router. Thus, measures have to be taken in order to ensure the consistency of the synchronization process.

Taking a snapshot of the tree information When a synchronization is started with a neighbor, the router takes a snapshot of all trees for which it considers itself being UPSTREAM regarding the neighbor. The snapshot will be completely transmitted to the neighbor, even if the tree information changes during the synchronization period. For example, a router may initially consider itself as UPSTREAM on a tree and then become NOT UPSTREAM during the synchronization period; in this case, the information on the tree will still be communicated to the neighbor.

The SnapshotSN Moreover, to distinguish among different synchronization periods, we introduce a new sequence number, called SnapshotSN. When a synchronization starts, the SN of the transmitting interface is incremented by one and this value is considered the SnapshotSN. Thus, the SnapshotSN belongs to the SN space of transmitting interfaces. The SnapshotSN along with the BootTime uniquely identify a synchronization period, and are both transmitted in Sync messages. These two SNs allow ignoring outdated Sync messages, which were transmitted during previous synchronization periods. As it will be discussed later, Sync messages have to include both the BootTime and SnapshotSN of the sending router and of its neighbor.

Moreover, when a Sync message is received, the SnapshotSN is assigned to all trees, as the highest SN received so far, except for the trees where the receiver has already tree information sent by the transmitter with a higher SN. Thus, the SnapshotSN allows ignoring all upstream and interest messages, of all trees, received with an SN equal or lower than SnapshotSN.

Once a snapshot is taken and a synchronization starts, subsequent upstream or interest messages may arrive at a router, which changes the upstream state being communicated to a neighbor. Recall that the snapshot is surely delivered to the neighbor, as taken. If these state changes trigger the transmission of new upstream or interest messages to the neighbor, the messages can be sent concurrently with the synchronization process. The messages will have an SN higher than the SnapshotSN and, therefore, will be accepted by the neighbor as having fresher information.

Example Figure 3.14 illustrates the synchronization process. We denote the SnapshotSN by SSN. R1 is initially ACTIVE regarding trees (S1,G1) and (S2,G2). When the Hello message is received by R1, R1 discovers the new neighbor and initiates the synchronization process. At this point, R1 takes a snapshot of the tree information that it will send to R2. Since R1 is ACTIVE regarding trees (S1,G1) and (S2,G2), and its interface with R2 is non-root for both trees, the snapshot includes both trees. Suppose that, when the synchronization starts, the SN of R1's non-root interface is SN=8. In this case, the interface SN is incremented by one and the SSN is given the interface SN value, i.e. SSN=9. Then, R1 transmits to R2 a Sync message with the tree snapshot and SSN=9. Suppose that now R2 receives an old Sync message, transmitted in a previous synchronization period with R1, with SSN=2. This message is rejected and does not trigger a new synchronization since its SSN is lower than the stored SSN. Suppose also that R2 receives an old IamUpstream message advertising an (S3,G3) tree with SN=3. This tree is unknown at R2, but it is rejected, since the SN of the IamUpstream message is lower than the current SSN. Finally, suppose that R1 receives an (S1,G1) IamNoLongerUpstream message from its parent on the tree which makes the router to become INACTIVE or UNKNOWN regarding the tree, during or after the synchronization period. R1 then sends an IamNoLongerUpstream message to R2 with SN=10. This message is accepted and processed by R2, since its SN is higher than the stored SSN.

Including the BootTime and SnapshotSN of the router and of the neighbor in Sync messages The Sync message transmitted by a router must include the BootTime and SnapshotSN of the sending router, and also of its neighbor (if known when the message is transmitted). A router receiving this message can acknowledge that the neighbor knows its BootTime and SnapshotSN. This avoids accepting outdated Sync messages when a router loses the sequencing information (BootTime and SnapshotSN) relative to

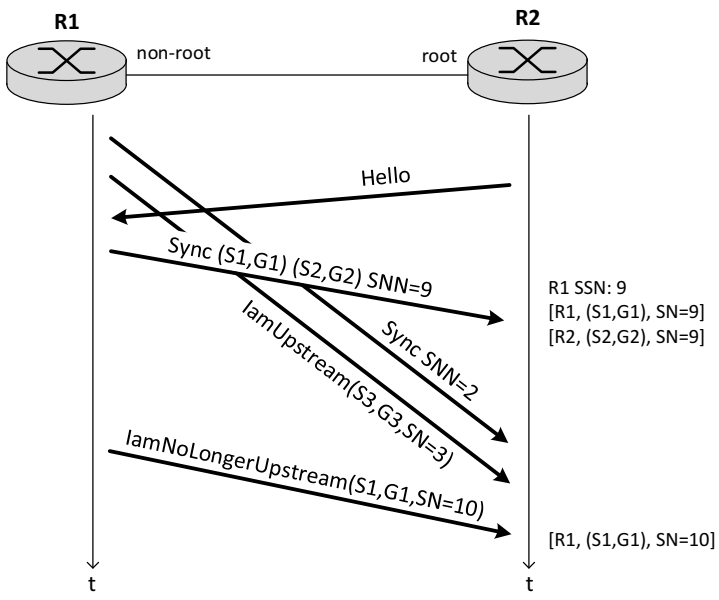

Figure 3.14: The tree's snapshot and the SnapshotSN.

previous synchronizations with a neighbor (e.g. because of temporary loss of connection). This mutual acknowledgment allows routers to ascertain that they are exchanging Sync messages belonging to the same synchronization period. We denote the BootTime and SnapshotSN of the sending router by myBT and mySSN, and the BootTime and SnapshotSN of the neighboring router by neiBT and neiSSN.

Figure 3.15 illustrates the consequences of not including the BootTime and SnapshotSN of the neighboring router in Sync messages. In Figure 3.15.a R1 sends initially a Sync message to R2 with myBT=1 and mySSN=10, and R2 replies with another Sync message with myBT=7 and mySSN=55. Suppose that R1 needs to retransmit its Sync message and one retransmission is delayed. In the meantime, R2 reboots and receives the retransmission from R1. At this point, R2 can only accept the message despite being from a previous synchronization and possibly having outdated information. In Figure 3.15.b the Sync messages include the BootTime and SnapshotSN of the sending router and of its neighbor. The message retransmitted by R1 arrives with myBT=1 and mySSN=10, and with neiBT=7 and neiSSN=55. However, since R2 rebooted its BootTime and SnapshotSN no longer coincide with the ones received in the Sync message sent by R1: the BootTime is 8 and the SnapshotSN is 2. Thus, the message is rejected.

3.6.4 Relationship with interest information

Information regarding the interest is not exchanged during the synchronization process. This is because interest information is not useful for all routers. Thus, the exchange of interest information follows the synchronization between routers. After a synchronization, a router may discover trees for which the neighbor is AW, that may trigger the unicast of interest messages to the neighbor, according to the procedure defined in section 3.3.

The snapshot carried in Sync messages is equivalent to receiving (one or more) IamUpstream mes-

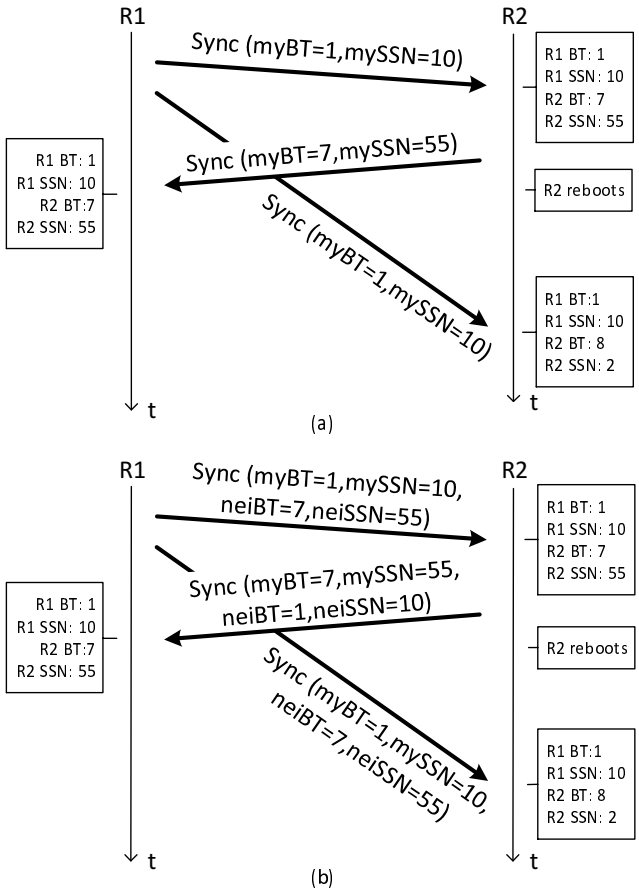

Figure 3.15: Including the BootTime and SnapshotSN of the sending router and of its neighbor in Sync messages.

sages, since the neighbor is telling that it is UPSTREAM for the trees contained in the snapshot. Thus, as when these messages are received, a router learns from a snapshot that the neighbor is not interested in receiving multicast data from the trees included in the snapshot.

The exchange of interest information following a synchronization between two routers, R1 and R2, has the following cases:

- If R1 includes a tree in its snapshot and R2 does not, then R1 is considered UPSTREAM (by R2) and R2 is considered NOT UPSTREAM (by R1). If R2 determines that R1 is the new AW for that tree, then R2 must send to R1 an interest message.

- If R1 and R2 include the same tree in their snapshots, then they are both considered UPSTREAM and NOT INTERESTED by their neighbors, and no interest message is sent.

- If R1 and R2 do not include a tree in their snapshots, then they are both considered NOT UPSTREAM by their neighbors, and none of them sends an interest message because none of them is considered AW.

3.6.5 Synchronization protocol

As referred above the protocol used for the exchange of Sync messages follows closely the database description process of OSPF. Similarly to OSPF, a router may have to transmit several Sync messages due to the size of the tree snapshot, and the message transmissions must be protected. The routers elect a Master while exchanging the first Sync messages and the Master controls the subsequent communications.

The Master is the router that initiates the synchronization, and Slave is the other router. If both routers initiate the synchronization at the same time and declare themselves as Master, the one with highest IP address becomes Master. To control this election a Master flag is included in all Sync messages, and is set whenever the transmitting router believes being the Master.

The transmission of Sync messages is protected by a Stop-and-Wait protocol controlled by the Master. The Sync messages are numbered using a sequence number called SyncSN, started at SyncSN=0. The Master is responsible for incrementing the SyncSN whenever a new Sync message is transmitted. When the Slave receives a Sync message from the Master, it must reply with another Sync message containing the same SyncSN. The Master retransmits a Sync message whenever the corresponding reply from the Slave is not received within a timeout period. This ensures reliable transmission of the Sync messages sent by the Master and the Slave.

The exchange of Sync messages ends when the routers have no more information to send to each other. To control this process, Sync messages include the More flag, which is set whenever the transmitting router has more information to send. Specifically, the More flag is only cleared when the reception of all transmitted snapshot fragments have been acknowledged. The synchronization ends when the Master transmits a Sync message with the More flag cleared, and the Slave replies with a Sync message with the same SyncSN and the More flag also cleared.

In order to guarantee the reliable transmission of the BootTime and SnapshotSN of both routers in Sync messages, the synchronization process must terminate necessarily with a SyncSN greater than 0. Thus, if the first two Sync messages exchanged by the routers (where SyncSN=0) have the More flag cleared (indicating that the routers have no information to send), a new round of messages is required. By having at least two rounds of Sync messages, we guarantee that each neighbor correctly acknowledged the exchanged sequence numbers.

Example Figure 3.16 illustrates the synchronization between two routers. In this example, R1 was already connected to the network and R2 is switched on. R1 considers itself UPSTREAM in relation to R2 for 5 trees. The initial BootTimes of R1 and R2 are denoted by BT1 and BT2, and the initial SNs are 50 for R1 and 0 for R2. Consider that each Sync message can only include information regarding 3 trees. In a real system the number would be much larger, and dependent on the interface's MTU and on the size of the Sync header. We denote the BootTime and SnapshotSN of the sending router by myBT and mySSN, and the BootTime and SnapshotSN of the neighboring router by neiBT and neiSSN.

When R2 is switched on, it obtains its BootTime BT2 and transmits an Hello message including this information. When R1 receives this message, it starts the synchronization process with R2. It assumes

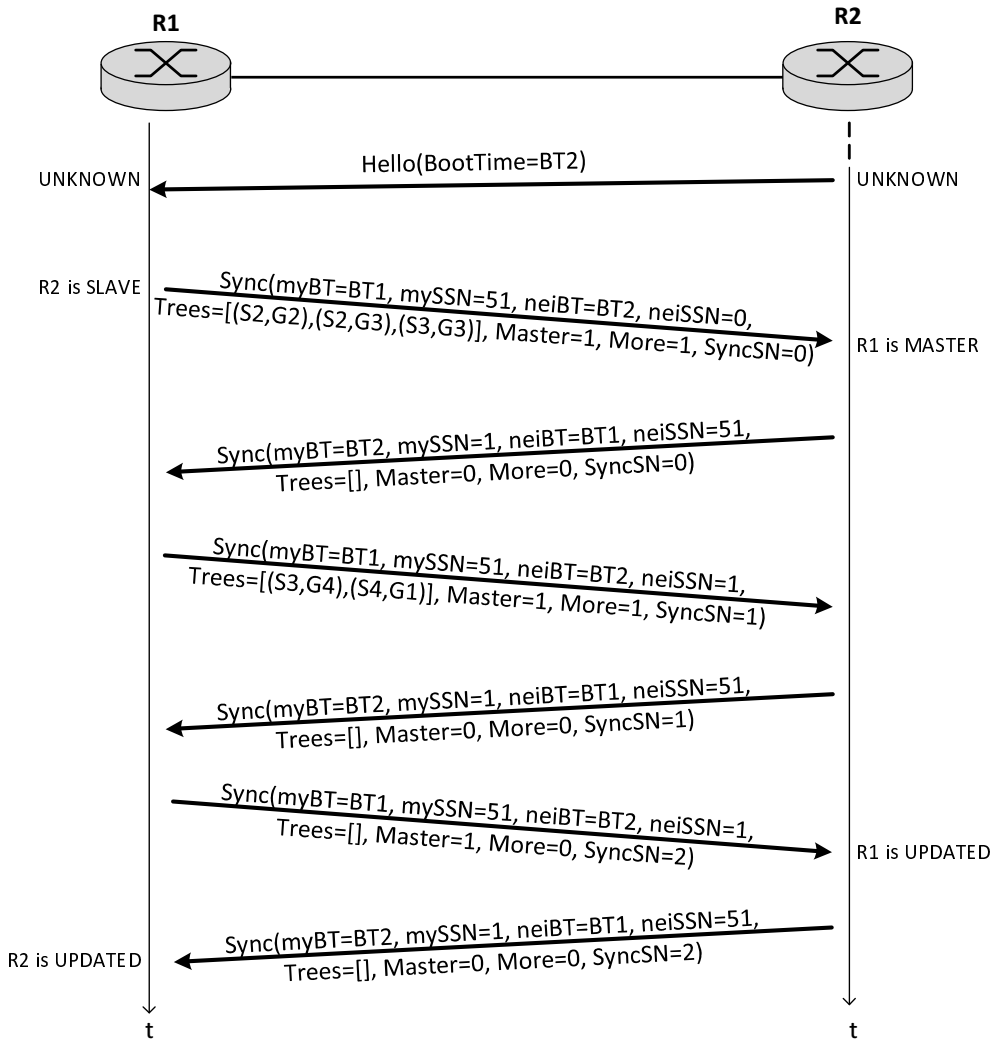

Figure 3.16: Synchronization protocol example.

itself as Master, takes a snapshot of all trees for which it considers itself UPSTREAM regarding R2, which includes 5 trees, and obtains its SnapshotSN as the current SN plus one, i.e. SnapshotSN=51. It then sends a Sync message to R2 containing information on the first 3 trees of the snapshot, and myBT=51, mySSN=51, and neiBT=BT2; the neiSSN is empty since it is unknown at this point. Since R1 believes being the Master and has additional information, it sets the Master and More flags. Moreover, since this is the first message sent by the Master, SyncSN=0.

When R2 receives this message, it starts its synchronization process with R1, assumes itself as Slave since R1 already declared itself as Master, and sets its SnapshotSN as SnapshotSN=0+1=1. It then replies sending a Sync message with the same SyncSN of the Master, i.e. SyncSN=0, and myBT=BT2, mySSN=1, neiBT=BT1 and neiSSN=51. The Master flag is cleared, since R2 is Slave, and the More flag is also cleared, since R2 has no tree information to send to R1.

R1 then receives this message and learns the SSN of R2. Since the remaining information is correct, i.e. neiBT and neiSSN coincide with R1's BT and SSN, and myBT coincide with the BT already stored for R2, R1 proceeds the synchronization process. It sends the next Sync message, with SyncSN

incremented by one, i.e. SyncSN=1, and information on the remaining 2 trees. When R2 receives this message it replies with a Sync message with the same SyncSN.

Finally, when R1 receives this message, it sends a Sync message with SyncSN=2 and the More flag cleared, since the transmission of its snapshot has now been completely acknowledged.

When R2 receives this message it terminates the synchronization process with R1 successfully and sets R1 as UPSTREAM neighbor for the 5 trees included in the snapshot. It also replies with a Sync message having SyncSN=2. When R1 receives this message it also terminates the synchronization process with R2 successfully.

Neighbor states regarding the synchronization process A router can consider a neighbor in one of four states regarding the synchronization process:

- UNKNOWN - unknown neighbor;
- MASTER - ongoing synchronization process and neighbor is Master;
- SLAVE - ongoing synchronization process and neighbor is Slave;
- UPDATED - neighbor has successfully synchronized.

A neighbor is initially in UNKNOWN state, until it is discovered through the reception of a control message. Then a synchronization is initiated and the neighbor changes to the MASTER or SLAVE states, depending on which router started the process. When the synchronization finishes successfully the neighbor changes to the UPDATED state. Finally, the neighbor returns to the UNKNOWN state, irrespective of the current state, if the neighbor is declared dead.

Triggering the synchronization process in case of SN overflow When a router suffers SN overflow at one interface, it obtains a higher BootTime and, therefore, is required to resynchronize with its neighbors. In this case, we require that the synchronization is only initiated by the neighbors. This allows avoiding to attempt synchronization with neighbors that failed but for which the failure is yet to be detected.

Establishing neighborhood relationships through the synchronization process The protocol allows that routers establish neighborhood relationships using only Sync messages. This speeds up the convergence of the protocol. As already noted, any control message can be used to detect the presence of a new neighbor. Moreover, we consider that two routers that have successfully synchronized have also established a neighborhood relationship. For that to be possible the Hello periodicity needs to be communicated during the synchronization process. This is required since, once a router establishes a neighborhood relationship, it needs to control the liveness of the neighbor, and for that it needs to know its Hello periodicity. To minimize the overhead this information is only included in Sync messages with the More flag cleared, which are the last ones to be exchanged during a synchronization process.

Validity of tree information exchanged during synchronization The tree information is only considered valid when the synchronization process finishes since, at this point, there is the guarantee that both routers are alive and have exchanged the tree information successfully. This avoids accepting tree information as valid sent by (i) a router that fails during the synchronization process and (ii) an attacker that replays Sync messages on behalf of a failed router.

3.7 Reliable message transmission

The control messages have to be transmitted reliably between neighboring routers. Hello messages are transmitted periodically, and this acts as a reliability mechanism. Sync messages are protected using a Stop-and-Wait protocol, as discussed in section 3.6. Upstream and interest messages use an ACK-protection mechanism, which is the main subject of this section. In this case, a router sets a timeout timer whenever it transmits a message to a neighbor and, upon receiving the message, the neighbor replies with an ACK message containing the SN of the received message. If the router does not receive the ACK within the timeout period it retransmits the message, and this procedure is repeated until the ACK is finally received. The timeout timer is called RetransmissionTimer.

Acknowledgment of multicasted messages Upstream messages are multicasted and interest messages are unicasted. Thus, in order to consider that an upstream message has been correctly received, an ACK must be received from all neighboring routers. ACK messages are unicasted to the router that transmitted the message being acknowledged.

Suppressing ACK transmissions The reception of a message needs not be acknowledged when the message conveys older information relative to previously received messages. Specifically, an ACK has only to be sent for a message received with an SN higher or equal than the SN stored for the corresponding neighbor and tree.

Suppressing pending ACKs There are several cases where the need for receiving an ACK of a transmitted message can be suppressed. This happens when a router is waiting for the ACK (or ACKs) of a previously transmitted message, and transmits a new message conveying fresher state. It can also happen in cases where the transmitted messages convey double meaning. There are three cases:

- When a new upstream message is transmitted, the pending ACKs related with older messages (with lower SN) of the same tree can be suppressed.
- When a new interest message is unicasted to some neighbor, the pending ACKs related with older messages (with lower SN) of the same tree transmitted to that neighbor can be suppressed.
- When a router starts a synchronization with some neighbor, the pending ACKs related to control messages transmitted previously by the router to that neighbor (unicasted or multicasted), i.e. with an SN lower than the SnapshotSN, can be suppressed.

Including the BootTime and SnapshotSN of the sending router and its neighbor in ACK messages The BootTime and the SnapshotSN of the sending router and its neighbor must be included in ACK messages. This is required due to the possibility of having upstream and interest messages transmitted concurrently with a synchronization process. In fact, it may happen that an ACK acknowledges a message that placed state information in a neighbor before the last synchronization with that neighbor; in this case, the state information may be removed without notice. Including the BootTime and SnapshotSN in ACK messages allows solving this problem. Specifically, an ACK is only accepted at a router if the values of myBT, mySSN, neiBT, and neiSSN match the ones stored at the router. Rejecting such

ACK, forces the retransmission of the message, which may now place the correct state information at the neighbor.

One alternative to solve this problem is to forbid the transmission of upstream and interest messages concurrently with a synchronization process. This is undesirable, since it would delay unnecessarily the protocol convergence.

Figure 3.17 illustrates the need for including the SnapshotSN of both the sending and replying routers in ACK messages; the BootTime needs also to be included for similar reasons. Suppose that R1 and R2 have previously synchronized with each other such that, initially, the SSN of R1 is 5 and the SSN of R2 is 30. Then, R1 loses contact with R2 (e.g. because it cease to receive Hello messages from R2 for a while) and removes its state and SSN relative to R2. Later R1 resumes receiving Hello messages from R2, and initiates a synchronization with it, by sending a Sync message with its snapshot and SSN=20. Soon after transmitting the Sync message, R1 sends an IamUpstream message relative to tree (S1,G1), which was not included in the previous snapshot. The IamUpstream message has an SN higher than the SSN, i.e. SN=21, but it arrives before the Sync at R2. At this point, R2 stores the information that router R1 is UPSTREAM for tree (S1,G1), and replies with an ACK. Suppose that this ACK does not include the SSN of the sending router and of its neighbor. When the ACK arrives at R1, R1 considers that R2 correctly stored the upstream information, and no further IamUpstream message is transmitted. Suppose now that R2 loses contact with R1 (e.g. because it cease to receive Hello messages from R1 for a while). In this case, R2 removes its state and SSN relative to R1. If now the Sync message arrives at R2, R2 accepts R1's snapshot, which does not include the (S1,G1) tree, and completes the synchronization process with R1 by sending a Sync message with its snapshot and SSN=47. No further control messages circulate and R1 is wrongly convinced that the upstream information previously sent in its IamUpstream message is stored at R2.

Suppose now that the ACK message carries the SSN of the sending router and of its neighbor. In this case, the ACK includes mySSN=30 and neiSSN=5 and will be rejected since the current SSN of R1 is SSN=20 and not the value indicated by R2, i.e. SSN=5. In this way, R1 detects that the IamUpstream message was received by R2 before the last synchronization between these two routers and, therefore, may have been removed at R2. Since the ACK was rejected, R1 is forced to retransmit the IamUpstream message relative to (S1,G1) which ensures that this information will be correctly stored at R2.

3.8 Message format

The format of the control messages is similar to the ones used by PIM-DM and PIM-SM, but with some additional fields. Since the protocol is not a standard yet, we do not have any IP protocol number. For test purposes, we used the one specified for PIM (103). Regarding the transmission of multicast messages we use the destination multicast IP address of PIM, i.e. 224.0.0.13.

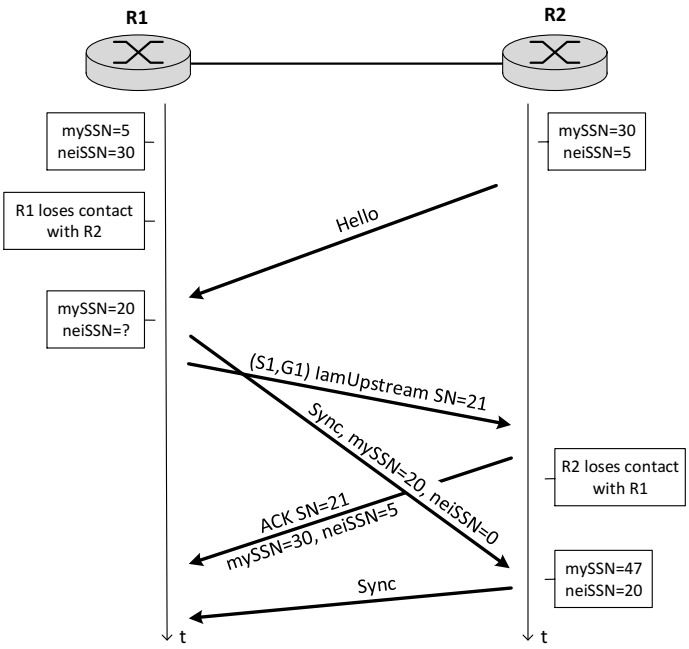

Figure 3.17: Motivation for the use of all sequence numbers in ACK messages.

3.8.1 Protocol Header

All control messages include a common header:

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1			
BootTime			
Version	Type	Integrity Identifier	Integrity Length
Integrity Value (length defined in Integrity Length)			

Figure 3.18: Protocol Header format.

BootTime - Sequence number of the router that is transmitting the message, used to detect reboots and SN overflows;

Version - Version of the protocol implementation (currently set to 0);

Type - Defines the type of the transmitted control message. The following types are available:

- 1 - Hello
- 2 - Sync
- 3 - IamUpstream
- 4 - IamNoLongerUpstream
- 5 - Interest
- 6 - NoInterest

- 7 - ACK

Security Identifier - Identifies the key and algorithm used in the integrity and authentication check (same as KeyID of OSPF messages [28]). In case the Identifier is 0, the message does not offer any integrity nor authenticity guarantees.

Security Length - Number of bytes of the following field. In case this is set to 0, the message does not offer integrity nor authenticity guarantees.

Security Value - Field that includes the cryptographic hash, used to verify the message integrity and authenticity. This value must be calculated in the following way: SecurityValue = function_cryptographic_hash(SourceIP + DestinationIP + Message with Security Value zeroed, Key), with:

- function_cryptographic_hash - function used to generate a cryptographic hash, having as input the secret key (Key) and the content of the message;
- SourceIP - source IP of the IP header;
- DestinationIP - destination IP of the IP header;

3.8.2 Hello

Hello message can include a variable number of options, since not all options are mandatory.

This message must be multicasted to all neighbors. Hello messages are composed by several options described in a TLV (Type-Length-Value) format.

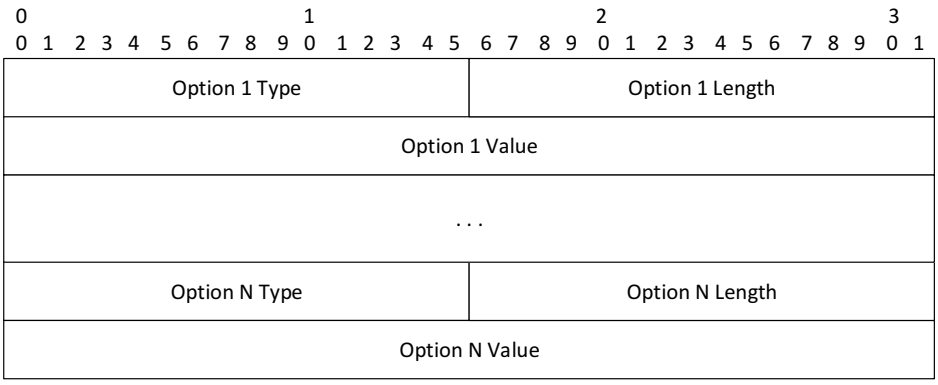

Figure 3.19: Hello message format.

Option Type - Identifies the option given in the Option Value field. Defined types are the following:

- 1 - Hello Hold Time
- 2 - CheckpointSN

Option Length - Number of bytes that are used by the Option Value.

Option Value - Value of a given option.

Hello Hold Time

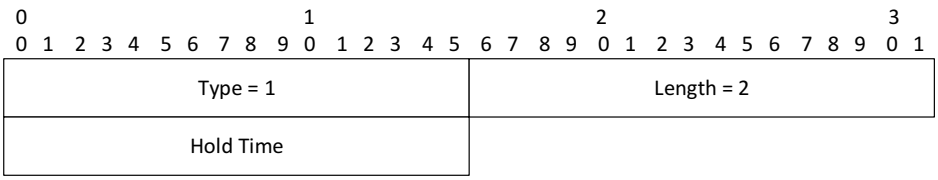

Figure 3.20: Hello Hold Time TLV.

Hold Time is the number of seconds a receiver must wait until a neighbor is declared dead. If the Hold Time is set to '0', the information is timed out immediately (to accelerate neighbor removals). The Hello Hold Time option is mandatory, so it must be included in Hello messages.

CheckpointSN

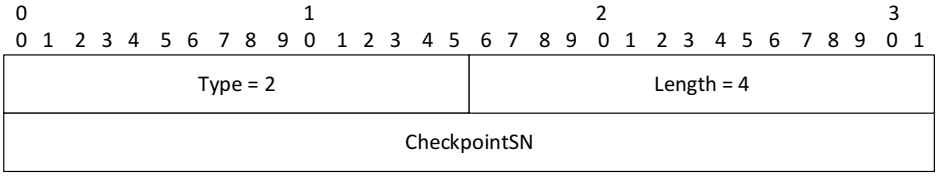

Figure 3.21: CheckpointSN TLV.

Option used for a router to advertise its CheckpointSN to its neighbors. This option is not mandatory. It corresponds to an optimization used to avoid storing excessive SNs (section 3.5.4).

3.8.3 Sync

This message must be unicasted to a single neighbor (the one that is performing the synchronization with the router). This was addressed in section 3.6.

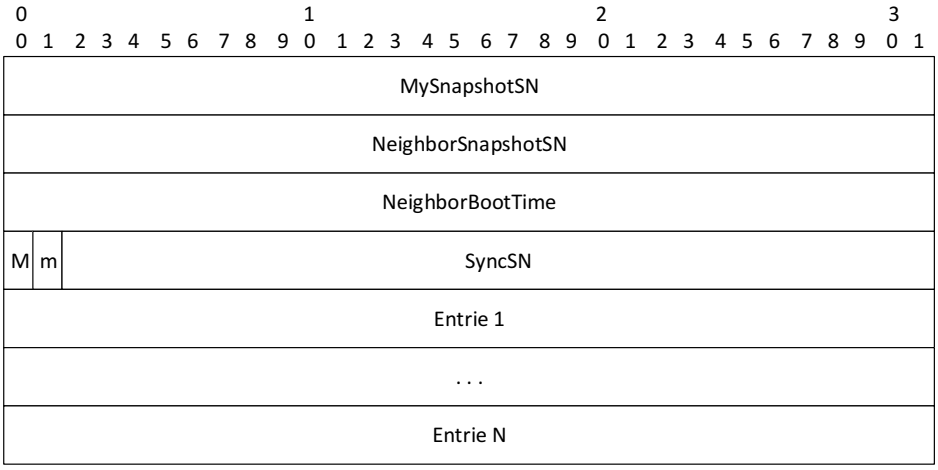

Figure 3.22: Sync message format.

MySnapshotSN - SnapshotSN of the router transmitting this message;

NeighborSnapshotSN - SnapshotSN of the neighboring router;

NeighborBootTime - BootTime of the neighboring router;

M - Master flag. Set to 1 if the router considers itself Master (neighbor is in SLAVE state);

m - More flag. Set to 1 if the router that is transmitting this message still has unacknowledged tree information;

SyncSN - Sequence number used to fragment Sync messages.

These fields were discussed in section 3.6. In that section, MySnapshotSN was referred as mySSN, NeighborSnapshotSN as neiSSN and NeighborBootTime as neiBT.

In case the More flag (m) is cleared, the Entry fields carry Hello options, such as the Hello Hold Time and CheckpointSN.

In case the More flag (m) is set, the Entry fields carry information on the trees for which the router considers itself as UPSTREAM (information from the snapshot). These fields must respect the structure below (Sync Tree Entry).

Sync Tree Entry

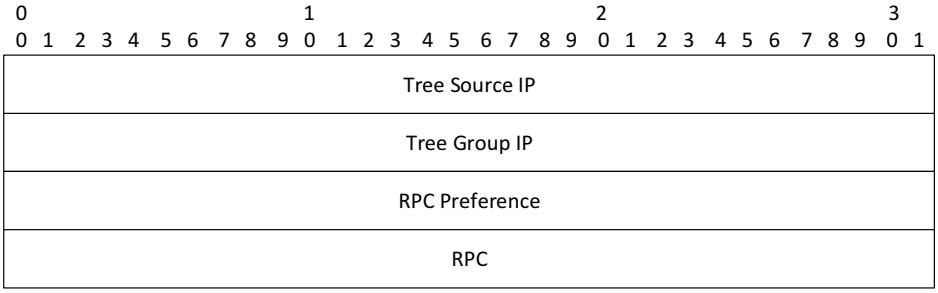

Figure 3.23: Sync tree entry message format.

Tree Source IP - IP address of the source of multicast traffic;

Tree Group IP - IP address of the multicast group;

RPC Preference - The preference value assigned to the unicast routing protocol that provided the route to the source;

RPC - The RPC to the source. The metric is in units applicable to the unicast routing protocol used.

3.8.4 IamUpstream

This message must be multicasted to all neighbors.

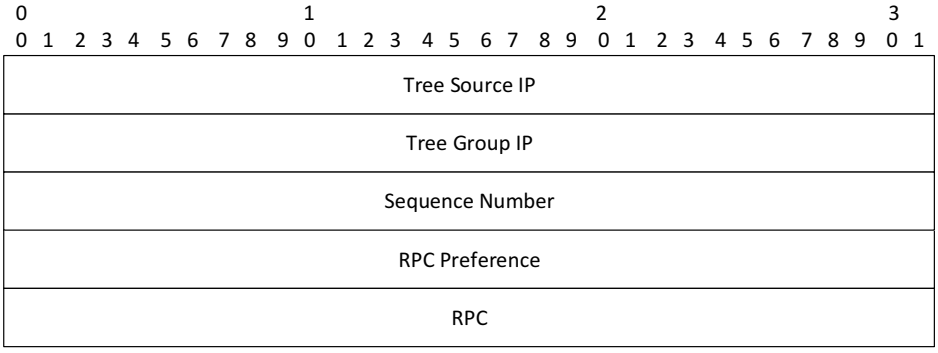

Figure 3.24: IamUpstream message format.

Tree Source IP - IP address of the source of multicast traffic;

Tree Group IP - IP address of the multicast group;

Sequence Number - Used to sequence messages;

RPC Preference - The preference value assigned to the unicast routing protocol that provided the route to the source;

RPC - The RPC to the source. The metric is in units applicable to the unicast routing protocol used.

3.8.5 IamNoLongerUpstream

This message must be multicasted to all neighbors.

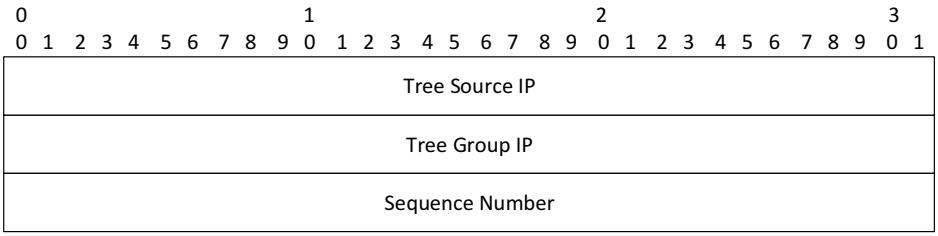

Figure 3.25: IamNoLongerUpstream message format.

Tree Source IP - IP address of the source of multicast traffic;

Tree Group IP - IP address of the multicast group;

Sequence Number - Used to sequence messages.

3.8.6 Interest

The format of this message is equal to IamNoLongerUpstream messages. These messages are distinguishable by the message's type at the Protocol Header.

This message should be unicasted to a single neighbor (the AW).

3.8.7 NoInterest

The format of this message is equal to IamNoLongerUpstream messages. These messages are distinguishable by the message's type at the Protocol Header.

This message should be unicasted to a single neighbor (the AW).

3.8.8 ACK

This message acknowledges the reception of IamUpstream, IamNoLongerUpstream, Interest or NoInterest messages. This message must be unicasted to the source of those messages.

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Tree Source IP
Tree Group IP
NeighborBootTime
NeighborSnapshotSN
MySnapshotSN
Neighbor Sequence Number

Figure 3.26: ACK message format.

Tree Source IP - IP address of the source of multicast traffic that this message is acknowledging;

Tree Group IP - IP address of the multicast group that this message is acknowledging;

NeighborBootTime - BootTime of the neighboring router (the one that will receive the ACK message);

NeighborSnapshotSN - SnapshotSN of the neighboring router (the one that will receive the ACK message);

MySnapshotSN - SnapshotSN of the own router (that is transmitting the ACK message);

Neighbor Sequence Number - Sequence number of the message that the ACK is acknowledging;

3.9 Security considerations

When a link connects an host/router that is not controlled by the network administrator, malicious behavior can happen. An attacker can on purpose change the order of messages, change its content, create messages on behalf of other routers and also retransmit old messages [29].

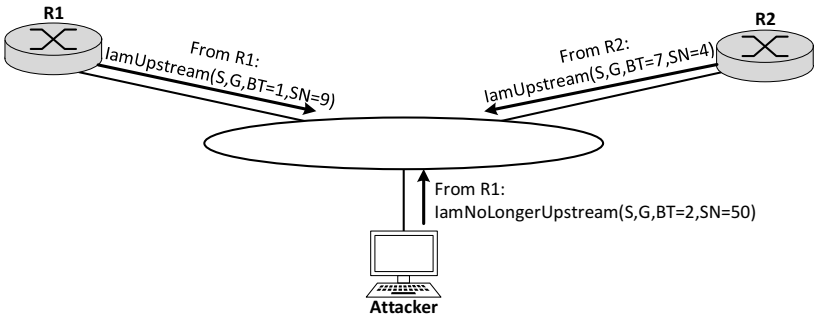

Figure 3.27: Attacker transmits control messages.

Figure 3.27 gives an illustration of a possible attack. We have two routers and one host, represented in this figure by R1, R2 and Attacker. R1 sends information regarding a new tree (S1,G2) with BT=1 and SN=9. R2 sends to the same link information regarding a new tree (S2,G2) with BT=7 and SN=4. This exchanged information can cause the creation of state regarding these two trees, if a router is hearing for the first time an IamUpstream regarding those trees. Since the messages that are exchanged are

not security protected, an attacker that has access to this link can create new messages with a spoofed source IP address. In this figure, the Attacker sends an IamNoLongerUpstream message regarding the tree that was previously created by R1, with the source IP address set to R1, and greater BTs and SNs. The reception of this message by R2, causes this router to consider incorrectly that R1 is NOT UPSTREAM. This causes the protocol to not work as expected.

Security requirements In order to avoid a malicious behavior, additional protection is required. This additional protection must guarantee message integrity, authenticity and freshness. Confidentiality is not required since control messages do not include any secret information.

Guaranteeing integrity and authenticity In order to guarantee authenticity and integrity, a MAC (Message Authentication Code) is included in all control messages. By including this information, routers can successfully determine if a control message has been tampered. Routers only accept control messages if the MAC received with the control message is correct, thus guaranteeing authenticity and integrity. Attackers are not able to create new messages or modify previously exchanged messages because they are unable to recreate the MAC.

The use of this security protection can be accomplished by including the MAC within the transmitted message. The Source IP and Destination IP of the control message must be considered in the construction of the MAC in order for an attacker to not be able to change the Source IP and/or Destination IP address at the IP layer. For a MAC, a secret key is required to be shared by all routers attached to the same link. A simple solution is to manually install this key in all routers or use a given protocol to exchange this information between routers. How routers discover/exchange this key is out of scope of this document.

Guaranteeing freshness The freshness property is meant to avoid replay attacks, in which an attacker simply replays a message that it has previously heard. The attacker does not change the message's content, so according to the MAC, the message is correct and further processed by the receiving router. This problem is similar to the out of order reception of control messages, in the sense that replayed messages carry older state. In our protocol we already deal with this ordering problem through message sequencing (section 3.5).

Some design decisions in our protocol were done having in mind the problem of replay attacks, which will be explained next.

Through messages that carry the BootTime and the SN (section 3.5), routers are able to determine if a received control message carries older state. The BootTime sequences reboots, thus being able to filter messages transmitted prior to a reboot. The SN sequences transmitted messages for a given BootTime. By having all control messages with monotonically increasing pairs $< BootTime, SN >$, and by having routers storing the BootTime and the last received SN for each neighbor, routers are able to filter replayed messages.

In our synchronization process, a router informs its neighbor about its SnapshotSN. This SN corresponds to the SN of the neighbor when it started the synchronization process. By knowing this SN, and by having exchanged the most up-to-date information in the synchronization process, a router filters

messages that were transmitted by its neighbor router prior to the synchronization (messages received with SN lower than the SnapshotSN). This in conjunction with the transmission of messages carrying the BootTime and SN, allows routers to filter replayed messages.

The synchronization process exchanges Sync messages between two routers. This process uses a challenge-response mechanism, in which all Sync messages regarding the same synchronization process must carry the BootTime and SnapshotSNs of both routers. Thus, if a router hears a Sync message in which its BootTime or SnapshotSN are not correct, this process will not progress. If an attacker replays Sync messages of routers that have failed, these messages could trigger a new synchronization (due to being originated from an unknown neighbor) but that synchronization process would never terminate successfully. This is due to the usage of a single SN space, causing newer synchronization processes to have necessarily greater pairs $< BootTime, SnapshotSN >$, thus previously exchanged Sync messages will include sequence numbers that are no longer valid for newer synchronizations.

Adjacency is not formed through Hello messages. In our protocol any control message can be used to discover new neighbors, not limited to Hello messages. In this protocol, what allows an adjacency to be fully established is the correct termination of the synchronization process. For this reason, any type of control message replayed from failed routers can trigger new synchronizations, but as explained above, even if Sync messages are replayed these new synchronization processes would never terminate successfully. This way of establishing adjacencies contrasts with existing routing protocols, in which the adjacency is fully established through the exchange of Hello messages (like OSPF). Since Hello messages are not sequenced, replayed Hello messages prior to the adjacency formation can break incorrectly neighborhood relationships in those protocols.

To correctly filter replayed messages, routers must store all SN information for each tree regarding each neighbor. This can cause an excessive overhead penalty in routers, in order to prevent replay attacks. As explained in section 3.5.4, we introduced a mechanism named CheckpointSN that allows to periodically remove SNs regarding neighbor routers, without having any impact in terms of security. A router advertises its CheckpointSN to its neighbors, referring that all messages with SN below the CheckpointSN have already been correctly acknowledged, thus will never be retransmitted. Neighbors by knowing this information store the CheckpointSN and remove all SNs below that value. Then following received messages, if there is no stored SN information, they only accept it if they carry a SN greater than the CheckpointSN. Eventually routers are able to only store a single SN per neighbor (the CheckpointSN), irrespective of the number of trees configured on the network.

By using the CheckpointSN we are also able to mitigate DOS attacks. If an attacker replays indefinitely often the messages that carried the most recent state, the receivers of those messages would necessarily acknowledge them because the stored SN is equal to the received SN (retransmission may be due to loss of the ACK). By using the CheckpointSN, routers are able to determine which messages have already been correctly acknowledged. For this reason, if the attacker is replaying messages that have a SN lower to the advertised CheckpointSN, its neighbors would no longer acknowledge those messages.

An attacker can also replay Hello messages right after the “source” of those messages fails. This

would cause routers to still consider the failing router to be alive, since the timer used to control the liveness of the failing router would never expire. In order to countermeasure this problem, this is detected indirectly. Replayed Hello messages are accepted, thus suggesting that the failed router is still alive, nevertheless, transmitted messages to the failed router would never be acknowledged (the attacker is not able to create new messages without being detected). For this reason, even if a neighbor transmits Hello messages, by failing successively the acknowledge of new control messages, a router should declare it as dead, breaking the neighborhood relationship. Following replayed messages may trigger a new synchronization process, but as explained above, would never terminate successfully.

Dependability on external factors A new BootTime can be determined by using the router's clock or by storing the last used value on non-volatile memory.

In case of using the clock, the value obtained from successive reads must be necessarily greater. Physical clocks do not preserve this property and for this reason a clock synchronization is required (for example with NTP). This clock synchronization must be necessarily protected, in order for attackers to not control the BootTime of routers.

In case of using non-volatile memory, we must be careful in case the memory of the router is repaired/changed. This would cause the new memory to not store the previously used BootTime.

The BootTime by having a finite number of bits, it can also overflow, so this must also be taken into consideration. An easy solution would be to change the secret key used for integrity and authentication guarantees. So even if following messages have lower BootTimes, a simple replay from previously transmitted control messages, with a greater BootTime, would be filtered by failing the integrity and authentication guarantees.

Data messages All considerations until now were regarding the exchange of control messages between routers. All mechanisms described until now allow the protection of control information. Attackers can cause the same attacks to data packets.

In terms of data packets, this is not controlled by the multicast routing protocol, since routers are "blind" to their content. In order to protect from attacks to data packets, these must be performed by the applications that will consume those messages, being out of scope of this document.

Nevertheless, the creation of data packets with spoofed source IP can cause the creation of trees and respective forwarding.

In case the spoofed IP address belongs to the subnet in which the attacker is at, this would trigger the creation of the tree, by having originator routers reacting to the reception of data packets. These messages would then be forwarded to possible interested receivers.

In case the spoofed IP address does not belong to the subnet in which the attacker is at:

- If the tree is ACTIVE, those data packets would be forwarded along with data packets originated by the real source;
- If the tree is UNKNOWN, those data packets could be flooded throughout the whole network. Since these data packets would be originated outside the source's subnet, the tree would never transition to ACTIVE state, not triggering the exchange of IamUpstream messages. By not exchanging

these messages, there would be no AW election and there would be no exchange of interest messages since these are triggered by the existence of UPSTREAM neighbors in a given link. This issue is only present if the network administrator configures initially to have all downstream routers interested in receiving data packets, before the tree is created. If the network administrator configures all routers to have initially no downstream routers interested, then data packets of the attacker would not be forwarded outside the local subnet in which they are being transmitted. This solution mitigates the issue but would cause the loss of initial data packets for every tree that is initially created.

There are solutions that avoid the continuous flood of data packets without sacrificing initial data packets of the source:

- Use a lighter version of the protocol in INACTIVE and UNKNOWN state. For example, using the Assert and Prune Override mechanism of PIM-DM could mitigate this issue, by electing an AW and controlling the interest at links, using a soft-state approach.
- Use a Rendezvous Point (RP). In this case data packets from unknown trees are not forwarded (not flooded). By having the creation of trees triggered by receivers, this avoids the flooding behavior, since these are only forwarded through previously created trees (from receivers). Initial data packets are simply transmitted from the source to a central router and if other routers already joined the tree rooted at the RP, these are forwarded to the receivers already attached to it. However in this case the protocol is of type Sparse Mode instead of Dense Mode, which requires maintaining trees using a different approach (our protocol by being Dense Mode is not designed to build trees this way).

Chapter 4

HPIM-DM correctness

In this chapter we discuss the correctness of the HPIM-DM protocol, in the sense that there is no unwanted behavior or deadlocks.

In order to argue about the global correctness of the HPIM-DM protocol, we can subdivide the protocol in multiple parts and argue about the correctness of each part individually. We will discuss the correctness regarding the synchronization, the creation and removal of trees, maintenance of interest information, message reliability and message sequencing. The correctness of some individual parts are dependent on the correctness of other parts, such as: interest maintenance is only correct if the tree maintenance is also correct; tree maintenance is only correct if the synchronization is also correct; all parts are dependent on the correctness of message sequencing and reliability.

In order to support the correctness of these individual parts of the protocol, we have performed some formal tests. This verification could be performed through simulation or model checking. Simulation is more error prone since we manually define the conditions of the tests and compare if the obtained result is the one expected. In model checking, a model of the solution is created and a software tool is used to generate all possible states in which the program can be at and verify if correctness properties hold in those states.

Since model checking allows to obtain better results, it was used for the verification of correctness properties in this protocol. SPIN [8, 30] was selected as the software tool used to perform those tests. In this tool, the model is specified in a programming language called Promela [7], which is similar to C. Then it is possible to verify if all correctness properties hold, by using linear temporal logic.

Linear temporal logic (LTL) allows to specify global correctness properties that are not associated with any specific control point of the model. By using LTL, it is possible to verify if a property holds in all states of the model or if it will eventually hold.

We start by explaining the correctness of the HPIM-DM components through logical reasoning. In addition, the correctness of the synchronization process and of the maintenance of trees were verified through model checking. Interest maintenance was not verified through model checking since this would require excessive computational resources. Message sequencing and reliability were not verified since these are obviously correct.

In section 4.1 we argue about the correctness of the synchronization process; in section 4.2 we argue about the correctness of the tree maintenance; in section 4.3 we argue about the correctness of the interest maintenance; in section 4.4 we discuss the correctness of message sequencing. Finally, in section 4.5 we discuss the correctness regarding the reliability of exchanged control messages.

The Promela models used to prove the HPIM-DM correction are detailed in Appendix C.

4.1 Synchronization

First we will use logical reasoning to argue that the synchronization process is correct. This process was described in detail in section 3.6.

4.1.1 Logical reasoning

The synchronization process in HPIM-DM is based on OSPF's database description process [28], a known and well accepted synchronization mechanism. However, our synchronization process has some differences that offer even stronger guarantees regarding its correctness, by preventing replay attacks.

Like in OSPF, the synchronization is controlled by a Master, elected between the two routers that are synchronizing with each other. In HPIM-DM we allow the Master to be the one that initiates the synchronization process, instead of performing this election through the comparison of both routers' identifiers like in OSPF. In case both routers consider themselves as Masters, the routers' identifiers takes place to elect a single router as Master, to avoid a possible deadlock.

In HPIM-DM each synchronization process is identified by the BootTime and SnapshotSN of both routers. This allows routers to ignore messages that were replayed from previous synchronization processes. Basically we are using the synchronization process of OSPF and forcing both routers to exchange synchronization messages with this information. In case a router hears a message with lower a BootTime or SnapshotSN (of itself or of its neighbor) the synchronization does not progress. This forces both routers to exchange information consistently since they can identify when both routers started the synchronization process:

- If a router hears a greater BootTime or same BootTime with greater SnapshotSN, compared to previous synchronizations with its neighbor, it knows that this is referring to a new synchronization;
- If it hears a lower BootTime or same BootTime with a lower SnapshotSN, this is referring an old synchronization process, thus the content of the message has old information and is ignored;
- Hearing the same BootTime and same SnapshotSN means that the message is referring to the current synchronization process. In this case, the message is interpreted according to the rules of OSPF, i.e. according to the Master-Slave election and to the SyncSN (fragmentation of Sync messages).

By including this information routers are able to perform synchronizations consistently.

4.1.2 Model checking

In order to confirm the correctness of our synchronization process, we have used model checking.

We modeled the synchronization between two routers requiring a random number of Sync messages to be exchanged in the presence/absence of failures concurrently to the synchronization process. The random number of Sync messages was used to test their fragmentation controlled by the Master (verify Master and More flags and all sequence numbers). The failures that were modeled were: (i) reboot of the neighbor and (ii) router falsely suspects that its neighbor has failed. Regardless of the presence of these failures, eventually both routers must terminate the synchronization process successfully, i.e. by considering their neighbors to be in UPDATED state and by storing the same exchanged sequence numbers (BootTime, SnapshotSN and the last SyncSN). The last exchanged SyncSN must necessarily be the maximum number of required Sync messages between both routers incremented by one (requires an additional message exchange with the More flag cleared). All variables that were tested in the model are listed in Tables 4.1 and 4.2. This was proved in the developed model.

	Test Description
1	Both routers require the same number of Sync messages to be transmitted to their neighbor
2	Both routers require different number of Sync messages to be transmitted to their neighbor

Table 4.1: Tests regarding fragmentation of Sync messages.

	Test Description
1	No failures
2	Reboot of one or both routers
3	One or both routers false suspect that their neighbor has failed

Table 4.2: Tests regarding failures during the synchronization process.

A detailed explanation of the model is described in Appendix C.

4.2 Creation and removal of the broadcast tree

First we will use logical reasoning to argue that the maintenance of trees is always correct. All details regarding the tree maintenance were addressed in section 3.2.

4.2.1 Logical reasoning

In HPIM-DM:

- Each router can consider a tree to be in one of three states: ACTIVE, INACTIVE and UNKNOWN;
- A router maintains the upstream state of all its neighbors, in both ACTIVE and INACTIVE states, as being UPSTREAM or NOT UPSTREAM. All neighbors are initially considered NOT UPSTREAM.

The storage of upstream state allows routers to determine in which state they are. This information is always correct. If a neighbor fails, upstream state is removed (equivalent to NOT UPSTREAM). If a new neighbor appears, the synchronization process sets the correct upstream state.

A router is ACTIVE if it has an UPSTREAM neighbor connected to its root interface, subject to having a lower RPC than the own router's RPC. Otherwise, the router is INACTIVE or UNKNOWN.

An ACTIVE router is said connected to the tree; INACTIVE and UNKNOWN are said disconnected.

Regarding the tree states:

- If a router becomes ACTIVE, sends IamUpstream message through non-root interfaces;
- If a router ceases being ACTIVE, sends IamNoLongerUpstream through all non-root interfaces;
- If interface roles change and remains/becomes ACTIVE:
  - new root interface sends IamNoLongerUpstream;
  - new non-root interfaces send IamUpstream.

This causes neighbors connected through non-root interfaces to be considered UPSTREAM if they are ACTIVE; neighbors connected through root interfaces are considered NOT UPSTREAM, irrespective of the state.

A router by storing the correct upstream state of its neighbors all the time, it can determine if it is connected to the tree (i.e. in ACTIVE state) or not.

Additionally, a router by correctly storing the upstream state of its neighbors and, for UPSTREAM neighbors, also their RPCs, it can always determine correctly the AW: it is the one among the UPSTREAM neighbors and itself (if attached to the link through a non-root interface and in ACTIVE state) that offers the lowest RPC. In case of a tie, the IP address breaks the tie (greatest IP wins).

4.2.2 Model checking

In order to confirm the correction of the tree maintenance mechanism, we have used model checking. We have modeled the formation of trees concurrently to failures and interface role changes. These are performed by having routers exchanging IamUpstream and IamNoLongerUpstream messages, in reaction to state transitions.

We have performed tests in three topologies, shown in Figures 4.1, 4.2 and 4.3. In these figures, all routers are identified by a given number and their interfaces are also identified by a given number (in Topology 1, Router 1 is connected by interface 3 to interface 1 of Router 0).

In all topologies, Router 0 was considered to be the originator router, by having interface 0 connected to the source of multicast traffic. The root interface was selected manually, in each router.

We pretended to verify if all routers consider the tree to be in a consistent state (ACTIVE/INACTIVE/UNKNOWN) according to the tree configuration, i.e. the existence of UPSTREAM neighbors and their RPCs. We have tested the tree formation in the presence of router failures and unicast changes (changes of interfaces' roles). We did not model the election of the AW, since this is correct due to the storage of upstream state of all neighbors at a router.

We used Topology 1, because it has a link that offers redundant paths (link that connects Routers 1, 2 and 3) and also has a link without redundant paths (link that connects Routers 3 and 4). By modeling

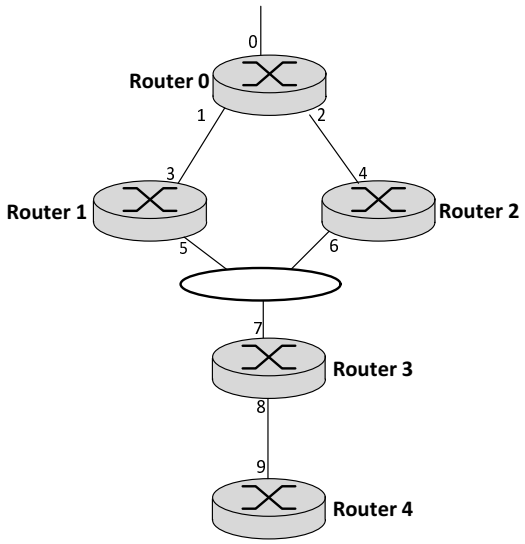

Figure 4.1: Correctness of the tree maintenance procedure (Topology 1).

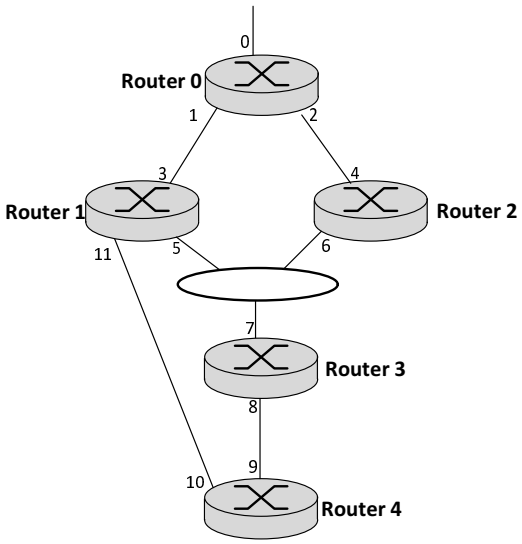

Figure 4.2: Correctness of the tree maintenance procedure (Topology 2).

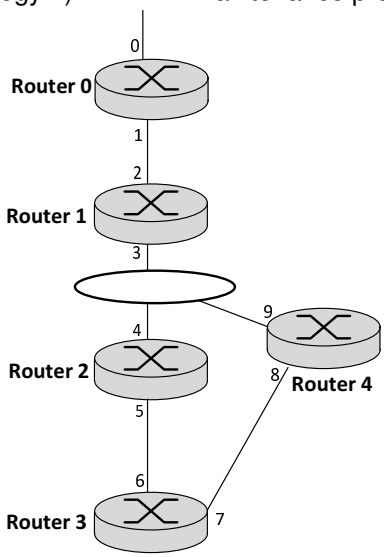

Figure 4.3: Correctness of the tree maintenance procedure (Topology 3).

the failure of different routers, the tree must be reconfigured, according to the existence of redundancy. In these tests we have verified, with LTL, if the tree is reconfigured correctly, i.e. if all routers reach a consistent tree state.

In Topology 1, the root interfaces were selected according to the number of hops. So interfaces 0, 3, 4, 7 and 9 were selected as root interfaces. The tests performed to Topology 1 are listed in Table 4.3.

	Test Description
1	Tree formation without router failures
2	Tree formation concurrent to failure of Router 1
3	Tree formation concurrent to failure of Router 2
4	Tree formation concurrent to failure of Router 3
5	Tree formation concurrent to failure of Router 0

Table 4.3: Description of all tests to our Topology 1.

In all tests, Router 0 triggers the formation of the tree through IamUpstream messages.

In Test 1, all routers start in UNKNOWN state and we expect that eventually all routers transition and remain in ACTIVE state.

Test 2 is similar to Test 1, but Router 1 fails concurrently to the tree formation. We expect that all routers become ACTIVE, except for the failing router. This is due to the existence of redundancy through Router 2 that allows a chain of upstream routers to still be formed.

Test 3 is similar to Test 2, but Router 2 fails instead of Router 1. In this case, eventually all routers become ACTIVE except for the failing router.

Test 4 is similar to Test 2, but Router 3 fails instead of Router 1. In this test, due to the absence of redundancy, all routers must become ACTIVE except for the failing router and Router 4. This last router must eventually be in UNKNOWN state.

In Test 5, Router 0 triggers the formation of the tree and concurrently fails. Eventually all routers must be in UNKNOWN state.

Topology 2 just introduced a link, between Router 1 and Router 4, to Topology 1. In this topology we wanted to test if all routers reach a consistent tree state, by having the unicast routing protocol also reacting to network changes. By having initially the same selected root interfaces, as the tests of Topology 1, the failure of Router 3 would cause a change of Router 4 root interface. In this test, we wanted to verify if Router 4, by changing its root interface to interface 10, would still consider the tree to be ACTIVE. This test, performed to Topology 2, is described in Table 4.4.

	Test Description
6	Tree formation concurrent to failure of Router 3 and concurrent to change of interfaces' roles at Router 4

Table 4.4: Description of all tests to our Topology 2.

In Test 6 it was expected that all routers become ACTIVE except for the failed router.

Topology 3 was used in order to verify the formation and removal of a tree, in all routers, in the presence of network loops. The root interfaces were manually selected, in order to form a loop. So, interfaces 0, 2, 4, 6 and 8 were selected as root interfaces. This caused Routers 2, 3 and 4 to form a loop, since non-root interface of Router 4 was connected to root interface of Router 2, non-root interface of Router 2 was connected to root interface of Router 3 and non-root interface of Router 3 was connected to root interface of Router 4.

The RPCs were selected carefully in order to respect the selected root interfaces, i.e. the RPC of a router must be greater than the RPC of its next-hop (at the unicast routing protocol). This would also be used by the multicast routing protocol to verify the presence of network loops, in order to not maintain a tree indefinitely.

All tests performed to Topology 3 are described in Table 4.5.

In Test 7, Router 0 triggers the formation of the tree. Eventually all routers must be in ACTIVE state.

In Test 8, Router 0 triggers the formation of the tree and then its removal. Eventually all routers must consider the tree to be UNKNOWN, even in the presence of a network loop.

	Test Description
7	Tree formation in the presence of network loop
8	Tree removal in the presence of network loop

Table 4.5: Description of all tests to our Topology 3.

All tests confirmed our expectations. A detailed explanation of the used Promela code is described in Appendix C.

4.3 Interest

Interest information must be maintained correctly at the AW. This has been described in section 3.3. In order to prove that this information is always maintained correctly, we will explain it by using logical reasoning.

Regarding interest information, this:

- Must be maintained correctly at the AW;
- Is stored at routers in ACTIVE state as long as they remain in this state;

Interest of UPSTREAM routers is always correct at all neighbors (also in the AW) due to the fact that this upstream state is controlled by the own routers.

Interest of NOT UPSTREAM routers is transmitted to the AW.

In order to guarantee the correct interest state maintenance of NOT UPSTREAM routers, when the AW changes, an interest message is transmitted to the new AW. This allows the new AW to obtain the interest information of its neighbors. However, routers may have different views regarding on who is the AW, nevertheless they trust their view regarding on who is AW, so only these four cases can happen:

- A router may transmit an interest message to the correct AW, that considers itself as AW and is ACTIVE;
- A router may transmit an interest message to the correct AW, but that neighbor considers itself as AL and is ACTIVE;
- A router may transmit an interest message to the wrong AW;
- A router may transmit an interest message to the correct AW but that neighbor loses its information.

In the first case, the interest state is correctly stored.

In the second case, the interest state is correctly stored at the neighbor router, by being ACTIVE, however this neighbor will not make a forwarding decision based on it, by considering itself as AL. If this neighbor then considers itself as AW, it will take into consideration the received state.

In the third case, the interest state may or may not be stored at the neighbor router depending on its tree state. However eventually the router that transmitted the interest will detect that its neighbor is no longer AW (due to failure or reception of IamNoLongerUpstream or other router is AW). If there is

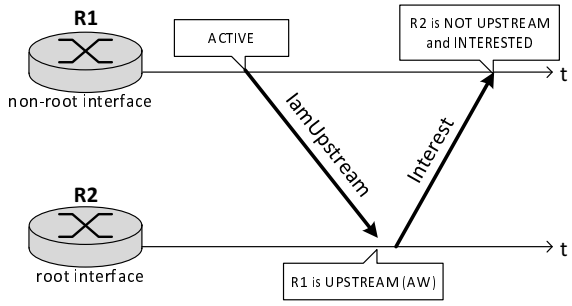

Figure 4.4: Reception of interest information.

another AW, transmitting interest to it allows the correct storage. If there is no AW, then no information is transmitted.

In the fourth case the interest may start initially to be stored correctly, however the neighbor loses the interest state. This can only happen if the neighbor was placed in a non ACTIVE state, triggering the transmission of IamNoLongerUpstream. If the neighbor transitions back again to ACTIVE, it transmits an IamUpstream message. The router that transmitted interest may consider that there was no AW change if the IamNoLongerUpstream is not received, however the IamUpstream message must be necessarily received and acknowledged. By forcing the transmission of interest messages when an IamUpstream message is received from the current AW, we guarantee that this information is stored correctly even if it was previously removed.

4.3.1 Examples

Receiving interest in ACTIVE state Consider Figure 4.4, in which two routers, R1 and R2, are connected to each other. R1 starts considering a given tree as ACTIVE, thus transmits an IamUpstream message. The reception of this message by R2, causes this router to consider R1 as UPSTREAM. By considering R1 as AW, due to be the only neighbor UPSTREAM, R2 transmits to it an interest message, which is received by that router and stored.

In this figure, it is possible to verify that the reception of IamUpstream, from the router that will be considered AW, causes R2 to know that its neighbor is in ACTIVE state, thus interest messages transmitted to it will be stored.

This figure illustrates a simple case, in which interest information is received during ACTIVE state.

Considered AW receives/loses interest in non ACTIVE state Consider Figure 4.5, that is similar to Figure 4.4, but now interest information is received at R1, in a non ACTIVE state. This means that the interest information would not be stored at R1. R2 transmitted its interest message, at a point in time in which it thought that R1 was ACTIVE, because it consider it as being UPSTREAM. Eventually, R2 would know that R1 is no longer ACTIVE, by receiving an IamNoLongerUpstream message from it. This means that R1 is no longer considered as AW.

If there are no other neighbors considered as UPSTREAM, this means that R2 would not consider any neighbor as AW. Then no additional message is transmitted.

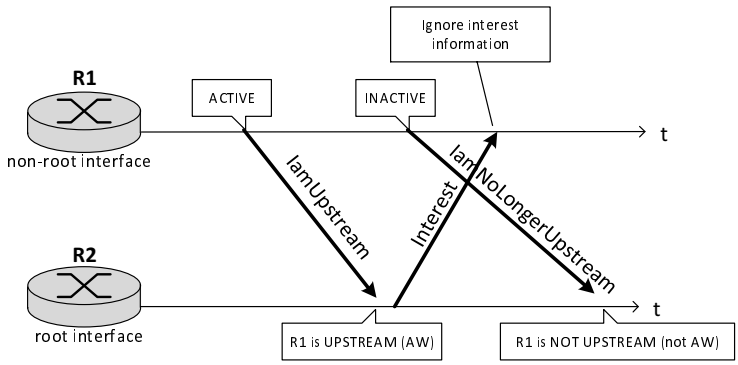

Figure 4.5: Loss of received interest information causing AW change.

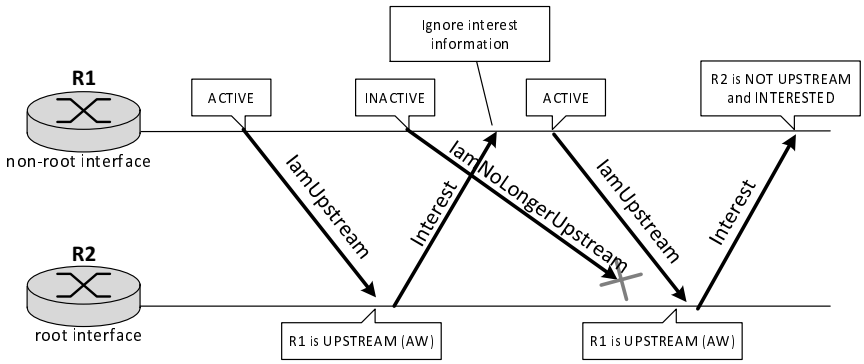

Figure 4.6: Reception of interest information during non ACTIVE state, but this router remains AW.

If there were other neighbors considered as UPSTREAM, for example R3, this means that it has previously transmitted an IamUpstream message. By being considered as UPSTREAM, the router also considers that neighbor to be in ACTIVE state. The transmission of interest information to the new AW (neighbor that is UPSTREAM and offers the best RPC), would be stored, in principle, by it. Router R3, if remains in ACTIVE state, would eventually consider itself as AW, which would take into consideration the received state from R2.

Router is considered always to be AW but loses interest information Now consider that a router always considers a neighbor as AW, but interest information is lost due to a transition to non ACTIVE state. In Figure 4.6, we have the same two routers, R1 and R2. When R1 considers R2 as AW, the former transmits its interest to the latter. This means that R1 considered that R2 was ACTIVE, at that point in time. Consider that this message is received in a non ACTIVE, meaning that the interest state is not stored. This would cause the transmission of IamNoLongerUpstream, like explained in the last example, but now consider that the router transitions back to ACTIVE, transmitting an IamUpstream message. If this last message is received first at R2, then it would consider that its neighbor remained in ACTIVE state, remaining the AW. By retransmitting a new interest message to R1, we force the correct storage of this information.

4.4 Message sequencing

Messages must be processed by the order they were transmitted. In this section we prove that this mechanism is correct.

Upstream and interest messages are sequenced using linear Sequence Numbers. The possibility of SN restart due to SN overflow or router reboot is correctly addressed through the BootTime sequence number.

Optimizations aimed at the minimization of sequencing information (i.e message sequencing per transmitting interface and SNs stored per neighbor and tree) do not compromise correctness.

Message sequencing is per transmitting interface. This was possible by increasing the semantic scope of transmitted messages, i.e. a single message carries upstream and interest information:

- IamUpstream carries UPSTREAM and NOT INTERSTED states by being only transmitted by non-root interfaces that are ACTIVE;
- IamNoLongerUpstream carries NOT UPSTREAM state;
- Interest carries NOT UPSTREAM and INTERESTED states by only being transmitted by root interfaces;
- NoInterest carries NOT UPSTREAM and NOT INTERESTED states by being transmitted by root interfaces and by non-root interfaces in which the router is not connected to the tree (non ACTIVE state).

At the receiver, sequencing information is interpreted per neighbor and tree, thus allowing to determine the order of transmitted messages of each neighbor and tree.

The CheckpointSN was also used to minimize the stored sequencing information. A router advertises a SN to its neighbors in which all messages carrying lower SNs have already been reliably acknowledged. All neighbors by knowing the CheckpointSN of a router can safely remove stored SNs lower than this value, since following control messages will necessarily carry higher SN information. The storage of the CheckpointSN allows a router to filter messages in which SN information has already been removed. This allows to remove SNs and still filter messages that carry older state.

4.5 Message reliability

Messages must be reliably delivered to neighbors. In this section we prove that this mechanism is correct.

Hello messages are transmitted periodically which acts as implicit ACK.

Sync messages are protected through a Stop-and-Wait protocol.

The transmission of upstream and interest messages is ACK-protected. An upstream message (multicasted) is only considered acknowledged when all neighbors acknowledge the message. An interest message (unicasted) is only considered acknowledged when the corresponding neighbor ACKs the

message. If a neighbor fails while a router is waiting for an ACK from it, the router removes the neighbor state (including the ACK pending information) as soon as the Hello protocol detects the failure. Additionally, since new transmitted messages for a given tree carry fresher state, all neighbors that have not acknowledged prior transmitted messages must only acknowledge the last transmitted message. This is possible due to a message carrying upstream and interest information, as explained in section 4.4.

Chapter 5

Implementation

In this MSc Dissertation we have implemented three protocols: IGMPv2, PIM-DM and HPIM-DM.

In this chapter we start by referring all important decisions regarding the implementation of these protocols. A brief description of the adopted software architecture of the HPIM-DM implementation is also illustrated in this section. A detailed description of all performed implementations during this MSc Dissertation can be found in Appendix D, along with a brief description of their specifications and the API offered by Linux regarding multicast.

5.1 Development Environment

In this section we describe the decisions that were made related to the implementation of the protocols.

The first decision was to choose in which operating system the protocol should be implemented in. Since both Microsoft Windows and Mac OS are not open platforms, Linux was selected as the platform for the code development. It turns out that Linux Kernel already has a multicast routing table, which allows a user-level process to manipulate its entries.

The second decision was to choose in which programming language the protocols should be implemented in. It was decided to code everything using Python programming language [31]. While this language can add an overhead penalty compared to low level programming languages like C, since the complex task of routing multicast data packets is accomplished by the kernel, this penalty can be neglected given that the implementation will only manipulate the kernel's routing table. Beside that, by being an high level programming language, it offers abstractions that ease the development.

5.2 Multicast specific features of Linux Kernel

Linux already offers strong APIs related to multicast. The kernel allows a user-level process to manipulate entries in the multicast routing table, allowing to define for each possible multicast stream,

the interface that should receive data packets (root interface), and by which interfaces that traffic should be forwarded (all non-root interfaces in FORWARDING state).

The kernel also implements part of the IGMP/MLD state machines, but only the host part. So if a host 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. The kernel automatically sends messages from the host to routers, answering their multicast queries.

A full description about multicast related APIs offered by Linux is described in Appendix D.1.

5.3 Protocol Implementations

We have implemented three protocols: IGMPv2, PIM-DM and HPIM-DM.

5.3.1 IGMP

As referred above, Linux already implements IGMP, but only the host side. The router side is not implemented, thus we had to implement it.

In order to determine if there are directly connected hosts interested in receiving multicast data packets, we have implemented IGMPv2. This implementation was written in Python. We opted to implement IGMPv2, instead of the more recent IGMPv3, because it is simpler and because the additional features of IGMPv3 were not crucial for the implemented multicast routing protocols. All details regarding the implementation of IGMPv2, along with a brief description of its specification, can be found in Appendix D.2.1.

5.3.2 PIM-DM

We searched for PIM-DM implementations and we have only found one [32], 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 [2]).

This implementation allowed us to better understand how PIM-DM performs its duty regarding the forwarding of multicast traffic. We have published this implementation in our GitHub repository [33], contributing to the open source and scientific communities.

All details regarding this implementation, along with a brief description of its latest specification, can be found in Appendix D.2.2.

5.3.3 HPIM-DM

We have implemented our specification in Python. This implementation is stored in our GitHub repository [34]. In this section we will briefly describe the opted software architecture. More details can be found in Appendix D.2.3.

Regarding the implementation of HPIM-DM, the diagram of Figure 5.1 represents the adopted 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:

- **Run** - this module is used to start/stop the protocol process and can be used for the user to interact with it, in order to add/remove HPIM-DM/IGMP interfaces, list neighbors, list entries and state of all multicast trees. This module is daemonized, in order to have the protocol running in the background;

- **Main** - is a service used to interact with the protocol. This module will receive all commands from Run and execute them in Kernel. If the command returns any output, this service will process the stored internal information and return it back to Run in a human readable format (table format);

- **Interface** - abstract class that has common code related with InterfaceIGMP and InterfaceProtocol, such as code related to the reception and transmission of control messages (through a socket);

- **InterfaceIGMP** - corresponds to a simplistic view of the implementation of IGMPv2. This module is responsible for monitoring the membership information of all hosts directly connected to this physical interface. All details regarding the implemenation of IGMPv2 can be found in Appendix D.2.1;

- **InterfaceProtocol** - this represents a physical interface that has enabled the routing protocol. It will simply send and receive control messages via a raw socket. This class will only process control messages and store received state in the corresponding neighbor's structure. Also, this object is 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;

- **ReliableTransmission** - responsible for guaranteeing the reliable transmission of control messages regarding a given (S,G) tree, from a given InterfaceProtocol;

- **Neighbor** - this object represents a known neighbor detected by a given InterfaceProtocol. This object will store all state regarding that neighbor, such as received upstream and interest state for each maintained (S,G) tree, along with the received sequence numbers. This object will also monitor the liveness of this neighbor. A neighbor object is also associated with a given state, used in the synchronization process;

- **State, Unknown, Master, Slave and Updated** - implementation of the Synchronization state machines;

- **Kernel** - this class is intended to be an abstraction regarding all interactions between this application and the Linux kernel. Besides that, this class stores tree information, representing our

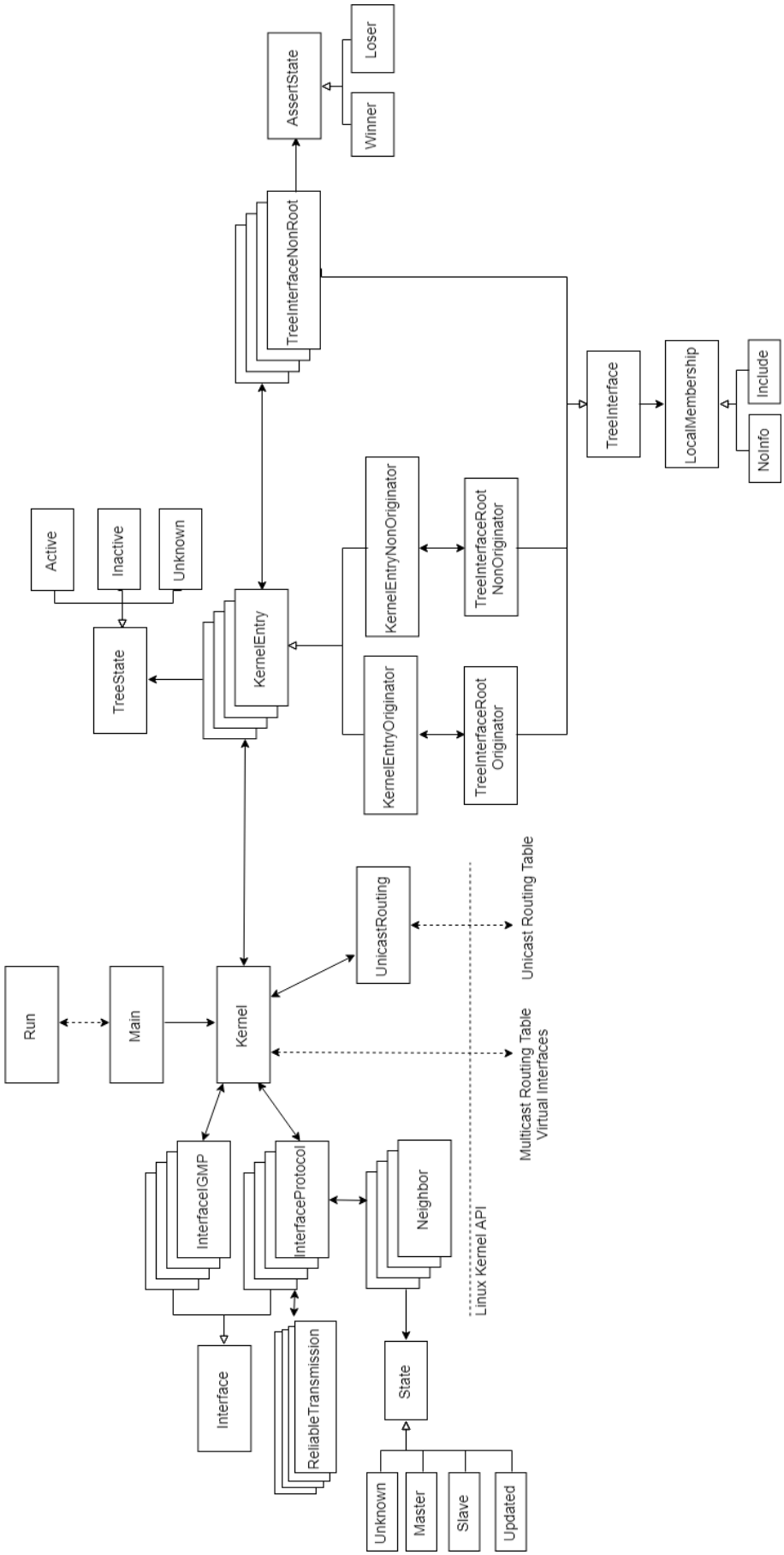

Figure 5.1: HPIM-DM implementation diagram.

abstraction of (S,G) entries of the multicast routing table, and the interfaces that have been enabled (for IGMP and for HPIM-DM).

- **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 in 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 as the root interface, which is represented by objects TreeInterfaceRootOriginator and TreeInterface-RootNonOriginator. The remaining interfaces will be non-root interfaces, represented by TreeInterfaceNonRoot object. It is required to distinguish between those types of interfaces, because depending on it, a router acts differently depending on the state of the tree;

- **TreeState, Active, Inactive and Unknown** - 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 and their RPCs, if the router is originator or non-originator and if data packets are being received;

- **TreeInterfaceRootOriginator** - represents an interface considered to be root, at a given (S,G) tree, 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, at a given (S,G) tree, that is not directly connected to the source of multicast traffic. This interface implements the state machine of root interfaces of non-originator routers in order to determine when control messages should be transmitted by this interface;

- **TreeInterfaceNonRoot** - represents an interface considered to be non-root at a given (S,G) tree. This interface implements the state machine of non-root interfaces in order to determine when control messages should be transmitted by this interface;

- **AssertState, Winner and Loser** - state used in non-root interfaces to determine if the interface is responsible for forwarding multicast data packets to a link;

- **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;

- **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 (RPC changes).

All details regarding data structures, threads and libraries is described in Appendix D.2.3.2. We have also implemented a bunch of commands in the Run module, in order for the user to interact with the protocol process. A full description of these commands can also be found in Appendix D.2.3.2.

Chapter 6

Tests

In order to check if the implementations were according to the specifications, some tests were done. All tests were performed using Netkit-NG [10], allowing to emulate a network environment. Additionally we have also used GNS3 [35] to verify the interoperability between our implementations executing in Linux devices and the implementations of Cisco routers, in case of IGMP and PIM-DM.

A test is defined by a certain topology in order to verify if a certain module of the protocol is working as expected. All messages exchanged between routers were captured using *tcpdump* [36] and further analyzed with *Wireshark* [37].

All tests were performed by knowing a priori the state in which some routers should be at. A centralized node knows this information and all routers send their state transitions to it. A test is considered to be successful when all predicted state transitions were made. This was accomplished by sending logs of the implementation to a server node, according to state transitions, exchange of control messages and expiration of timers. The remote logging was accomplished by the Python's *logging* module, that has an handler that can transmit all recorded logs to a given node. Regarding the exchanged control messages, these were manually verified with *Wireshark* after executing the tests.

We have performed tests to our three implementations. In this chapter, we list all tests performed to our protocol, HPIM-DM. The description and corresponding results of all our implementations can be found in Appendix E.

6.1 IGMP

To test IGMP we only require two or more routers connected to the same subnetwork. A detailed description of all tests and the corresponding results can be found in Appendix E.1.

6.2 PIM-DM

To test PIM-DM we require a topology with multiple routers and subnetworks in order to verify if trees are correctly constructed. We have subdivided these tests in multiple categories in order to verify if each

component of the protocol performs correctly.

A detailed description of all tests and corresponding results can be found in Appendix E.2.

6.3 HPIM-DM

All tests performed to our multicast routing protocol are enumerated here. These were subdivided in order to test each module of the protocol separately. A detailed description of the results can be found in Appendix E.3.

We have tested the formation and maintenance of neighborhood relationships, creation and maintenance of the broadcast tree, election of the AW, multicast forwarding decision based on interest, discovery of trees through synchronization and maintenance of trees in the presence of loops. All tests were successful.

6.3.1 Maintenance of neighborhood relationships

The tests regarding the establishment and maintenance of neighborhood relationships, without having any tree information exchanged in the synchronization process, are listed in Table 6.1. These tests were performed by having two routers connected to the same link. In these tests we have focused on the correct establishment and maintenance of neighborhood relationships in the presence of reboots and fails, without having any tree involved in the synchronization process.

	Test Description
1	Establishment of neighborhood relationship between two unknown neighbors, without any existing trees
2	Re-establishment of neighborhood relationship after a known neighbor reboots
3	Neighborhood relationship break after known neighbor fails

Table 6.1: Description of all tests to our protocol implementation, regarding the establishment and maintenance of neighborhood relationships.

In these tests, each router transmitted state transitions (logs) regarding their neighbors to the centralized node. Additionally, we used Wireshark in order to verify if messages were exchanged correctly (the fields of exchanged messages).

Each neighbor can be in one of four states: UNKNOWN, MASTER, SLAVE and UPDATED.

In Test 1, the test is successful when both routers consider each other to be in UPDATED state, after detecting and performing the synchronization successfully.

After Test 1, with both routers considering their neighbor to be in UPDATED state, we have restarted one router. In Test 2, we have verified if the router that is still alive detected correctly that its neighbor has rebooted. The centralized node considers this test to be successful when both routers eventually consider their neighbor to be in UPDATED state again, after they trigger a new synchronization, i.e. both routers first transition to the MASTER and SLAVE states before considering their neighbors to be in UPDATED state.

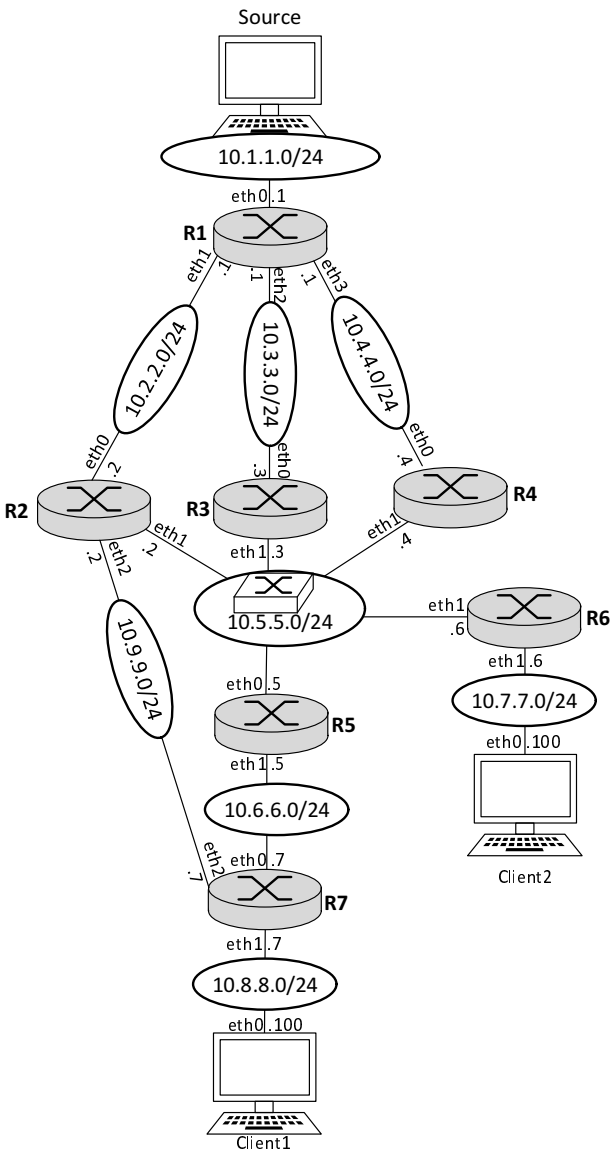

Figure 6.1: Topology used to test the creation and maintenance of the trees in both PIM-DM and HPIM-DM implementations.

In Test 3, with both routers considering their neighbors to be in UPDATED state, we have verified if the failure of one router is detected. In this test, due to lack of Hello messages, the router still alive considers its neighbor to be in UNKNOWN state. The centralized node considers this test to be correct when it obtains the log referring to this state transition.

In all tests we have verified manually if all fields of Sync messages were correct (Master and More flags, BootTime and SnapshotSN of both routers in exchanged Sync messages).

6.3.2 Formation and maintenance of trees

All following tests require a topology with multiple routers connected to each other in order to verify if all components of the protocol are correct. We used the topology represented by Figure 6.1 to perform most of the tests to each module of the protocol. We selected this topology because it offers multiple

redundant paths and has a shared link that connects multiple non-root interfaces. This allowed to tests each module of the protocol, allowing to verify its reaction to changes on the network.

Regarding the unicast routing protocol, each interface was set with a default cost of 10, this meant that the root and non-root interfaces were selected according to the "number of hops" in the sense that paths with fewer hops were preferred due to having a lower RPC. For this reason, the root interface, according to the subnet of Source, must be interface eth0 in all routers except for R7 that must be eth2.

	Test Description
4	Initial creation of the broadcast tree (root interface selection and tree state)
5	Broadcast tree reconfiguration in reaction to RPC changes detected by unicast routing protocol
6	Broadcast tree reconfiguration in reaction to router failure, detected by both the unicast routing protocol and the multicast routing protocol

Table 6.2: Description of all tests to our protocol implementation, regarding the formation of the broadcast tree.

The tests regarding the creation and maintenance of the broadcast tree are listed in Table 6.2. These were performed in the topology of Figure 6.1. In these tests we have focused on the creation of the broadcast tree, by verifying if root interfaces were selected correctly, and if that tree was being maintained correctly in the presence of network changes (RPC changes and failures) that could cause a change of interfaces' roles. Additionally, we have verified if the expected upstream messages were exchanged.

In these tests, each router by detecting the tree formation, transmitted to the centralized node the identifiers of their root and non-root interfaces. Additionally, their tree state transitions (ACTIVE/INACTIVE/UNKNOWN) were also transmitted to the centralized router. This allowed to verify if the tree was formed correctly at each router.

In Test 4, Source started sending multicast data packets triggering the formation of the tree. The centralized node considered this test to be correct when all routers transmitted to it that eth0 was their root interface, except for R7 that considered eth2 as its root interface. We have then manually verified if all routers transitioned to ACTIVE state, by analyzing the received logs at the centralized node. Additionally, we have verified through Wireshark if the expected IamUpstream messages were exchanged, and if they carried correct RPCs.

In Test 5, after having all routers in ACTIVE state and their root interfaces correctly selected, we have changed the cost of interface eth2 of R7 to 100. This allowed to verify if the router reacted to this change. The centralized node considered this test to be correct when R7 transmitted to it logs referring the new interface roles, i.e. at R7 eth2 was now non-root and eth0 was now root. Additionally we have manually verified if all routers remained in ACTIVE state and if R7 transmitted the correct IamUpstream and IamNoLongerUpstream by their new interface roles (IamUpstream through the new non-root interface and IamNoLongerUpstream through the new root interface).

In Test 6, after Test 5 we have crashed router R5. In this test we have verified if the multicast and unicast routing protocols detected correctly this failure: RPC changes and interface roles change at R7. This test was considered to be correct when the centralized node received logs of R7 in which it considered that eth2 was again root and eth0 now non-root (initial interface roles before the RPC change).

We have manually verified if all routers remained in ACTIVE state and if the expected control messages were transmitted by R7 (IamUpstream through the new non-root interface and IamNoLongerUpstream through the new root interface).

6.3.3 Election of AW

The tests regarding the election of the AW are listed in Table 6.3. These tests focused on the shared link that connects routers R2, R3, R4, R5 and R6 at the topology of Figure 6.1. We verified if all routers elected correctly the AW in the presence of network changes and failures.

	Test Description
7	Election of the AW in a shared link
8	Re-election of the AW in a shared link, in case the previous one fails
9	Re-election of the AW in a shared link, in case the previous one changes its interface role (its interface becomes root type)

Table 6.3: Description of all tests to our protocol implementation, regarding the election of the AW.

Routers R2, R3 and R4 are connected to the link by non-root interfaces. These can be in one of two states regarding the Assert state machine: AW or AL. Routers R5 and R6 elect the AW based on the stored upstream state, i.e. router UPSTREAM that offers the lowest RPC, also called BestUpstream-Neighbor. The Assert state transitions were transmitted to the centralized node, in order to verify if non-root interfaces correctly performed the election of the AW. Additionally, changes to upstream state at root interfaces triggered the election of their BestUpstreamNeighbor (at R5 and R6), being this information also transmitted to the centralized node.

Recall that all routers have set 10 as their interfaces' costs. This meant that R2, R3 and R4 offered the same RPC to the Source subnet (20). For this reason the election of the AW had the IP address of all UPSTREAM neighbors in consideration, meaning that R4 must be the elected AW by all routers.

In Test 7, the tree was initially built and we verified if R4 was elected correctly as the AW. The centralized node considered this test to be correct when R2 and R3 transmitted to it that they were in AL state and R4 transmitted to it that it was in AW state. We have manually verified if the centralized node received logs in which R5 and R6 considered R4 as the AW (BestUpstreamNeighbor).

In Test 8, we have crashed the AW (R4) on purpose. In this test we verify if all routers react to this failure correctly by reelecting a new AW. This test was considered to be correct when the centralized node received a log in which R3 considered itself to be in AW state. Additionally we have verified manually if R5 and R6 considered R3 as the new AW (BestUpstreamNeighbor).

In Test 9, after Test 8 we have changed the cost of eth0 at R3 to 100, in order for the current AW at the shared link to be connected to it by a root interface. The centralized node considered this test to be correct when R2 transmitted to it that it was in AW state. Additionally we have manually verified if R3, R5 and R6 considered R2 as the new AW (BestUpstreamNeighbor) and if R3 transmitted the correct upstream messages (IamNoLongerUpstream message to the shared link).

6.3.4 Forwarding decision

The tests regarding the forwarding decision based on interest of neighbors are listed in Table 6.4. These were performed in the topology of Figure 6.1. In these tests we have focused on the forwarding decision of the AW at the shared link that connects R2, R3, R4, R5 and R6 by having multiple routers changing interest.

	Test Description
10	Correct multicast forwarding decision based on neighbors' interests, after the tree is initially built
11	A neighbor becomes not interested (R6) but there are still other routers interested (R5)
12	A neighbor becomes not interested (R5) and there are no interested neighbors (R6)
13	All neighbors were not interested (R5 and R6) and one becomes interested (R6)

Table 6.4: Description of all tests to our protocol implementation, regarding the forwarding decision.

In these tests we have set the initial interface costs (10) at all routers, except for interface eth2 at R7 that was set to 100. In this configuration all routers consider eth0 as their root interface. This way the interest at the share link is controlled by both clients (Client1 controls the interest of R5, while Client2 controls the interest of R6).

A non-root interface can be in one of two states regarding the interest of its neighbors and hosts: DOWNSTREAM INTERESTED and NOT DOWNSTREAM INTERESTED. State transitions were transmitted to the centralized node in order to verify if the AW performed a correct forwarding decision.

The AW is R4. Routers R2 and R3 by being UPSTREAM are NOT INTERESTED. The interest of R5 and R6 control the downstream interest state machine at R4.

In Test 10, we have initially both clients interested in receiving multicast data. The centralized node considers this test to be correct when R4 transmits a log in which it refers that it is in DOWNSTREAM INTERESTED. We have manually verified if R5 and R6 transmitted correct interest messages to R4 (AW).

In Test 11, after Test 10, Client2 becomes not interested. In this test, router R4 must still be in DOWNSTREAM INTERESTED state. This test is considered correct when R6 transmits a log in which its non-root interface is placed in NOT DOWNSTREAM INTERESTED. Additionally we have manually verified if R6 transmitted a NoInterest message to R4.

In Test 12, after Test 11, Client1 becomes not interested. In this test, router R4 must be in NOT DOWNSTREAM INTERESTED. This test was considered correct when non-root interfaces of R7, R5, R4 and R1 were placed in NOT DOWNSTREAM INTERESTED, detected at the centralized node that received logs from these routers. Additionally we have manually verified if all routers exchanged correct interest messages.

In Test 13, after Test 12, Client2 becomes interested again. In this test, router R4 must be placed in DOWNSTREAM INTERESTED. The centralized node considers this test correct when non-root interfaces of R4 and R1 inform that they were placed in DOWNSTREAM INTERESTED. Additionally we have manually verified if interest messages were exchanged correctly.

6.3.5 Tree discovery through synchronization process

The tests regarding the formation of the tree by having this information exchanged in the synchronization process are listed in Table 6.5. These were performed in the topology of Figure 6.1. In these tests we have focused on the synchronization process, that exchanges information between routers that have been previously connected to a link but that have restarted their interface. In these tests we have created multiple trees with the same source but different groups, in order to tests the fragmentation of Sync messages.

	Test Description
14	Restart of non-root interface (R4)
15	Restart of root interface (R5)

Table 6.5: Description of all tests to our protocol implementation, regarding the discovery of a tree through the synchronization process.

After all trees are created, by IamUpstream messages, we have disabled and re-enabled some interfaces of routers in order to verify if they discover all trees through the synchronization process. We have focused on the shared link, by restarting interfaces of routers that are connected to it by a root and non-root interfaces: R4 and R5.

In Test 14, we have restarted eth1 at R4. In this test we have verified if a new synchronization is triggered with all neighbors. In this case all trees must be transmitted to all neighbors in Sync message, since R4 is ACTIVE and connected to the shared link by a non-root interface. This test is considered correct when the centralized node receives from R2, R3, R5 and R6 logs in which they consider R4 to be in UPDATED state (regarding the synchronization). We have manually verified if all tree were included in the Sync messages and if all routers considered R4 as UPSTREAM and AW and if it considers R2 and R3 as UPSTREAM.

In Test 15, we have restarted eth0 at R5. In this test we have verified if R5 discovered all UPSTREAM neighbors connected to this link, through the synchronization process. This test was considered correct when all routers considered R5 to be in UPDATED state (regarding the synchronization). We have manually verified if R2, R3 and R4 transmitted all trees in Sync messages and if R6 did not transmit those same trees in Sync messages, through Wireshark. Also, before re-enabling the interface we have manually verified through the multicast routing table if all routers below R5 no longer considered the tree to be alive (the chain relationship of UPSTREAM routers is broken due to the lack of UPSTREAM neighbors connected to the root interface of R5).

In these tests we have verified through Wireshark if the fragmentation of Sync messages was performed correctly. This was verified by analyzing the SyncSN and corresponding trees listed in each transmitted Sync message.

6.3.6 Tree maintenance in the presence of network loops

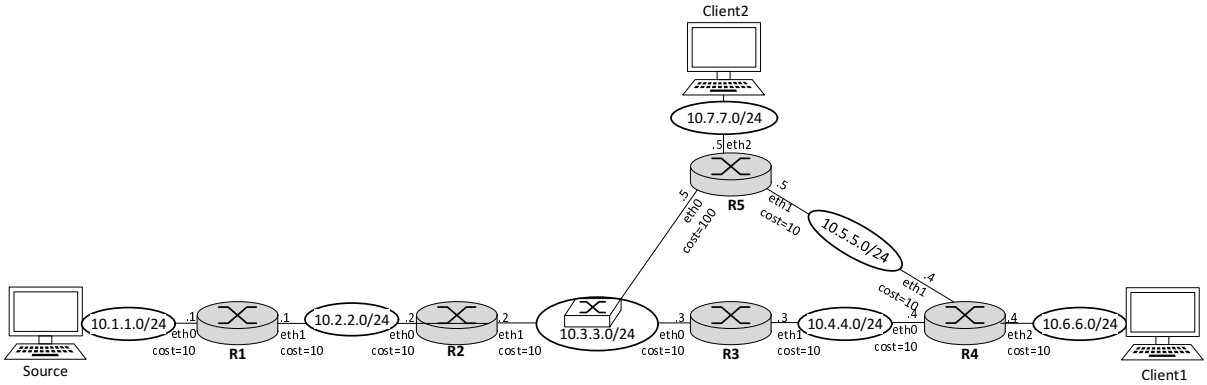

Figure 6.2: Topology used to test the maintenance of a tree in the presence of loops.

The existence of network loops can be dangerous in the maintenance of a (S,G) tree. For this reason we used a new topology, represented by Figure 6.2, allowing us to test this. We choose this topology because it forms a loop composed by routers R3, R4 and R5 (non-root interface of R5 is connected to root interface of R3; non-root interface of R3 is connected to root interface of R4; non-root interface of R4 is connected to root interface of R5).

Tests regarding the creation and removal of a tree in the presence of network loops are listed in Table 6.6. In these tests we have verified if the tree was correctly built and if it was then correctly removed, at all routers.

	Test Description
16	Tree formation in the presence of network loops
17	Tree removal in the presence of network loops

Table 6.6: Description of all tests to our protocol implementation, regarding the creation and removal of a tree in the presence of loops.

In Test 16, Source started transmitting multicast data, triggering the formation of the tree. This test was considered correct when all routers transmitted logs to the centralized node in which they considered to be in ACTIVE state and in which they have selected correctly their root interfaces. We have manually verified if all exchanged IamUpstream messages were correct.

In Test 17, after Test 16, Source stopped transmitting multicast data triggering the removal of the tree. This test was considered correct when all routers transmitted logs to the centralized node in which they considered to be in UNKNOWN state. We have manually verified if all exchanged IamNoLongerUpstream messages were correct.

A full description of the results to all these tests can be found in Appendix E.3.

Chapter 7

Conclusions

From the analysis regarding existing IP multicast routing protocols, we have identified several issues that limit their correctness and convergence time. These issues can cause multicast distribution trees to have sub-optimal configurations, in case there are network changes. These protocols do not have any order guarantees, regarding transmitted messages, which can cause routers to perform wrong decisions in case messages arrive out of order. Also, not all messages are reliably protected and, therefore, can be lost and not delivered to their destinations. In order to overcome these issues, existing protocols use a soft-state approach and reconstruct the multicast distribution trees periodically, even in stable configurations, causing excessive control and data overhead.

In this work we have specified, implemented and tested a novel multicast routing protocol, which is a hard-state version of PIM-DM, aiming at overcoming its limitations. The protocol is named HPIM-DM (Hard-state PIM-DM). This protocol has distinguishing features that stand out from existing protocols, such as maintenance of multicast distribution trees using an hard-state approach, allowing to reduce network overhead and the periodic reconstruction of trees present in PIM-DM. Routers maintain multicast distribution trees exclusively by control messages, instead of data packets, allowing routers to avoid the periodic reconstruction of trees in pruned links. The creation and removal of a tree is triggered by the originator router, but it can also be triggered by non-originator routers, in case they lose logical connectivity to the source of multicast traffic (in response to router failures). The multicast distribution tree is reconfigured only in reaction to network changes, as soon as they are detected, avoiding solving issues by the periodical reconfiguration of existing trees. This protocol has in consideration arbitrary behavior that can happen in network links, which may cause messages to arrive out of order and also to be lost, by sequencing all control messages and by having an ACK-protection mechanism, allowing to have order and reliability guarantees. Moreover, new routers that appear in the network, use a synchronization mechanism to exchange information regarding existing trees, allowing them to discover all previously formed trees as soon as possible.

The correctness of the specification was verified with logical reasoning and model checking, using SPIN. The synchronization and tree maintenance were verified with SPIN, confirming that all routers can reach a consistent state, even in the presence of router failures, network changes and network loops.

The resulting specification was implemented with Python programming language and was extensively tested in a network emulated environment.

7.1 Future Work

All implementations were performed having IPv4 in mind. It was implemented IGMPv2 for a router to understand when there are hosts interested in receiving multicast data packets destined to a given group. IGMPv3 was not implemented because the additional features were not crucial for the development of our protocol, HPIM-DM. For future work, IGMPv3 could be implemented in order for an host to specify interest regarding multicast traffic from a specific source and group.

Also, an IPv6 implementation of our protocol (HPIM-DM) could be performed. This can be easily achieved, since the only thing that would require modification is the interface that communicates with the API offered by Linux, to set and unset entries in the multicast routing table. Also, it would be required to implement the specification of MLD, to understand when IPv6 hosts are interested in receiving multicast traffic.

Finally, a hard-state version of PIM Sparse Mode could also be specified and implemented, to overcome the current limitations of PIM-SM. Many of the ideas that make HPIM-DM could also be used to improve PIM-SM.

Bibliography

[1] B. Williamson. *Developing IP Multicast Networks*. Cisco Press, 2000.

[2] A. Adams, J. Nicholas, and W. Siadak. Protocol Independent Multicast - Dense Mode (PIM-DM). RFC 3973, RFC Editor, January 2005. URL `https://tools.ietf.org/html/rfc3973`.

[3] D. Waitzman, C. Partridge, and S. Deering. Distance Vector Multicast Routing Protocol. RFC 1075, RFC Editor, November 1988. URL `https://tools.ietf.org/html/rfc1075`.

[4] B. Fenner, M. Handley, H. Holbrook, I. Kouvelas, R. Parekh, Z. Zhang, and L. Zheng. Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised). STD 83, RFC Editor, March 2016. URL `https://tools.ietf.org/html/rfc7761`.

[5] J. Moy. Multicast Extensions to OSPF. RFC 1584, RFC Editor, March 1994. URL `https://tools.ietf.org/html/rfc1584`.

[6] Protocols for IP Multicast (pim), IETF working group. URL `https://datatracker.ietf.org/wg/pim/documents/`. Online; accessed 08 October 2018.

[7] Promela Manual Pages. URL `http://spinroot.com/spin/Man/promela.html`. Online; accessed 05 October 2018.

[8] SPIN - Formal Verification. URL `http://spinroot.com/spin/whatispin.html`. Online; accessed 05 October 2018.

[9] W. C. Fenner. Internet group management protocol, version 2. RFC 2236, RFC Editor, November 1997. URL `http://www.rfc-editor.org/rfc/rfc2236.txt`.

[10] Netkit-NG. URL `https://netkit-ng.github.io/`. Online; accessed 14 November 2017.

[11] IPv4 Multicast Address Space Registry, . URL `https://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml`. Online; accessed 14 November 2017.

[12] IPv6 Multicast Address Space Registry, . URL `https://www.iana.org/assignments/ipv6-multicast-addresses/ipv6-multicast-addresses.xhtml`. Online; accessed 14 November 2017.

[13] E. Rosenberg. *A Primer of Multicast Routing*. Springer-Verlag New York, 2012.

[14] Y. K. Dalal and R. M. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 21(12):1040–1048, Jan 1978. doi: 10.1145/359657.359665.

[15] S. Deering, W. Fenner, and B. Haberman. Multicast listener discovery (mld) for ipv6. RFC 2710, RFC Editor, October 1999.

[16] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. Internet group management protocol, version 3. RFC 3376, RFC Editor, October 2002. URL `http://www.rfc-editor.org/rfc/rfc3376.txt`.

[17] R. Vida and L. Costa. Multicast listener discovery version 2 (mldv2) for ipv6. RFC 3810, RFC Editor, June 2004. URL `http://www.rfc-editor.org/rfc/rfc3810.txt`.

[18] C. Hedrick. Routing information protocol. RFC 1058, RFC Editor, June 1988. URL `http://www.rfc-editor.org/rfc/rfc1058.txt`.

[19] J. Moy. Ospf version 2. RFC 2178, RFC Editor, July 1997. URL `http://www.rfc-editor.org/rfc/rfc2178.txt`.

[20] R. Coltun, D. Ferguson, J. Moy, and A. Lindem. Ospf for ipv6. RFC 5340, RFC Editor, July 2008. URL `http://www.rfc-editor.org/rfc/rfc5340.txt`.

[21] E. W. Dijkstra. *A Note on Two Problems in Connexion with Graphs*. Numerische Mathematlk. URL `http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf`.

[22] J. Chroboczek. The babel routing protocol. RFC 6126, RFC Editor, April 2011. URL `https://tools.ietf.org/html/rfc6126`.

[23] C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.

[24] J. Garcia Luna Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Transactions on Networking 1:1*, February 1993.

[25] B. Albrightson and J. Garcia-Luna-Aceves and J. Boyle. EIGRP – a Fast Routing Protocol Based on Distance Vectors. *Proc. Interop 94*, 1994.

[26] M. Bhatia, S. Hartman, D. Zhang, and A. Lindem. Security extension for ospfv2 when using manual key management. RFC 7474, RFC Editor, April 2015.

[27] U. Chunduri, W. Lu, A. Tian, and N. Shen. Is-is extended sequence number tlv. RFC 7602, RFC Editor, July 2015.

[28] J. Moy. Ospf version 2. STD 54, RFC Editor, April 1998. URL `http://www.rfc-editor.org/rfc/rfc2328.txt`.

[29] A. Barbir, S. Murphy, and Y. Yang. Generic threats to routing protocols. RFC 4593, RFC Editor, October 2006. URL https://tools.ietf.org/html/rfc4593.

[30] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer-Verlag London.

[31] Welcome to Python.org. URL https://www.python.org/. Online; accessed 05 October 2018.

[32] K. Windisch. Protocol Independent Multicast - Dense Mode (PIM-DM): Routing Daemon Software. URL http://antc.uoregon.edu/PIMDM/pimd-dense.html. Online; accessed 1 November 2017.

[33] P. Oliveira. PIM-DM implementation. https://github.com/pedrofran12/pim_dm, 2018. Online; accessed 16 September 2018.

[34] P. Oliveira. HPIM-DM implementation. https://github.com/pedrofran12/hpim_dm, 2018. Online; accessed 16 September 2018.

[35] GNS3. URL https://www.gns3.com/. Online; accessed 1 January 2018.

[36] TCPDUMP/LIBCAP public repository. URL http://www.tcpdump.org/. Online; accessed 05 October 2018.

[37] Wireshark - Go Deep. URL https://www.wireshark.org/. Online; accessed 05 October 2018.

[38] J. Nilsson. SMCRoute - A static multicast routing daemon. https://github.com/troglobit/smcroute, 2018. Online; accessed 20 July 2018.

[39] J. Nilsson. An implementation of DVMRP for Linux & BSD. https://github.com/troglobit/mrouted, 2017. Online; accessed 20 July 2018.

[40] J. Nilsson. PIM-SM/SSM Multicast Routing for UNIX. https://github.com/troglobit/pimd, 2018. Online; accessed 20 July 2018.

[41] P. Radoslavov. *multicast – Multicast Routing*, September 2003. URL https://www.freebsd.org/cgi/man.cgi?query=multicast&sektion=4&manpath=FreeBSD+5.2-RELEASE.

[42] M. Kerrisk. *IP(7) Linux Programmer's Manual*, February 2018. URL http://man7.org/linux/man-pages/man7/ip.7.html.

[43] J. Schulist, D. Borkmann, and A. Starovoitov. Linux Socket Filtering aka Berkeley Packet Filter (BPF). URL https://www.kernel.org/doc/Documentation/networking/filter.txt. Online; accessed 07 October 2018.

[44] P. Jakma. Quagga Routing Suite. URL https://www.quagga.net/.

Appendix A

Alternative interest solution for HPIM-DM

The specified interest solution relaxes how interest state is maintained at routers. In the current solution, a router must only maintain its neighbors' interest state, as long as the router considers to be part of the tree, i.e. tree is in ACTIVE state. This means that a router by transitioning from ACTIVE to INACTIVE or to UNKNOWN states, it could remove interest state regarding its neighbors. This removal would be detected by its neighbors indirectly, through the exchange of IamUpstream/IamNoLongerUpstream messages.

In the current solution, a router knows that its neighbor is in ACTIVE state, through the reception of IamUpstream messages, from it. This means that after hearing an IamUpstream message, if a router then transmits an interest message, this state will be stored until the neighbor transmits an IamNoLongerUpstream message or a new IamUpstream message. In the first case, the reception of IamNoLongerUpstream message determines that the neighbor is no longer in ACTIVE state, thus it might no longer store our interest state. In the second case, by hearing a new IamUpstream message, after transmitting the interest, we have no guarantee if the router remained in ACTIVE state, thus it might no longer store our interest state, i.e. we do not know what triggered the transmission of this new message (e.g. it may remained in ACTIVE state and changed its RPC or the router transitioned from ACTIVE to INACTIVE and then to ACTIVE state again).

This means that an interest control message by being reliably transmitted, through the reception of ACKs, it does not guarantee that this state will be stored indefinitely by the neighbor router. For this reason, whenever the AW transmits a new IamUpstream message, if this router remains the AW, all neighbors will retransmit their interest to it. By retransmitting a new interest message after hearing the IamUpstream message, we guarantee that a router interest state will be stored, because the IamUpstream message was transmitted by the neighbor router when it was in ACTIVE state. By transmitting the interest message after the IamUpstream message, we guarantee that the interest message will be received in the ACTIVE state, if no more control messages from the neighbor router are exchanged.

This causes excessive exchange of messages, because there is no way to distinguish the following

events:

- Transmission of two IamUpstream messages, due to RPC change;
- Transmission of IamUpstream, followed by IamNoLongerUpstream and a new IamUpstream. The loss of IamNoLongerUpsteam causes both events to be indistinguishable.

If the interest state is intended to be stored at all routers in order to accelerate the convergence of the protocol in case the AW changes, the transmission of new interest messages when there is no interest change causes more messages to be exchanged. This is problematic, because a multicasted interest message requires to be acknowledged by all neighbors.

For this reason, in order to not exchange an interest message in case there is no change of the router's interest, instead of relying on the IamUpstream/IamNoLongerUpstream messages, in order to determine when the router may no longer store the interest of a router, the solution would be to preserve the interest state of a router, until that router informs otherwise. This solution would be similar to the upstream state, that is maintained until informed otherwise, through IamNoLongerUpstream message. For the interest state, we could use a new message, called RemoveInterest, that would allow a router to remove the interest state, from its neighbors, when the router no longer connects to Upstream neighbors. This allows to have the router that transmits control messages to decide when its neighbors should remove upstream and interest state, explicitly.

For this reason, we need to change the definitions of tree state in the current protocol specification:

- Interest state must be maintained until informed otherwise (through RemoveInterest).

  This means that the UNKNOWN state, is a state in which the router has no state information regarding all its neighbors (about upstream and interest), which have been removed explicitly by its neighbors, through IamNoLongerUpstream and RemoveInterest.

- ACTIVE state remains the same (a non-originator router is connected to an UPSTREAM neighbor via its root interface and the RPC of the own router is greater than the RPC of its neighbor; in case of an originator router the router is receiving data packets from the source, through its root interface);

- INACTIVE state must be reformulated, in order to include the maintenance of interest state. In the case of originator routers, this state defines that a router does not receive data packets from the directly connected source and at least one interface stores state about a neighbor (interest or upstream); In the case of non-originator routers, this state defines that a router is not connected to an UPSTREAM neighbor through its root interface, but stores interest or upstream state regarding one of its neighbors, connected to non-root interfaces, or the router is connected to an UPSTREAM neighbor via its root interface but the own router offers an RPC not greater than its UPSTREAM neighbor, or the router stores interest state regarding any neighbor;

The interest and upstream state will be removed from a router, by all its neighbors. Meaning that when a router transitions from ACTIVE to INACTIVE state, eventually all neighbors will remove state about themselves, causing the router to transition to UNKNOWN state.

When a router no longer connects to UPSTREAM neighbors, through any of its interfaces, it transmits a RemoveInterest through all its interfaces. This allows all neighbors connected to this router to remove interest state about it. Eventually all routers will perform this operation to each other, causing the total removal of interest state, which would cause all routers to transition to UNKNOWN state and no longer maintain state about this tree.

This way of maintaining interest state allows the own router to control the maintenance of its state, in all its neighbors. This means that the router knows exactly if all neighbors are maintaining interest state, by receiving the acknowledges of Interest and NoInterest messages, and when they no longer maintain this state, by receiving the acknowledge to the RemoveInterest message. This allows to guarantee that a neighbor maintains a router interest state, reducing the number of multicasted Interest/NoInterest messages, which would be only transmitted when the router changes its interest.

Also, interest information would be required to be included in Sync messages, in order for new routers to know about the interest of its neighbors.

Appendix B

HPIM-DM State Machines

This Appendix defines all specified state machines:

- **Appendix B.1** defines all state that must be stored at routers;
- **Appendix B.2** defines all events that the protocol must react to;
- **Appendix B.3** describes how message sequencing is performed;
- **Appendix B.4** defines the transmission of Hello messages;
- **Appendix B.5** defines the state machines related with the formation and maintenance of neighborhood relationships (synchronization process and reception of Hello messages);
- **Appendix B.6** defines the maintenance of upstream and interest state of neighbors through the reception of control messages, along with the transmission of ACKs and maintenance of SN information;
- **Appendix B.7** defines the state of trees;
- **Appendix B.8** defines the Assert state of non-root interfaces;
- **Appendix B.9** defines the Downstream Interest state of non-root interfaces;
- **Appendix B.10** defines the Forwarding state of non-root interfaces;
- **Appendix B.11** defines the interest state of the own router;
- **Appendix B.12** defines when an interface must transmit control messages;
- **Appendix B.13** defines reliability of transmitted control messages;
- **Appendix B.14** defines when routers can remove state about neighbors/trees;
- **Appendix B.15** defines an optimization to remove sequence information (CheckpointSN);
- **Appendix B.16** defines the default values of timers.

B.1 Stored state

For each interface:

- **BootTime** – Timestamp obtained when the interface booted up or when the SN overflowed;
- **InterfaceSN** – Last transmitted SN;
- **HelloTimer** – Timer used to control transmission of Hello messages;
- **NeighborList** - List/Dictionary with information regarding all known neighbors of this interface;
- **ReliableTransmissionDictionary** - Dictionary with information regarding all messages that are currently being reliably transmitted, for each (S,G) tree.

For each neighbor N:

- **State** – Synchronization state;
- **BootTime** – Timestamp obtained when the neighbor's interface booted up. Used to detect reboots/sequence number overflow and to filter messages transmitted prior to the synchronization;
- **MySnapshotSN** – SN when the own router created a snapshot, used to synchronize with this neighbor;
- **NeighborSnapshotSN** – SN of the neighbor router when it took its snapshot;
- **CheckpointSN** – SN of the neighbor router, used to save resources regarding stored sequence numbers;
- **CurrentSyncSN** – SN used for the fragmentation of Sync messages;
- **UpstreamStateDictionary** – Dictionary with information regarding which (S,G) trees neighbor N is considered as UPSTREAM (and its corresponding RPC);
- **InterestStateDictionary** – Dictionary with information regarding interest of neighbor N for each (S,G) tree;
- **LastSN** – Dictionary with information regarding the last received SN from this neighbor for each (S,G) tree;
- **SyncRetransmissionTimer** – Timer used to regulate the retransmission of Sync messages, while the synchronization is occurring with neighbor N;
- **NeighborLivenessTimer** – Timer used for determining when neighbor N is declared to have failed.

Each router stores:

- **Trees** - Dictionary with information regarding all known trees (that are in state different than UNKNOWN).

For each (S,G) tree:

- **State** – State of the tree (ACTIVE/INACTIVE/UNKNOWN), depending on the existence of Upstream neighbors connected to an interface and also whether the router is considered to be or not an originator;
- **Root Interface** – One interface elected as root;
- **Non-Root Interfaces** – All the remaining interfaces not considered as root;
- **MyRPC** – RPC of the own router to the source S (via unicast routing table).

Each root interface of non-originators' routers:

- **BestUpstreamNeighbor** – Neighbor connected to the root interface that considers itself as UPSTREAM and offers the best RPC to source S. This router will be responsible for forwarding data packets to the link that connects this root interface. Interest and NoInterest messages should be destined to this router.

Each root interface of originators' routers:

- **SourceActiveTimer (SAT)** – Timer used to determine when the tree/source is no longer declared to be active.

Each non-root interface:

- **BestUpstreamNeighbor** – Neighbor connected to a non-root interface that considers itself as UPSTREAM and offers the best RPC to the source S. This information will be used to determine if this interface should be considered the Assert Winner or Assert Loser of the link and also to transmit NoInterest messages in case the router considers the tree to be in INACTIVE state.

B.2 Definition of Events

Our protocol reacts to the following events:

- Reception of control messages;
- New neighbor;
- Failure of neighbor;
- Change of RPC;
- Change of interfaces' roles;
- New interface;
- Interface removal;
- IGMP/MLD state transitions.

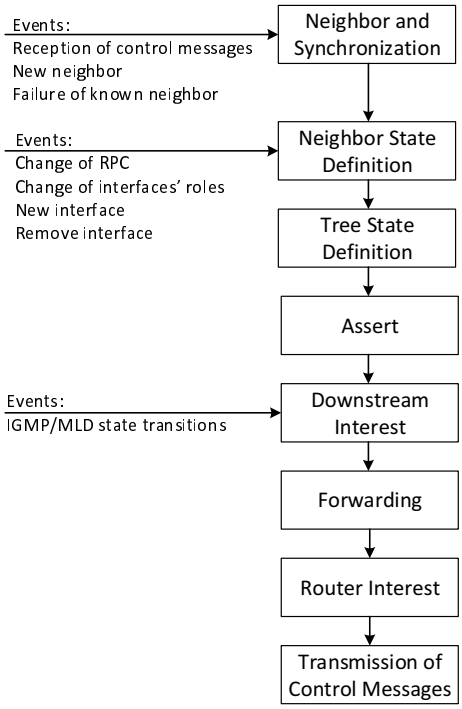

Figure B.1: Events and corresponding order by which state machines must be verified.

Any of the above mentioned events may cause the discovery of new trees, changes in existing trees or the complete removal of a tree. When any of these events happens, a bunch of verifications must occur through all state machines. Figure B.1 defines the order by which these verification must occur. This order must be respected since state machines below this hierarchy are dependent on the state of state machines above them. For example, the router interest is dependent on the Forwarding state of non-root interfaces.

B.3 Message sequencing

Each interface requires to store its BootTime.

Additionally, each interface must also store its last transmitted SN. This stored information will be called InterfaceSN. Following control messages and synchronization processes will use this information (increment the InterfaceSN).

The CheckpointSN is calculated per interface having in consideration all messages that are currently being transmitted and all message that have already been acknowledged. This information does not require to be stored and can be calculated on-demand (when it is required to be transmitted).

B.4 Hello protocol

This mechanism is used to discover new routers and also to detect failures of already known neighbors. By not hearing Hello messages from a given neighbor, we suspect that it has failed.

Table B.1 describes when Hello messages should be transmitted. This state machine uses one timer to regulate the transmission of those messages, called Hello Timer (HT).

The Hello messages must include the BootTime in order for neighbors to detect reboots/sequence number overflows. These messages do not include the other sequence number (SN), since they are not transmitting state. These messages must include the Hello Hold Time option to advertise to all neighbors their transmission periodicity of Hello messages (to detect failures). The Hello Hold Time must be 3.5 times the periodicity of Hello messages.

	Event	Action
1	Start routing process in interface I	Send Hello message Set HT
2	HT expires	Send Hello message Set HT
3	Interface I no longer participates in the routing process	Send Hello message informing that I no longer participates in routing process Clear HT

Table B.1: Transmission of Hello messages.

In event 1 of Table B.1, when a new interface is enabled in the multicast routing process, this interface requires to transmit an Hello message, in order to establish neighborhood relationships. Besides transmitting the message, the interface sets the HT, used to regulate the periodic transmission of those types of messages.

In event 2 of Table B.1, when the HT expires, a new Hello message must be transmitted. The HT is reset, in order to schedule the next Hello message transmission.

In event 3 of Table B.1, when an interface no longer participates in the multicast routing process, an Hello message should be transmitted in order to notify its neighbors. This is just to accelerate the removal of the neighbor, instead of relying on the absence of reception of Hello messages. Neighbors can be informed that a neighbor is being removed, by transmitting an Hello message with zero as lifetime (Hello Hold Time), like in PIM-DM. Since this interface will no longer transmit Hello messages, the HT is canceled.

Regarding the detection of new neighbors, this is described in the next subsection, by the Synchronization mechanism.

B.5 Neighbors and Synchronization mechanism

Each neighbor can be in one of four states:

- UNKNOWN - unknown neighbor;
- MASTER - neighbor is in an ongoing synchronization process and is the Master of this process;
- SLAVE - neighbor is in an ongoing synchronization process and is the Slave of this process;
- UPDATED - neighbor has successfully synchronized.

All neighbors are initially in UNKNOWN state, until they are discovered through the exchange of control messages. When a new synchronization process is triggered the neighbor transitions to a MASTER or SLAVE state, according to the one that initiates this process or according to the IP address of both routers. After the synchronization finishes successfully the neighbor is placed in UPDATED state and will remain in this state as long as it is alive. By considering a neighbor to have failed, it transitions to UNKNOWN state, regardless of which state it is at.

A router must store all types of sequence numbers, regarding itself and the neighbor router:

- neighbor's BootTime
- MySnapshotSN
- NeighborSnapshotSN
- CurrentSyncSN

Additionally, there are two specific timers used for each neighbor:

- NeighborLivenessTimer (NLT) - Used to detect failures of a known neighbor;
- SyncRetransmissionTimer - Used to control the retransmission of previously sent Sync messages, in case there is no proper response from the neighbor.

Listing B.1, describes in pseudocode how to process received Sync and Hello messages from a neighbor router and when its state must change.

```
New neighbor detected via non-Sync message or
NeighborSnapshotSN changes via Sync message or BootTime
changes during or after synchronization (via Sync or non-Sync message):
  -> SLAVE
  Clear stored info from neighbor
  MySnapshotSN = InterfaceSN++
  Take snapshot
  CurrentSyncSN = 0
  Set SyncRetransmissionTimer
  Set NLT
  Send Sync

Receive Sync message:
  If neighbor is SLAVE and Interface.BootTime == Message.NeighborBootTime and
        NeighborBootTime == Message.MyBootTime and
        Message.SyncSN == CurrentSyncSN:
    If Message.SyncSN == 0 and Message.MasterBit:
      If MyIP < NeighborIP:
        -> MASTER
```

```
        Send Sync                                                          20
      Else:                                                                21
        Resend last transmitted message                                    22
  Else If not Message.MasterBit and                                        23
        NeighborSnapshotSN == Message.MySnapshotSN and                     24
        MySnapshotSN == Message.NeighborSnapshotSN:                        25
      If Message.SyncSN > 0 and not Message.MoreBit and                    26
            ''I dont need to transmit more messages'':                     27
        -> UPDATED                                                         28
        Set NLT                                                            29
        Clear SyncRetransmissionTimer                                      30
      Else:                                                                31
        CurrentSyncSN++                                                    32
        Set NLT                                                            33
        Set SyncRetransmissionTimer                                        34
        Send Sync                                                          35
                                                                           36
If neighbor is MASTER and Interface.BootTime == Message.NeighborBootTime and 37
      NeighborBootTime == Message.MyBootTime and                           38
      NeighborSnapshotSN == Message.MySnapshotSN and                       39
      Message.SyncSN == CurrentSyncSN:                                     40
  If Message.MasterBit and (Message.SyncSN > 0 and                         41
          MySnapshotSN == Message.NeighborSnapshotSN                       42
          or Message.SyncSN == 0):                                         43
      Send Sync                                                            44
      If Message.SyncSN > 0 and not Message.MoreBit                        45
              and ''I dont need to transmit more messages'':               46
        -> UPDATED                                                         47
        Clear SyncRetransmissionTimer                                      48
      Else:                                                                49
        CurrentSyncSN++                                                    50
                                                                           51
If neighbor is UPDATED:                                                    52
  If Message.MasterBit and CurrentSyncSN == Message.SyncSN and             53
        Interface.BootTime == Message.NeighborBootTime and                 54
        NeighborBootTime == Message.MyBootTime and                         55
        MySnapshotSN == Message.NeighborSnapshotSN and                     56
        NeighborSnapshotSN == Message.MySnapshotSN:                        57
      Resend last transmitted message                                      58
```

```
    If neighbor is UNKNOWN and InterfaceBootTime == Message.NeighborBootTime:
        If Message.SyncSN == 0 and Message.MasterBit:
            -> MASTER
            MySnapshotSN = InterfaceSN++
            Take snapshot
            CurrentSyncSN = 0
            Send Sync
            Set NLT
            Set SyncRetransmissionTimer
        Else:
            -> SLAVE
            MySnapshotSN = InterfaceSN++
            Take snapshot
            CurrentSyncSN = 0
            Send Sync
            Set NLT
            Set SyncRetransmissionTimer

SyncRetransmissionTimer expires:
    Resend last transmitted message
    Set SyncRetransmissionTimer

NLT expires:
    -> UNKNOWN
    Clear SyncRetransmissionTimer
    Clear stored info from neighbor

Receive Hello message:
    If neighbor is UPDATED:
        Set NLT
```

Listing B.1: Pseudocode used in the synchronization process.

1) A new synchronization process must start, in which the router declares itself to be Master, when:

- New neighbor is detected via non-Sync message - by discovering a new neighbor that is not triggering the synchronization process, the interface takes the initiative, by declaring itself to be the Master of the synchronization process.
- Known neighbor sends a Sync message with same BootTime but a greater SnapshotSN - by hearing a Sync message, in which the known neighbor declares a greater SnapshotSN, this means

that it is triggering a new synchronization process.

- Known neighbor sends any type of control message with a greater BootTime - by hearing any type of control message, Sync or non-Sync message, with a greater BootTime, a new synchronization process must be started. Since there is no way to distinguish between router resets and sequence number overflows, a greater BootTime triggers always a new synchronization.

In the last two cases, if a router hears a Sync message from a known neighbor, that is triggering a new synchronization process, the router must declare itself to be the Master of this process. Both routers will initially declare to be Masters of this process, being the IP address used to break the "tie".

In any of the three cases described, the router must take a snapshot of its trees, obtain a SnapshotSN (which will be called MySnapshotSN), consider the neighbor router to be in SLAVE state and transmit a Sync message to it. The first Sync message must have information regarding the neighbor's BootTime, the own router's BootTime and MySnapshotSN. The NeighborSnapshotSN is not known, thus set to 0. The NeighborLivenessTimer and SyncRetransmissionTimer are set, for detecting neighbor failures during the synchronization and to control the retransmission of Sync messages, respectively.

2) When a router hears a Sync message that has correct information regarding the own router's BootTime, the neighbor's BootTime, the SyncSN (according to the CurrentSyncSN) and the NeighborSnapshotSN (except for the first transmitted message):

- From a neighbor that is considered to be in SLAVE state:
  - If that neighbor transmits a Sync message declaring to be the Master of the synchronization process (and if it is the first message - SyncSN=0), this means that both neighbors consider themselves as Masters of this process. In order to have progress in this process, the IP address of both neighbors is used, being the neighbor with the greatest IP address the Master and the other the Slave. If the router that receives this message has the lowest IP address, the neighbor is placed in MASTER state, and a Sync message is transmitted (in which it does not declare to be Master - Master flag cleared).
  - If the neighbor transmits a message in which it does not declare to be the Master of that process, it means that the synchronization is making progress. If no more messages are required to be transmitted (SyncSN>0, received More flag cleared and the last transmitted message, by the own router, had the More flag cleared), the neighbor is placed in UPDATED state, otherwise a Sync message must be transmitted.

    The SyncSN is required to be greater than 0 in order to terminate the synchronization process because both routers are required to reliably transmit all sequence numbers, which is only guaranteed if the SyncSN is greater than 0.

- From a neighbor that is considered to be in MASTER state:

  In this case, the received Sync message must have the Master flag set. Also, the MySnapshotSN (received message's NeighborSnapshotSN) must be correct (if the received SyncSN is greater than 0).

If it is guaranteed that the message corresponds to the current synchronization process and all sequence numbers are correct, a newer Sync message is transmitted, in response to the Master's message (with same SyncSN).

  - If both routers transmitted messages with the More flag cleared and the SyncSN is greater than 0, this means that the synchronization process has finished and the neighbor is placed in UPDATED state.
  - Otherwise, the router will wait for the reception of more Sync messages from it.

- If the neighbor is in UPDATED state and receives a Sync message having all sequence numbers correct:

  The router should resend the last transmitted Sync message if the neighbor declares itself to be Master (Master flag set). This retransmission is required because there is the possibility that the last transmitted Sync message, from the Slave, was lost.

- If the neighbor is in UNKNOWN state and receives a Sync message (with the correct BootTime):

  If the neighbor declares itself as Master (Master flag set) and is its first transmitted Sync message (SyncSN=0), then the router declares itself as the Slave of the synchronization process, meaning that the neighbor is placed in MASTER state and a newer snapshot is taken with its corresponding sequence number (MySnapshotSN). A Sync message is transmitted to the neighbor router declaring itself as Slave (Master flag cleared). Information about the SNs of the neighbor router (BootTime and neighbor's SnapshotSN), present in the received Sync message, must be stored.

  If the neighbor does not declare itself as Master (Master flag cleared) or is not its first transmitted Sync message (SyncSN not 0) then this corresponds to a previous synchronization process. For this reason the router takes a snapshot with the corresponding sequence number (MySnapshotSN). The router discovers the BootTime of the neighbor through the received Sync message. The router declares itself as the Master of the synchronization process, placing its neighbor in SLAVE state and transmits a new Sync message to it, with information regarding all known sequence numbers (BootTime of both routers and MySnapshotSN), SyncSN set to 0 and Master flag set.

When a newer Sync message is transmitted both the NeighborLivenessTimer and the SyncRetransmissionTimer are set, in order to detect failures during the synchronization and retransmit a Sync message in case no proper response is obtained. When the neighbor is placed in UPDATED state, the SyncRetransmissionTimer is canceled and the NeighborLivenessTimer must be set according to the Hello HoldTime transmitted in the Sync message or a default value is set;.

3) When the SyncRetransmissionTimer expires, it means that no proper response was obtained in due time. For this reason, the last transmitted Sync message is retransmitted and this timer is reset.

4) When the NLT expires, it means that the neighbor is considered to have failed. This can happen in an ongoing synchronization process or after this process has finished correctly. In both cases, the neighbor is placed in UNKNOWN state and all information about it is removed.

5) If an Hello message is received from a neighbor that is in UPDATED state, the NLT is reset, according to the value present in the received message. Reception of Hello messages during a synchronization do not cause a reset of this timer, because progress in the synchronization process is what defines the neighbor to be alive.

All trees received during the current synchronization process must be stored. All stored state is only valid after the neighbor router finishes successfully this synchronization process (neighbor transitions to the UPDATED state).

B.6 Neighbor state definition regarding (S,G) Tree

For a given neighbor, a router must monitor two types of information for each (S,G) tree:

- Upstream information - Whether the neighbor declares itself to be Upstream regarding the (S,G) tree. Two states are possible: UPSTREAM or NOT UPSTREAM;
- Interest information - Whether the neighbor is interested or not in receiving data packets regarding the (S,G) tree. Two states are possible: INTERESTED or NOT INTERESTED.

For a neighbor that declares itself as UPSTREAM, it is also required to store its RPC to source S, in order to elect the router responsible for forwarding multicast data packets in a given link. Also, in the case of a neighbor declaring itself as UPSTREAM, this also means that the neighbor is NOT INTERESTED.

Absence of upstream information, regarding a neighbor, must be considered as NOT UPSTREAM. Absence of interest information, regarding a neighbor, can be manually configured to be considered as INTERESTED or NOT INTERESTED, in order to allow or prevent the initial flooding behavior.

The maintenance of a neighbor state is controlled by the reception of control messages such as IamUpstream, IamNoLongerUpstream, Interest, NoInterest and Sync. The first four messages include a SN in order to filter messages that may arrive out of order and also to not reinterpret messages that have already been processed. So when one of these control messages is received, it is required to verify if it is considered to be “fresh”. The Sync messages will exchange initial state between two routers, and state regarding a router must only be considered as valid when this synchronization successfully finishes. When a synchronization finishes successfully, we have the guarantee that the information exchanged in Sync messages is correct.

In event 1 of Table B.2, if a router receives a “fresh” IamUpstream message from a given neighbor N, this means that N considers itself to be Upstream. The router reacts to this message by changing the state of N, in order to consider it as being UPSTREAM. The RPC included in this message must also be stored in order to determine the router responsible for forwarding data packets on the link. Since N is considered to be UPSTREAM, this implicitly means that it is not interested in receiving data packets, being this information also stored.

In event 2 of Table B.2, if a router receives a “fresh” IamNoLongerUpstream message from a given neighbor N, this means that N no longer considers itself to be Upstream. The router reacts to this message by setting N as NOT UPSTREAM regarding this tree. Regarding the interest, the router can

	Event	Action
1	Receive IamUpstream(S,G,RPC) with SN and BT from neighbor N in interface I and shouldProcess(S,G,N,SN,BT)	Set N as UPSTREAM regarding (S,G) with RPC Set N as NOT INTERESTED regarding (S,G)
2	Receive IamNoLongerUpstream(S,G) with SN and BT from neighbor N in interface I and shouldProcess(S,G,N,SN,BT)	Set N as NOT UPSTREAM regarding (S,G)
3	Receive Interest(S,G) with SN and BT from neighbor N in interface I and shouldProcess(S,G,N,SN,BT)	Set N as NOT UPSTREAM regarding (S,G) Set N as INTERESTED regarding (S,G)
4	Receive NoInterest(S,G) with SN and BT from neighbor N in interface I and shouldProcess(S,G,N,SN,BT)	Set N as NOT UPSTREAM regarding (S,G) Set N as NOT INTERESTED regarding (S,G)
5	Receive Sync message from neighbor N that included information about (S,G), with the corresponding RPC, from an ongoing synchronization process and no fresher state is stored	Set N as UPSTREAM regarding (S,G) with RPC Set N as NOT INTERESTED regarding (S,G)
6	Neighbor N fails or N is resynchronizing	Remove previously stored upstream and interest information about N

Table B.2: Maintenance of neighbor state regarding a given (S,G) tree.

set the interest of N as the default/omission interest. The neighbor would then transmit an Interest or NoInterest if the router that received the IamNoLongerUpstream is the one responsible for forwarding multicast traffic to the link, which would cause a correct set of its interest state.

In event 3 of Table B.2, if a router receives a “fresh” Interest message from a given neighbor N, this means that N is NOT UPSTREAM and is INTERESTED in receiving data packets.

In event 4 of Table B.2, if a router receives a “fresh” NoInterest message from a given neighbor N, this means that N is NOT UPSTREAM and is NOT INTERESTED in receiving data packets.

Event 5 of Table B.2 represents the reception of Sync messages from a neighbor that it is in an ongoing synchronization process with the own router. If the synchronization has already finished, following Sync messages referring the same synchronization process (same BootTimes and SnapshotSNs) must be ignored (in terms of storage of state). If the synchronization has not finished yet, and the Sync message is from an ongoing synchronization that is making progress, then tree information can only be stored if fresher state was not stored in the meantime (this can happen if control messages transmitted during the synchronization have already been processed regarding the neighbor router). In order to verify if more recent information regarding a given (S,G) tree, included in a Sync message, has already been stored, the router compares the SnapshotSN of the ongoing synchronization process with the SN of the last received non-Sync message regarding (S,G). If the router already stores a SN greater than the neighbor’s SnapshotSN, the router does not do anything (because it already stores fresher information), otherwise if there is no SN stored about (S,G) then N is considered as UPSTREAM (because the Sync message is transmitting fresher information).

In event 6 of Table B.2, if a router fails or if it triggers a new synchronization process, all stored upstream and interest state about it is removed. In case of a failure, the neighbor by no longer participating in the multicast routing process, all its state can be safely removed. In case of a new synchronization with a known neighbor, all stored state, before the new synchronization starts can be safely removed,

since fresher state will be exchanged in new Sync messages.

The shouldProcess referred in Table B.2 corresponds to a function used to determine whether a message must be processed. The following pseudocode shows which tests are used to determine if a received message is considered “fresh ”. If shouldProcess returns True, the message must be processed and the state of the neighbor must be set, otherwise the message must be ignored and the corresponding neighbor’s state not changed.

```
function shouldProcess(S,G,N,SN,BT):                                  1
If ''Neighbor N is UNKNOWN or N.CurrentSyncSN==0'':                   2
    Return False                                                      3
Else If SN<=N.NeighborSnapshotSN or BT!=N.BootTime or                 4
        SN<=N.CheckpointSN:                                           5
    Return False                                                      6
Else If SN==N.LastSN(S,G):                                            7
    Send ACK(S,G,SN) to N                                             8
    Return False                                                      9
Else If SN > N.LastSN(S,G):                                           10
    N.LastSN(S,G)=SN                                                  11
    Send ACK(S,G,SN) to N                                             12
    Return True                                                       13
Else:                                                                 14
    Return False                                                      15
```

Listing B.2: Pseudocode used to determine “fresh” messages.

Regarding Listing B.2, the first test to determine if a message should be processed (line 2) is if the source of the transmitted message is already considered to be a neighbor. If the neighbor is in UNKNOWN state, the message must be ignored, and possibly trigger a synchronization with it. If the router is in an ongoing synchronization process with it (neighbor in MASTER or SLAVE state), then the message must not be processed if the SyncSN of the current synchronization is 0. In case the synchronization is currently with a SyncSN equal to 0, then there is no guarantee that the exchanged BootTimes and SnapshotSNs are correct.

If the neighbor is in UPDATED state or if it is in an ongoing synchronization process with the CurrentSyncSN greater than 0, then there is the guarantee that the exchanged BootTimes and SnapshotSNs were reliably exchanged. In that case, a message must not be processed if the NeighborSnapshotSN or CheckpointSN are higher than the SN received in this message or if the BootTime is different from the one obtained during the synchronization process (lines 4-5). If the NeighborSnapshotSN is greater than the SN, this means that the synchronization process already exchanged fresher state, compared to the state included in this message. If the CheckpointSN is greater than the received SN, then the received message corresponds to a retransmission which has already been acknowledged by the own router. If the BootTime is different from the one obtained during the synchronization process, this

means that the message was transmitted before or after a reboot. In both cases the message must not be processed, but in case the received BootTime is greater than the one stored, a new synchronization with this neighbor should be triggered, since the stored state regarding the router may be inconsistent.

If the previous two test conditions were not true for the received message, then it may correspond to a simple retransmission of already stored state or may include fresher state that was not stored yet. If the last SN received from this neighbor, regarding the tree that this message is referring to, is equal to the message's SN (line 7), this represents a retransmission of a message that has already been processed. It may happen that the retransmission is occurring due to the loss of the acknowledgment, so in this case a ACK must be retransmitted and the content of the message should not be processed.

If the previous three test conditions were not true for the received message, then it may correspond to fresher state that was not stored yet. The message must only be processed if the received SN is greater than the last received SN regarding the same tree (line 10). In this case an ACK must be transmitted to the neighbor router, the SN must be stored as the most recent SN received regarding this tree and the message processed.

In any other case (line 14) the packet must be ignored, since it does not have fresher state compared to the one already stored by the router.

Required calculations based on stored state.

Based on the stored state regarding each neighbor, for all trees, an interface must determine which neighbor is considered to be UPSTREAM that offers the lowest RPC. To this neighbor we will call it BestUpstreamNeighbor. This can change whenever an interface receives any type of control messages, new neighbors appear on the network and known neighbors fail.

In order to facilitate the explanation of the following subsections, we define two sets of BestUpstream-Neighbors, for a given (S,G) tree:

- BURI - The BestUpstreamNeighbor of the (S,G) tree and its corresponding RPC, that is connected to the root interface, i.e. neighbor connected to the root interface that is considered to be UPSTREAM and that offers the lowest RPC, regarding the (S,G) tree;

- LBUNI - List of BestUpstreamNeighbors connected to non-root interfaces and their corresponding RPC, regarding the (S,G) tree. This list includes all non-root interfaces UPSTREAM neighbors, that offer the lowest RPC on their corresponding interface.

As we can see, the BURI and LBUNI are defined depending on the root and non-root interfaces. This means that changes of interface roles or the addition/removal of interfaces can change the BURI and LBUNI structures.

Also, changes of upstream state in neighbors can modify these structures, like the reception of any control message, the failure of neighbors that were considered to be UPSTREAM and also the discovery of new neighbors that are considered to be UPSTREAM.

If the BestUpstreamNeighbor of any interface suffers an RPC change this can modify the corresponding BURI/LBUNI structures.

In case there is no reachability regarding the source of a tree, due to the absence of entries in the unicast routing table regarding the subnet of the source, this means that there is no root interface. For this reason, all interfaces are considered to be non-root. This implies that the BURI must be empty.

Please note that if the BURI or LBUNI is empty or not set, this is because the router does not connect to any UPSTREAM neighbor in its root or non-root interfaces, respectively.

The BURI and LBUNI structures are very important because they define which directly connected neighbors offer the lowest RPC to the source of multicast traffic. This allow us to determine if the tree is built correctly and also if an interface is responsible for forwarding multicast traffic.

B.7 (S,G) Tree definition

A router to maintain state regarding a given tree, it must be logically connected to the source of multicast traffic. The originator will maintain state regarding a tree as long as the source keeps transmitting multicast data packets. A non-originator router will maintain state of a tree as long as there are UPSTREAM neighbors connected to it.

Since the state of a tree depends whether the source is or is not directly connected to a router, we will distinguish the state machines of originator and non-originator routers.

B.7.1 (S,G) Tree definition state machine of Originator router

This state machine controls the state of a given (S,G) tree, in which the source is directly connected to the router.

The state of a tree for originator routers depends on the reception of data packets from the directly connected source and also on the existence of UPSTREAM neighbors connected to non-root interfaces.

Trees of originator routers require a timer, called Source Active Timer, used to control the state of a tree. When a data packet is received from a directly connected source, the timer is set and the tree will keep in ACTIVE state as long as the timer does not expire. When this timer expires, the tree is no longer considered to be ACTIVE, due to the absence of data packets from the source.

Table B.3 shows how originator routers determine the state of a tree.

		ACTIVE	**INACTIVE**	**UNKNOWN**
1	Receive (S,G) data packets in root interface	Set SAT	→ ACTIVE Set SAT	→ ACTIVE Set SAT
2	SAT expires and LBUNI is non-empty	→ INACTIVE	NA	NA
3	SAT expires and LBUNI is empty	→ UNKNOWN	NA	NA
4	Received IamUpstream regarding (S,G) tree or new neighbor finishes synchronization and is considered as Upstream regarding (S,G) causing LBUNI to get non-empty			→ INACTIVE
5	Neighbor fails or new synchronization with known neighbor is triggered or received IamNoLongerUpstream(S,G) causing LBUNI to get empty		→ UNKNOWN	

Table B.3: Maintenance of (S,G) tree state of Originator router.

In event 1 of Table B.3, the reception of data packets from the root interface, i.e. interface that directly connects to the source, means that the tree is in ACTIVE state. This causes the SAT to be set. The tree will keep in ACTIVE state while the SAT does not expire.

In event 2 and 3 of Table B.3, the expiration of the SAT means that the tree is no longer ACTIVE due to the absence of data packets from the source. If the router connects to UPSTREAM neighbors via one of its non-root interfaces (event 2), this means that the tree transitions to INACTIVE state, otherwise (event 3) the tree transitions to UNKNOWN state. If the router transitions to INACTIVE state, this means that there are other routers that can connect to the source, due to the existence of redundant paths.

In event 4 of Table B.3, if the tree was in UNKNOWN state, this means that the router did not consider the source to be "active" and it was not connected to any UPSTREAM neighbor. If the tree was in UNKNOWN state and the router receives an IamUpstream message from a non-root interface, causing the router to be connected to at least one neighbor UPSTREAM in one of its non-root interfaces, this causes the tree to transition to INACTIVE state. The same thing can occur when a synchronization finishes successfully with a neighbor router that is connected to a non-root interface, and that neighbor is considered to be UPSTREAM regarding (S,G).

In event 5 of Table B.3, if the tree was in INACTIVE state, this means that the router did not consider the source to be "active" and it was connected to at least one UPSTREAM neighbor in one of its non-root interfaces. If due to a failure of a neighbor, removal of an interface or reception of a control message that causes this router to no longer be connected to UPSTREAM neighbors in any of its non-root interfaces, this causes the router to transition to UNKNOWN state. If a new synchronization is triggered with a known neighbor, all its state is ignored during this process, i.e. the router is not considered as UPSTREAM during a synchronization.

The definition of the tree state is important to determine which type of control messages an interface must transmit (explained in the next sections).

B.7.2 (S,G) Tree definition state machine of Non-Originator router

This state machine controls the state of a given (S,G) tree, in which the source is not directly connected to the router.

The state of a tree for a non-originator router depends solely on the existence of UPSTREAM neighbors connected to the router's interfaces and also on the RPC of the own router and of its UPSTREAM neighbors.

The (S,G) tree will be in:

- ACTIVE state - if BURI is non-empty and it offers a lower RPC compared to the own router's RPC to source S;
- INACTIVE state - if BURI is empty and LBUNI is non-empty or BURI is non-empty and it offers an RPC that is greater or equal to the own router's RPC to source S;
- UNKNOWN state - if BURI is empty and LBUNI is empty.

If the router is connected to an UPSTREAM neighbor via its root interface and a loop is impossible to exist (RPC offered by the BestUpstreamNeighbor connected to the root interface is lower than the own router' RPC), this means that the tree is in ACTIVE state.

If the router is not connected to UPSTREAM neighbors in its root interface (BURI is empty) but connects to at least one UPSTREAM neighbor in one of its non-root interfaces (LBUNI non-empty), this means that the tree is in INACTIVE state. The same INACTIVE state applies when there are UPSTREAM neighbors connected to the router's root interface (BURI non-empty) but there is the possibility of a loop (RPC of the BestUpstreamNeighbor not lower than the own router's RPC).

If the router does not connect to UPSTREAM neighbors in any of its interfaces (BURI and LBUNI empty), the tree is in UNKNOWN state.

The maintenance of UPSTREAM neighbors is very important and the tree's state should be rechecked whenever the BURI and/or LBUNI structures are manipulated. This manipulation may be due to the addition or removal of UPSTREAM neighbors, that can change the state of a tree. Changes regarding the root interface can also change the state of a tree, due to the existence/absence of UPSTREAM neighbors connected to the new root interface.

The RPC of the own router and of the BestUpstreamNeighbor connected to the root interface (BURI) is also important in the determination of a tree state. For this reason, any change to the RPC of the BestUpstreamNeighbor or of the own router, must trigger the verification of the new tree state.

When the interface's roles change, i.e. the root interface changes, the tree state must also be rechecked, since the new root interface may or may not be connected to UPSTREAM neighbors (changes to BURI and LBUNI).

The definition of the tree state is important to determine which type of control messages an interface must transmit (explained in the next sections).

B.7.3 (S,G) Non-Originator becomes Originator router

If a non-originator router has a new interface that directly connects to the source of a tree that was already being maintained, the router becomes originator.

If the router was non-originator and considered the tree as being ACTIVE, when it becomes originator, the tree should remain ACTIVE even if no data packet was received by the new root interface. This is to not trigger the removal of the tree. In this case, the router sets the Source Active Timer, in order to control when it should transition from ACTIVE to INACTIVE or UNKNOWN states. This is because if the router was previously ACTIVE then there was already a redundant path that connected this router to the source of multicast traffic, through another originator router, and if it was ACTIVE then the source was transmitting data packets.

If the router was INACTIVE (BURI empty and LBUNI non-empty), it should remain INACTIVE until data packets are received through the new root interface.

From this point on, the calculation of the tree state will be performed using the "Originator tree state definitions" (in section B.7.1).

B.7.4 (S,G) Originator becomes Non-Originator router

If a originator router disables its root interface, that directly connected to the source of multicast traffic, it becomes a non-originator router if it is no longer directly connected to the source.

This means that the tree must now be recalculated not considering the reception of data packets, through the new root interface. This calculation must be performed only by verifying the existence of UPSTREAM neighbors connected to the new root and non-root interfaces (BURI and LBUNI), like referred in section B.7.2.

B.8 Assert state of (S,G) regarding Non-Root interfaces

When two or more non-root interfaces connect to the same link, only one of them requires to transmit data packets in case there are routers interested in receiving those data packets, otherwise there will be duplication of data packets.

The Assert state machine has two states:

- **ASSERT WINNER (AW)** - interface is responsible for forwarding data packets to a given link;
- **ASSERT LOSER (AL)** - interface is not responsible for forwarding data packets to a given link.

The interface will be in an AW state, when:

- Tree is in ACTIVE state, and
- The interface offers the best RPC to source S, compared to all UPSTREAM neighbors connected to the interface. In case of a tie, the IP will be used to break the tie, being the greatest IP the winner

OR

- Tree is in INACTIVE state, and
- No UPSTREAM neighbor is connected to the interface

OR

- Tree is in UNKNOWN state

The first condition means that the tree is ACTIVE and the interface by offering the best reachability to source S must be responsible for forwarding data packets to the link.

In the second and third conditions, the tree is not ACTIVE, meaning that there is no logical connection between the root interface of a router with the source of multicast traffic. If the non-root interface does not connect to UPSTREAM neighbors (no BestUpstreamNeighbor connected to this non-root interface), there are no routers connected to the same link that logically connect to the source. In order to be able to forward the first multicast data packets that might arrive at the root interface before information

regarding UPSTREAM neighbors is known, the non-root interface should be in AW state. If the flood of initial data packets is not intended, the default interest information will be used to control this behavior.

Interfaces in AL state will not forward any multicast data packets, regardless of their neighbors' interests.

B.9 Downstream Interest in (S,G) at Non-Root interfaces

A non-root interface must determine if there is downstream interest of neighbors or directly connected hosts.

The detection of interest from neighbors is through their interest state (INTERESTED or NOT INTERESTED).

The detection of interest from hosts is through an independent protocol used for this matter. IGMP is used in IPv4 networks, while MLD is used for IPv6 networks.

A non-root interface can be in one of two states regarding downstream interest:

- **DOWNSTREAM INTERESTED** - there is interest through this non-root interface;
- **NOT DOWNSTREAM INTERESTED** - there is no interest through this the non-root interface.

A non-root interface is DOWNSTREAM INTERESTED when:

- At least one neighbor has its interest state set to INTERESTED, OR
- IGMP/MLD define this tree in a state where there is interest from directly connected hosts.

Or is NOT DOWNSTREAM INTERESTED otherwise.

This state machine must be reverified in reaction to changes at IGMP/MLD, due to the join/leave of hosts, and also in reaction to changes of interest of neighbors, their removal or their addition.

Initially an interface may not know the interest state of all its neighbors. All neighbors that are NOT UPSTREAM and that have not set their interest state, are initially considered as INTERESTED or NOT INTERESTED (configurable), until the router receives their corresponding interest.

This is used to set the initial state of the Downstream Interest state machine. This will be used to control the initial flooding behavior of the protocol.

By considering that all neighbors are initially in INTERESTED state (of the ones that have not transmitted yet their interest message), this can be used to flood initial data packets, in order for these to not be lost.

By considering that all neighbors are initially NOT INTERESTED (of the ones that have not transmitted yet their interest message), this can be used to prevent the flooding behavior of the multicast routing protocol. This will cause the first data packets to be lost, since all neighbors are initially not interested in receiving data packets.

B.10 Forwarding of (S,G) data packets

A non-root interface is responsible for forwarding multicast data packets that arrive at the root interface. Depending on the Assert state and on the Downstream Interest state, a non-root interface may or may not forward those data packets.

A non-root interface can be in one of two states, regarding the forwarding of data packets:

- **FORWARDING** - the non-root interface must forward data packets;
- **PRUNED** - the non-root interface must not forward data packets.

A non-root interface will be in a FORWARDING state when:

- The interface is in AW state, and
- The interface is in DOWNSTREAM INTERESTED state.

Or in PRUNED state otherwise.

As we can see, the Forwarding state of a non-root interface depends on the Assert and Downstream Interest state machines. For this reason, whenever one of these state machines changes its state, the forwarding decision must be reverified.

As we can see, by controlling the interest state of neighbors that have not transmitted yet their interest message, the router determines its initial Downstream Interest state. Routers by being in AW and DOWNSTREAM INTERESTED states, forward data packets. By configuring the initial Downstream Interest, this allows or prevents the flooding behavior.

B.11 Router/Root interface interest in (S,G) data packets

Depending on the forwarding state of non-root interfaces, a router may be interested in receiving data packets from its root interface.

There are two states in which a router/root interface can be at:

- **INTERESTED** - router is interested in receiving data packets because it must forward them to at least one non-root interface;
- **NOT INTERESTED** - router is not interested in receiving data packets because no non-root interface will forward them.

A router will be in INTERESTED state when:

- At least one non-root interface is in a FORWARDING state.

And NOT INTERESTED, otherwise (all non-root interfaces are in a PRUNED state).

This state will control the type of interest message that a router will transmit through its root interface, in order to notify the router responsible for forwarding multicast traffic about its interest.

B.12 Control message transmission regarding (S,G)

A router is responsible for notifying its neighbors about its state:

- Whether it is UPSTREAM or NOT UPSTREAM regarding a given tree;
- If NOT UPSTREAM, if it is INTERESTED or NOT INTERESTED in receiving data packets regarding a given tree.

In this subsection, we define when each type of control messages must be transmitted by a given interface type.

B.12.1 Control message transmission regarding (S,G) from Root interfaces of Originator routers

Root interfaces of originator routers do not transmit any type of control message, since they are directly connected to the source of multicast data packets.

B.12.2 Control message transmission regarding (S,G) from Root interfaces of Non-Originator routers

Root interfaces of non-originator routers can transmit three types of control messages, which are the IamNoLongerUptream, Interest and NoInterest. These control messages can notify respectively, that a router is no longer considered to be UPSTREAM and if it is INTERESTED or NOT INTERESTED in receiving data packets.

The transmission of IamNoLongerUpstream can happen when interfaces change roles, being the new root interface an interface that has previously notified its neighbors that it was considered to be UPSTREAM. Since the interface is now root type, it is required to inform its neighbors that this interface is no longer considered to be UPSTREAM, which is accomplished by transmission of this type of message.

The transmission of Interest or NoInterest messages are destined to the BestUpstreamNeighbor, which is the UPSTREAM neighbor responsible for forwarding data packets to the link that connects the root interface. The transmission of these types of messages can happen when the router changes interest or when the BestUpstreamNeighbor changes. The type of message that is transmitted depends whether the own router is INTERESTED or NOT INTERESTED.

Table B.4 summarizes all events and corresponding actions, regarding the transmission of control messages through the root interface of a non-originator router.

In event 1 of Table B.4, when interfaces change roles and the tree remains in an ACTIVE state, due to the existence of an UPSTREAM neighbor connected to the new root interface with a lower RPC than the own router's RPC, this interface must inform all its neighbors that it is no longer considered to be UPSTREAM. This is accomplished by the transmission of an IamNoLongerUpstream message. Also, the new root interface, by being an interface that receives data packets and forwards to non-root

	Event	Action
1	Interface role changes (Non-Root→Root) and tree remains in ACTIVE state	Send IamNoLongerUpstream(S,G) Send Interest (if INTERESTED) or NoInterest (if NOT INTERESTED) to BestUpstreamNeighbor
2	Interface role changes (Non-Root→Root) and tree was INACTIVE and transitions to ACTIVE state	Send Interest (if INTERESTED) or NoInterest (if NOT INTERESTED) to BestUpstreamNeighbor
3	Interface role changes (Non-Root→Root) and tree was ACTIVE and transitions to non ACTIVE state (INACTIVE or UNKNOWN) and BestUpstreamNeighbor is null	Send IamNoLongerUpstream(S,G)
4	Interface role changes (Non-Root→Root) and Tree was ACTIVE and transitions to INACTIVE and BestUpstreamNeighbor≠null (possible loop detected)	Send IamNoLongerUpstream(S,G) Send Interest (if INTERESTED) or NoInterest (if NOT INTERESTED) to BestUpstreamNeighbor
5	Interface role changes (Non-Root→Root) and tree remains INACTIVE and BestUpstreamNeighbor≠null	Send Interest (if INTERESTED) or NoInterest (if NOT INTERESTED) to BestUpstreamNeighbor
6	Interface role does not change and BestUpstreamNeighbor changes and is not null	Send Interest (if INTERESTED) or NoInterest (if NOT INTERESTED) to BestUpstreamNeighbor
7	Interface role does not change and BestUpstreamNeighbor does not change and router's interest changes (NOT INTERESTED→INTERESTED or INTERESTED→NOT INTERESTED)	Send Interest (if INTERESTED) or NoInterest (if NOT INTERESTED) to BestUpstreamNeighbor
8	Interface role does not change and receive IamUpstream from current BestUpstreamNeighbor and BestUpstreamNeighbor does not change	Send Interest (if INTERESTED) or NoInterest (if NOT INTERESTED) to BestUpstreamNeighbor

Table B.4: Message transmission through Root interface of non-originator routers.

interfaces, requires to inform the BestUpstreamNeighbor about its interest regarding (S,G) data packets via an Interest or NoInterest message.

In event 2 of Table B.4, when interfaces change roles and the tree transitions from INACTIVE to ACTIVE state, the new root interface requires to inform the BestUpstreamNeighbor about its interest regarding (S,G) data packets via an Interest or NoInterest message. This event is caused by both changes in the unicast routing table and also due to the presence of UPSTREAM neighbors connected to the new root interface which offer a lower RPC compared to the own router's RPC. In case of this event, it is not required to transmit an IamNoLongerUpstream, since all neighbors already know that this router is NOT UPSTREAM, because the tree was previously in INACTIVE state.

In event 3 and 4 of Table B.4, when interfaces change roles that cause the tree to transition from an ACTIVE state to INACTIVE state, the new root interface requires to first inform all neighbors that it is no longer considered to be UPSTREAM. The same is required when the router transitions from ACTIVE to UNKNOWN state. This is accomplished by the transmission of an IamNoLongerUpstream message. This transmission is required since the tree was previously in an ACTIVE state and the new root interface, which was an old non-root interface, informed its neighbors that it was considered to be UPSTREAM. If the new root interface does not connect to UPSTREAM neighbors (event 3), then no additional action is required. If the new root interface connects to UPSTREAM neighbors, but none of

them offer a RPC lower than the own router's RPC, then it is possible that a loop exists in the network and for this reason the tree is considered to be in INACTIVE state (event 4). In this last case, since there are UPSTREAM neighbors connected to the root interface, the router still requires to inform the UPSTREAM neighbor that offers the lowest RPC, about its interest regarding the reception of multicast data packets, via an Interest or NoInterest message.

In event 5 of Table B.4, when interfaces change roles, the tree remains in INACTIVE state and the new root interface connects to UPSTREAM neighbors, an interest message must be transmitted to the router that offers the lowest RPC. That router, the BestUpstreamNeighbor, will make a forwarding decision based on the interest of all neighbors, thus requiring the new root interface to transmit to it its intent.

Event 6 of Table B.4, describes what a root interface must do when the router responsible for forwarding multicast data, the BestUpstreamNeighbor, changes. This can happen due to a failure of the previous BestUpstreamNeighbor or reception of IamUpstream with a lower RPC from the new BestUpstream-Neighbor or reception of IamUpstream with a greater RPC from the previous BestUpstreamNeighbor or reception of IamNoLongerUpstream from the previous BestUpstreamNeighbor or there was no previous BestUpstreamNeighbor. When this happens, the new BestUpstreamNeighbor must be informed about the interest of this router, via the transmission of an Interest or NoInterest message.

In Table B.4, when there is no change of the BestUpstreamNeighbor but the router changes its interest regarding the reception of multicast data packets (event 7), the root interface must inform about its new interest to the BestUpstreamNeighbor, via an Interest or NoInterest message, if the router transitions respectively to INTERESTED or NOT INTERESTED state. This can happen due to a change of forwarding state in all or some of the router's non-root interfaces.

Event 8 of Table B.4, describes what a root interface must do when the current BestUpstreamNeighbor transmits an IamUpstream message, not causing a change of the BestUpstreamNeighbor. Since there is the possibility that the BestUpstreamNeighbor lost the interest state of this interface, it is required to retransmit an interest message to it.

In the case of the actions that require to transmit two messages, the order of these transmissions must be respected in order to have an interest message with a greater sequence number.

B.12.3 Control message transmission regarding (S,G) from Non-Root interfaces

Non-root interfaces of any router, originator or non-originator, can transmit three types of control messages, which are the IamUpstream, IamNoLongerUpstream and NoInterest. These control messages can notify respectively, that a router considers itself to be UPSTREAM, NOT UPSTREAM or NOT UPSTREAM and NOT INTERESTED in receiving data packets regarding a given tree.

The transmission of IamUpstream can happen when the tree transitions to an ACTIVE state or when the tree was already in an ACTIVE state but the router suffers an RPC change. This message will notify all neighbors connected to the interface, that this router considers itself to be UPSTREAM, with a given RPC.

The transmission of IamNoLongerUpstream can happen when the tree was ACTIVE and transitions to an INACTIVE or UNKNOWN state. This requires to notify all neighbors that this router no longer considers itself to be UPSTREAM regarding a given tree.

The transmission of a NoInterest message is destined to the BestUpstreamNeighbor and can happen when the tree is in INACTIVE state and there are UPSTREAM neighbors connected to the non-root interface. Since a non-root interface is just responsible for forwarding data packets, not receiving them, and by not being logically connected to the source via the root interface, this router requires to inform the BestUpstreamNeighbor connected to it about its disinterest.

Table B.5 summarizes all events and corresponding actions, regarding the transmission of control messages through a non-root interface.

	Event	Action
1	Interface role does not change and tree transitions to ACTIVE state (UNKNOWN→ACTIVE or INACTIVE→ACTIVE)	Send IamUpstream(S,G,MyRPC)
2	Interface role changes (Root→Non-Root) and tree remains or transitions to ACTIVE state	Send IamUpstream(S,G,MyRPC)
3	Interface role does not change and tree state ACTIVE→INACTIVE and BestUpstreamNeighbor==null	Send IamNoLongerUpstream(S,G)
4	Tree state ACTIVE→UNKNOWN	Send IamNoLongerUpstream(S,G)
5	Interface role does not change and tree state ACTIVE→INACTIVE and BestUpstreamNeighbor≠null	Send IamNoLongerUpstream(S,G) Send NoInterest(S,G) to BestUpstreamNeighbor
6	Interface role changes (Root→Non-Root) and tree state ACTIVE→INACTIVE and BestUpstreamNeighbor≠null	Send NoInterest(S,G) to BestUpstreamNeighbor
7	Tree remains in INACTIVE state and receive IamUpstream from BestUpstreamNeighbor and BestUpstreamNeighbor does not change	Send NoInterest(S,G) to BestUpstreamNeighbor
8	Tree remains in INACTIVE state and receive IamUpstream or IamNoLongerUpstream or neighbor fails and BestUpstreamNeighbor changes and BestUpstreamNeighbor≠null	Send NoInterest(S,G) to BestUpstreamNeighbor
9	Tree state UNKNOWN→INACTIVE due to reception of IamUpstream and BestUpstreamNeighbor≠null	Send NoInterest(S,G) to BestUpstreamNeighbor
10	Tree remains in ACTIVE state and MyRPC changes	Send IamUpstream(S,G,MyRPC)

Table B.5: Message transmission through Non-Root interface.

In event 1 of Table B.5, when the state of a tree transitions to ACTIVE, not due to an interface role change, the router must inform its neighbors that it is UPSTREAM regarding that tree. This is accomplished by the transmission of an IamUpstream message through non-root interfaces. The transition to ACTIVE state is due to the existence of UPSTREAM neighbors connected to the root interface of a router, which happens when the router hears for the first time an IamUpstream message through the root interface (with the neighbor RPC lower than the own router's RPC).

In event 2 of Table B.5, when the router suffers an interface role change causing a transition to an ACTIVE state, the router must inform its neighbors that it is UPSTREAM regarding this tree. This is accomplished by the transmission of an IamUpstream message through all non-root interfaces. If the router remains in ACTIVE state after the interface role changes, the new non-root interface must inform its neighbors that it is UPSTREAM, via the transmission of an IamUpstream message.

In event 3 of Table B.5, when the state of a tree transitions from ACTIVE to INACTIVE and there are no UPSTREAM neighbors connected to the non-root interface, the interface must inform its neighbors that it is no longer considered as UPSTREAM regarding that tree. This is accomplished by the transmission of an IamNoLongerUpstream message. This transition happens due to the absence of UPSTREAM neighbors or data packets on the root interface, depending whether the router is originator or non-originator, but the router is still connected to UPSTREAM neighbors via one of its non-root interfaces.

In event 4 of Table B.5, when the state of a tree transitions from ACTIVE to UNKNOWN, the interface must inform its neighbors that it is no longer considered as UPSTREAM regarding that tree. This is accomplished by the transmission of an IamNoLongerUpstream message. This transition happens when the router no longer connects to UPSTREAM neighbors and/or no longer receives data packets from the source, depending on whether the router is originator or non-originator.

In event 5 and 6 of Table B.5, when the state of a tree transitions from ACTIVE to INACTIVE, all its neighbors must not consider the router as UPSTREAM. If the interface role did not change, i.e. the interface is still non-root type, the router must first transmit an IamNoLongerUpstream message. If the non-root interface connects to UPSTREAM neighbors, the interface must inform the BestUpstream-Neighbor that it is not interested in receiving data packets, because it is a non-root interface. This is accomplished by the transmission of a NoInterest message destined to the BestUpstreamNeighbor, regardless of whether this interface changed or not its role.

In event 7 of Table B.5, when the state of a tree remains in INACTIVE state and the BestUpstream-Neighbor transmits an IamUpstream message that does not cause a change of the BestUpstream-Neighbor, the interface must retransmit a NoInterest message to it. This is performed because there is no guarantee that the BestUpstreamNeighbor is still maintaining interest state of this non-root interface, thus a NoInterest message is transmitted to it.

In event 8 of Table B.5, when the state of a tree remains in INACTIVE state and there is still an UPSTREAM neighbor connected to the interface and the BestUpstreamNeighbor changes, due to a failure or reception of IamUpstream or IamNoLongerUpstream messages, the router requires to inform the BestUpstreamNeighbor that it is not interested in receiving data packets, via a NoInterest message.

In event 9 of Table B.5, when the state of a tree transitions from an UNKNOWN to INACTIVE state and there is an UPSTREAM neighbor connected to the non-root interface, which will be the BestUpstreamNeighbor, the router must inform it that it is not interested in receiving data packets, via the transmission of a NoInterest message. This can happen when there was no UPSTREAM neighbors connected to any interface and a non-root interface receives an IamUpstream message.

In event 10 of Table B.5, when the state of a tree remains in ACTIVE state and the router suffers an RPC change, the non-root interface must notify all neighbors about this change. This is accomplished by the transmission of an IamUpstream message with the new RPC. This is required in order to reelect the router responsible for forwarding data packets.

In the case of the actions that require to transmit two messages, the order of these transmissions must be respected in order to have an interest message with a greater sequence number.

B.13 Control message reliability

Regarding Sync messages, these are reliably protected by a Stop-and-Wait mechanism, which has already been addressed in section B.5.

Control messages such as IamUpstream, IamNoLongerUpstream, Interest and NoInterest are reliably protected using an ACK-protection mechanism, meaning that those messages must be acknowledged by the corresponding receiver(s). The transmission of ACK messages, in reaction to the reception of control messages has already been addressed in Appendix B.6.

The reliability of transmitted control messages will be addressed in this section.

First of all, IamUpstream and IamNoLongerUpstream messages are destined to all neighbors attached to a link (destination IP address is multicast). This means that in order to consider that these messages have been reliably transmitted, an interface must receive the ACK from all known neighbors attached to that link. Regarding Interest and NoInterest messages, by being destined to a single router (destination IP address is unicast), the interface that transmitted those messages requires to wait for the reception of the ACK from the corresponding destination.

An interface must wait a given amount of time to allow its neighbor(s) to acknowledge the message. If after a given amount of time, the interface did not get the ACK from at least one neighbor that was supposed to acknowledge it, the message must be retransmitted. This retransmission will occur until all supposed neighbors ACK the message or the ones that have not acknowledged yet are declared to have failed. For this reason, a timer called RetransmissionTimer is used, that is reset whenever a newer message or retransmission occurs, for messages regarding a given interface and a given tree.

When a new multicast message is transmitted (IamUpstream or IamNoLongerUpstream), regarding a given (S,G) tree, all messages that refer the same tree, that have been previously transmitted and are waiting for the corresponding acknowledge can be “canceled”, in the sense that those previous messages are transmitting old state. Since a newer state is being transmitted, to all neighbors (multicast), messages with older state no longer are required to be monitored. This difference between old and newer state is given by the SN.

In the case of a unicast transmission (Interest or NoInterest), referring a given (S,G) tree and destined to a given neighbor, all previous unicast messages referring the same tree and same destination neighbor can be “canceled”, in the sense that newer state is being transmitted to the same destination neighbor. This can happen for example when the router changes interest and the previous interest message has not been acknowledged yet.

If there are being reliably transmitted multicast and unicast messages concurrently, being the unicast messages transmitted after the multicast message, an acknowledge to the unicast message also represents an acknowledge to the multicast message. This can only happen when IamNoLongerUpstream is being transmitted concurrently to Interest or NoInterest messages. Since the interest message has been transmitted after the IamNoLongerUpstream and since both messages indicate that a router is NOT UPSTREAM regarding a given tree, there is no loss of information if the neighbor does not hear the IamNoLongerUpstream message. Also, since the interest message would have a SN greater than

the other message, if the network changes the order of these messages, the neighbor would never acknowledge the IamNoLongerUpstream message, since it has a SN lower than the last received message regarding this tree (the interest message).

Also, in case of a message being transmitted concurrently to a neighbor that is synchronizing, if the router's SnapshotSN transmitted to the neighbor router is greater than the SN of the transmitted message, the neighbor router will never acknowledge this message, since fresher state was included in the synchronization process via Sync messages. For this reason, those neighbors are considered to have implicitly acknowledged.

The ACK of a given message must be unicasted to its source and it must include the identification of the tree, the corresponding SN, the BootTime of the source and destination routers and also the MySnapshotSN and NeighborSnapshotSN of the synchronization process. The ACK must only be accepted if these fields are correct.

B.14 Removal of neighbor's state regarding a tree

A router must maintain all information of its neighbors regarding upstream and interest. This information is obtained by the reception of control messages such as IamUpstream, IamNoLongerUpstream, Interest, NoInterest and Sync messages.

When the interface of a router transitions from root to non-root or vice-versa, and the tree is in a ACTIVE or INACTIVE state, it is safe to remove information regarding the interest state stored on those interfaces that was received until that happened. Information regarding neighbors in UPSTREAM state must never be lost and must be transitioned to the new interface type.

Regarding trees in an UNKNOWN state, that have been previously in an ACTIVE or INACTIVE state, all information of neighbors regarding those trees can be safely removed, since the router no longer is logically connected to the tree.

Regarding trees that transition to INACTIVE state, interest information can be safely removed since neighbors no longer consider this router as UPSTREAM and interest information will no longer be transmitted to it. Information regarding UPSTREAM neighbors must never be lost.

Interest information received in a root interface can be safely ignored, since the router is not responsible for forwarding data packets. If this interface then changes its role, i.e. becomes a non-root type, the transmission of an IamUpstream message would cause its neighbors to retransmit interest messages to it, if it is the UPSTREAM neighbor that offers the best RPC.

Sequence numbers must not be removed even when the tree transitions to UNKNOWN state. This is to prevent the recreation of the tree if an IamUpstream is replayed or received out of order. The next subsection suggests a mechanism to free resources that are being wasted in sequence numbers. Only that mechanism should be used to remove those sequence numbers.

B.15 Periodic removal of sequence numbers (CheckpointSN)

The CheckpointSN allows to periodically remove stored SNs at neighbors. This mechanism is optional and only allows to free resources at neighbor routers.

The CheckpointSN is included in transmitted Hello messages. These can be calculated and/or inserted in Hello messages at a periodicity greater than the one used to transmit Hello messages, i.e. include the CheckpointSN from N to N messages.

The CheckpointSN that a router should inform its neighbors must be:

- If control messages are being currently transmitted, the CheckpointSN must be the lowest SN of a control message that is currently being transmitted (and not yet acknowledged), minus one;
- If there are no messages being currently transmitted, the CheckpointSN must be the SN of the last transmitted message (stored at InterfaceSN).

The reason for decrementing by one the lowest sequence number that is currently being transmitted in a control message is to not cause a deadlock. If a message is currently being transmitted, and it was not acknowledged by all routers yet, this means that those routers still require to process a message with that sequence number, for that reason the CheckpointSN must be lower than the SN that is currently being transmitted in an ongoing message.

If the CheckpointSN is intended to be used, a router must store the CheckpointSN of each of its neighbors. This is received in Hello messages. A router by receiving the CheckpointSN from neighbor N:

- If the received CheckpoinSN is greater than the stored one (for N), then the stored CheckpointSN is updated and all stored SNs, about N, that are lower than it are removed.

B.16 Timers

HelloTimer (HT): controls the periodicity of Hello messages. Default value = 30 seconds.

NeighborLivenessTimer (NLT): Controls the liveness of a neighbor. In case of an ongoing synchronization, the default time is set to 10 seconds (neighbor in a state different than UPDATED). In case the neighbor has finished the synchronization successfully, this timer is set according to the received Hello Hold Time present in Sync or Hello messages (in case this option is not present, set to 105).

SyncRetransmissionTimer: Controls the retransmission of Sync messages in case the neighbor does not answer in due time. Default value = 3 seconds.

RetransmissionTimer: Controls the retransmission of non-Sync messages in case the neighbor(s) does not acknowledge in due time. Default value = 3 seconds.

SourceActiveTimer (SAT): Controls the liveness of an (S,G) source in originator routers. Default value = 210 seconds.

Appendix C

Promela verification of specified state machines

In this section we describe the models that were developed in order to prove that some specified state machines are correct, in the sense that there are no deadlocks or unwanted behavior. We opted to prove the correctness of these state machines through model checking.

All used code is stored in our GitHub repository [34] in a branch named “promela”.

In order to test any of the developed models, it is required to run the following commands:

```
spin -a FILE_NAME.pml
gcc -O2 -o pan pan.c
./pan -a
```

Listing C.1: Perform a test using spin.

Some files may fail execution due to lack of memory resources or due to the model being too big. One of the first lines, after running the third command would suggest what to do. In those cases the second command should be substituted by gcc -O2 -o pan pan.c -DCOLLAPSE -DVECTORSZ=1500, or a greater number in -DVECTORSZ argument.

We have modeled the specification of the synchronization process in order to verify if two routers always finish this process consistently. We have also verified if the maintenance of trees, through Upstream neighbors, is consistent.

We have not modeled the forwarding decision based on the exchange of interest information, neither the correct election of the router responsible for forwarding multicast traffic, because these are guaranteed by the reliability and ordering of exchanged messages, thus not tested.

We will start by describing the tests regarding the synchronization and then the maintenance of trees.

C.1 Synchronization

The first module to test was the specification of the synchronization process between two routers. In this process, two routers that initially did not know each other, must exchange information reliably. The Promela file used to verify the synchronization process is “new_specification_sync.pml” and can be accessed by visiting our GitHub repository [34] in a branch named “promela”. Parts of its code will be described in this section.

The synchronization process requires to monitor the following information:

- BootTime - of the own router and of the neighbor in order to detect reboots and SN overflows;
- MySnapshotSN - SN of the own router, when the snapshot was taken;
- NeighborSnapshotSN - SN of the neighbor router, when its snapshot was taken;
- CurrentSyncSN - SN of the fragmentation of Sync messages;
- Master flag - flag used to inform if the sender of the message considers itself as Master;
- More flag - flag used to determine if additional Sync messages are required to be exchanged.

In the model, trees will not be included in messages since those are not important for the test. What is important to test is if both routers finish the synchronization process correctly.

Besides the exchange of Sync messages, Hello messages will also be initially exchanged in order to trigger the synchronization. In the Hello messages, only the BootTime of the own router is included.

Routers communicate with each other using channels, which is already integrated in Promela. Loss of messages is not be tested in this model since the real specification implements a retransmission mechanism of the last transmitted message until the neighbor replies or is declared to have failed.

Regarding routers, these are modeled by proctypes which can be seen as “independent processes”. Each proctype monitors the state of the neighbor router (UNKNOWN, MASTER, SLAVE or UPDATED states) and all sequence numbers. This information is stored in global variables:

```
mtype neighbor_state[N] = unknown;  // store state of neighbor router
byte current_sync_sn[N] = 0;        // store info about current sync sn
byte my_boot_time[N] = 0;           // store boot time of own router
byte neighbor_boot_time[N] = 0;     // store boot time of neighbor router
byte my_snapshot_sn[N] = 0;         // store snapshot sn of own router
byte neighbor_snapshot_sn[N] = 0;   // store snapshot of neighbor router
```

Listing C.2: Global variables used to store state.

A router stores its information in these arrays according to its index, for example the router identified by index zero stores its information in index 0 of all referred arrays.

In order to perform tests of reboots and broken bidirectional relationships, during the synchronization process, an additional channel is used to notify a router when it should simulate those types of failures. A broken bidirectional relationship happens when there was a bidirectional relationship (during or after

synchronization) but one of those routers determines incorrectly that the neighbor router has failed, while the other router still considers its neighbor to be alive. The reboot is modeled by generating a greater BootTime and send messages with this new information to the neighbor router. The broken bidirectional relationship is modeled by removing all information stored regarding the neighbor router and transmitting an Hello message to this router, in order to trigger a new synchronization process.

```
chan ch[N] = [BUFFER_SIZE] of {mtype, byte, byte, byte, byte, byte, byte, bool, bool};
//<message_type, neighbor_id, my_boot_time, neighbor_boot_time, my_snapshot_sn,
//neighbor_snapshot_sn, sync_sn, master_bit, more_bit>;
chan fail_ch[N] = [BUFFER_SIZE] of {mtype}; //<failure_type>;
```

Listing C.3: Declaration of channels.

The declaration of channels is represented above, in Listing C.3. Channel ch represents the channel used to exchange Sync and Hello messages. In this channel there are 9 variables exchanged, being declared the type of each variable. The first mtype represents the type of message (Sync or Hello). The following 6 bytes represent respectively the identifier of the message source, the BootTime of the message source, BootTime of the neighbor router, MySnapshotSN, NeighborSnapshotSN and SyncSN. The last two bools represent respectively the Master and More flags. In the case of Hello messages, only the first three components are used (message type, neighbor identifier and BootTime of the own router).

The channel represented by fail_ch is used to inform the router that it should simulate a reboot or a broken bidirectional relationship. This last channel only declares one type of variable, mtype, which defines the type of failure to simulate.

The proctype of each Router, continuously verifies if it has messages in one of its channels.

When a message is in channel fail_ch, the corresponding failure is simulated. When a reboot behavior must be simulated, the router increments its BootTime, removes all stored information about the neighbor and transmits an Hello message with the new BootTime. When a broken bidirectional relationship must be simulated, the router removes all state regarding the neighbor router and then receives an Hello message.

When a message is in channel ch, the router acts accordingly to the state of its neighbor and all stored SNs. A greater BootTime or the same BootTime and greater NeighborSnapshotSN correspond to a new synchronization process. In case the BootTime and the SnapshotSN do not change, this corresponds to a message of the same synchronization process. The functions represented by `new_neighbor()`, `recv_sync_and_neighbor_is_unknown()`, `recv_sync_and_neighbor_is_master()`, `recv_sync_and_neighbor_is_slave()` and `rcv_sync_and_neighbor_updated()`, shown in Listing C.4, implement all actions that must be taken according to all stored state and the received Sync message.

```
proctype Interface(byte node_id; byte my_initial_boot_time;
   byte my_initial_snapshot_sn; byte total_of_sync_msgs){
...
do
```

```
5  :: nempty(ch[node_id]) && empty(fail_ch[node_id]) ->
6     atomic {
7     ch[node_id] ? msg_type(neighbor_id, rcv_neighbor_boot_time, rcv_my_boot_time,
8        rcv_neighbor_snapshot_sn, rcv_my_snapshot_sn, rcv_sync_sn, rcv_master_bit,
9        rcv_more_bit);
10    if
11    :: msg_type == hello ->
12       if
13       :: neighbor_state[node_id] == unknown ||
14             (rcv_neighbor_boot_time > neighbor_boot_time[node_id]) ->
15          new_neighbor(node_id, total_of_sync_msgs, neighbor_id, rcv_neighbor_boot_time,
16             rcv_neighbor_snapshot_sn, rcv_my_snapshot_sn, rcv_sync_sn, rcv_master_bit,
17             rcv_more_bit);
18       :: else ->
19          skip;
20       fi;
21    :: msg_type == sync ->
22       if
23       :: rcv_my_boot_time == my_boot_time[node_id] &&
24                neighbor_state[node_id] == unknown ->
25          recv_sync_and_neighbor_is_unknown(node_id, total_of_sync_msgs, neighbor_id,
26                rcv_neighbor_boot_time, rcv_neighbor_snapshot_sn, rcv_my_snapshot_sn,
27                rcv_sync_sn, rcv_master_bit, rcv_more_bit);
28       :: rcv_my_boot_time == my_boot_time[node_id] &&
29                neighbor_state[node_id] == master ->
30          recv_sync_and_neighbor_is_master(node_id, total_of_sync_msgs, neighbor_id,
31                rcv_neighbor_boot_time, rcv_neighbor_snapshot_sn, rcv_my_snapshot_sn,
32                rcv_sync_sn, rcv_master_bit, rcv_more_bit);
33       :: rcv_my_boot_time == my_boot_time[node_id] &&
34                neighbor_state[node_id] == slave ->
35          recv_sync_and_neighbor_is_slave(node_id, total_of_sync_msgs, neighbor_id,
36                rcv_neighbor_boot_time, rcv_neighbor_snapshot_sn, rcv_my_snapshot_sn,
37                rcv_sync_sn, rcv_master_bit, rcv_more_bit);
38       :: rcv_my_boot_time == my_boot_time[node_id] &&
39                neighbor_state[node_id] == updated ->
40          rcv_sync_and_neighbor_updated(node_id, total_of_sync_msgs, neighbor_id,
41                rcv_neighbor_boot_time, rcv_neighbor_snapshot_sn, rcv_my_snapshot_sn,
42                rcv_sync_sn, rcv_master_bit, rcv_more_bit);
43       :: else ->
```

```
44            ch[neighbor_id] ! hello(node_id, my_boot_time[node_id], 0, 0, 0, 0, false,
45                    false);
46        fi;
47    fi;
48    }
49 :: nempty(fail_ch[node_id]) ->
50    atomic {
51    fail_ch[node_id] ? (msg_type);
52    if
53    :: msg_type == reboot ->
54        neighbor_snapshot_sn[node_id] = 0;
55        neighbor_boot_time[node_id] = 0;
56        current_sync_sn[node_id] = 0;
57        neighbor_state[node_id] = unknown;
58        my_boot_time[node_id] = my_boot_time[node_id] + 1;
59        my_snapshot_sn[node_id] = 1;
60        ch[(node_id + 1) % 2] ! hello(node_id, my_boot_time[node_id], 0, 0, 0, 0, false,
61                false);
62    :: msg_type == broken_bidirectional_relationship ->
63        neighbor_snapshot_sn[node_id] = 0;
64        neighbor_boot_time[node_id] = 0;
65        current_sync_sn[node_id] = 0;
66        neighbor_state[node_id] = unknown;
67        my_snapshot_sn[node_id] = my_snapshot_sn[node_id] + 1;
68    fi;
69    }
70 od;
71 }
```

Listing C.4: Code of router (proctype).

The test was performed in the following conditions:

- random required number of Sync messages to be exchanged (each proctype required between 0 to 2);
- the initial BootTime was explicitly selected (router 0 initiated with 15 and router 1 initiated with 125);
- the initial interface SN that will be used to obtain MySnapshotSN was also explicitly selected (router 0 initiated with 12 and router 1 initiated with 75);
- random selection of the router that triggers the synchronization process;

- random selection of the failure and of the router that must simulate it.

These conditions are specified in the init function, which represents the “main”, where the proctypes can be started and where initial configurations can be chosen. Regarding the random selection of required Sync messages, Listing C.5 represents the initial code of the init, where SPIN randomly selects the total number of Sync messages required to be exchanged by each process. The function `select()` is already implemented by Promela and allows the model to select a random number to be set at a given variable. In this code, the model will test all combinations of required Sync messages to be exchanged. The number of required Sync messages will be used to set the More flag of each transmitted message. Then, after selecting the total number of required messages, for each router, we calculate the final SyncSN that both routers must have in order to declare the synchronization process as finished. This would be the maximum required number of Sync messages, between both routers, plus one (last exchange will have the More flag cleared). The variable `TOTAL_MSGS_SYNC` will be used to verify if both routers end the synchronization process with the same SyncSN.

```
1  init {
2     byte total_msgs_of_process_0, total_msgs_of_process_1;
3     select(total_msgs_of_process_0: 0 .. 2);
4     select(total_msgs_of_process_1: 0 .. 2);
5     TOTAL_MSGS_SYNC = total_msgs_of_process_0 + 1;
6     if
7     :: total_msgs_of_process_1 > total_msgs_of_process_0 ->
8        TOTAL_MSGS_SYNC = total_msgs_of_process_1 + 1;
9     :: else ->
10       skip;
11    fi;
12
13 ...
```

Listing C.5: Model the random selection of required Sync messages to be exchanged by each process.

Regarding the explicit selection of the initial BootTime and interface SN of each process, these are not important to be chosen randomly by the model checker, thus not modeled since these do not influence the synchronization behavior. In a real system, these initial configurations would be determined at runtime when both routers discover each other.

The transmission of initial Hello messages that trigger the synchronization process is also modeled in order to test the Master and Slave behavior. It can happen that one or both routers transmit the Hello message. A router by receiving an Hello message from an unknown neighbor would declare itself as the Master of the synchronization process. This is also modeled in the init function and is shown below:

```
1 init{
2 ...
3 if
```

```
:: true ->
   ch[0] ! hello(1, 15, 0, 0, 0, 0, false, false);
:: true ->
   ch[1] ! hello(0, 125, 0, 0, 0, 0, false, false);
:: true ->
   ch[0] ! hello(1, 15, 0, 0, 0, 0, false, false);
   ch[1] ! hello(0, 125, 0, 0, 0, 0, false, false);
fi;
...
}
```

Listing C.6: Model the transmission of the first Hello message.

Listing C.6 shows how to model the transmission of the first Hello message. This if statement has 3 branches that have their conditions evaluated as true. This allows the model checker to test all conditions. In the first branch, router 1 sends to router 0 an Hello message. In the second branch, router 0 sends to router 1 an Hello message. In the third branch, both routers send Hello messages to each other. The model checker, SPIN, will verify if the LTL properties hold whatever branch is selected.

After the first Hello message is transmitted, both routers would start the synchronization mechanism. In order to test what happens when one or both routers reboot or have a broken bidirectional relationship during or after the synchronization process, also in the init function the following if statement is used:

```
init {
...
if
:: true ->
   fail_ch[0] ! reboot;
:: true ->
   fail_ch[1] ! reboot;
:: true ->
   fail_ch[0] ! reboot;
   fail_ch[1] ! reboot;
:: true ->
   fail_ch[0] ! broken_bidirectional_relationship;
   ch[0] ! hello(1, my_boot_time[1], 0, 0, 0, 0, false, false);
:: true ->
   fail_ch[1] ! broken_bidirectional_relationship;
   ch[1] ! hello(0, my_boot_time[0], 0, 0, 0, 0, false, false);
:: true ->
   fail_ch[0] ! broken_bidirectional_relationship;
   fail_ch[1] ! broken_bidirectional_relationship;
```

```
20      ch[0] ! hello(1, my_boot_time[1], 0, 0, 0, 0, false, false);
21      ch[1] ! hello(0, my_boot_time[0], 0, 0, 0, 0, false, false);
22  fi;
23  ...
24  }
```

Listing C.7: Model failures during the synchronization process.

To test these types of failures during or after the synchronization, an if statement with six branches evaluated as true is used. If the first branch is selected, router 0 is the one that reboots. If the second branch is selected, router 1 is the one that reboots. If the third branch is selected, both routers reboot during the synchronization process. If the fourth branch is selected, router 0 declares incorrectly that router 1 has failed. If the fifth branch is selected, router 1 declares incorrectly that router 0 has failed. If the sixth branch is selected, both routers declare incorrectly that their neighbor has failed. In the case of broken bidirectional relationships, an Hello message is transmitted to the router that suffered this type of failure in order to trigger a new synchronization process. In the real protocol this is accomplished by the periodic transmission of Hello messages, which is not modeled here. The transmission of Hello messages is modeled here by transmitting this type of message whenever it is required.

All branches will send a message to the fail_ch channel of a given router, in order to inform it that it should simulate a specified behavior.

In order to verify if the specification is correct, LTL was used. The correctness property that we want to check is the following: "Eventually, both routers will always finish the synchronization process considering their neighbors to be in UPDATED state and will always have the same view regarding their BootTimes, SnapshotSNs and SyncSNs".

This property can be specified in LTL by: <>([](neighbor_state[0] == updated && neighbor_state[1] == updated && current_sync_sn[0] == TOTAL_MSGS_SYNC && current_sync_sn[1] == TOTAL_MSGS_SYNC && my_snapshot_sn[0] == neighbor_snapshot_sn[1] && my_snapshot_sn[1] == neighbor_snapshot_sn[0] && my_boot_time[0] == neighbor_boot_time[1] && my_boot_time[1] == neighbor_boot_time[0])))

C.2 Tree creation and maintenance

The second module to test was the formation and removal of a tree. This process is triggered by the exchange of IamUpstream(S,G,RPC) and IamNoLongerUpstream(S,G) messages between routers.

Since this formation is dependent on the existence of UPSTREAM routers connected to a root interface, in order to test correctly, the definition of a network topology was required for these tests.

We have performed tests in three topologies that have already been illustrated in section 4.2, in Figures 4.1, 4.2 and 4.3. In these figures, all routers are identified by a given number and their interfaces are also identified by a given number (in Topology 1, Router 1 is connected by interface 3 to interface 1 of Router 0). These identifiers are important in the model that simulates the behavior of each router.

We consider that Router 0 is the originator router, with interface 0 being directly connected to the

source. All the remaining routers are non-originators by not being directly connected to the source.

We will start by describing how the model was implemented and then the reason for using these three topologies and the explanation of each test.

```
typedef NEIGHBOR_STATE{
    mtype neighbor_state[NUMBER_OF_INTERFACES] = no_info;
    byte my_rpc[NUMBER_OF_INTERFACES] = 255;
}

typedef NODE_CONFIGURATION {
    mtype tree_state = unknown_tree;
    mtype node_interface[NUMBER_OF_INTERFACES] = not_interface;
    bool luri = false;
    short luni = 0;
    byte my_rpc = 0;
    short neighbors_at_each_interface[NUMBER_OF_INTERFACES] = 0;
    NEIGHBOR_STATE neighbor_state[NUMBER_OF_INTERFACES];
}


NODE_CONFIGURATION node_info[N];
```

Listing C.8: Data strutures that store state for each router.

Each router stores all information in global variable “node_info”. A router manipulates this variable at an index represented by its router identifier (Router 0 manipulates only index 0 of node_info). This global variable is of type NODE_CONFIGURATION which is a typedef composed by multiple variables:

- tree_state - represents the current state of the tree (ACTIVE, INACTIVE or UNKNOWN). Initially all routers have the tree set to UNKNOWN state.
- node_interface - is an array of mtypes. This variable defines the types of interface at each router. For example, Router 0 (index 0) would consider interface 0 as root and interface 1 as non-root in all topologies. In Topology 1 and 2, interface 2 is also considered to be non-root type. All the remaining interfaces are set as not_interface.
- luri - is a boolean that controls if the root interface has UPSTREAM neighbors connected to it, with an RPC lower than the own router’s RPC. This is used for recalculating the tree state whenever an UPSTREAM neighbor is added or removed.
- luni - is the same as luri but controls which non_root interfaces have UPSTREAM neighbors. This is represented as a short because it is modeled in a binary format (if interface 1 is the only non_root interface that connects to UPSTREAM neighbors, luni would be set to 2 (00000010) - this is modeled this way to save resources).

- my_rpc - represents the router's RPC at its unicast routing table.
- neighbors_at_each_interface - is an array of shorts that represents which neighbors are connected to each interface. A neighbor is represented by its interface identifier. Regarding Router 0, in Topology 1 and 2, all indexes would be set to 0 except for index 1 (interface 1) that is connected to interface 3 (would be set to 8 - 00001000) and index 2 (interface 2) that is connected to interface 4 (would be set to 16 - 00010000).
- neighbor_state - same as above, but used to set the state of each neighbor connected to each interface of a router (set to upstream, not_upstream or no_info and also the RPC of each UPSTREAM neighbor). This variable is of type NEIGHBOR_STATE and an index at this variable corresponds the identifier of an interface. NEIGHBOR_STATE is a typedef composed by another neighbor_state and my_rpc. These two variables store the state of each neighbor connected to the router's interface. An index in these two variables correspond to the interface identifier of a neighbor. For example, if interface 3 of Router 1 considers interface 1 of Router 0 as being UPSTREAM with an RPC of 10, this would be represented by the following code:
  node_info[1].neighbor_state[3].neighbor_state[1] = upstream and
  node_info[1].neighbor_state[3].my_rpc[1] = 10.

```
chan ch[NUMBER_OF_INTERFACES] = [BUFFER_SIZE] of {mtype, byte, byte};
//<msg_type, neighbor_id, rpc>;
```

Listing C.9: Channel used to exchange messages.

Each interface has a channel used to exchange messages between its neighbors. Each message carries information regarding its type (IamUpstream or IamNoLongerUpstream), the message's source (interface index) and also the advertised RPC.

Instead of modeling a router, we have decided to model each interface. Each interface is modeled by two proctypes, one for the reception of messages and the other for the transmission of messages. Each interface has information regarding its index and the identifier of the router it belongs. Both proctypes are represented in Listings C.10 and C.11.

```
proctype InterfaceReceive(byte node_id; byte interface_id) {
...
do
:: nempty(ch[interface_id]) ->
   atomic {
   ch[interface_id] ? msg_type(neighbor_id, neighbor_rpc);
   if
   :: IS_NEIGHBOR(node_id, interface_id, neighbor_id) ->
      if
      :: msg_type == msg_i_am_upstream ->
         NEIGHBOR_STATE(node_id, interface_id, neighbor_id) = upstream;
```

```
12            NEIGHBOR_RPC(node_id, interface_id, neighbor_id) = neighbor_rpc;
13         :: else ->
14            NEIGHBOR_STATE(node_id, interface_id, neighbor_id) = no_info;
15            NEIGHBOR_RPC(node_id, interface_id, neighbor_id) = 255;
16         fi;
17
18         recalculateTreeState(node_id, interface_id);
19      :: else ->
20         skip;
21      fi;
22      }
23   od;
24   }
```

Listing C.10: Proctype used to model the reception of messages at a given interface.

Regarding the proctype used to model the reception of messages, represented by Listing C.10, this is defined by an infinite do cycle in which it is always verified if the channel has messages (line 4). When a message is inside the channel, that message is obtained (line 6) and it is verified if it was originated from a known neighbor (line 8). If both conditions are true, the message is processed causing the neighbor's state to be set (to upstream or no_info depending on the received message's type) and the storage of its RPC. Then a recalculation of the tree is performed, in order to verify if the tree state of a router changes, depending on the existence of UPSTREAM routers connected to each interface and also on their RPC.

Sequence numbers are not used since the used channel models a FIFO buffer. Also, loss of messages are not modeled since the real specification models a retransmission mechanism until the neighbor acknowledges it or is declared to have failed. ACKs are also not modeled for the same reason (messages are not lost).

The function recalculateTreeState verifies if there are UPSTREAM neighbors connected to the interface that has invoked this function. Based on this information, the luni, luri and also on the RPC of the own router and of its UPSTERAM neighbors connected to the root interface (Listing C.8), the state of a tree at a router can change. This change at the tree state will be detected by all proctypes responsible for the transmission of messages of a router, being described below.

```
1  proctype InterfaceSend(byte node_id; byte interface_id) {
2  mtype last_interface_type = INTERFACE_TYPE(node_id, interface_id);
3  mtype last_tree_state = unknown_tree;
4  mtype last_msg_type = msg_i_am_no_longer_upstream
5  byte last_rpc = MY_RPC(node_id);
6
7  atomic {
8  do
```

```
9  // interface is root
10 :: INTERFACE_TYPE(node_id, interface_id) == root &&
11    (CURRENT_TREE_STATE(node_id)!=last_tree_state || last_interface_type==non_root) ->
12       if
13       :: last_interface_type == root &&
14                last_tree_state != CURRENT_TREE_STATE(node_id) ->
15          last_tree_state = CURRENT_TREE_STATE(node_id);
16       :: last_interface_type == non_root && last_tree_state == active_tree ->
17          last_interface_type = root;
18          last_tree_state = CURRENT_TREE_STATE(node_id);
19          sendMsg(node_id, msg_i_am_no_longer_upstream, interface_id, 0);
20       :: else ->
21          last_interface_type = root;
22       fi;
23 // interface is non-root
24 :: INTERFACE_TYPE(node_id, interface_id) == non_root  &&
25        (CURRENT_TREE_STATE(node_id) != last_tree_state || last_interface_type == root ||
26        last_rpc != MY_RPC(node_id)) ->
27    last_rpc = MY_RPC(node_id);
28    if
29    :: last_interface_type == non_root &&
30             CURRENT_TREE_STATE(node_id) == active_tree ->
31       last_tree_state = active_tree;
32       sendMsg(node_id, msg_i_am_upstream, interface_id, MY_RPC(node_id));
33    :: last_interface_type == non_root && last_tree_state == active_tree &&
34             CURRENT_TREE_STATE(node_id) != active_tree ->
35       last_tree_state = CURRENT_TREE_STATE(node_id);
36       sendMsg(node_id, msg_i_am_no_longer_upstream, interface_id, 0);
37    :: last_interface_type == root && CURRENT_TREE_STATE(node_id) == active_tree ->
38       last_interface_type = non_root;
39       sendMsg(node_id, msg_i_am_upstream, interface_id, MY_RPC(node_id));
40    :: else ->
41       last_tree_state = CURRENT_TREE_STATE(node_id);
42       last_interface_type = non_root;
43    fi;
44 od;
45 }
```

```
46 }
```

Listing C.11: Proctype used to model the tranmission of messages at a given interface.

Regarding the proctype used to model the transmission of messages, represented by Listing C.11, this is composed by an infinite do cycle in which it is verified if there are changes to the role of an interface and/or to the state of a tree and/or to the RPC of the own router. In any of these cases, a router may be required to send a message in order to notify its neighbors about those changes. For this reason, it is required to store the last role of the interface (last_interface_type), the last tree state (last_tree_state) and also the last known RPC (last_rpc).

When a change is detected, each condition of the if statement is verified in order to act accordingly to the protocol specification. Regarding the transmission of a message, this is accomplished by the function sendMsg() that verifies which interfaces are considered to be neighbors and sends to them the message.

Lines 10-22 model the behavior of root interfaces, while lines 24-43 model the behavior of non-root interfaces. In the case of a root interface, this must only transmit an IamNoLongerUpstream message if it was previously non-root and the tree was in ACTIVE state (lines 16-19). In the case of non-root interfaces, these must transmit IamUpstream messages if the tree transitions to an ACTIVE state (lines 29-32) or if it was root type and the tree remained ACTIVE (lines 37-39). Non-root interfaces must also transmit IamNoLongerUpstream messages in case the tree was previously non-root and ACTIVE but transitions to a non ACTIVE state (lines 33-36).

We pretend to test if all routers get a consistent state regarding the tree even when there are concurrent changes to the interfaces' roles (root $<->$ non-root) and router failures.

The failure of routers is modeled by the execution of function nodeFailure(node_id). This function will determine which interfaces belong to the failed router (node_id) and then will iterate all routers and stop considering those interfaces to be neighbors. This requires to recalculate the tree state in order to verify if the state of a tree changes. This change would then be detected by the proctype InterfaceSend of all interfaces that were neighbors with the failed router, which would act accordingly.

The change of interfaces' roles is modeled by the execution of function unicastChange(node_id, interface_id1, interface_id2). This function receives as arguments the node_id of the router that will suffer this change and two identifiers of interfaces. One identifier is the root interface and the other is a non-root. This function changes the role of both interfaces and recalculates the tree state. Both changes would be detected by the proctype InterfaceSend of all interfaces of the own router, which would act accordingly.

Both conditions are executed in the init method, depending on what we pretend to test. We have multiple Promela files with the same code but different test configurations (changes at the init function). These files are stored in a branch named "promela" in our GitHub repository [34]. The Promela files used to verify the formation and maintenance of a tree for each topology are the following:

- Topology 1:

  This topology was chosen because it has a link that offers redundant paths (link that connects

Routers 1, 2 and 3) and also has a link without redundant paths (link that connects Routers 3 and 4). By modeling the failure of different routers, the tree must be reconfigured according to the existence of redundancy. In these tests we have verified, with LTL, if the tree is reconfigured correctly, i.e. if all routers reach a consistent tree state.

In this topology the following interfaces were considered to be root type: 0, 3, 4, 7 and 9. The remaining interfaces were considered to be non-root type.

In these tests we considered that the RPC was set according to the hop distance of a router to the "source" of multicast traffic. We needed to set the RPC consistently since the state of a tree depends on the RPC of the own router and also on the RPC of its UPSTREAM neighbors connected to a root interface.

- **Test 1** - Tree formation without router failures

  In this test we started by having all routers initially considering the tree to be in UNKNOWN state. Then the originator router (Router 0) by manually considering the tree to be ACTIVE, it would trigger the creation of the broadcast tree via IamUpstream messages. Eventually, each router would be connected to UPSTREAM neighbors via their root interfaces with an RPC worser than one of their UPSTREAM neighbors. Each router would eventually consider the tree to be ACTIVE.

  The property that we wanted to verify was that eventually all routers would consider the tree to be indefinitely in ACTIVE state. This can be expressed in LTL by:

  $<> ([](CURRENT\_TREE\_STATE(0) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(1) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(2) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(3) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(4) == active\_tree))$

  With CURRENT_TREE_STATE(i) being a macro that obtains the state of the tree at Router i.

  This is modeled in file "new_specification_all_active.pml", which can be found in our repository [34] at a branch named "promela".

- **Test 2** - Tree formation concurrent to failure of Router 1

  Similar to Test 1 but Router 1 fails concurrently to the tree formation.

  Since there is a redundant path, all routers should eventually consider the tree to be in ACTIVE state, except for the failing router. This can be expressed in LTL the following way:

  $<> ([](CURRENT\_TREE\_STATE(0) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(2) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(3) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(4) == active\_tree))$

  This is modeled in file "new_specification_node_1_fail.pml", which can be found in our repository [34] at a branch named "promela".

- **Test 3** - Tree formation concurrent to failure of Router 2

  Same as Test 2 but Router 2 fails instead of Router 1.

Since there is still a redundant path, through Router 1, all routers should eventually consider the tree to be in ACTIVE state, expect for the failing router. This can be expressed in LTL the following way:

$<> ([](CURRENT\_TREE\_STATE(0) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(1) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(3) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(4) == active\_tree))$

This is modeled in file “new_specification_node_2_fail.pml”, which can be found in our repository [34] at a branch named “promela”.

– **Test 4** - Tree formation concurrent to failure of Router 3

Same as Test 2 but Router 3 fails instead of Router 1.

Since there is no redundant path that can connect Router 4 to the originator router (Router 0), all routers should eventually consider the tree to be indefinitely in ACTIVE state except for the failing router and Router 4. This last router must eventually consider the tree to be indefinitely in UNKNOWN state. This can be expressed in LTL the following way:

$<> ([](CURRENT\_TREE\_STATE(0) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(1) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(2) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(4) == unknown\_tree))$

This is modeled in file “new_specification_node_3_fail.pml”, which can be found in our repository [34] at a branch named “promela”.

– **Test 5** - Tree formation concurrent to failure of Router 0

Same as Test 2 but Router 0 fails instead of Router 1.

By having the only originator failing, this must trigger the removal of the tree (if it was previously formed).

Eventually all routers must consider the tree to be indefinitely in UNKNOWN state. This can be expressed in LTL the following way:

$<> ([](CURRENT\_TREE\_STATE(1) == unknown\_tree \,\&\&\, CURRENT\_TREE\_STATE(2) == unknown\_tree \,\&\&\, CURRENT\_TREE\_STATE(3) == unknown\_tree \,\&\&\, CURRENT\_TREE\_STATE(4) == unknown\_tree)) \,\&\&$

This is modeled in file “new_specification_originator_fail.pml”, which can be found in our repository [34] at a branch named “promela”.

- Topology 2:

Topology 2 just introduced a link, between Router 1 and Router 4, to Topology 1. In this topology we wanted to test if all routers reach a consistent tree state by having the unicast routing protocol also reacting to network changes. By having initially the same selected root interfaces as the tests to Topology 1, the failure of Router 3 would cause a change of the root interface of Router 4. In this test, we wanted to verify if Router 4, by changing its root interface to interface 10, would still consider the tree to be ACTIVE.

- **Test 6** - Tree formation concurrent to failure of Router 3 and concurrent to change of interfaces' roles at Router 4

  This test is similar to Test 4 but it also models the unicast routing protocol that would eventually detect the failure of Router 3 and would change the root interface at Router 4.

  Eventually Router 4 would have interface 10 as the new root interface, causing this router to still be attached to the tree. For this reason we would like to verify if eventually all routers consider the tree to be indefinitely in ACTIVE state, except for the failing router. This is expressed in LTL the following way:

  $<> ([](CURRENT\_TREE\_STATE(0) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(1) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(2) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(4) == active\_tree)) \,\&\&$

  This is modeled in file "new_specification_node_3_fail_with_redundant_path.pml", which can be found in our repository [34] at a branch named "promela".

- Topology 3:

  Topology 3 was used in order to verify the formation and removal of a tree, in all routers, in the presence of network loops. The root interfaces were manually selected in order to form a loop. So, interfaces 0, 2, 4, 6 and 8 were selected as root interfaces. This caused Routers 2, 3 and 4 to form a loop since non-root interface of Router 4 was connected to root interface of Router 2, non-root interface of Router 2 was connected to root interface of Router 3 and non-root interface of Router 3 was connected to root interface of Router 4.

  The RPCs were selected carefully in order to respect the selected root interfaces, i.e. the RPC of a router must be greater than the RPC of its next-hop (at the unicast routing protocol). This would also be used by the multicast routing protocol in order to verify the presence of network loops to maintain trees correctly.

  Since we have considered interfaces 0, 2, 4, 6 and 8 as root types, the RPC at Router 0 must be lower than the RPC at Router 1. The RPC at Router 1 must be lower than the RPC at Router 2. The RPC at Router 3 must be lower than the RPC at Router 4. For this reason we have selected 0 as the RPC at Router 0, 20 as the RPC at Router 1, 30 as the RPC at Router 2, 40 as the RPC at Router 3 and 50 as the RPC at Router 4.

  - **Test 7** - Tree formation in the presence of network loop

    The originator router, Router 0, would trigger the creation of the tree. Each router would eventually consider the tree to transition to ACTIVE state, causing the transmission of an IamUpstream message downwards the tree.

    This successive propagation of IamUpstream messages would cause any router to be connected to UPSTREAM neighbors via their root interfaces, with the RPC of a router greater than the RPC of its UPSTREAM neighbor. This means that eventually all routers would consider the tree to be indefinitely in ACTIVE state. This can be expressed in LTL the following

way:

$<> ([](CURRENT\_TREE\_STATE(0) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(1) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(2) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(3) == active\_tree \,\&\&\, CURRENT\_TREE\_STATE(4) == active\_tree))$

This is modeled in file “new_specification_tree_creation_with_loop.pml”, which can be found in our repository [34] at a branch named “promela”.

– **Test 8** - Tree removal in the presence of network loop

Similar to Test 7, in the sense that the tree would be created, but eventually the removal of the tree would be triggered by the originator router.

This means that in the end, eventually all routers would consider the tree to be indefinitely in UNKNOWN state even in the presence of a network loop. This can be expressed in LTL by:

$<> ([](CURRENT\_TREE\_STATE(0) == unknown\_tree \,\&\&\, CURRENT\_TREE\_STATE(1) == unknown\_tree \,\&\&\, CURRENT\_TREE\_STATE(2) == unknown\_tree \,\&\&\, CURRENT\_TREE\_STATE(3) = unknown\_tree \,\&\&\, CURRENT\_TREE\_STATE(4) == unknown\_tree))$

This is modeled in file “new_specification_tree_removal_with_loop.pml”, which can be found in our repository [34] at a branch named “promela”.

Appendix D

Implementation

This chapter 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).

D.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.

D.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 statical solution is not the one we pretend and it is already implemented by SMCRoute [38].

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 mrouted [39] (implementation of DVMRP) and pimd [40] (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 [41] 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 multicast 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 multicast 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 D.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 D.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 D.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 mfcctl {
2          struct in_addr mfcc_origin;              /* Origin of mcast      */
3          struct in_addr mfcc_mcastgrp;            /* Group in question    */
4          vifi_t   mfcc_parent;                    /* Where it arrived     */
5          unsigned char mfcc_ttls[MAXVIFS];        /* Where it is going    */
6          unsigned int mfcc_pkt_cnt;               /* pkt count for src-grp */
7          unsigned int mfcc_byte_cnt;
8          unsigned int mfcc_wrong_if;
9          int          mfcc_expire;
10 };
```

Listing D.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 D.3: Python example displaying creation of mroute socket.

Listing D.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 interfaces i0 and i2 of router R1 of Figure 2.1 have 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) that corresponds in the figure to (S2,G1), 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).

D.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 [42].

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 D.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 D.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).

D.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 D.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.

D.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.

D.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 D.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 [9] since it is simpler compared to IGMPv3 [16] and because the additional features of IGMPv3 are not required for this work.

D.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 [9] and are shown in Figures D.1, D.2 and D.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

```
                                        ------------------------------
                                 ______|________  gen. query timer      |
 ---------                      |               |        expired       |
| Initial |---------------->|               | (send general query,  |
 ---------  (send gen. q.,    |               |  set gen. q. timer)   |
       set initial gen. q.    |               |<---------------------
             timer)           |    Querier    |
                              |               |
                        -----|               |<---
                       |     |               |    |
                       |     |_______________|    |
query received from a  |                          | other querier
router with a lower    |                          | present timer
IP address             |                          | expired
(set other querier     |      _______________     | (send general
 present timer)        |     |               |    |  query,set gen.
                       |     |               |    |  q. timer)
                       |     |               |    |
                        ---->|      Non      |----
                              |    Querier    |
                              |               |
                              |               |
                        ---->|               |----
                       |     |_______________|    |
                       | query received from a    |
                       | router with a lower IP   |
                       | address                  |
                       | (set other querier       |
                       |  present timer)          |
                        --------------------------
```

Figure D.1: IGMPv2 Querier state machine.

```
                                   _______________
 ---------------------------|               |<-----------------------
|                           |               |timer expired           |
|                timer expired|               |(notify routing -,      |
|           (notify routing -)|   No Members  |clear rxmt tmr)         |
|                     ------->|    Present    |<-------                |
|                   |         |               |        |               |
|v1 report rec'd    |         |               |        |  ------------ |
|(notify routing +, |         |_______________|        | | rexmt timer| |
| start timer,      |                   |            | |  expired   | |
| start v1 host     |  v2 report received|            | | (send g-s  | |
|  timer)           |  (notify routing +,|            | |  query,    | |
|                   |        start timer)|            | |  st rxmt   | |
|         _________|_____              |       _____|_|_____  tmr)| |
|        |                |<------------       |             |     | |
|        |                |                    |             |<----- |
|        |                | v2 report received |             |       |
|        |                | (start timer)      |             |       |
|        | Members Present |<------------------|   Checking  |       |
|  ----->|                | leave received     |  Membership |       |
| |      |                | (start timer*,     |             |       |
| |      |                |  start rexmt timer,|             |       |
| |      |                |  send g-s query)   |             |       |
| |  --->|                |------------------->|             |       |
| | |    |_______________|                    |_____________|       |
| | |v2 report rec'd |  |                        |                 |
| | |(start timer)   |  |v1 report rec'd         |v1 report rec'd  |
| |  ----------------   |(start timer,           |(start timer,    |
| |v1 host              | start v1 host timer)   | start v1 host   |
| |tmr     _____________V__                       | timer)          |
| |exp'd |                |<---------------------                    |
|  ------|                |                                          |
|        |    Version 1   |timer expired                             |
|        | Members Present |(notify routing -)                       |
 ------->|                |-----------------------------------------
        |                |<--------------------
 ------->|_______________| v1 report rec'd     |
| v2 report rec'd |    |   (start timer,        |
| (start timer)   |    |    start v1 host timer) |
 -----------------      -------------------------
```

Figure D.2: IGMPv2 Membership state machine of Querier routers.

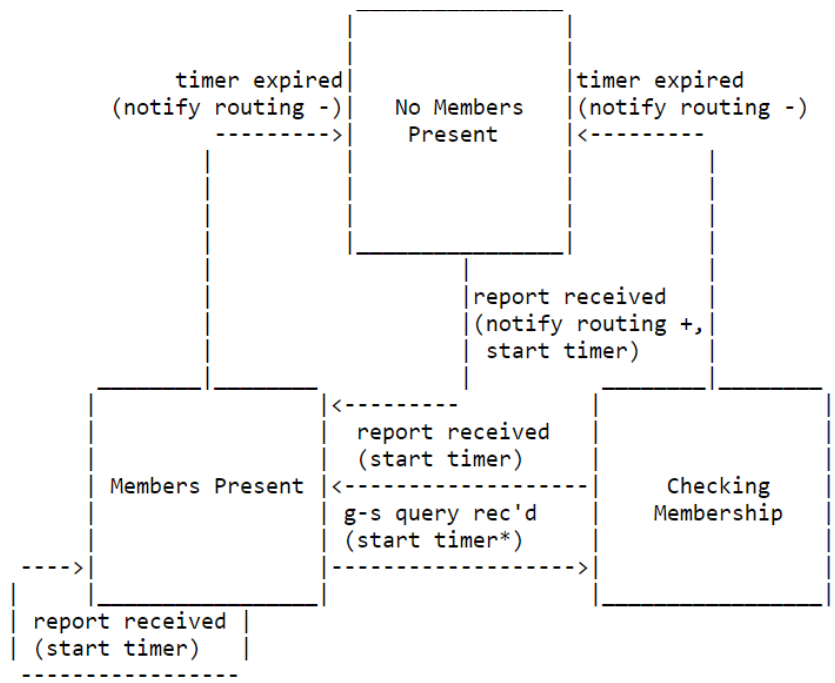

Figure D.3: IGMPv2 Membership state machine of Non-Querier routers.

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 Query 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;
- **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;
- **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).

D.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 [33, 34].

Regarding our implementation, the following diagram (Figure D.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 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 D.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 D.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 D.2;
- **NonQuerier** - implements the Non-Querier state of the Querier state machine. The protocol specification is also represented in Figure D.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

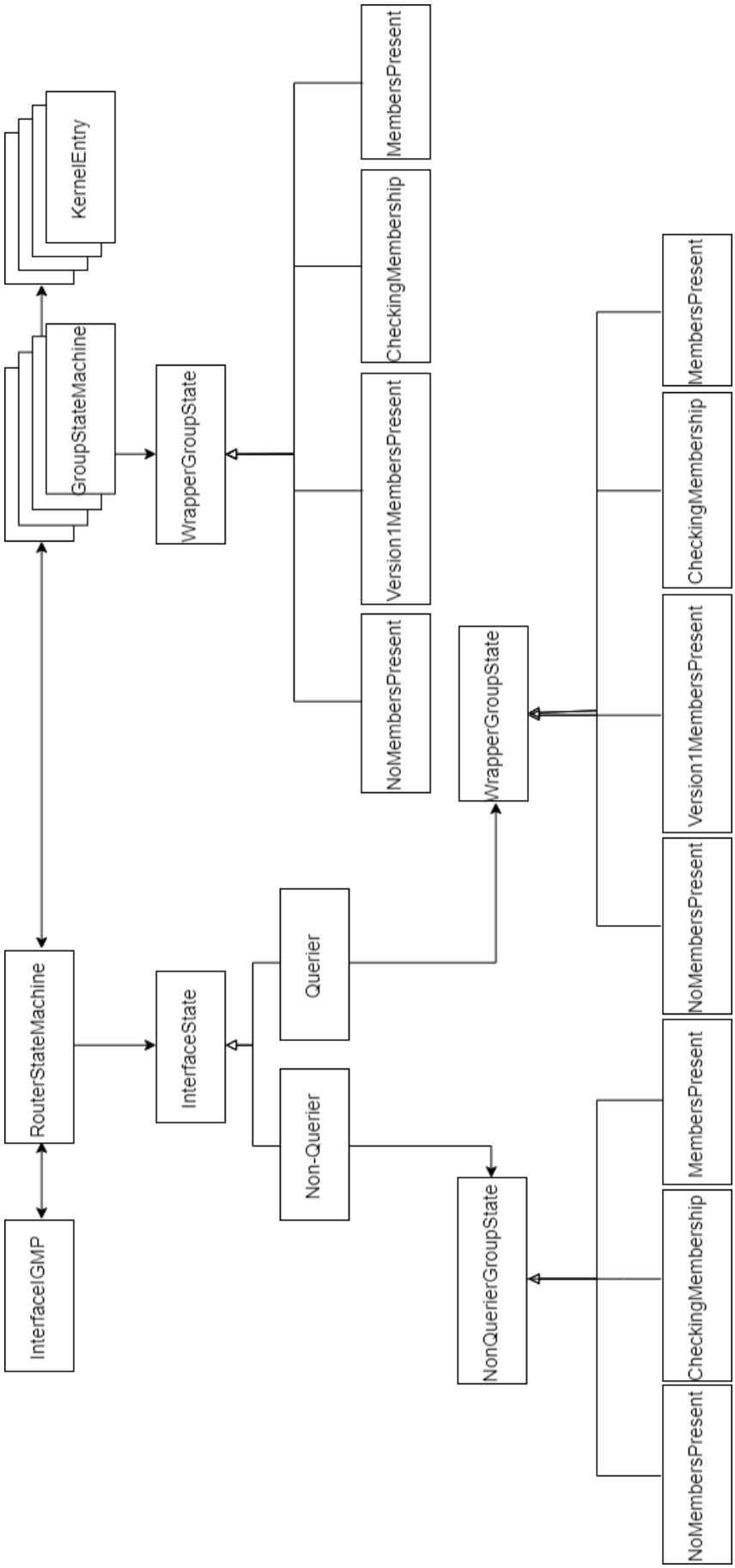

Figure D.4: IGMPv2 state machine implementation diagram.

a group can be at (NoMembersPresent, CheckingMembership and MembersPresent), which are represented in Figure D.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 D.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 D.2 and D.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 D.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 object[1], 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 D.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;
- 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) [43] 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:

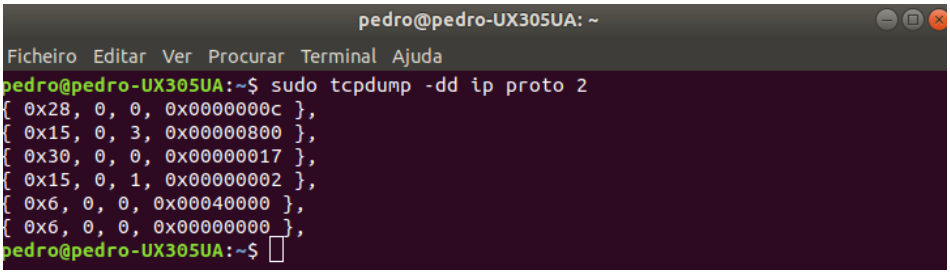

Figure D.5: Get BPF bytecode.

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:

```
FILTER_IGMP = [
        struct.pack('HBBI', 0x28, 0, 0, 0x0000000c),
        struct.pack('HBBI', 0x15, 0, 3, 0x00000800),
        struct.pack('HBBI', 0x30, 0, 0, 0x00000017),
```

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

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

Lines 1-8 define the filter obtained by Figure D.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.

D.2.2 PIM-DM

We searched for PIM-DM implementations and we have only found one [32], 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 [2]).

D.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 [2]. Figures D.6, D.7, D.8 and D.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:

```
+-------------+                        +-------------+
|             | Rcv Pref Assert or SR  |             |
|   Winner    |----------------------->|    Loser    |
|             |                        |             |
+-------------+                        +-------------+
     ^   |                                  ^   |
     |   |                Rcv Pref Assert or|   |
     |   |AT Expires OR        State Refresh|   |
     |   |CouldAssert->FALSE                |   |
     |   |                                  |   |
     |   |         +-------------+          |   |
     |   +-------->|             |----------+   |
     |             |   No Info   |              |
     +-------------|             |<-------------+
  Rcv Data from dnstrm +-------------+ Rcv Inf Assert from Win OR
OR Rcv Inferior Assert                 Rcv Inf SR from Winner OR
    OR Rcv Inferior SR                 AT Expires OR
                                       CouldAssert Changes OR
                                       Winner's NLT Expires
```

Figure D.6: Simplistic PIM-DM Assert state machine.

```
+-------------+                        +-------------+
|             |     olist == NULL      |             |
|   Forward   |----------------------->|   Pruned    |
|             |                        |             |
+-------------+                        +-------------+
     ^   |                                  ^   |
     |   |                                  |   |
     |   |RPF`(S) Changes      olist == NULL|   |
     |   |                                  |   |
     |   |         +-------------+          |   |
     |   +-------->|             |----------+   |
     |             | AckPending  |              |
     +-------------|             |<-------------+
       Rcv GraftAck OR +-------------+ olist != NULL
     Rcv State Refresh
        With (P==0) OR
    S Directly Connect
```

Figure D.7: Simplistic PIM-DM Upstream state machine.

```
+-------------+                        +-------------+
|             |      PPT Expires       |             |
|PrunePending |----------------------->|   Pruned    |
|             |                        |             |
+-------------+                        +-------------+
     |   ^                                      |
     |   |                                      |
     |   |Rcv Prune                             |
     |   |                                      |
     |   |         +-------------+              |
     |   +---------|             |              |
     |             |   NoInfo    |<-------------+
     +------------>|             | Rcv Join/Graft OR
   Rcv Join/Graft OR +-------------+ PT Expires OR
RPF_Interface(S)->I                RPF_Interface(S)->I
```

Figure D.8: Simplistic PIM-DM Downstream state machine.

```
+-------------+  Rcv Directly From S   +-------------+
|             |----------------------->|             |
|NotOriginator|                        | Originator  |
|             |<-----------------------|             |
+-------------+     SAT Expires OR     +-------------+
                 S NOT Direct Connect
```

Figure D.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 [2] 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:

$$olist(S,G) \;=\; immediate\_olist(S,G) \quad (-) \quad RPF\_interface(S)$$

with $$immediate\_olist(S,G) \;= pim\_nbrs \quad (-) \quad prunes(S,G) \quad (+) \quad (pim\_include(*,G) \quad (-) \quad pim\_exclude(S,G)) \quad (+) \quad pim\_include(S,G) \quad (-) \quad lost\_assert(S,G) \quad (-) \quad boundary(G);$$

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.

D.2.2.2 Protocol Implementation

The implementation of PIM-DM is stored in our GitHub repository and can be accessed here [33].

The following diagram (Figure D.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

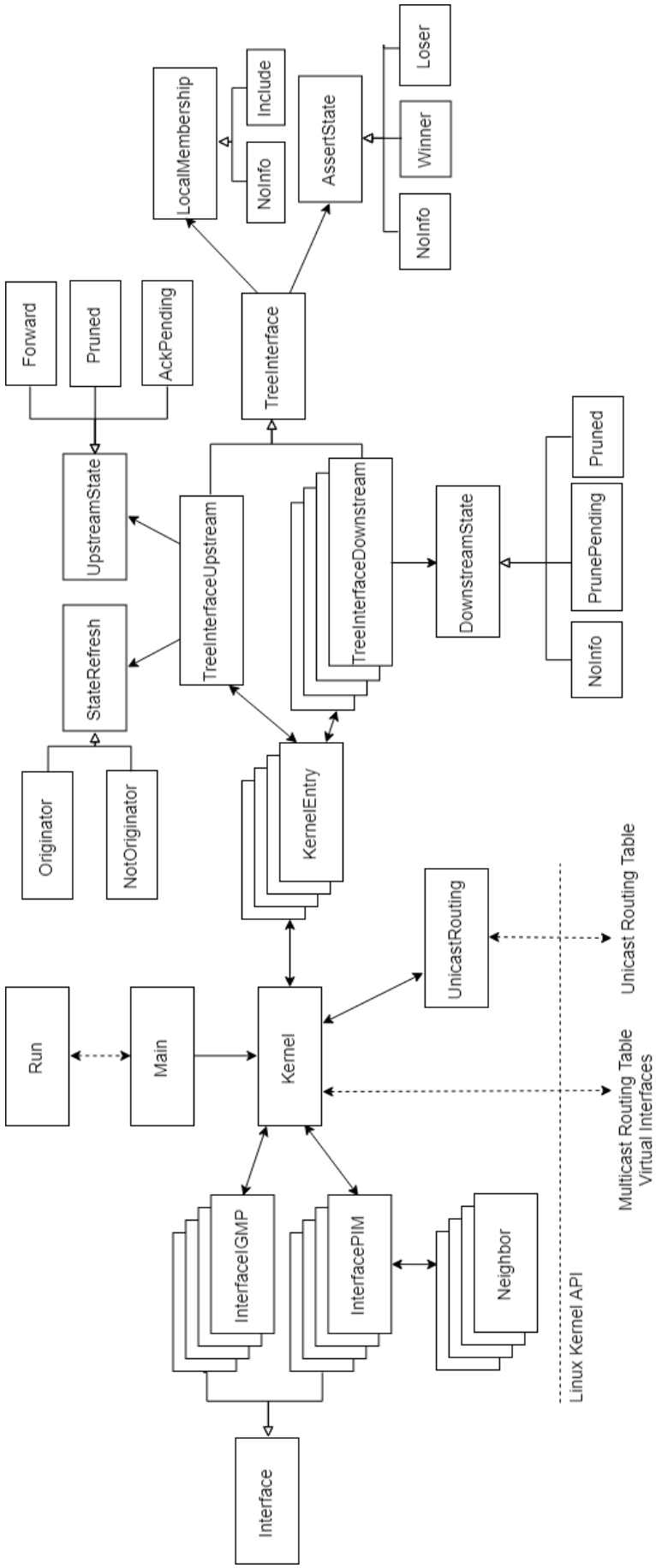

Figure D.10: PIM implementation diagram.

is removed);

- **InterfaceIGMP** - corresponds to a simplistic view of the implementation of IGMPv2, that has been described in section D.2.1.2. Before creating an InterfaceIGMP, it is required to create a virtual interface associated with the physical interface (in Kernel module);
- **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 D.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 [2]. 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 [2], 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 tests, which will be detailed in section E. 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 D.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 here[2].

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.

D.2.3 HPIM-DM Protocol

D.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 Appendix B. 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 UPDATED. 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 Slave of that process. In the UPDATED 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

[2] A 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/

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, INACTIVE and UNKNOWN. 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 INACTIVE 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 INACTIVE 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 UNKNOWN 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 INACTIVE state when the source does npt 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 UNKNOWN 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 INACTIVE 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 UNKNOWN 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.

D.2.3.2 Protocol Implementation

The implementation of HPIM-DM is stored in our GitHub repository and can be accessed here [34].

The following diagram (Figure D.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.

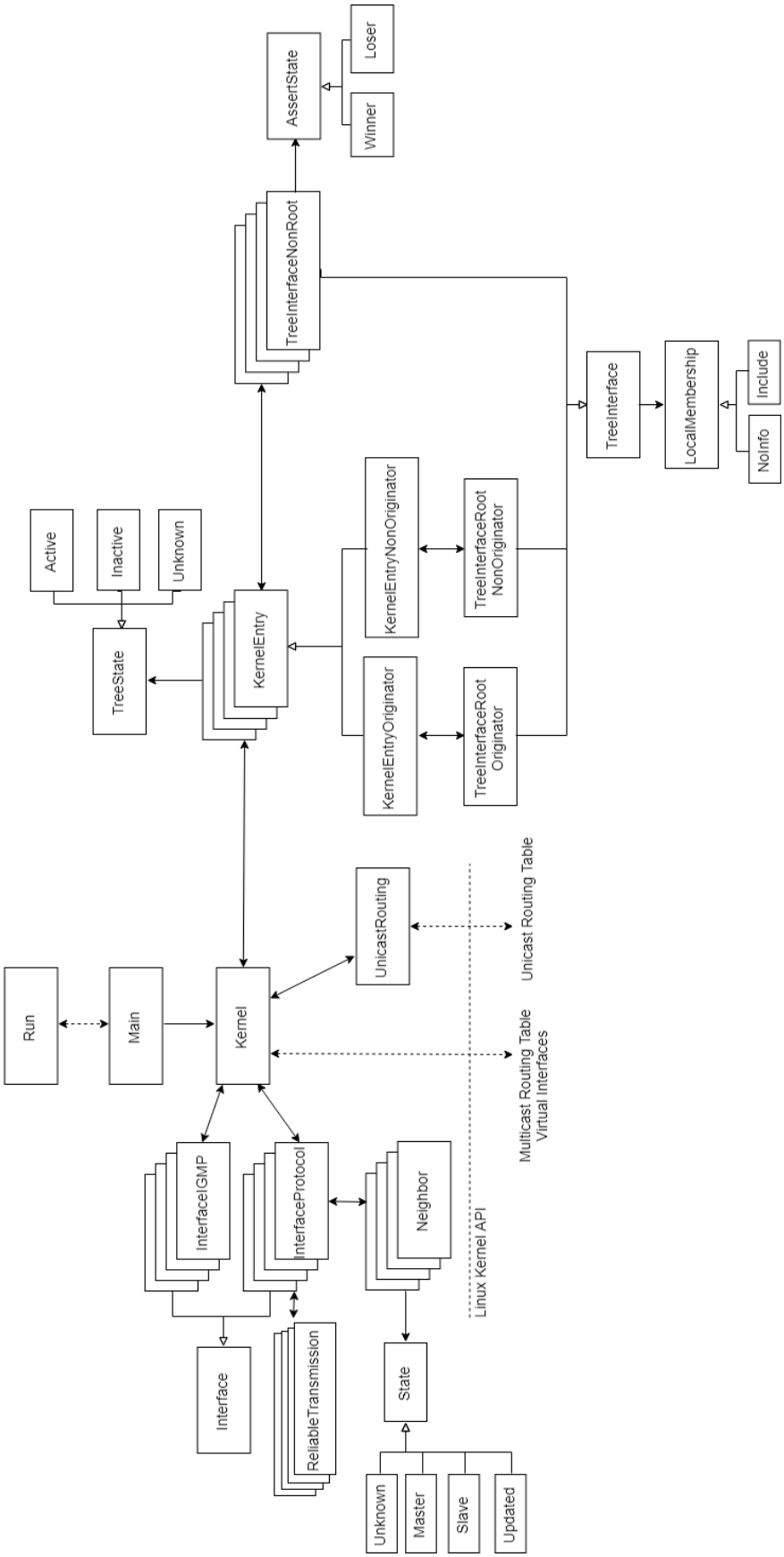

Figure D.11: HPIM-DM implementation diagram.

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 UPDATED).

  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 Updated** - 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, Inactive and Unknown** - 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.

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 D.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 D.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.

Appendix E

Implementation Tests

In order to check if the implementations were according to the specifications, some tests were done. All tests were performed using Netkit-NG [10], allowing to emulate a network environment. Additionally, we have also used GNS3 [35] to verify the interoperability between our implementations executing in Linux devices and the implementations of Cisco routers.

A test is defined by a certain topology in order to verify if a certain module of the protocol is working as expected. All packets exchanged between routers were captured using *tcpdump* [36] and further analyzed with *Wireshark* [37].

All tests were performed by knowing a priori the state in which some routers should be at. A centralized node knows this information and all routers send their state transitions to it. A test is considered to be successful when all predicted state transitions were made. This was accomplished by sending logs of our implementation to a server node, according to state transitions, exchange of control messages and expiration of timers. The remote logging was accomplished by the Python's *logging* module, that has an handler that can transmit all recorded logs to a given node. In order to do this, all routers would execute the Test command that was referred in sections D.2.2.2 and D.2.3.2. Regarding the exchanged control messages, these were manually verified with *Wireshark* [37] after executing the tests.

We will describe all performed tests and corresponding results to our IGMPv2, PIM-DM and HPIM-DM implementations.

E.1 IGMP Implementation Test Results

In order to test IGMP, we only need one subnet with one or more routers and one or more hosts connected to it. The used topology is shown in Figure E.1. Since routers did not require to communicate beyond this local subnetwork, it was not configured any unicast routing protocol on routers.

Hosts manifested their interest, reporting and leaving groups, by opening and closing sockets, like it was explained in sections D.1.2 and D.1.3. The host part of IGMPv2 was totally controlled by the Linux implementation of the network devices.

One Linux machine had the job of a switch, in order to connect more than one router to a single

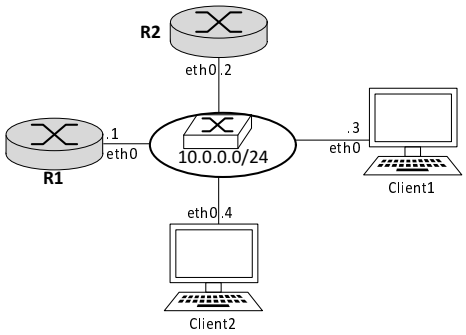

Figure E.1: Topology used to test IGMP, PIM-DM and HPIM-DM implementations.

subnet. This was possible by executing special commands, such as “brctl” on that Linux machine. This command simply allows to aggregate multiple physical interfaces into one logical interface, belonging all to the same subnetwork.

The server process that received all logs was running on the switch, which printed to the console all received logs regarding IGMPv2 (of routers) and stored them in a file.

The switch also captured all exchanged packets, using *tcpdump*, by executing the command “tcpdump -i br0 -Q in -w Testx.pcap”. This command recorded all packets that entered interface br0, which was the logical bridge interface that aggregated all physical interfaces of the switch, and wrote those packets to a capture file named Testx.pcap, being x the number of the test.

The implementation of the IGMPv2 protocol was integrated in both PIM-DM and HPIM-DM implementations. Since the IGMPv2 implementation was not intended to be used alone, all tests were performed in HPIM-DM implementation. Also, all implemented timers were set to their default values, according to the IGMPv2 specification [9].

We have subdivided the tests of IGMP in two parts. The first part tested if all routers elected the Querier correctly and if that router transmitted the expected control messages. The second part tested if all routers were determining correctly the interest of hosts in a particular group.

The Netkit network configurations, tests and corresponding results are stored in a branch named “Test_IGMP” at our GitHub repository [34].

E.1.1 Querier election and maintenance

First we will describe the test results regarding the Querier state machine:

- **Test 1 -** Election of the Querier, after two routers start the IGMP process

  In this test we started the IGMP protocol first on router R1 and then on router R2. We have verified that both routers initially transmitted IGMP General Membership Queries, destined to the All-Systems multicast IP address (224.0.0.1). Both routers started initially in Querier state and since router R1 transmitted its control message when R2 had not started its protocol, this meant that R2 did not interpret that message. For this reason, both routers remained in the Querier state, until R1 retransmitted an IGMP General Membership Query.

When R1 retransmitted an IGMP General Membership Query, controlled by the “general query timer”, R2 by hearing this message, transitioned to Non-Querier state. From this point on, R2 did not transmit anymore IGMP General Membership Query messages.

It was verified in Wireshark that all messages were correctly built, by being properly dissected. Also, it was possible to check the periodicity of IGMP Membership Query message transmissions, which was around 125 seconds. This periodicity was according to the specification.

With this test, it was possible to verify that the Querier state machine made the correct transitions and correctly elected the Querier.

- **Test 2 -** Reelection of the Querier after the previous one fails

  We have repeated Test 1, but now we have started first the IGMP process in R2 and only then in R1. This allowed a quicker transition to the correct state (Non-Querier) at R2.

  We have waited for the first IGMP General Membership Query retransmission of R1 and then stopped the protocol process at R1. This message was transmitted at time 131 seconds, according to Wireshark.

  After a given amount of time, in the logs it was possible to verify that the “other querier present timer” of R2 had expired, causing a transition to the Querier state in this same router. Then an IGMP General Membership Query was transmitted by R2 at time 386 seconds. The delta between the last transmission of R1 and the first transmission of R2 was around 255 seconds, which was according to the specification ((the Robustness Variable) x (the Query Interval) + (one half of one Query Response Interval) = 2 x 125 + 0,5 x 10 = 255).

  With this test it was possible to verify that the implementation of IGMP correctly reacted to failures of the Querier, due to the absence of its IGMP Membership Query messages. Also, the timer used to control this state was set correctly.

- **Test 3 -** Failure of a Non-Querier must not cause a reelection

  We have started the protocol first in R2 and then in R1, like described in Test 2. Also, we have waited for the first retransmission of an IGMP Membership Query by R1 and then stopped the protocol at R2.

  In this test it was possible to verify that R1 remained in the Querier state, not being affected by the failure of R2. This test and Test 1 are indistinguishable in terms of router R1, since it remained in the Querier state.

  The periodicity of exchanged control messages was according to the previous tests and to the specification.

E.1.2 Membership maintenance

Now, we will describe the test results regarding membership detection by routers:

- **Test 4 -** Detection of membership when hosts are not interested in receiving multicast traffic (all groups remain in “No Members Present” state)

  This is similar to all tests that were performed until now (from Test 1 to 3), since on those tests hosts did not manifest their interest in receiving data packets.

  On those tests we did not observe any IGMP Report or IGMP Leave message. Also, by manually verifying the list of groups that the protocol process was monitoring, we found that list to be empty. This was expected, since the process only starts monitoring a given group when it receives an IGMP message regarding it. By having the list empty, there are no groups being monitored, which means that all groups were considered to be in a “No Members Present” state.

- **Test 5 -** Detection of membership when a previously not interested group starts having interested hosts (hosts were not interested in receiving multicast traffic but one of them declares interest)

  Two hosts were not interested in any group but one of them declares interest in group “224.12.12.12”. It is expected that all groups remain in “No Members Present” state, except for the reported group that will transition to “Members Present”, in both routers.

  In this test the two routers, R1 and R2, were running the protocol process and Client1 declared interest in group 224.12.12.12, by transmitting an IGMP Membership Report message.

  We have observed an unexpected behavior regarding the transmission of IGMP Report messages from the host. We have observed three consecutive transmissions of this type of message, when the specification only specifies two. The additional transmission may be for reliability purposes. Since we are only interested in testing the IGMP implementation of routers, we were not concerned with this unexpected behavior, since it does not cause any malfunction to the protocol.

  By hearing the first IGMP Membership Report, both routers transitioned group “224.12.12.12” to “Members Present” state. From time to time, the Querier transmitted IGMP Membership Query messages, which were answered with an IGMP Membership Report regarding group “224.12.12.12” by the same host.

  When the Querier transmits an IGMP Membership Query message, it specifies a given amount of time that it will wait for the reception of IGMP Report messages. This amount of time is called Max Response Time and is included in a field of the transmitted IGMP Query message. The default amount of time, according to the specification, is 10 seconds. It was observed that the host responded within this time interval. The first Report message was transmitted 2 seconds after the Query message has been received, while the second Report was transmitted 4 seconds after the second Query message was received and a third Report was transmitted 8 seconds after the third Query message has been received. The amount of time a host waits before it transmits a Report is chosen randomly, in order to support the suppression mechanism, i.e. a host does not require to transmit a Report if other host has already reported the same group.

  Since both routers remained in “Members Present” state in group “224.12.12.12”, we can conclude that the reporting of membership was correctly detected. Also, the field Max Response Time and

the corresponding timer were set correctly.

- **Test 6 -** Detection of membership when a previously interested group starts having interest by another host (an host manifested interest regarding a group that already had interested hosts)

  Two hosts were not interested in any group but one of them declares interest in group “224.12.12.12”. Then the second host reports interest in the same group. It is expected that all groups remain in “No Members Present” state, except for that reported group, which will transition and remain in “Members Present” state, after the first host reports it.

  In this test the two routers were running the protocol process and both hosts reported interest in group “224.12.12.12”.

  Like Test 5, we have observed that both hosts transmitted initially three consecutive IGMP Report messages and then only retransmitted this message after hearing an IGMP Membership Query message.

  By looking to the packet capture, it looked unexpected at first sight because both hosts were “Reporting” the same group after hearing IGMP Membership Query messages, suggesting that the suppression mechanism was not working. This was due to a technology built-in in some switches called IGMP Snooping. Basically this allows the switch to interpret IGMP messages and define which interfaces of the switch have interest in receiving multicast traffic. This allows multicast traffic to not behave like broadcast traffic in a local network. For this reason, IGMP Membership Report messages transmitted from a switch port were not received by a host connected to other switch port. This requires to specify which interfaces of a switch have multicast routers connected to it, in order to receive all multicast traffic and all IGMP message types. Since we were only interested in testing the router’s IGMP state machines, this suppression mechanism is out of scope because it is only used at the host part.

  The difference between this test and Test 5 is that two hosts Report group “224.12.12.12” instead of just one. Like the other test, all hosts reported within the Max Response Time, which allowed all routers to remain in the correct state regarding this group (“Members Present”).

- **Test 7 -** Detection of membership when an host leaves a group in which there is still interest by other hosts

  Two hosts declared interest in group “224.12.12.12” and one of them leaves it, by transmitting an IGMPv2 Leave message. Since the other host remains interested in the same group, it is expected that this group remains in “Members Present”. Since there was a Leave message involved, first all routers will transition to “Checking Membership” state and then transition again to “Members Present”.

  In this test the two routers, R1 and R2, were running the protocol process and Client1 and Client2 “Reported” initially interest in the same group “224.12.12.12”. After a given amount of time, Client2 left group “224.12.12.12”, by transmitting an IGMP Leave message, regarding it.

In the logs we can observe that both routers received this message. Router R2 did not react to this message, because it was in Non-Querier state. Router R1 by receiving the same message, transitioned to state “Checking Membership” state and transmitted an IGMP Group-Specific Query. Only the reception of this last message by R2 caused this router to transition to “Checking Membership” state.

After a given amount of time, the host still interested in group “224.12.12.12”, responded to the IGMP Group Specific Query, by transmitting an IGMP Membership Report message. In the logs it was possible to verify that the reception of this message caused both routers to transition group “224.12.12.12” back to “Members Present” state.

The transmitted IGMP Group Specific Query had Max Response Time of 1 second, like specified in the specification (Last Members Query Interval = 1 second). The response from the interested host was within this time interval, being observed a difference of 0.2 seconds between the reception and the transmission of those messages.

All state transitions were expected and the routers did not transition to “No Members Present” during this process. The implementation reacted accordingly to the received messages and the timers were set with the expected values.

- **Test 8 -** Detection of membership when an host leaves a group and there are no other hosts interested in this same group

  Initially only one host is interested in group “224.12.12.12”, then stops being interested. Since there are no more hosts interested on this group, it is expected a transition to “No Members Present” state by both routers.

  We have started by recreating Test 7, i.e. two hosts initially interested and then one host leaves the group. Then we have observed what happened when the last interested host leaves that same group.

  When the last host leaves group “224.12.12.12”, by transmitting an IGMP Leave message, we have observed the same initial transition to “Checking Membership” state and the transmission of an IGMP Group Specific Query.

  Since there was no response from the hosts to the IGMP Group Specific Query, within the Max Response Time of 1 second, the Querier retransmitted this message (due to the expiration of the retransmit timer). By not hearing any response from the hosts to the second IGMP Group Specific Query, both routers transitioned group “224.12.12.12” to “No Members Present” state. This transition, in both routers, was caused by the expiration of the timer, which was possible to verify in the logs.

  As it was expected, both routers correctly determined when a given group had no members interested in receiving data packets.

- **Test 9 -** Detection of membership when an host that manifested interest fails, without leaving the group through an IGMP Leave message

One host declares interest in group “224.12.12.12” and then fails, without transmitting an IGMPv2 Leave message. Since the “Members Present” state is only maintained for a given amount of time, it is expected that eventually all routers transition to the correct state (“No Members Present”), after the timer associated with it expires.

In this test, only one host reported interest in group “224.12.12.12”, which was Client1. We started by initializing the protocol in both routers, R1 and R2, and waited for them to correctly transition to their Querier/Non-Querier states. Then Client1 reported interest in group “224.12.12.12”, causing a transition of this group to “Members Present”, in both routers.

Then, after three successive IGMP Membership Queries and corresponding IGMP Membership Reports, we simulated the failure of Client1 by shutting down its interface that connected this host to the local network.

It was possible to verify in the logs that after a given amount of time, both routers transitioned correctly to the “No Members Present”, due to the expiration of the timer regarding group “224.12.12.12”. This expiration happened around 4 and a half minutes after the last IGMP Membership Report was received, which is consistent to the expected time. This time is defined in the specification by Group Membership Interval = (Robustness Variable) x (Query Interval) + (Query Response Interval) = 2 x 125 + 10 = 260 seconds = 4.333 minutes.

With this test, it was possible to verify that the absence of IGMP Report messages was correctly detected. This caused both routers to no longer consider the group to have members. Also, the timer used to control this behavior was set according to the specification.

E.2 PIM-DM Implementation Test Results

Since PIM-DM requires information from the unicast routing table, in order to form and shape multicast distribution trees, routers need to run a given unicast routing protocol or configure statically all subnetworks. It was opted to use OSPF [28] as the underlying unicast routing protocol. For this reason all Linux routers executed *Quagga* [44], which offered an implementation of this protocol.

E.2.1 Neighborhood maintenance

First we will describe the test results regarding the establishment and maintenance of neighborhood relationships. These were performed using the topology of Figure E.1.

The corresponding Netkit network configurations, tests and results are stored in a branch named “Test_PIM_Hello” at our GitHub repository [33].

The timer associated with the transmission of Hello messages was set to its default value, i.e. transmission periodicity of 30 seconds. In case a new or rebooting neighbor is detected, a new Hello message is transmitted within 5 seconds (randomly). The packet captures must confirm these values.

Also, State Refresh was not enabled on these tests, since we were only testing if two routers correctly established a neighborhood relationship. By having this feature disabled, the State Refresh option was

not present in the transmitted Hello messages.

- **Test 1 -** Establishment of neighborhood relationship between two unknown neighbors

  In this test we have two routers, R1 and R2, directly connected to each other. We have first started the protocol process in R1 and then on R2.

  In the logs, we can confirm that both routers established a neighborhood relationship due to the reception of an Hello message from the corresponding neighbor.

  In the packet capture, we can verify that R1 transmitted Hello messages having 105 as the Hello Hold Time and Generation ID set to 1217533332. R2 also transmitted Hello messages with the same Hello Hold Time and Generation ID set to 3312157476. Following messages by both routers transmitted the same options and values, as it was supposed. In the logs, we can verify that both routers correctly interpreted these values.

  The transmitted Hello Hold Time was correct, in both routers, since this value is set to 3.5 times the periodicity of Hello messages (default of 30), being 3.5*30=105 seconds.

  Regarding the message periodicity, according to Wireshark, when both routers started detecting each other, they transmitted Hello messages within the expected range of 5 seconds (R2 transmitted at 15.35 seconds due to the boot up; R1 transmitted at 19.89 seconds due to the discovery of R2; R2 transmitted at 20.88 seconds due to the discovery of R1). Then after discovering each other, they transmitted Hello messages every 30 seconds. R1 transmitted at 19.89 seconds, 49.93 seconds, 79.99 seconds, ..., while R2 transmitted at 20.88 seconds, 50.92 seconds, 80.98 seconds, ...

  According to this test, both routers correctly interpreted the exchanged Hello messages and established a neighborhood relationship. Also, message transmissions were within the expected time.

- **Test 2 -** Re-establishment of a neighborhood relationship after a known neighbor reboots

  We started by recreating Test 1, i.e. start the protocol process in both routers, in order to establish a neighborhood relationship. Note that since we have restarted the protocol process in both routers, they transmitted Hello messages with different Generation IDs (R1 with Generation ID set to 3164982407 and R2 set to 3806743712) compared to the previous test. This was normal since these values change every time the protocol starts/restarts in a given interface.

  After R1 and R2 established a correct neighborhood relationship, we rebooted the protocol process in R2. This caused R2 to transmit Hello messages with a newer Generation ID, set to 1471879794.

  In the logs we can observe that the reception of an Hello message with a different Generation ID, caused R1 to correctly detect the reset of R2.

  The periodicity of Hello messages was within the expected values, like it was observed in the previous test. Also, it was possible to determine that a router correctly interpreted a neighbor reset.

- **Test 3 -** Neighborhood relationship break after known neighbor fails

  In this test, we started by recreating Test 1 in order to have both routers, R1 and R2, with a established neighborhood relationship. Like Test 2, since we have restarted the protocol process, both routers transmitted Hello messages with different Generation IDs, compared to the previous tests.

  After both routers established a neighborhood relationship, we waited around 60 seconds and stopped the protocol process in R2. This caused R2 to no longer transmit Hello messages.

  According to the logs, R1 received the last Hello message from R2 at time 18:02:21. At time 18:04:06, R1 considered R2 to have failed due to the absence of received Hello messages. This time interval is of about 1 minute and 45 seconds, which represents 105 seconds. It was possible to verify that the detection of a neighbor fail was detected correctly, since the last transmitted Hello message from R2 had the Hello Hold Time option set to this precise time.

  In this test we could verify that a neighbor failure was detected correctly. Also the timer used to control the liveness of a known neighbor was set correctly.

E.2.2 Formation and maintenance of the broadcast tree

Now we will describe the test results regarding the formation of the broadcast tree in order to verify if it was being constructed correctly, according to the unicast routing table. These were performed in the topology represented by Figure 6.1. In order to accomplish these tests, we first needed to verify if the unicast routing table had already stabilized.

Regarding the unicast routing protocol, each interface was set with a default cost of 10, meaning that the root and non-root interfaces were selected according to the "number of hops", in the sense that paths with fewer hops were preferred due to having a lower Root Path Cost. For this reason, the root interface, according to the subnet of Source, must be interface eth0 in all routers except for Router7 that must be eth2.

The corresponding Netkit network configurations, tests and results regarding the formation of the broadcast tree are stored in a branch named "Test_PIM_BroadcastTree" at our GitHub repository [33].

- **Test 4 -** Formation of the broadcast tree (flood of data packets and selection of root interface)

  In order to test this, the two receivers manifested interest regarding group "224.12.12.12". Host represented by Source transmitted multicast data packets destined to this group. In order verify if the broadcast tree was correctly formed, the two hosts must receive those multicast data packets and the multicast routing table, in all routers, must be correctly set (i.e. their root and non-root interfaces).

  When the unicast routing table at all routers had information regarding all subnets, Source started transmitting data packets to group "224.12.12.12".

  By looking to the logs, we have verified that all types of interfaces were chosen correctly (root and non-root). Also, multicast data packets were received in both clients. For this reason, it was

possible to verify that the broadcast tree regarding (10.0.0.1, 224.12.12.12) was correctly built.

Regarding the packet captures, we ran *tcpdump* in all links. It was possible to verify that UDP data packets originated from Source and destined to group 224.12.12.12 were initially present in all captures, confirming that the broadcast tree was formed due to the flood of multicast data packets.

Control messages exchanged by routers were not verified in this test, since those will be further analyzed on following tests and because the creation of the broadcast tree is only triggered by the reception of data packets.

- **Test 5 -** Broadcast tree reconfiguration in reaction to RPC changes detected by the unicast routing protocol.

  To perform this test, we selected interface eth2 of Router7 to suffer an RPC change. This must cause a change in the interfaces' roles of this router (root <-> non-root interface). Since all interfaces have a default cost of 10, we have changed the cost of interface eth2 of Router7 to 100, in order for eth0 to be selected as the new root interface.

  By analyzing the logs, we have verified that interface eth2 of R7 transitioned to non-root and interface eth0 transitioned to root, as expected.

  With this test, we have verified that the protocol correctly reacted to changes at the unicast routing table, reshaping the already formed (10.0.0.1,224.12.12.12) tree.

- **Test 6 -** Broadcast tree reconfiguration in reaction to router failures, detected by both the unicast routing protocol and the multicast routing protocol.

  After performing Test 5, without restarting the protocol in any router, the broadcast tree was set in all routers, with eth0 being set as the root interface on all routers. In this test, we have opted to fail router R5 in order to verify if the root interface of router R7 changes again to eth2.

  In order to "simulate" the failure of a network device, we have disabled (shutdown) interface eth0 of R5. Eventually, the unicast routing protocol reacted to this change, modifying the unicast routing table of some routers. This modification was visible in the logs, with the change of the root interface at R5 and R7.

  With this test, we have verified that all interfaces were set correctly, even when the unicast routing protocol suffers changes due to router failures.

With all performed tests regarding the creation of the broadcast tree, it was possible to verify that the selection of root and non-root interfaces was performed correctly. Also, by analyzing the packet captures, it was possible to verify that multicast data packets were initially exchanged in all links, like it was supposed to, due to the flooding behavior of this protocol.

E.2.3 Assert mechanism

Now we will describe the test results regarding the Assert mechanism. These tests were performed on the same topology of the previous tests, represented by Figure 6.1. In theses tests, we only focused

on what happened on the subnet that connected routers R2, R3, R4, R5 and R6. The corresponding Netkit network configurations, tests and results are stored in a branch named “Test_PIM_Assert” at our GitHub repository [33].

- **Test 7 -** Election of a single Assert Winner

  The flood of multicast data packets from host Source, must trigger the Assert mechanism, causing the exchange of Assert message with RPC information. Since all routers have their interfaces set with default costs, routers R2, R3 and R4 have the same RPC to Source, causing the Assert mechanism to use the IP address to perform the election. It was expected that router R4 would won the election, since its non-root interface had the greatest IP address.

  In the packet capture, we have verified that routers started exchanging Assert messages after they forward the first multicast data packets. We have observed a triplication of the same initial data packet, due to this being transmitted by R2, R3 and R4. The Assert messages, transmitted after the first data packet, were well dissected and had the same RPC (20), as expected.

  It was also possible to verify that in reaction to the Assert messages, downstream routers transmitted Graft messages whenever a new Assert Winner was elected. We have observed that routers R5 and R6 transmitted a Graft message destined to R2 because this router was the first one to transmit an Assert message, which was initially elected as the Assert Winner by these two routers. Then R4 transmitted an Assert message and by having a greater IP address than R2, both downstream routers considered a change of the Assert Winner, causing the transmission of a new Graft message but now destined to R4. Then R3 transmitted an Assert message but since this message lost the Assert compared to the already elected Assert Winner (R4), downstream routers did not transmit new Graft messages. These Graft messages were acknowledged back with Graft-Ack messages.

  Also, whenever an upstream router lost the Assert mechanism, it transmitted a Prune due to the reception of an Assert with the same RPC but a greater IP address, which was observed in the packet capture. Routers R2 and R3 transmitted Prune messages due to the reception of the Assert from R4.

  In the log, we have observed that all routers connected to the switch, transitioned from NoInfo to the Loser state, except for router R4 that transitioned to the Winner state.

  With this test we have verified that the implemented Assert mechanism elected correctly an Assert Winner. Also, routers reacted correctly to this election, by transmitting Prune messages whenever a non-root interface lost the Assert and by transmitting Graft messages whenever a root interface detected a change of the Assert Winner.

- **Test 8 -** Reaction to the failure of the current Assert Winner (without having data packets being forwarded)

  After Test 7, we have stopped the protocol process only in interface eth1 of the Assert Winner (R4). This caused the interface to transmit an Hello message with 0 as the Hello Hold Time, causing all

routers to stop considering R4 as being “alive”. Since this router was considered to be the Assert Winner, all routers reacted to its failure by transitioning to the NoInfo state. In the logs, we have observed this expected behavior.

It was not transmitted any data packet during this test, which caused all routers to remain in NoInfo state. This proved that the implementation reacted correctly to the removal/failure of the Assert Winner.

- **Test 9 -** Reelection of the Assert Winner, after the previous one fails, with data packets being forwarded

  After Test 8, the transmission of data packets should reelect a new Assert Winner (R3).

  The host Source by transmitting data packets, caused those data packets to be transmitted by routers R2 and R3. Since data packets were received in non-root interfaces, this triggered the transmission of new Assert messages by both R2 and R3.

  The Assert messages had the same RPC to the Source subnet, like it was supposed to. This caused the election of the Assert Winner to be accomplished by the comparison of the IP address. Since router R3 had a greater IP address, it was elected as the Assert Winner, causing a transition to the Winner state. All the other routers transitioned to the Loser state, in reaction to the exchanged messages. This behavior was observed in the logs and in the packet capture.

  After the exchange of data packets and Assert messages, the router that lost the Assert (R2) transmitted a Prune message, like it was supposed to. Also, in reaction to the new Assert Winner (R3), downstream routers sent Graft messages destined to it. These messages were acknowledged back with a Graft-Ack message.

  This test proved that after a failure/removal of the Assert Winner, all non-root interfaces by transmitting data packets, triggered a new Assert Winner election with the exchange of new Assert control messages.

- **Test 10 -** Reelection of the Assert Winner in case the previous one changes its interface role (becomes root type)

  After Test 9, R3 was elected as the Assert Winner of the link. We have changed the cost of interface eth0 of this router to 100, in order for interface eth1, of this router, to become root type.

  The change of the interface cost and corresponding change of its role caused the transmission of an Assert Cancel message via interface eth1 of R3, as it can be observed in the packet capture. This caused all routers to transition to the NoInfo state, as it was observed in the logs. This Assert Cancel message represents an Assert message with RPC of infinite.

  Following data packets were forwarded by router R2 due to its transition to the NoInfo state.

  With this test, we have verified that the implementation reacted correctly to the change of the interface role at the Assert Winner, by having all routers transitioning to the NoInfo state with the reception of an Assert Cancel message from the current Assert Winner.

With these tests, we have verified that the Assert Winner was elected correctly. This election was performed correctly even when the current winner fails and whenever its interface becomes root type.

E.2.4 Pruning, Joining and Grafting

Now we will describe the test results regarding the Pruning, Joining and Grafting of the tree and if the Prune Override mechanism is not pruning branches of the tree that have routers still interested in receiving data packets. To test this, we used the same topology of the previous tests, represented by Figure 6.1. In this topology, we have explicitly set interface eth2 of Router7 to have a cost of 100. This way, interface eth0 was the root interface of Router R7, allowing to test if the Prune Override mechanism, at the switch, was working correctly, being controlled by the membership of both Client1 and Client2. In order to perform these tests, we have captured all packets exchanged at all links of the network topology, in order to verify if the correct control and data packets were exchanged. These tests will focus on the Upstream Interface and Downstream Interface state machines.

The corresponding Netkit network configurations, tests and results are stored in a branch named "Test_PIM_JOIN_PRUNE_GRAFT" at our GitHub repository [33].

To perform these tests, all timers were set to their default values, meaning that a non-root interface after hearing a Prune message will remain in PrunePending state for 3 seconds (controlled by the PrunePending timer), in order to give time for downstream routers to override the Prune. Downstream routers that want to override the Prune with a Join message, must perform this within 2.5 seconds (Override timer). The time set for the Override timer is randomly chosen in order to support the suppression mechanism (a router does not need to send a Join if other downstream router has already transmitted that same message).

- **Test 11 -** Pruning of redundant paths from the broadcast tree

  In this test, we started by having Client1 and Client2 interested in receiving multicast data packets destined to group 224.12.12.12. This way, we could test if redundant paths at the broadcast tree were pruned from it.

  Host Source by transmitting its data packets, these got flooded on the network, which was possible to verify in all packet captures.

  We have observed the following state transitions at all root and non-root interfaces, and we verified that these were correct:

  - R1: eth0 set to Forwarding state in Upstream Interface state machine; eth1 set to Pruned state in Downstream Interface state machine; eth2 set to Pruned in Downstream Interface state machine and eth3 set to NoInfo in Downstream Interface state machine.

    The Pruned interfaces were due to the routers that were directly connected to those interfaces, R2 and R3, have lost the Assert mechanism at the switch.

    Interface eth3 was in NoInfo state because it was connected to router R4 that won the Assert and had downstream routers interested in receiving those data packets.

Interface eth0 was in Forwarding state because one non-root interface (eth3) must forward data packets received by the root interface.

- R2 and R3: eth0 set to Pruned state in Upstream Interface state machine.

  The state of non-root interfaces at these routers was not relevant, since what defined if they must forward multicast data traffic was the Assert state machine. These interfaces could be placed in Pruned/NoInfo state depending on the last received control message.

- R4: eth0 set to Forwarding state in Upstream Interface state machine and eth1 set to NoInfo in Downstream Interface state machine.

  Interface eth1 was placed in NoInfo state since it connected to routers that reached interested receivers. For this reason, it was placed in this state.

  Interface eth0 was placed in Forwarding state because one non-root interface must forward data packets (eth1).

- R5: eth0 set to Forwarding in Upstream Interface state machine and eth1 set to NoInfo state in Downstream Interface state machine.

  Interface eth1 was set to NoInfo state because it connected to one interested router (R7). For this reason, interface eth0 was set to Forwarding state.

- R6 and R7: eth0 set to Forwarding in Upstream Interface state machine.

  Interface eth0 was set to Forwarding state in these routers because they must forward data packets, due to one of their non-root interfaces being directly connected to interested receivers.

  The state of interface eth1, on these routers, was not relevant for this test, because they were not connected to neighbor routers, so they remained in NoInfo state. Regarding interface of eth2 of R7, the forwarding of data traffic was controlled by the Assert state machine, for this reason its state was not relevant for this test.

Regarding exchanged control messages, we have verified that the link that connects R2 to R7 has exchanged a Prune message, due to the Assert mechanism (R7 lost the Assert).

In the switched network, downstream routers transmitted Graft messages whenever a new Assert Winner was elected, like referred in the Assert tests. These messages were acknowledged back with a Graft-Ack message. This was supposed to happen since a new Assert has been elected and downstream routers were interested in receiving data packets (due to the interest of Client1 and Client2).

Also in the switched network, we have observed that Assert Losers transmitted Prune messages. This caused the Prune Override mechanism to be triggered, since downstream routers were still interested in receiving data packets. We have observed that the last transmitted Prune message had a response from R5 with a Join message. This time difference between the transmission of the Prune and the Join was of 1 second, which was within the time range of the default time (2.5 seconds).

The two clients have received all transmitted data packets from Source. The initial transmitted data packets were triplicated, due to the three upstream routers connected to the switched network, that flooded the first data packet. Then, after the Assert, transmitted data packets were received correctly without duplications.

With this test, we have verified that all routers transitioned to the correct state regarding their Upstream and Downstream Interface state machines, when the tree flooded for the first time.

- **Test 12 -** Correct multicast forwarding decision in case Client2 leaves the group

  Client2 by leaving the group, this should cause the pruning of the multicast distribution tree. The Prune Override mechanism should not prune the switched network due to the interest of Client1.

  We have started by closing the socket of Client2, in order for it to send an IGMP Leave message.

  Regarding the state transitions, we have observed that interface eth0 of router R6 was placed in Pruned state, in reaction to the IGMP Leave of Client2. This caused R6 to transmit a Prune message.

  The Prune Override mechanism was triggered, causing router R4 to transition to PrunePending state. We have observed that router R5 transmitted a Join message in reaction to the expiration of its Override timer. This expiration happened, according to the logs, 2 seconds after the reception of the Prune message, which was within the range of 2.5 seconds specified by the protocol.

  The transmission of the Join, by R5, caused the interface of the Assert Winner (R4) to transition back to the NoInfo state.

  With this test, we have observed that the Prune Override mechanism worked correctly. Also, the timers used to control this mechanism were set accordingly.

- **Test 13 -** Correct multicast forwarding decision in case both hosts leave the group

  Client1 by leaving the group must cause the pruning of the tree since there is no host interested in receiving data packets. After Test12, Client1 left group 224.12.12.12 in order to not have any member interested in this traffic.

  We have observed, according to the logs, that previously non-Pruned branches transitioned to a Pruned state (interfaces that connect R7 to R5, R5 to R4 and R4 to R1).

  According to the *Wireshark* capture, in the switched network, the reception of the Prune, from router R5, caused R4 to transition to a PrunePending state. Since no router was interested in receiving multicast data packets, no one did override the Prune with a Join message. Eventually, the PrunePending timer of R4 expired, causing this router to transition to the Pruned state. According to the logs, the expiration of this timer happened 3 seconds after the reception of the Prune message, like specified.

  After the expiration of the PrunePending timer at R4, which caused interface eth1 of this router to transition to the Pruned state, we have observed in the packet capture, that this router, Assert Winner of the switched network, transmitted a Prune message. This message is known as Prune

Echo message and has the goal of adding extra reliability to the Prune Override mechanism. Since no one has transmitted a Join after this message, the router remained in the Pruned state.

With this test, we have confirmed that the protocol reacted accordingly to the removal of interested members, causing the pruning of the whole tree. Also, the Prune Override mechanism pruned the tree due to the absence of a Join message. We have also observed the transmission of a Prune Echo message from the current Assert Winner, adding extra reliability guarantees.

- **Test 14 -** Correct multicast forwarding decision by having the tree "grafted" (Client2 becomes again interested in receiving data packets)

  Client2 by being again interested in the group must cause the "grafting" of the tree. After Test 13, Client2 became interested again in receiving data packets of group 224.12.12.12. According to the logs, this caused router R6 to transition to AckPending state. In the *Wireshark* capture we can verify that a Graft message destined to R4 was transmitted by R6.

  Also in the logs, the reception of the Graft message, by R4, caused this router to transition its non-root interface to NoInfo state. In the *Wireshark* capture, we can observe that R4 acknowledged back with a Graft-Ack message, destined to router R6. In the logs, we can observe that the reception of the Graft-Ack message, by R6, caused this router to transition from an AckPending state to a Forwarding state, as expected.

  Router R4, by being connected to downstream routers, caused a graft of the tree via its root interface (eth0), in order for traffic originated from Source to be forwarded from router R1 to R4 again. The same Graft and Graft-Ack exchange happened at that link.

  Then Source transmitted data packets and we have correctly verified that those packets reached the new interested receiver.

  With this test, we have verified that the tree was correctly grafted, in order to forward data packets through previously pruned links.

- **Test 15 -** Correct multicast forwarding decision in reaction to the timeout of pruned branches (re-flood of multicast data traffic after this timeout)

  After Test 14, we have waited a given amount of time. We have verified in the logs that approximately at time 16:05:00 all routers transitioned to state NoInfo, in reaction to expiration of timers and Graft messages being exchanged. The first test was started at time 16:02:00, which confirmed that the pruned tree only lasted three minutes, like defined in the specification. In the logs, we can verify that this happened due to the expiration of the timers that control the Assert and Downstream Interface state machines. The expiration of the Assert timer caused a transition to the NoInfo state, while the expiration of the PruneTimer caused non-root interfaces, that were in a Pruned state, to transition back to the NoInfo state.

  Data packets by being again flooded, recreated the multicast distribution tree by pruning again redundant paths and branches that do not lead to interested receivers.

With this test, we have verified that the tree only lasted 3 minutes, as specified. Also, after the expiration of this state, we have observed the recreation of the tree due to the flood of data packets through the whole network.

By performing these tests, regarding the interest of hosts, we could verify that the transitions in the Upstream and Downstream Interface state machines were made correctly. Also, the transmitted messages and corresponding state transitions were made within the time intervals specified by the protocol specification, which confirmed that the timers were set correctly.

E.2.5 State Refresh

Now we will describe the test results regarding the State Refresh mechanism. The same topology of the previous tests was used (Figure 6.1). All interfaces were set to their default costs (10), except for interface eth2 of R7 that was set to 100. This meant that interface eth0, in all routers, must be set as the root interface for data traffic originated from Source.

In order to perform these tests, all routers enabled the protocol with the State Refresh feature. This caused all routers to include an additional option in the exchanged Hello messages and also to periodically transmit and flood through the whole network a State Refresh message. This information was present in the packet capture of all links.

All corresponding Netkit network configurations, tests and results regarding State Refresh are stored in a branch named "Test_PIM_StateRefresh" at our GitHub repository [33].

Regarding timers, they were set to their default values, which meant that the periodicity of State Refresh messages was 60 seconds (used in the State Refresh timer). Regarding the router directly connected to the source, placed in a Originator state, it will consider the source to be active as long as its Source Active timer does not expire, which was set to 210 seconds whenever it received data packets.

- **Test 16 -** Tree maintenance by State Refresh messages

  All routers should be in a NotOriginator state, except for the router that directly connects the source of multicast traffic, R1, that must be in Originator state. After starting the protocol process in all routers, host Source started transmitting data packets destined to group 224.12.12.12. We executed this test having both receivers interested in multicast traffic of this group.

  In the logs, we have verified that only router R1 transitioned to Originator state regarding this tree. All other routers transitioned to NotOriginator state, by not being directly connected to the source of multicast traffic.

  Then we have continued transmitting data packets and verified that State Refresh messages were being flooded in the network, according to all packet captures. In the switched network, that connects R2, R3, R4, R5 and R6, the first data packet was received at time 49.18 seconds and the first State Refresh message was received at time 109.00 seconds, which was according to the

expected periodicity of these messages (60 seconds). Following State Refresh messages also respected this periodicity (at times 169.05, 229.14, 289.15, ...).

We have observed that only the Assert Winner, of a given link, transmitted State Refresh messages. Other routers did not transmit these types of messages, as expected.

Regarding the content of State Refresh messages, these were correct, by being properly dissected and by having their fields set to expected values. Those fields include the RPC of the router that transmitted the message, TTL which was decremented whenever the message was forwarded to a new link and also the flags. Regarding the flags, these included the Prune indicator, Prune Now and Assert Override. The Prune indicator was set according to the Downstream Interface state machine, the Prune Now flag was set on every third transmitted State Refresh message and the Assert Override flag was set according to the Assert state machine.

Regarding the Prune indicator flag, this was visible on links that were pruned out from the broadcast tree, like the link that connects R1 to R2. In this link, after R2 transmitted a Prune message to R1, because the first router lost the Assert in the switched network, this caused following State Refresh messages, originated by R1, to be transmitted with the Prune indicator always set. In the switched network, we have observed that State Refresh messages did not have the Prune indicator flag set due to the interest of both downstream routers, R5 and R6, by reaching interested receivers. Pruned links did not transmit any data packets while the source was active, in a time window of 10 minutes, due to the refresh of the Pruned state, in the Downstream Interface state machine, caused by the reception of State Refresh messages.

Regarding the Assert Override flag of State Refresh messages, in links that elected the Assert Winner with Assert messages, the Assert Winner transmitted these messages with the flag cleared. In links that did not require to elect an Assert Winner with Assert messages, such as the link that connected R1 to R2, because R1 was connected with a non-root interface and R2 was connected to the same link with a root interface, the flag was set. The value of this flag was set accordingly because only routers in a NoInfo state at the Assert state machine, must send these messages with the flag set, like R1 did in the link that connected this router to R2.

Regarding the Assert election, we have just observed the exchange of Assert messages on the arrival of the first data packets. After electing correctly the Assert Winner, through the first Assert messages, following State Refresh messages refreshed the state of the Assert Winner on all routers. This was possible to verify in the logs, by the reception of the State Refresh from the Assert Winner that caused the restart of the Assert timer, used to maintain state regarding the winner of a given link.

During the transmission of data packets, around 10 minutes, R1 remained in the Originator state due to the reception of data packets from Source, causing the reset of its Source Active Timer.

With this test we have verified that State Refresh messages were correctly exchanged by routers. Only the Assert Winner transmitted these messages, as expected.

Regarding the State Refresh state machine, only the router that directly connected to the source of multicast traffic transitioned and remained in the Originator state. All other routers, that were not directly connected to the source, remained in NotOriginator state. By not hearing data packets in pruned links, we have confirmed that these messages correctly refreshed the Assert and Pruned states.

- **Test 17 -** Stop maintaining a tree due to the absence of data and State Refresh messages

  Absence of data packets must cause the router directly connected to the source of multicast traffic, R1, to transition to the NotOriginator state in the State Refresh state machine. State Refresh messages should no longer be transmitted by this router. After Test 16, host Source stopped transmitting data packets.

  We have observed in all links that around 3 additional StateRefresh messages were exchanged after the source stopped transmitting data packets. Then no more State Refresh messages were observed. By looking to the logs, we have verified that this behavior was due to the expiration of the Source Actime timer of router R1, that caused a transition to the NotOriginator state. In this state, the router that directly connects to the source, no longer transmits State Refresh messages.

  In the switched network, after the last transmitted State Refresh message, there were no more messages exchanged regarding this tree.

  In a previously pruned link, like the one that connects R1 to R2, we have observed that after the last State Refresh message was transmitted, Graft and Graft-Ack messages were exchanged. This was due to the expiration of both the Prune state (in the Downstream Interface state machine of router R2) and the Loser state (in the Assert state machine of router R2). The expiration of both states, caused this router to have a non-root interface included in the OIL at its multicast routing table. This caused router R2 to have its OIL non-empty, triggering the exchange of Graft messages, since the link no longer was considered to be Pruned.

  We have observed in the logs, that all routers, at around the same time, timed out the state used to control the Assert and Prune states. This happened around 3 minutes after the last exchanged State Refresh message, as expected.

  With this test, we have verified that the router directly connected to the source of multicast traffic controlled its State Refresh state machine, by transitioning to the NotOriginator state when it no longer heard data packets. Also, all routers in the network correctly reacted to the absence of State Refresh messages, by expiring the Assert and Prune states.

We have verified that the router directly connected to the source of multicast traffic reacted correctly to the forwarding and absence of data packets. This regulated the transmission of State Refresh messages that were forwarded throughout the whole network. We have confirmed that these messages were correctly built, by verifying them in *Wireshark*.

All tests proved that each module of the protocol was implemented into accordance to the specification. All timers were set into accordance to the specification, by observing that messages and state

transitions were performed in a given time window or after a given amount of time. Control messages were correctly built, being fully dissected by *Wireshark*.

E.3 HPIM-DM Implementation Test Results

Like PIM-DM, a unicast routing protocol was configured in order for HPIM-DM to correctly determine root and non-root interfaces. For this reason, OSPF [28] was used as the underlying unicast routing protocol, having all routers executing *Quagga* [44], that had an implementation of this protocol.

E.3.1 Neighborhood maintenance

Regarding the tests of neighborhood relationships, we just needed a single subnetwork with one or more routers connected to it, for this reason we opted to use the topology of Figure E.1 that is composed by two routers, R1 and R2. In these tests, we focused only on the formation of neighborhood relationships, without exchanging any tree information with the neighbor router (no trees were previously set). The exchange of tree information during the synchronization will be described on following sections.

All corresponding Netkit network configurations, tests and results regarding establishment of neighborhood relationships, without having trees set up in the network, are stored at our GitHub repository [34] in a branch named "Test_NewProtocol_Sync_Without_Trees".

A synchronization to be successful requires both routers to exchange Sync messages with a bunch of sequence numbers (BootTime, SnapshotSN and SyncSN). In order to accomplish this, both routers must exchange these sequence numbers using a Stop-and-Wait protection, in which there is a Master-Slave election. At the end of this process, both routers need to know the same sequence numbers and a router must have its neighbor router in the UPDATED state.

- **Test 1 -** Establishment of neighborhood relationship between two unknown neighbors, without any existing trees

  In this test, we started first the routing process on R1 and then on R2. We verified that by enabling the process in an interface, this triggered the transmission of an Hello message.

  According to the logs, the first transmitted message by R1 was an Hello message having the BootTime set to 1534977381. Then R2 also transmitted an Hello message with BootTime set to 1534977383. According to the logs, the first transmitted Hello message from R1 was not received by R2 since at that time this router did not have the protocol process started. This explains why the synchronization process only started after the Hello message from R2 was transmitted.

  The first Sync message was transmitted by R1, in response to the Hello message from an unknown neighbor router, R2. Router R1 by being the first one triggering the synchronization process, considered itself as Master of it, which is possible to verify in the logs by R1 transitioning its neighbor R2 from an UNKNOWN to SLAVE state. The Sync message transmitted by R1 had the BootTimes of both routers set correctly (own BootTime set to 1534977381 and neighbor BootTime

set to 1534977383), just like in the Hello messages. Regarding the SnapshotSNs, R1 transmitted a Sync message with its own SnapshotSN set to 1 and the neighbor SnapshotSN set to 0, just like it was expected since R1 has still no information regarding the neighbor router. The SyncSN was set to 0, as expected since this was the first phase of the synchronization process. Regarding the flags, the Master flag was set and the More flag was cleared. These flags were correctly set, since R1 considered itself as the Master of this process and had no more tree information to transmit in following Sync messages.

According to the logs, we verified that R2 heard R1 for the first time due to its Sync message. For this reason, R2 had neighbor R1 transition from an UNKNOWN to MASTER state. In response to the Sync message from R1, R2 transmitted a new Sync message. This message had all SNs correct (own BootTime set to 1534977383, neighbor BootTime set to 1534977381), just like the Hello messages. Regarding the SnapshotSNs, this message had information regarding the neighbor SnapshotSN, which was set to 1, due to the received message and had its own SnapshotSN also set to 1. Regarding the flags, both Master and More were cleared, as expected since the router considered itself as the Slave of the synchronization process and did not have more tree information to be transmitted. The SyncSN was set to 0, since this was the first phase of the synchronization.

R1 by receiving the message from R2, transmitted a new Sync message equal to its first message but now having the SyncSN set to 1. R2 answered back with a message equal to its second message, having 1 also set as the SyncSN. After this, both routers finished successfully the synchronization process, transitioning to UPDATED state, and had exchanged and stored the same SNs.

In this test we have verified that both routers have established a correct neighborhood relationship, having exchanged all information consistently.

- **Test 2 -** Re-establishment of neighborhood relationship after a known neighbor reboots

  After Test 1, we have restarted the protocol process in R2. This caused R2 to transmit an Hello message with a greater BootTime, 1534977413, as expected.

  In the logs, we have verified that R1 by receiving the message from R2, considered its neighbor to have restarted, causing a transition in the neighbor state machine, regarding R2, from UPDATED to SLAVE. This caused R1 to consider itself as the Master of the synchronization process, since it has detected a reboot of a known neighbor due to the reception of a control message with a greater BootTime. This is visible in the packet capture by having R1 transmitting a new Sync message to R2.

  The new Sync message transmitted by R1 had its own BootTime set to 1534977381, like the previous Hello and Sync messages, neighbor BootTime set to 1534977413, due to the received Hello message from R2, own SnapshotSN set to 2, which is greater than the sequence number used in the previous synchronization process as expected, neighbor SnapshotSN set to 0, since this information was still not known, and SyncSN set to 0, due to this Sync message belong to the

first phase of the Sync message exchange. The Master flag was set and More flag was cleared, just like the previous synchronization process.

In response to the message transmitted by R1, R2 considered R1 to be in MASTER state, since the router discovered its neighbor by a Sync message. This caused a new message from R2 to be transmitted to R1, having its own BootTime set to 1534977413, neighbor BootTime set to 1534977381, own SnapshotSN set to 1, neighbor SnapshotSN set to 2 and SyncSN set to 0. The Master and More were cleared, as expected. As we can see, all SNs had consistent information, like the first Sync message transmitted by R1.

Two more Sync messages were exchanged having all SNs and flags consistent and SyncSN set to 1. At the end, both routers considered each other to be in an UPDATED state, as expected.

In this test, we have verified that a router can detect when its neighbor has restarted, causing a newer synchronization process to be started. This detection was caused by the greater BootTime obtained after the protocol process restarted on R2.

- **Test 3 -** Neighborhood relationship break after known neighbor fails

  After Test 2, we have stopped the protocol process in R2. This caused R2 to no longer transmit Hello messages in the switched network.

  After a given amount of time we have verified that in the logs, router R1 stopped considering R2 as a neighbor.

  In this test we have verified that the absence of Hello messages from a neighbor router causes a router to break a neighborhood relationship.

E.3.2 Formation and maintenance of the broadcast tree

Now we will describe the test results regarding the formation of the broadcast tree. In these tests we have verified if the interfaces' roles were selected correctly, if the transmitted control messages were correct and if the state of the tree was set correctly. In this protocol, the formation of the broadcast tree happens due to the exchange of IamUpstream and IamNoLongerUpstream messages. The synchronization between routers, in the formation of neighborhood relationships, did not include the trees of these tests.

All corresponding Netkit network configurations, tests and results regarding the formation of the broadcast tree are stored in a branch named "Test_NewProtocol_BroadcastTree" at our GitHub repository [34].

In these tests and on the following tests, we have enabled the option of "Flooding Initial Data Packets". This meant that when an UPSTREAM router had not received interest information from its neighbors, it considered those neighbors to be interested. This allowed initial data packets to be flooded in order to not be "lost" during the formation of the tree.

- **Test 4 -** Formation of the broadcast tree (root interface selection and tree state)

In this test, after all routers synchronized with each other, Source transmitted data packets for the first time. The two hosts reported interest in order to verify if data packets were received.

In the logs, we have verified that all interfaces' roles were correctly selected, i.e. eth0 was selected as the root interface in all routers, except for router R7 that selected eth2 as its root interface. All the remaining interfaces were selected as non-root type. It was also possible to verify that the tree state regarding (10.1.1.100, 224.12.12.12) transitioned from UNKNOWN to ACTIVE state in all routers. In some routers, prior to the transition to ACTIVE state, there was a transition to INACTIVE state. This meant that all routers received IamUpstream messages through their root interface, but some routers received first IamUpstream messages through their non-root interfaces.

We can observe that in the following links there were IamUpstream messages exchanged:

- Links that connects R1 to R2, R1 to R3 and R1 to R4: Only R1 transmitted an IamUpstream(S,G) message.

  This was expected since only R1 was connected by a non-root interface to those links. All messages transmitted by R1 had an RPC of 0 because this router was directly connected to the source of multicast traffic.

- Link that connects R2, R3, R4, R5 and R6: Only R2, R3 and R4 transmitted IamUpstream(S,G) messages.

  This was expected since only those routers were connected by non-root interfaces to that link. These routers transmitted IamUpstream(S,G) messages with the same RPC of 20. This RPC was expected because all routers set their interfaces' cost to 10.

  We have observed in the packet capture that the first IamUpstream message was transmitted by R2. According to the logs, this caused routers R5 and R6 to transition to ACTIVE state due to the presence of an UPSTREAM neighbor connected to their root interfaces. In the case of routers R3 and R4, these transitioned to INACTIVE state, suggesting that these routers received their first IamUpstream message through their non-root interfaces.

  Then according to the logs, R3 and R4 transitioned to ACTIVE state due to the reception of R1's message. This caused the transmission of IamUpstream messages from both R3 and R4.

- Link that connects R2 to R7: Only R2 transmitted an IamUpstream(S,G) message.

  This was expected because only R2 was connected by a non-root interface to that link. That message was transmitted with an RPC of 20, as expected. This IamUpstream message by being received through R7's root interface caused a transition to ACTIVE state in this router.

- Link that connects R5 to R7: both routers transmitted IamUpstream(S,G) messages.

  Since both routers were connected to each other by non-root interfaces, the transition to ACTIVE state caused the transmission of IamUpstream(S,G) messages from both routers. These messages were transmitted with RPC of 30, as expected.

As expected, root interfaces did not transmit any IamUpstream messages. They only transmitted interest messages, such as Interest or NoInterest. These interest messages will be further analyzed in the next tests.

We have observed in all packet captures that after an IamUpstream message was received, all neighbors acknowledged that message with an ACK message. Those ACK messages were correct by having information from the IamUpstream message, such as a reference of the tree (source IP and group IP) and its sequence number. Beside this information, they also included information regarding the synchronization process, such as BootTimes and SnapshotSNs. These sequence numbers will be further analyzed in the next tests. Also, we did not observe any retransmissions of IamUpstream messages, which confirmed that the ACKs were correctly received and had consistent information to properly acknowledge those IamUpstream messages.

During the formation of the tree, both members received all data packets regarding this tree, which confirmed that the protocol was flooding data packets during the formation of the broadcast tree, as expected.

With this test we have verified that the broadcast tree was correctly built by selecting interfaces' roles according to the unicast routing table (root and non-root interfaces). We have confirmed that all routers ended up in ACTIVE state, in the Tree state machine, due to the exchange of IamUpstream messages, which were correctly acknowledged.

- **Test 5 -** Broadcast tree reconfiguration in reaction to RPC changes detected by unicast routing protocol

  After Test 4, we have changed the cost of interface eth2 of R7 to 100 in order to change its role. This change should cause interface eth0 of R7 to be selected as the new root interface.

  By analyzing the logs we can confirm this expected behavior. Also, the tree state in this router remained in ACTIVE state due to the presence of an UPSTREAM neighbor connected to the new root interface (R2).

  We have verified that interface eth2 of R7, the new non-root interface that was previously root type, transmitted an IamUpstream message. The transmitted IamUpstream included an RPC of 40, as expected. Also, interface eth0, the new root interface that was previously non-root type, transmitted an IamNoLongerUpstream message and then an interest message destined to R5 (router responsible for forwarding data packets in that link). These transmitted messages were expected according to the protocol specification.

  Then data packets were transmitted and these were received by both members, without any duplications, through the reshaped tree. This test proved that the protocol reacted correctly to an RPC change that resulted in the modification of interfaces' roles.

- **Test 6 -** Broadcast tree reconfiguration in reaction to router failure, detected by both the unicast routing protocol and the multicast routing protocol

After Test 5, we opted to shutdown interface eth0 of router R5. This should cause a change of the interfaces' roles at router R7, in order to have eth2 as its new root interface.

In this test we started by disabling the protocol process in interface eth0 of R5, which simulated that the multicast routing protocol reacted first to this change. Since this interface was root type, this caused the protocol to no longer consider to be connected by UPSTREAM neighbors to that interface, causing router R5 to consider the tree to be in UNKNOWN state and transmit an IamNoLongerUpstream message through its non-root interface, eth1, which was received by router R7's root interface (eth0). Since R5 was the only UPSTREAM neighbor connected to the root interface of R7, this caused a Tree state transition from ACTIVE to INACTIVE in R7. The transition to INACTIVE was due to the presence of UPSTREAM neighbors connected to interface eth2 of R7, a non-root interface. This transition caused the transmission of an IamNoLongerUpstream through R7's non-root interface, eth2, as observed in the packet capture of that link.

Then we have shutdown interface eth0 of R5, causing the unicast routing protocol to detect and react to this change. This caused a change in the unicast routing table of R7, modifying the interfaces' roles of this router, being eth2 selected as its new root interface. This change in the interfaces' roles caused a transition in the Tree state from INACTIVE to ACTIVE at R7, as observed in the logs, due to the presence of an UPSTREAM neighbor connected to its new root interface (R2). The new non-root interface, eth0, in reaction to this Tree state transition transmitted an IamUpstream message, as expected. The RPC present in this last message was of 120, as expected due to the interface cost of eth2 being 100 (in Test 2).

Following data packets transmitted by Source were correctly forwarded and received by both hosts.

In this test it was possible to verify that all control messages were correctly transmitted in reaction to transitions in the Tree state machine. These caused the transmission of IamUpstream and IamNoLongerUpstream messages depending on the state transition and on the interface type. All messages were correctly acknowledged, like the previous tests.

According to these tests, the tree was correctly formed even in the presence of RPC changes and router failures that caused a modification of the interfaces' roles. The tree state was set according to the presence of UPSTREAM neighbors in all interfaces, like the protocol specified. All messages were correctly transmitted based on those state transitions and were correctly acknowledged by all neighbors connected to the same link.

E.3.3 Assert mechanism

Now we will describe the test results regarding the Assert mechanism. These tests were similar to the ones that were performed on the PIM-DM implementation, but now we did not require to retransmit data packets in order to cause a new reelection since whenever the current Assert Winner suffered a change, previously stored state regarding UPSTREAM neighbors was used in the new reelection. We used the same network topology of previous tests, represented by Figure 6.1. In these tests we only focused on what happened in the switched network, that connected routers R2, R3, R4, R5 and R6.

All corresponding Netkit network configurations, tests and results regarding the election of the router responsible for forwarding multicast traffic are stored in a branch named “Test_NewProtocol_Assert” at our GitHub repository [34].

The same configuration of the previous tests, in the unicast routing protocol, was made in order to have all routers knowing about all subnetworks. Each interface was set with a cost of 10. Also, we only started these tests after all routers established a neighborhood relationship and correctly synchronized with each other. The tree that was used on these tests was not previously formed in order to test the Assert mechanism triggered only by IamUpstream messages. As in the case of PIM-DM, by having all UPSTREAM routers (R2, R3 and R4) offering the same RPC to the Source subnet, 20, the IP address was used in this election (greater IP wins the Assert).

- **Test 7 -** Election of the AW in shared links

  Both hosts initially reported interest in order to verify if initial data packets were received. After that, the Source started transmitting data packets.

  In the switched network, according to the packet capture, router R2 was the first one transmitting an IamUpstream message. This was followed by R4 and then by R3. These transmissions were caused by these routers having an UPSTREAM neighbor, router R1, connected to their root interfaces.

  According to the logs, router R2 was the first one that transitioned to an ACTIVE state in its Tree state machine. This explains why this router was the first one transmitting an IamUpstream message. Also, according to the logs routers R3 and R4 transitioned first from an UNKNOWN state to an INACTIVE state in their Tree state machines. This was due to the first UPSTREAM neighbor being discovered by a non-root interface (R2’s message). Routers R5 and R6 transitioned immediately from UNKNOWN to ACTIVE state in their Tree state machines, due to the IamUpstream message, from R2, being received on their root interfaces.

  Then we verified that router R4 transitioned to ACTIVE state, due to the IamUpstream message from R1 being received in its root interface. Then the same happened to router R3. The transition to ACTIVE state caused both routers to transmit IamUpstream messages to the switched network, by their non-root interfaces.

  Regarding the Assert state machine, according to the logs, all routers started initially in ASSERT WINNER state. When R2 transmitted its IamUpstream message, both R3 and R4 transitioned to ASSERT LOSER state, as expected, due to their Tree state machines being in INACTIVE state.

  We have verified that when R4 transitioned to ACTIVE state, in its Tree state machine, this caused a transition to ASSERT WINNER state in interface eth1 of R4, in its Assert state machine, due to the presence of an UPSTREAM neighbor connected with a worse “Assert state” than itself (R2). This message by being received by R2 and R3, caused both routers to transition to ASSERT LOSER in their Assert state machines.

  We have also verified in the packet capture that after the exchange of IamUpstream messages, both R5 and R6, transmitted their interest messages destined to R4 (destination IP address was

unicast). This proved that both routers elected correctly R4 as the new ASSERT WINNER, since interest messages are always destined to the router that wins the Assert.

As the previous tests, all messages were acknowledged correctly. IamUpstream messages were acknowledged by all routers since they were destined to all routers (destination IP address was multicast) and interest messages were only acknowledged by a single router (destination IP address was unicast). The ACK messages had all correct information such as Sequence Numbers and tree identification. Regarding IamUpstream messages, all routers transmitted the same RPC, 20, as expected.

Client2 received initially a triplication of the first data packets, as expected, due to those packets being initially forwarded by R2, R3 and R4. Then following data packets were received without duplications, due to having a single Assert Winner connected to the switched network.

In this test we have verified that all routers elected correctly the Assert Winner by exchanging IamUpstream messages. These messages were correct, by being correctly dissected and by being correctly acknowledged. Downstream routers by sending their interest information destined to R4, proved that those routers elected that router as the new Assert Winner.

- **Test 8 -** Reelection of the AW in a shared link, in case the previous one fails

  After Test 7, we stopped the protocol process in interface eth1 of R4. This interface was considered to be the Assert Winner of the switched network. By stopping the protocol process in that interface, just like in PIM-DM, a new Hello message was transmitted, informing all routers that this interface was shutdown. This allowed all routers to react quickly to its removal.

  We could verify in the packet capture, that at time 83.007 seconds an Hello message from R4 was transmitted, having the option Hello Holdtime set to 0. The reception of this message caused all routers to stop considering R4 as a neighbor.

  According to the logs, router R3 transitioned from ASSERT LOSER to ASSERT WINNER state, as expected. Router R2 remained in ASSERT LOSER state, due to the presence of an UPSTREAM neighbor with a better Assert state, R3.

  In the packet capture we can verify that right after the referred Hello message was transmitted, at time 83.037 and 83.041 seconds, interest messages from R5 and R6 were transmitted to the new Assert Winner (R3). This proved that both downstream routers elected correctly R3 as the new Assert Winner.

  The Source by transmitting data packets, these were received by both hosts. The reception in Client2 was due to these being forwarded by the new Assert Winner (R3).

  In this test we have confirmed that the removal of a router that was considered to be the Assert Winner of a given link, caused all routers to recalculate the new Assert Winner based on the presence of UPSTREAM information that was previously stored. IamUpstream messages were not exchanged proving that this election took into account only the information that was previously stored.

- **Test 9 -** Reelection of the AW in a shared link, in case the previous one changes its interface role (its interface becomes root type)

  After Test 8, interface eth1 of R3 was elected as the new Assert Winner of the switched network. In this test, like it was performed in the same test of PIM-DM, we changed the cost of interface eth0 of R3 to 100, in order for eth1 of the same router to become root type.

  After changing the cost of R3's interface, we have verified in the logs that router R3 had its interfaces' roles changed. Interface eth1 became root type. The change of the interfaces' roles in R3, caused it to transmit an IamNoLongerUpstream message though its new root interface, eth1, as it was observed in the packet capture.

  This packet by being received by R2 caused this router to consider itself as the new Assert Winner of the switched network, which was observed in the logs, by the transition to ASSERT WINNER state in its Assert state machine. This happened because interface eth1 of R2 did not have any information regarding UPSTREAM neighbors connected to that interface.

  Also in the packet capture we can observe that after the transmission of the IamNoLongerUpstream, from R3, all downstream routers transmitted interest messages destined to R2, which confirms that all routers considered that router as the new Assert Winner of the switched network. These interest messages were transmitted by R3, R5 and R6, via their root interfaces.

  Source by transmitting data packets, these were received by both hosts. The reception of data packets in Client2 proved that the new Assert Winner, R2, correctly forwarded them.

  All exchanged messages, such as IamNoLongerUpstream and interest messages were correctly acknowledged, just like in the previous tests.

  As it was verified in this test, a non-root interface of an Assert Winner by becoming root type, caused the new root interface to transmit an IamNoLongerUpstream, in order for all routers to not consider it as UPSTREAM. This triggered a reelection of the new Assert Winner, based on the information that was already stored. There were no additional IamUpstream messages transmitted, which proves that only previously stored information was used in the election. All routers by transmitting interest messages destined to the new Assert Winner, R2, proved that this election was performed correctly.

Like in the tests of PIM-DM, all tests elected the same Assert Winners, since this election took into account the same type of information, RPC and IP address. We had the same results, differing only that in this protocol this election was not performed by the exchange of data packets, i.e. the reception of data packets in non-root interfaces did not trigger the election, instead already stored information was used. All exchanged control messages and all state transitions in both the Tree and Assert state machines were expected, in accordance to the protocol specification.

E.3.4 Forwarding based on Interest information

Now we will describe the tests regarding the decision of data forwarding based on the interest of NOT UPSTREAM neighbors. These tests were similar to the ones that were performed on the PIM-DM implementation, regarding the Prune, Join and Graft messages. Like previous tests, we used the same network topology, represented by Figure 6.1. In these tests we focused on the exchanged control messages of all routers and also on the decision of the Assert Winner of all links regarding the interest of NOT UPSTREAM neighbors. In order to detect this decision, we logged all events of non-root interfaces, regarding the global decision of interest, i.e. there is downstream interest if at least one neighbor is interested or no downstream interest otherwise.

All corresponding Netkit network configurations, tests and results regarding the interest decision of the router responsible for forwarding multicast traffic are stored in a branch named "Test_NewProtocol_Interest" at our GitHub repository [34].

Regarding the unicast routing protocol, we changed the cost of interface eth2 of router R7 to 100, in order for eth0 of this same router be selected as root type. This allowed the interest decision on the switched network to be controlled by the membership of both hosts.

We performed these tests after all neighbors correctly formed neighborhood relationships with each other. The tree formation was done only by the exchange of IamUpstream messages. In these tests we did not focus on the synchronization process.

- **Test 10 -** Correct multicast forwarding decision based on neighbors' interests, after the tree is initially built

  We started initially by having both hosts interested in receiving multicast traffic of group 224.12.12.12. Then Source started transmitting data packets and all routers formed the tree.

  Since all routers had the option "Flood Initial Data Packets" enabled, we verify in the logs that all interfaces, that connected to at least one neighbor, started initially by considering that there was interest by downstream routers. This allowed to flood initial data packets through the whole network, being received by both hosts. Interfaces eth1 of R7 and eth1 of R6 did not consider that there was interest from downstream neighbors since they were not connected to other routers.

  With the exchange of IamUpstream messages due to the tree formation we observed the following in each link:

  - Link that connects R1 to R2:

    According to the packet capture, R2 started by informing R1 that it was interested, via an Interest message. This is explained by this router initially consider itself as the Assert Winner of the switched link and of the link that connects R2 to R7 and by considering that there was downstream interest in both links (due to the enabled option).

    Then we verified, with the election of the Assert Winner in the switched link and by the absence of NOT UPSTREAM neighbors connected to the link R2-R7, this router transmitted a NoInterest message to R1.

R1 then no longer considered the link that connected this router to R2 to have downstream interest, as expected.

- Link that connects R1 to R3:

  We verified that R3 started by informing R1 that it was not interested. This can be explained by R3 consider itself initially as the Assert Loser of the switched link, since R2 transmitted first an IamUpstream message when R3 was considered to be in UNKNOWN state in its Tree state machine, causing a transition to INACTIVE state.

  Since R3 was in INACTIVE state and there was an UPSTREAM neighbor connected to its non-root interface, its interface was not part of the broadcast tree (it was pruned). By not having any interface that would forward data packets, when R3 detected the existence of an UPSTREAM neighbor connected to its root interface, it transmitted initially a NoInterest message. Then R3 by transitioning to ACTIVE state and consider itself as the Assert Winner of the switched link, because at that time only R2 had transmitted the IamUpstream message and R3 had a better Assert state, this router transmitted an Interest message.

  Then R3 considered again to be the Assert Loser of the switched link, due to presence of R4, causing the transmission of a NoInterest message to R1. From this point on there were no more interest messages exchanged between both routers. At the end, interface eth2 of R1 considered this link to not be connected to interested downstream routers, causing the pruning of this interface from the distribution tree.

- Link that connects R1 to R4:

  The same thing that was described in link R1-R3 happened in this link. The only difference was that the final message transmitted by R4 to R1 was an Interest message, due to this router consider itself as the Assert Winner of the switched network and by being connected to interested downstream neighbors.

- Link that connects R2, R3, R4, R5 and R6:

  We have observed that the first router that transmitted an IamUpstream message was R2, then R4 and finally R3.

  When R2 transmitted its message, all routers transmitted interest messages to this router, including R3 and R4. These two routers transmitted NoInterest messages to R2 since they were connected by non-root interfaces to the switch. The other routers transmitted Interest messages, as expected.

  Then, when R4 transmitted its IamUpstream message, only R5 and R6 transmitted Interest messages to it, since it was the new Assert Winner of the link. R3 did not transmit an interest message to R4 because at that time it already considered the tree to be in ACTIVE state.

  Finally, when R3 transmitted its IamUpstream message, there were no interest messages exchanged. This was expected because all routers already were storing at that time information regarding the RPC of R4. When R3 transmitted its IamUpstream message, R4 was already considered to be the Assert Winner of the switched network.

The reception of Interest messages at R4 caused this router to consider the switched network to have interested downstream neighbors, as expected. The conclusion reached by R2 and R3, regarding the existence of interested downstream neighbors, was not relevant since only the Assert Winner of a link must have a consistent view regarding this matter. NOT UPSTREAM neighbors only send their interest information to the current Assert Winner, so Assert Losers do not require to have that same view since they do not make forwarding decisions in this link. If an Assert Loser becomes Assert Winner, all NOT UPSTREAM neighbors would retransmit their interest to the new winner.

- Link that connects R5 to R7:

  When R5 transmitted its IamUpstream message, R7 sent an Interest message to it. There were no more interest messages exchanged, since Client1 remained interested in multicast data packets.

  The reception of the Interest message by R5, caused this router to still consider this link to have interested downstream routers.

- Link that connects R2 to R7:

  Both routers were connected to this link by non-root interfaces. The exchange of IamUpsteam messages by both routers caused these two routers to consider that there was no downstream interest in this link. Interest/NoInterest messages were not exchanged, but both routers reached this conclusion correctly, because they both considered to be UPSTREAM regarding this tree. Since the link only connected UPSTREAM neighbors, they reached to this conclusion without having interest messages involved.

All links reached the right conclusion regarding the existence of interest from downstream neighbors. This happened due to the exchange of IamUpstream and Interest/NoInterest messages.

Like previous tests, all messages were correctly interpreted and acknowledged, which explains the transmission of interest messages destined to the Assert Winner of each link.

- **Test 11 -** Correct multicast forwarding decision after a neighbor changes its interest (a neighbor becomes not interested but there are still other neighbors interested)

  After Test 10, we closed the socket used to receive data packets at Client2. This caused the transmission of IGMP Leave, being received by R6.

  Since interface eth1 of R6 was not connected to any router and the state of group 224.12.12.12 at IGMP was in a state in which it indicated that there was no interest from hosts, this caused interface eth0 of R6, the root interface, to transmit to the switched link a NoInterest message destined to R4.

  We have observed that this message was received by R4, which answered back with an ACK message. We have verified in the logs that R4 still considered that there was interest from NOT UPSTREAM neighbors, due to this router having stored state regarding the Interest message that

was transmitted in the previous test, from router R5. Router R5 did not transmit a new interest message to R4, as expected.

We have then transmitted data packets and these were correctly received by Client1 that was still interested. These packets appear on the packet capture.

With this test, we have verified that R4 still considered that there was interest from downstream neighbors, due to the storage of interest information. The NoInterest message did not cause the switched link to be removed from the tree, due to the existence of an INTERESTED neighbor (R5).

- **Test 12 -** Correct multicast forwarding decision after a neighbor changes its interest (a neighbor becomes not interested and there is no interested neighbor)

  After Test 11, we closed the socket from the remaining interested host. This triggered the transmission of an IGMP Leave, from Client1, causing interface eth1 of R7 to stop considering to be connected to interested hosts.

  We have verified in the packet capture that R7 transmitted to R5 a NoInterest message, which was properly acknowledged. In the logs we can verify that interface eth1 of R5 stopped considering to be connected to interested downstream neighbors upon the reception of this message. This is explained by R5 being connected to only one neighbor, in that interface, that was no longer interested in receiving data packets from this tree. This caused the transmission of a NoInterest message from interface eth0 of R5, its root interface, destined to R4.

  We observed that the reception of the NoInterest message from R5, caused R4 to no longer consider that there was interest from downstream neighbors. This is explained by R4 not storing information regarding interested neighbors. At that time, R4 considered R2 and R3 to be UPSTREAM and R5 and R6 to be NOT INTERESTED. Since it was not connected to INTERESTED neighbors, this caused the Assert Winner of the switched network to prune its non-root interface. R4 by having its only non-root interface pruned, caused the transmission of a NoInterest message by its root interface, eth0, in order to prune the distribution tree.

  The message received by R1 was properly acknowledged. Then Source transmitted data packets and these were not forwarded to any link.

  In this test, we proved that the forwarding decision was made taken into account the state that was already stored, regarding all neighbors. When a router no longer connected to INTERESTED neighbors, this caused the prune of the tree.

- **Test 13 -** Correct multicast forwarding decision after a neighbor changes its interest (all neighbors were not interested and one them becomes interested)

  After Test 12, we opened the socket at Client2. This caused the IGMP implementation of Client2 to inform router R6 that there was interest from an host connected to that link, via an IGMP Report message.

  In the packet capture we can verify that router R6 transmitted an Interest message destined to R4. In the logs, after the reception of this message by R4, this router considers that there is again

downstream interested neighbors. This caused interface eth1 to belong again to the OIL of this tree. R4 then transmitted an Interest message through its root interface, eth0, in order to inform about this change to R1.

In the packet capture we have verified that R4 transmitted an Interest message to R1, which was properly acknowledged. In the logs, we have observed that this message caused R1 to consider again that there was interest from downstream neighbors regarding this tree.

By having the Source transmitting again data packets, these were correctly received by Client2, proving that the tree has been reshaped in order to attach the new member to it.

In this test we have verified that the tree reacted to the interest of the new host. This caused the exchange of Interest messages in order for data packets from Source to be forwarded to the new member.

In these tests we have confirmed that the implementation reacted correctly to all events regarding interest information. The tree was "pruned" when hosts manifested their disinterest and was "grafted" when they manifested interest in those same data packets. This behavior was caused by the exchange of IamUpstream, Interest and NoInterest messages, causing all routers to store information regarding their neighbors. Just like the previous tests, all messages were correctly acknowledged.

E.3.5 Discovery of tree information based on synchronization

We will now describe the tests regarding the exchange of tree information using Sync messages. This can happen whenever a new neighborhood relationship is formed and there were already trees set up in the network. In order to test this, we will reuse the same topology of previous tests (Figure 6.1). In these tests we will start the protocol process in all routers, then form multiple trees and finally restart a router (considered as UPSTREAM and NOT UPSTREAM for those same trees). Multiple trees will be formed in order to verify if the fragmentation of this information is being done correctly. In order to do this, we have manually configured all routers to only include information about 5 trees per Sync message, in order to test the fragmentation with a lower number of trees per message, instead of fragmenting Sync messages using MTU information.

All corresponding Netkit network configurations, tests and results regarding the synchronization of trees between routers are stored in a branch named "Test_NewProtocol_Sync_With_Trees" at our GitHub repository [34].

Before we started these tests, all routers synchronized with each other when there were no trees configured on routers. Then Source transmitted data packets to 20 groups (from 224.12.12.12 to 224.12.12.32). All routers formed these trees with the exchange of IamUpstream messages. Then we focused on what happened when an UPSTREAM and NOT UPSTREAM routers rebooted. In these tests we focused on the switched network, being R4 the UPSTREAM router that would reboot and R5 the NOT UPSTREAM router that would reboot.

- **Test 14 -** Tree formation triggered by the synchronization process with a known neighbor (reboot

of an UPSTREAM neighbor)

After all trees were set correctly, we have rebooted the router elected as Assert Winner in the switched network, R4. This was performed by removing interface eth1 from the protocol process and then re-enable this interface. Since the root interface of R4 was still being monitored by the protocol process, all trees remained stored in R4, since the root interface was connected to an UPSTREAM neighbor (R1).

After re-enabling the interface in R4, we have verified in the packet capture and in the logs that this interface formed correct neighborhood relationships with all neighbors (R2, R3, R5 and R6), by exchanging Sync messages with them. We have verified that R5 and R6 did not include any tree in their transmitted Sync messages, since these routers were connected by a root interface to the link that connected them to the rebooted router, i.e. they were not considered to be UPSTREAM. Regarding the remaining routers, R2, R3 and R4 have included all the same trees in Sync messages (from 224.12.12.12 till 224.12.12.32).

Regarding the synchronization process, all neighborhood relationships were formed with Sync messages ending at SyncSN of 5. This was expected since there were 20 trees to be exchanged, each Sync message only included 5 trees and an additional Sync message was required to terminate the synchronization process (when both routers transmitted Sync message with More flag cleared).

Regarding the sequence numbers, we have verified that these were consistent in all exchanged Sync messages between each pair of neighbors. Also, all SnapshotSNs of routers were greater than the SNs used to transmit messages prior to this synchronization, in IamUpstream and in Interest/NoInterest messages. For example, the last transmitted message by R5 was a NoInterest message regarding tree (10.1.1.100, 224.12.12.17) with SN=107 and the SnapshotSN of this same router used to synchronize state with R4 was 108.

After the Sync process has finished, we have verified that all NOT UPSTREAM routers, R5 and R6, transmitted interest messages to R4. This was expected because R4 was considered the new Assert Winner and whenever there is a change of the router responsible for forwarding data packets to a given link, routers need to manifest their interest to it. These interest messages were acknowledged by R4, with ACK messages that included all SNs that were exchanged in the synchronization process. Example, R4 and R5 have synchronized with the following SNs: R4's BootTime=1535061072, R4's SnapshotSN=1, R5's BootTime=1535060980, R5's SnapshotSN=108, which was included in all ACKs transmitted by R4 to R5 in response to R5's interest messages. Routers that were UPSTREAM did not transmit additional messages after the synchronization finished, as expected.

With this test we have verified that routers were able to exchange tree information whenever an existing neighbor restarted. All tree information was included in Sync messages and interest information was transmitted after the synchronization was successful.

- **Test 15 -** Tree formation triggered by the synchronization process with a known neighbor (reboot

of a NOT UPSTREAM neighbor)

A NOT UPSTREAM router by rebooting must send interest information regarding all trees discovered during the Sync process. In this test, we have restarted the protocol process in interface eth0 of R5. When the interface was enabled, all routers triggered a new synchronization with it.

Only R2, R3 and R4 included information regarding all existing trees, like the previous test, since these routers were the only ones connected by non-root interfaces to the link, i.e. were the only routers that were considered as UPSTREAM. Routers R5 and R6 exchanged Sync messages with no trees, because they were both connected by root interfaces to the link, i.e. they were considered as NOT UPSTREAM.

After the synchronization finished, router R5 transmitted interest messages destined to R4, which was the router elected as the Assert Winner of the link. These messages were then acknowledged by R4, with all information obtained during the synchronization (SNs), just like the previous tests.

In this test we have confirmed that only UPSTREAM routers included tree information in Sync messages. After a NOT UPSTREAM router finished synchronizing with all routers, it transmitted interest messages to the router that was elected as the Assert Winner of each exchanged tree.

E.3.6 Tree maintenance in the presence of loops

All described tests up until now have tested the maintenance of a tree in topologies that do not have loops. While the topology of Figure 6.1 might look like it has a loop, in the connection of routers R2, R7 and R5, this is not true, since the selection of the root interface can by itself break the loop, since router R2's root interface is eth0.

In order to test the maintenance of tree state in the presence of loops, we used a new topology, illustrated by Figure 6.2. In this figure, near each interface it is detailed its cost, which would be used by the unicast routing protocol to calculate the RPC to every subnet. This information would be obtained by the multicast routing protocol in order to determine the root interface and also the RPC to the source of multicast traffic.

In Figure 6.2, router R1 is the originator because it is directly connected to the Source, being all the remaining routers considered as non-originators. Interface eth0 is the root interface in routers R1, R2, R3 and R4, while interface eth1 is the root interface in R5, since these interfaces offer the lowest RPC, in each router, to the source subnet.

As we can see, routers R3, R4 and R5 can form a loop, since the root interface of R3 is connected to the non-root interface of R5, the root interface of R5 is connected to the non-root interface of R4 and the root interface of R4 is connected to the non-root interface of R3. For this reason, a router may falsely determine that a tree is ACTIVE, due to these routers informing each other that they are UPSTREAM (maintenance of state due to a loop).

In order to not maintain a tree indefinitely, a router uses the feasibility condition in order to determine a tree state. In order for a tree to become ACTIVE, the feasibility condition must hold, for this reason the RPC is used in the calculation of a tree state. In these tests we will verify if when the source

transmits data packets, all routers can determine that the tree is in ACTIVE state. Then if the source stops transmitting data packets, if all routers can correctly determine that the tree is in UNKNOWN state.

All corresponding Netkit network configurations, tests and results regarding the maintenance of tree state in the presence of network loops are stored in a branch named "Test_NewProtocol_Loop" at our GitHub repository [34].

- **Test 16 -** Tree formation in the presence of network loops

  Data packets originated by the originator must trigger the formation of the tree. All routers, eventually will be in an ACTIVE state regarding this (S,G) tree.

  We started by initiating the multicast routing protocol in all routers and then the Source started transmitting multicast data packets destined to group G (224.12.12.12). Regarding the hosts, both have reported interest in receiving multicast traffic regarding group G.

  According to the logs all routers reached an ACTIVE state regarding this tree, which confirms that the tree was correctly built.

  Regarding the exchange of messages:

  - Link R1-R2:

    Router R1 started by transmitting an IamUpstream(S,G) with RPC set to 0. Then an Interest message, from R2, was transmitted to R1. Both messages were correctly acknowledged.

    According to the logs, the reception of the IamUpstream message, by R2, causes this router to consider the tree to be in ACTIVE state.

  - Switched network:

    Router R2 started by transmitting an IamUpstream(S,G) with RPC set to 20.

    According to the logs, the reception of this message by R3 causes this router to consider the tree to be in ACTIVE state since it was received in its root interface. The reception of the same message by R5 causes this router to consider the tree to be in INACTIVE state since it was received in a non-root interface.

    Then both routers transmitted interest messages destined to R2, the Assert Winner. R3 transmitted an Interest message, since there was downstream interest regarding this tree, while R4 transmitted a NoInterest message, because it was connected to the link by a non-root interface.

    Then router R4 transmitted an IamUpstream(S,G) message with RPC set to 50. This was due to the propagation of Upstream information downwards in the tree (from R3 to R4 and then from R4 to R5), causing R4 to consider the tree to be in ACTIVE state.

    No more messages were exchanged. All described messages were correctly acknowledged.

  - Link R3-R4:

    Router R3 by considering the tree to be ACTIVE transmitted an IamUpstream(S,G) with RPC set to 30.

The reception of this message by R4, according to the logs, caused this router to consider the tree to be ACTIVE. Also in reaction to the reception of this message, R4 transmitted an Interest message destined to R3.

No more messages were exchanged and all described messages were correctly acknowledged.

- Link R4-R5:

  The transition to ACTIVE state by router R4, caused this router to transmit an IamUpstream(S,G) with RPC set to 40.

  This message was received by R5, which caused the already maintained tree to transition from INACTIVE to ACTIVE state. In reaction to this transition, router R5 transmitted an Interest message destined to R4.

  Just like in all links, no more messages were exchanged and all described messages were correctly acknowledged.

As we can verify, all routers reached an ACTIVE state regarding the (S,G) tree. This was due to the presence of UPSTREAM neighbors connected to a router's root interface and also by respecting feasibility condition.

This condition was respected in all links:

- Link R1-R2: RPC advertised by R1 (0) is lower than the RPC of R2 (20);
- Switched network: RPC advertised by R2 (20) is lower than the RPC of R3 (30). Since this RPC is better than the one advertised by R5, the latter is not checked;
- R3-R4: RPC advertised by R3 (30) is lower than the RPC of R4 (40);
- R4-R5: RPC advertised by R4 (40) is lower than the RPC of R5 (50);

As it was verified, all routers reached to the correct state regarding the (S,G) tree.

- **Test 17 -** Tree removal in the presence of network loops

  Absence of data packets must trigger the removal of a tree. All routers must eventually transition to UNKNOWN state regarding (S,G) tree. By having the Source stopped transmitting data packets, this caused the originator to eventually trigger the removal of state regarding it, in all routers. This happened around 213 seconds after the tree was initially formed.

  According to the logs, the exchange of IamNoLongerUpstream messages, caused all routers to eventually consider the tree to be in UNKNOWN state, as expected. This confirmed that all routers eventually stopped having UPSTREAM neighbors connected to any interface, causing the removal of the tree.

  The following happened in each link:

  - Link R1-R2:

The first router to transition to UNKNOWN state was R1. This transition was performed on its own, due to the expiration of its Source Active timer.

The transition of R1's tree state caused this router to transmit an IamNoLongerUpstream(S,G) by its non-root interface.

This message was received by R2, causing a transition from ACTIVE to INACTIVE state, according to the logs. This was accomplished since R2, at that point in time, was still connected to an UPSTREAM neighbor by its non-root interface (R5).

R2 eventually transitioned to UNKNOWN state, in reaction to the absence of UPSTREAM neighbors connected to its non-root interface.

Only the IamNoLongerUpstream(S,G) message was transmitted, being acknowledged by R2. No more messages were exchanged.

– Switched network:

  The initial transition from ACTIVE to INACTIVE state, at R2, caused a transmission of an IamNoLongerUpstream(S,G) message.

  The reception of this message caused all routers to consider that the Assert Winner of this link was router R5 since it was the only UPSTREAM neighbor. Router R3 by recalculating its tree state, transitioned to INACTIVE state, due to the violation of the feasibility condition, by having the RPC advertised by R5 (50) not lower than R3's own RPC (30). This caused R3 to correctly suspect about the existence of a loop, causing a transition to INACTIVE state.

  By having the Assert Winner changed, this caused new interest messages to be transmitted to R5.

  Router R3 transmitted an Interest message since it was still connected to downstream routers interested in receiving traffic from this tree.

  Router R2 transmitted a NoInterest message because its interface was non-root type.

  Eventually R5 transmitted an IamNoLongerUpstream(S,G) causing all routers in this link to no longer be connected to UPSTREAM neighbors. In reaction to this message, all routers connected to this link, transitioned the tree to UNKNOWN state.

  Like the previous link, all messages were correctly acknowledged.

– Link R3-R4:

  Router R3 by transitioning to INACTIVE state, caused a transmission of an IamNoLongerUpstream(S,G) message, which was correctly acknowledged by R4.

  According to the logs, the reception of this message by R4, caused the tree in this router to transition directly from ACTIVE to UNKNOWN state.

  No more messages were exchanged in this link.

– Link R4-R5:

  The transition to UNKNOWN state in R4 caused a transmission of an IamNoLongerUpstream(S,G) message, which was properly acknowledged by R5.

According to the logs, the reception of this message by R5 caused router R5 to transition from ACTIVE to UNKNOWN state, due to the absence of UPSTREAM neighbors connected to this router.

No more messages were exchanged.

As we have verified, the feasibility condition allowed all routers to correctly remove state of a tree, when the source stopped transmitting data packets. The use of this condition was crucial in this topology, due to the presence of a loop.

With all performed tests we have proved that the protocol worked as expected. We have verified that the protocol kept working, by maintaining trees consistently, even in the presence of router failures, reboots and network loops. The synchronization mechanism allowed a router that recovered from a failure to quickly exchange information with its neighbor routers, accelerating its recovery. All control messages were correctly built, being fully dissected by *Wireshark* with the implemented dissector.