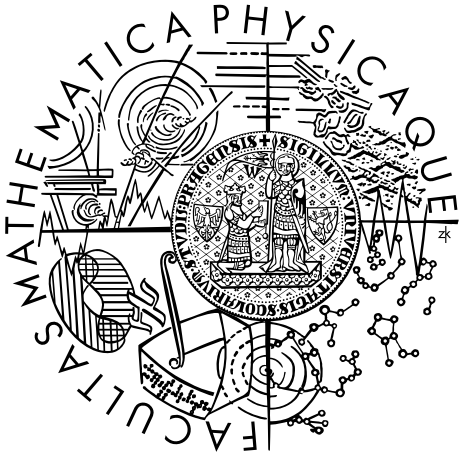
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Ondřej Hlavatý

Multicast routing

Department of Applied Mathematics

Supervisor of the bachelor thesis: Mgr. Martin Mareš, Ph.D.

Study programme: Computer Science

Study branch: IOI

Prague 2016

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Multicast routing

Author: Ondřej Hlavatý

Department: Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš, Ph.D., Department of Applied Mathematics

Abstract: We implement multicast routing in the BIRD routing daemon. At first, we review basic principles of IP multicast and different modes of the PIM multicast routing protocol. Then we outline the architecture of the BIRD and discuss its aspects related to our goal. We present our solution based on adding IGMP, bidirectional PIM, and kernel interface as three separate BIRD protocols. Finally, we describe our framework for testing network software in complex virtual networks.

Keywords: multicast routing, BIRD, PIM, networking

I would like to thank Ondřej 'Santiago' Zajíček from the CZ.NIC association for giving me directions on the topic and quickly cutting the dead ends, and my supervisor Martin Mareš for many interesting discussions.

Contents

Introduction

There are multiple ways how to use a computer network to communicate between two or more machines. In the most common technology, based on the Internet Protocol (IP), one can send a message (called a packet) from one machine, directed to another one. That is usually called a *unicast* and is by far the most well-known method. Or one can send a packet to all machines in some network – that is called a *broadcast*. However, there are applications where both of these are suboptimal. A good example can be conference telephony, games or video streaming.

When a public television company broadcasts the final match of Olympic games, it can have millions of receivers. The company does not want to send every packet in millions of copies by unicast. However, it does not want to flood every computer connected to the Internet by broadcast either. That is why *multicast* has been invented. Multicast is a general way how to transmit traffic from a group of senders to a group of receivers.

In order to deliver packets between networks, the network itself has to do some computations. One of them is routing – collaboratively finding an optimal path for a packet, for some definition of optimal. The world of unicast routing is well inhabited. There are lots of unicast routing protocols based on completely different approaches. When it comes to routing multicasts, it is much different.

Multicast itself was added as an extension to IPv4 and it requires special handling in network hardware. When just one router on the way does not support it, multicast will not work. As such it suffered from a lack of support for a long time. And because this support was often broken, it is still not widely used.

Supporting multicast on the side of the receiver is easy, and one can receive multicast in all current major operating systems. For example in Unix-like operating systems, it is a matter of a few syscalls. But routing multicast traffic is a different thing – one needs to run a routing daemon that communicates with hosts and other routers, speaks at least two different network protocols and controls the kernel's routing tables. There already are such daemons that are traditionally used: `mrouted` and `pimd`.

The purpose of this thesis is to explain how multicast works, introduce the reader to the world of multicast routing and implement one of existing multicast routing protocols. It would be possible to implement a standalone routing daemon, but its chance to catch on would be small.

There once was a student project at MFF UK, that grew to worldwide usage – the BIRD internet routing daemon. It currently supports a handful of unicast routing protocols, that can run at the same time. There are use-cases in which having closer communication between unicast and multicast routing can be an advantage. Let the primary goal of this thesis be to teach BIRD multicast routing.

The protocol chosen for this thesis is PIM-BIDIR. It is a somehow simplified protocol from the PIM family because it doesn't need to hold state information based on traffic source. That makes it less demanding on network hardware.

1. Multicast in IPv4

IP multicast was added as an additional addressing model. The first references can be found in RFC 966 [1]. This RFC defines multicast traffic with many responsibilities for routers (called "multicast agents" at that time) and it was soon obsoleted and served only as an inspiration. The latest specification can be found in RFC 1112 [2].

In local networks, multicasts are similar to broadcasts. Every packet is to be transmitted to every host on the same link, and each host can decide according to the destination address whether or not the packet should be processed or altogether dropped. In other words, you need neither a multicast routing capable router nor any special network hardware to use multicast in one network. Differences are notable on inter-network level. While broadcasts should be either forwarded to all links or filtered completely by routers, multicasts should have a more controlled behavior.

1.1 Multicast groups

The first term we need to understand is the *multicast group*. It is represented by an IP address allocated from a dedicated region `224.0.0.0/4`. This region is further divided into blocks, which have additional semantics. From routing point of view, only the Local Network Control Block (`224.0.0.0/24`) is interesting. Multicasts in this group must never be forwarded and they stay in the boundary of a local network. This block is often used by unicast routing protocols.

Host membership in a group is fully optional. Neither the number of members in a group nor the number of groups a host belongs to is limited and it can be even zero. A host does not have to belong to a group to send packets into it. Multicast packets are distributed in the same "best effort" manner as unicast ones – they can be delivered to all, some or no hosts, and in any order.

There are three levels of conformance with IP multicast. Level 0 means no conformance. Level 1 requires hosts to be able to send packets to multicast groups. This level still requires little code in the networking stack, because sending a packet to a multicast group is very similar to sending it as an unicast packet with a destination address set to the group address. While it doesn't make much sense to send unicast packets destined outside the local network prefix directly to the link, it is the expected behavior with multicast. Level 2 means fully conforming, able to send and receive multicast traffic.

1.2 Internet Group Management Protocol

As already said, membership in a multicast group is optional. By default, there is only one group that every host listens to — the `ALL-HOSTS` group with address `224.0.0.1`. There must be a way for hosts to express group membership. A protocol exists for this purpose and it is called IGMP.

The most recent version of IGMP is 3. It is described in RFC 3376 [3]. However, all versions are backwards compatible. Whenever there is a router which only knows an older version, all the newer ones stop sending any packets and they only receive and observe.

The purpose of IGMP is to exchange information about group membership between hosts and routers. Note that it is mandatory to implement IGMP in order to participate in a multicast group, but until a multicast packet crosses the boundary of a single network, it is delivered as broadcast to all hosts. As such all hosts receive it without the routers being involved.

Let us start with IGMP version 1 specified in Appendix I of RFC 1112 [2]. There are just two types of messages: *membership query* and *membership report*. Membership query is sent periodically by routers. It announces to hosts that there is an IGMPv1 capable router. This query is sent to `ALL-HOSTS` group with TTL 1 and therefore is limited to the local network. When a host receives a query, it replies with a membership report for every group it wants to join, with a random delay up to 10 seconds. No host should send reports for the `ALL-HOSTS` group, membership there is implicit and mandatory.

The report is sent to the group being reported. As noted above, the host does not have to send reports in order to listen. Filtering multicast traffic is made by other means. Sending a report declares there is someone interested in this group on this particular link. When, during the random delay, the host receives a report, it stops his own timer and remains silent. This reduces the amount of control traffic. When the host decides to leave a group, it does nothing. If it was the only host in that group, on the next query no one will report and the routers will stop forwarding.

IGMPv2 [4] comes with major changes. First, a router can send a query to a single specific group. That means it can keep a timer for every group separately, and asking for reports does not generate unnecessary traffic in every other group. Second, the maximum delay between a query and the report (previously fixed to 10 seconds) is now specified in each query and it is configurable. A network administrator can tune this variable for his needs. When the value is decreased, hosts will reply faster and routers can faster stop forwarding. Giving longer delays can improve stability on very slow or unreliable links.

The most notable change though is a new message type – *leave group*. When a host sends a report, it sets a flag. Other hosts without the flag remain silent. When the flagged host leaves a group, it sends the leave group message. Other hosts react to other hosts' leave in a way similar to a query. When a router hears the leave, it starts a timer. This timer can be stopped by receiving a report. Expiring this timer immediately changes this group state to not having any members. This way less unnecessary traffic is forwarded.

IGMPv3 comes with additional fields and more complex protocol logic. It is dedicated to source-specific multicast forwarding and includes messages to manage a set of sources for every link. Routers should then forward only traffic from these sources onto this link.

For purpose of this thesis, only IGMPv2 was implemented, as PIM-BIDIR is not a source-specific routing protocol.

1.3 Link layer

Even though multicast is specified on the network layer, it needs some level of support from the link layer. Let us explain how multicast traffic integrates with the most common combination of IP over Ethernet.

The first problem we need to solve is which MAC address to use for multicast group addresses. When sending an unicast packet, a mechanism called Address Resolution Protocol (*ARP*, RFC 826 [5]) is used to determine the MAC address. It is done simply by broadcasting questions "Who has this IP address?" and collecting answers and caching them.

As the multicast is technically a broadcast on the link layer, it would be possible to use the broadcast address `FF:FF:FF:FF:FF:FF`. But that would mean that every multicast packet on the network must be received by all hosts and examined by their operating systems, because there is no way how to filter it on the hardware level. For this purpose, every Ethernet address with the least-significant bit of the first byte set is considered to be a broadcast address and is to be flooded to all nodes on this network. Network hardware generally does not make a difference between them, so we can associate specific meaning to different MAC addresses.

For IPv4 multicast, a region of MAC addresses was allocated. This region starts at `01:00:5E:00:00:00` and ends with `01:00:5E:7F:FF:FF`. You can see that this address space is smaller (only 23 bits) than the multicast IP region (28 bits). To construct the MAC address from the multicast group address, one puts lower 23 bits of the IP address to the lower bits of the MAC address. This way, network interfaces can implement a rough filter, that can drop frames the host knows nothing about. Additional filtering is still needed because of those missing bits.

A multicast router needs to receive packets for all groups because IGMP reports are sent there. To enable this behavior, interfaces often have an "all multicast" mode, which lets all multicast frames pass through. This mode is also used when the filter table on the interface is too small to contain all the filters needed. Or we can put the interface into a so-called promiscuous mode, which passes all frames to the operating system.

1.3.1 IGMP snooping

What was written above is true according to the specifications, but not really reflecting the real state of the art. Link layer hardware such as switches often does make a difference and it uses a technique called *IGMP snooping*. This is described in RFC 4541 [6]. This RFC is only informational, because a lot of network hardware with this functionality was produced before this RFC and it does not behave in any standardized way. Network switches performing IGMP snooping understand IGMP. They listen to routers communicating with hosts and they filter multicast traffic on the link layer.

At the first glance, this seems reasonable, because it stops forwarding packets to network segments without any receivers, thus saving bandwidth. On the

other hand, this high-level functionality is expected from the network layer, and reimplementing it in the link layer brings many corner cases, which can prevent local multicast from working at all. Especially when connecting two multicast routers with a bridge, you must disable IGMP snooping, because it will filter out all traffic for which the router is not a local host – most probably everything you want to route.

For instance, Linux virtual bridges have IGMP snooping turned on by default. You can check its status and configure it through special files located in `/sys/devices/virtual/`*netdev*`/bridge/`.

1.4 Kernel implementation

The system interface for multicast is pretty complicated. At least, differences between Unixes are not big. Let us start with code to receive multicast traffic in an ordinary application. It is not much harder than opening a socket to receive unicasts:

```
int fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

struct sockaddr_in addr = { 0 };
addr.sin_family = AF_INET;
addr.sin_port = htons(42000);
addr.sin_addr.s_addr = htonl(INADDR_ANY);
bind(fd, (struct sockaddr *) &addr, sizeof(addr));

struct ip_mr mr = { 0 };
inet_aton("224.42.0.1", &mr.imr_multiaddr);
setsockopt(fd, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mr, sizeof(mr));
```

You might have expected that we would *bind* the socket to the multicast group address. That is not true, a socket can join an arbitrary number of groups. Instead, we're binding it to a wildcard address and joining the group issuing a `setsockopt` call. You can join more groups using multiple calls.

One thing to note here. When there is a socket opened which joined a group, the host itself joins the group by means of IGMP. And when the host is joined, every other multicast-enabled socket will get all packets addressed to that group. So, you still need to filter those at the transport layer (using ports for example) or by yourself in the application.

When the router daemon wants to receive all the IGMP packets for all groups, it would have to join every group possible. That is certainly not the way to go. IGMPv2 suggests using the IP option *Router Alert*. Every IP packet with this option set should be passed to all sockets that order it by this `setsockopt` call:

```
int true = 1;
setsockopt(fd, IPPROTO_IP, IP_ROUTER_ALERT, &true, sizeof(true));
```

There are multiple problems with this. Backward compatibility with IGMPv1 is broken, and standard does not force hosts to set the option. Some (e.g., old Cisco) routers do not do it.

Surprisingly, there is only one way how to get all IGMP packets coming on some interface. That is opening the IGMP control socket.

1.4.1 Multicast kernel interface

When we want to control kernel multicast routing tables, we need to open a control socket. That is a raw IP socket with a type `IPPROTO_IGMP`. Then, a `setsockopt` call is needed:

```
int igmp_sock = socket(AF_INET, SOCK_RAW, IPPROTO_IGMP);

int true = 1;
setsockopt(igmp_sock, IPPROTO_IP, MRT_INIT, &true, sizeof(true));
```

Syscalls `setsockopt` related to the multicast have option names starting with `MRT_`. These constants are defined in `<linux/mroute.h>`. Because they are essentially function calls, allow us to say "call `MRT_INIT`" with the meaning of calling `setsockopt` with the option name `MRT_INIT`.

There can be only one socket on which the call `MRT_INIT` succeeds. That socket is stored in the kernel and it is used for multiple purposes. By issuing further `setsockopt` calls, one can control multicast routing tables. What is not well documented is that this is the only socket on a host which can receive all IGMP packets. Because of this limitation, it is not possible to run more than one multicast routing daemon.

Let us have a look on how to control the multicast routing. As the first thing we need to add *virtual interfaces* (VIFs). A VIF can be built on top of a real network device, an IPIP tunnel, or it can create a special purpose device. Multicast routing table entries operate only on subsets of VIFs.

There can be only a very limited number of VIFS. This number is currently controlled by a kernel compile-time constant `MAXVIFS`, which defaults to 32. It is not easy to change it, because it breaks the kernel ABI. Every VIF has an index from the range $0 \ldots 31$. To add a VIF, we must fill `struct vifctl` and call `MRT_ADD_VIF`:

```
struct vifctl vc = { 0 };
vc.vifc_vifi = 1;
vc.vifc_flags = VIFF_USE_IFINDEX;
vc.vifc_lcl_ifindex = 2;

setsockopt(igmp_sock, IPPROTO_IP, MRT_ADD_VIF, &vc, sizeof(vc));
```

We are creating a VIF representing real network interface with index 2. This VIF will get assigned the index 1. We can use an interface address to target a real

interface instead of interface index, or specify destination address for the IPIP tunnel. If we set the `VIFF_REGISTER` flag, special device will be created.

The kernel does not provide regular routing tables. The structure used for forwarding is called Multicast Forwarding Cache (MFC). A MFC entry can be added by filling a `struct mfcctl` and calling `MRT_ADD_MFC` or `MRT_ADD_MFC_PROXY`. The difference between these two will be explained later.

```
struct mfcctl mc = { 0 };

mc.mfcc_origin = in_source;
mc.mfcc_mcastgrp = in_group;
mc.mfcc_parent = vifi;

for (int i = 0; i < MAXVIFS; i++)
    mc.mfcc_ttls[i] = ttl_threshold[i];

setsockopt(igmp_sock, IPPROTO_IP, MRT_ADD_MFC, &mc, sizeof(mc));
```

The basic meaning of an MFC entry is hidden in the field `mfcc_ttls`. It is an array of `MAXVIFS` integers, indexed by the VIF index. Whenever a field for the VIF is either 0 or 255, we call the VIF inactive for this entry, otherwise we call it active.

The entries are indexed by source and group address. When a packet for a pair (S, G) arrives on any interface, it is first checked whether it came from the interface `mfcc_parent`. If not, it is dropped immediately. Otherwise, for every active VIF i, the packet's TTL is compared to the threshold in `mfcc_ttls[`i`]`. If the packet's TTL is greater, it is copied to the VIF i.

In Linux, there are two more types of MFC entries: $(*, G)$ and $(*, *)$. These are added by a call `MFC_ADD_MFC_PROXY`, and they have a completely different meaning. I haven't found any explanation for these, my assumptions are based on studying the kernel source code [7]. The proxy entries are used to build a shared tree, possibly statically, common for all groups. First, you have to choose one uplink interface i, and add a $(*, *)$ entry with a `mfcc_parent` set to i. Whenever a packet comes to an active interface, and no $(*, G)$ MFC entry is present yet, it is forwarded to the uplink.

For a simple example, imagine your home router. It probably has three interfaces: the uplink `wan`, wired `lan` and wireless `wlan` downlinks.[1] A $(*, *)$ entry with `wan` and `lan` as active interfaces would enable you to send multicast packets from hosts on `lan`, which would then be forwarded to `wan`. Multicast packets received on `wlan` would be dropped.

Then, you can specify some $(*, G)$ entries. Having a $(*, G)$ entry with parent p, when a packet comes on an interface j, and there is a $(*, *)$ entry with both p and j active, the packet is forwarded to all interfaces active in the $(*, G)$ entry with the exception of j. Following the previous example, you would add an $(*, G)$ entry with all three interfaces active for a group that you want to be delivered to

[1] Those two downlinks are probably internally bridged in your router, but not in this example.

both of your links. Any multicast packet received on either `wan` or `lan` would be forwarded to the remaining interfaces.

Note that even when a $(*, G)$ entry is present, the kernel looks for a $(*, *)$ entry to filter the incoming interface. There can be more $(*, *)$ entries with different uplinks, but it will work only when the active interfaces are disjoint. There are no checks to prevent you from adding an entry that would never be found, so it is hard to guess which behavior is intentional and which is not.

You may wonder whether it is possible to add unicast routes for multicast addresses. It is, but their meaning is different. These routes are never used to forward packets. Instead, when you open a socket and do not specify an outgoing interface for a multicast packet (`IP_MULTICAST_IF`), the interface will be selected according to these routes. If no matching route exists, the sending will fail.

2. Protocol Independent Multicast

Unicast routing protocols try to build a tree for every destination network cooperatively. This tree is a spanning tree over some graph of networks and it is rooted at its destination. We can call it the shortest-path tree with respect to the destination. Every packet travels up the tree until it reaches its destination. Alternatively, you can imagine them building a shortest-path delivery tree for every source, rooted at the source, with traffic traveling down the tree. This is what OSPF builds.

Multicast packets have no single destination, so multicast routing protocols have to take a slightly different approach. For every multicast group, they still try to build a tree connecting all traffic sources and receivers. Multicast routing protocols often reuse the unicast routing topology to create the multicast routing information base (MRIB).

Protocol Independent Multicast (PIM) is a whole family of multicast routing protocols. Different protocols in this family are called PIM modes. The members of this family share common mechanisms, but they differ in the routing logic. There are situations where one mode can be better than the others.

Every group managed by PIM has a *Rendezvous-point* (RP). It is defined by Rendezvous-point address (RPA) assigned by local configuration. The RP is a router specially configured for this purpose. Unicast delivery tree for the RPA is then called a *RP tree*. With respect to this tree, we can define a few more terms:

- *Rendezvous-point link* (RPL) is simply a link where the RPA belongs. It is the root of the RP tree.
- *Reverse path forwarding interface* (RPF interface) with respect to an address is the next-hop interface. RPF neighbor is the next-hop router.
- *Upstream* is the direction towards the RPA, up the RP tree. *Downstream* is the opposite direction.
- *Upstream interface* is then the RPF interface for the RPA.

While a unicast packet at first hops over a number of routers, then it is sent to its destination, multicast packets are always sent on the link directly. This way every router on the link can hear it. In order to avoid duplicate forwards, on every link other than the RPL, election of a *Designated Router* (DR) takes place. This router is responsible for distributing the information taken from IGMP messages to the DR on its upstream interface.

This information is distributed using PIM *Join/Prune* messages. This mechanism is almost a reimplementation of IGMPv2 in PIM context. When there is a host on the link which joined a group G by IGMP, or any router that joined G by PIM, the DR joins G on its upstream. Let us clarify the notation for joins and prunes:

- join(S, G) joins the source-specific tree for source S and group G.
- join$(*, G)$ joins the shared RP tree for RPA(G) and group G.
- join$(*, *, \text{RPA})$ joins the shared RP tree for a RPA given, effectively joining every group with this RPA.

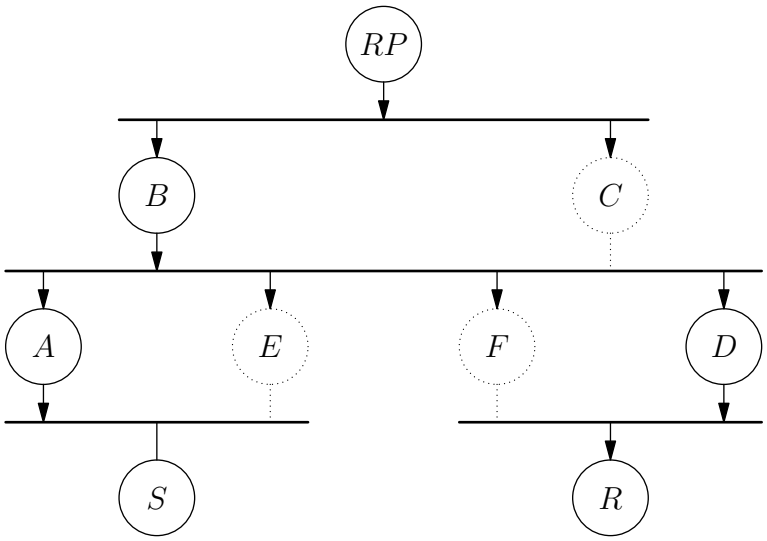


Figure 2.1: An example RP tree

Let us first explain briefly how PIM Sparse Mode works. Suppose there is already a RP tree built for a group G with some receivers that joined $(*, G)$. One such network can be seen on figure 2.1. Suddenly node S starts sending multicast packets on a link. The DR on this link (A, also called the first-hop router) takes these packets and encapsulates them into ordinary unicast packets – PIM Register messages. These are then sent to the acting RP. The encapsulation is done by the routing daemon. On BSD and Linux, the PIM daemon creates a virtual interface, where the kernel forwards unroutable multicast packets.

Incoming Register packets are decapsulated by the RP, which then temporarily distributes them downstream. Routers are responsible for picking packets coming on upstream and forwarding them down on links where they are acting as designated routers according to the shared tree for G.

In the example, traffic between S and R flows from S to A, then encapsulated in register messages over B to RP, then as multicast over B and D to R.

In the meantime, RP initiates a Join for (S, G), which creates a distribution tree for G rooted at S. After all routers in between A and RP joins the source tree, packets will be forwarded from A to RP naturally. At that time RP starts

to discard Register packets and it sends a Register-Stop message to A. Encapsulating is then stopped and the traffic will flow as usual. So, now the traffic follows the (S, G) tree from S to the RP, then the $(*, G)$ tree down to all receivers.

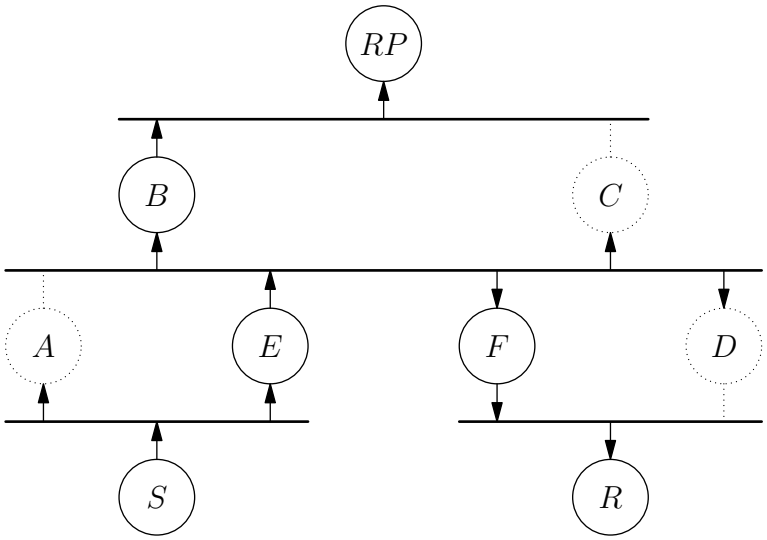


Figure 2.2: The same network with a (S, G) tree

Eventually, for a receiver R, the DR on R's link will send a (S, G) join. This join will propagate towards S until reaching first router with (S, G) state. At that time the receiver joins the source-specific tree for (S, G) and the packets use the shortest path from S to R. The situation is then similar to figure 2.2. Traffic uses the shortcut and it flows as multicast directly from S over E and F to R.

PIM Sparse Mode and PIM Dense Mode differ in their default downstream behavior. While PIM-SM requires receivers to explicitly join, PIM-DM floods everything over the RP tree by default. Branches without receivers are pruned afterwards. The difference is that PIM-SM requires state information for every group with receivers, while PIM-DM needs to hold state for every group with sources. In these modes, the RP has to be an existing router and it must be running and have enough processing power to decapsulate traffic from all sources until multicast forwarding is established. Bidirectional PIM takes a different approach.

2.1 Bidirectional PIM

Everything is made a bit simpler when we do not expect building specific trees for every source. Routers can have a lot smaller routing tables and even build them proactively. No encapsulation is then needed and processing power is spared. Let us first clarify terms in the context of PIM-BIDIR.

Designated router is called the *designated forwarder* (DF) and it takes additional responsibilities. There are differences in how the election works. The RPA can be just an arbitrary unicast-routable address and it is used just as the virtual root of the distribution tree. It does not matter whether or not any router has the RPA assigned. Every multicast packet from any source then travels up the tree, being forwarded by the DFs, and then down to branches having receivers.

2.1.1 Designated forwarder election

Every router running PIM-BIDIR knows from the MRIB what its upstream interface is, and its metric towards RP. Router cannot be a DF on its upstream link, so whenever it sends an offer, it advertises an infinite metric. Election never takes place on the RPL, because it is the upstream link for every router.

There are four types of messages in the election: Offer, Winner, Backoff, and Pass. Routers advertise their metrics as pairs of (protocol preference, distance). The state machine with all details is described in RFC 5015 [8]. The main idea of the election is that every router advertises its metric in Offer messages. When a router hears a better Offer, it remains silent for a while. If it does not, after advertising its metric three times, it claims itself a winner. Winning is announced by Winner messages, and every other router notes the sender as the winner and remembers its metric.

Two noteworthy events may happen. If the winner changes its metric to a worse value, it reannounces itself as a winner. When any other router has a better metric, it replies with an Offer immediately. Similarly, when a losing router changes its metric to be better than the winner's, it sends an Offer.

When the metric changes at one router, it is often a reaction to some router losing its link and it means that this change will propagate to other routers, changing their metrics too. In order to spare bandwidth, the current winner reacts to a better offer with a Backoff message. It instructs the offering router to wait for a while until unicast routing stabilizes. After a short period without other Offers, the winner passes the DF role in a Pass message.

In the ideal conditions, there would be exactly one DF on every link. There are situations, where no DF is elected, for example when all paths to the RPL are lost. Then every node advertises an infinite metric, and no router is elected winner. Also, when the DF fails silently, other nodes know only after few missing hello messages. For the in-between period, no router is the DF.

Much worse would be if there were two DFs on a link, as that would lead to routing loops. The election protocol is therefore biased to the safe side. For instance, an old DF sending a Pass immediately stops being a DF, even if all Pass messages were to be lost. Even when they are delivered, there is a short period without any router forwarding.

2.1.2 Packet forwarding

Suppose a source S starts transmitting packets on its link L_0. Then the currently acting DF will forward them immediately to its upstream link L_1. The DF on L_1 forwards them to its upstream, L_2, and so on. This is repeated until the packet reaches the RPL. As there is no DF on the RPL, forwarding upstream stops there.

Simultaneously, all routers listen for packets on their upstream links, and they forward packets from there to links where they are acting DFs. This way, a packet is forwarded from $L_0, L_1, \ldots$ in the reverse direction – downstream. Downstream forwarding is conditioned on having receivers active. Upstream forwarding needs to be done even when there are no receivers.

You can see that every DF can forward packets for one group upstream and downstream at the same time, explaining the name “bidirectional”.

As opposed to other PIM modes, there is no need to encapsulate packets, thus there is no different behavior on the first-hop router. Also, the RPA does not have to be assigned to any router. This means that every router behaves the same, no matter where in the tree it is, simplifying the overall protocol design.

3. BIRD Internet Routing Daemon

BIRD is a routing daemon. It does not mean that the BIRD forwards the traffic itself, it just instructs the kernel how to do so. And how does the BIRD know? It speaks with other routers using routing protocols. They share information about the network topology and decide, which way should traffic take. Details on how the information is shared and the path is calculated are up to the specific protocol used. The BIRD currently, in version 1.6.0, supports BGP, OSPF, RIP and Babel. All the information about BIRD can be found on its webpage [9].

It is usually enough to run only one routing protocol in your network, but there can be use cases when you need more protocols running simultaneously. For instance, in a border router between two network systems already using different protocols.

From the user point of view, the main units are the *protocol* and the *routing table*. Routing tables are simply tables containing routes, the same kind of tables as those in the operating system underneath the BIRD. Routing tables in BIRD are kept only in its memory and the OS knows nothing about them. You can have as many routing tables as you want, and some of them do not have to have a corresponding table in the OS.

Protocols are instances of independent modules. Every protocol is connected (typically) to a single routing table. Protocols can add, modify and delete entries in routing tables, and such changes are then announced to all connected protocols. Different instances of one protocol, or even different protocols can communicate through the table and exchange routing information in a language understood by both protocols.

3.1 Filters

Every protocol-table connection has two *filters* attached: the *import* and the *export* filter. They are named from the point of view of a table. The import filter controls routes going from the protocol to the table, the export filter controls routes going from the table to the protocol. The filter is called for every route and it decides whether the route should be accepted or rejected.

Default filters also show the most basic behavior. The import filter is by default `all`, which means every route is accepted. The export filter is `none` by default, causing every route to be rejected. This default behavior is the reason why the BIRD does not magically work out of the box, because running protocol instance have no knowledge of the network topology, thus no information to share.

Before accepting the route, filters may also modify the route's attributes. This mechanism is targeted to, but not limited to, setting route's metrics while transferring the route between different protocols. But you can do almost anything with the route, including, e.g., changing the next hop.

3.2 Non-routing protocols

Routing protocols often have some method to detect a neighbor failure using some form of hello messages. But these often have long reaction period in the order of tens of seconds. A protocol named BFD tries to provide faster, low overhead failure detection. It serves as an adviser to other protocols.

Another protocol implemented in the BIRD is RAdv. It is used by IPv6 routers to advertise an available network prefix allowing hosts to autoconfigure.

3.3 Special protocols

Nothing from what we have discussed communicates with the kernel, yet protocols and tables are all what the BIRD is made of. The synchronization of BIRD's tables with the kernel routing table is made through a special protocol called, surprisingly, `kernel`. The kernel protocol behaves like an usual protocol, and it is connected to one table through some filters. Every route it receives propagates to the kernel, and every route from the kernel is announced back. On systems where there can be more than one routing table (e.g., Linux), you have to run a separate instance of the kernel protocol for each. The scanning is done periodically. Do not forget to set the export filter for kernel protocol to something more sensible than `none`.

Another necessary protocol is the `device` protocol. It does not send nor receive route updates, but periodically scans available network interfaces. Unless you have a very specific setup, this protocol is needed because other protocols need to know what devices do they have under control.

Three more special protocols are implemented. You can specify some routes in the configuration, then the `static` protocol will add them to the BIRD's tables. The `pipe` connects two tables and it copies routes between them. Of course, this is useful only with filters, otherwise you could have just used only one table. Finally, the `direct` protocol generates routes that directly belong to some interface. That is useful when your operating system cannot track route origin and it would allow the BIRD to permanently overwrite these routes. This is the case for BSD, on Linux the BIRD cannot change routes from other sources.

3.4 IP agnosticity

When BIRD was initially designed, its creators decided that the it will be able to route both IPv4 and IPv6, but not at the same time. The choice is made during compilation and a bunch of macro definitions will compile the BIRD either for IPv4, or for IPv6. If you have a dual-stack network, you have to run both BIRDs at the same time and configure them separately.

This choice is good because of performance and memory reasons. Internal structures can be optimized for the given address length, which means they can be nearly four times faster and lighter for IPv4. However, nowadays it is usual to have tunnels mixing both internet protocols together and a routing daemon

must be able to route IPv6 traffic in a tunnel over IPv4 network. That is one of the reasons the BIRD team have chosen to make a big change and generalize internal structures to run both versions at the same time.

My work is based on this dual-stack integrated version, where protocol-table connections are generalized and called `channels`. A protocol can have any number of channels of any type, thus receive route updates from more tables. PIM heavily uses this, because it is connected to three tables all the time.

Having two IP versions in one daemon introduces differences between routing tables. Every routing table has a type attached, and channels can usually work only with specific table types. The default routing table for the type `ipv4` is the table `master4`. The notation will be important in the configuration.

3.5 Configuration

Because of channels and IP agnosticity, configuration differs between the latest released version of BIRD and the version my work is based on. As we would like to give configuration samples in the following chapters, it wouldn't make much sense to give them for a different version. But these configuration files will not work in the officially released BIRD 1.6.0.

Let us have a look at a simple configuration file:

```
protocol kernel {
  scan time 20;
  ipv4 { export all; };
}

protocol device {
  scan time 10;
}

protocol rip {
  ipv4 { export all; import all; };
  interface "*";
}
```

This is the most basic configuration actually doing something. We can see the definition of a kernel protocol, scanning kernel routing table every 20 seconds for new routes. The only channel `ipv4` of the kernel protocol have the export filter set to `all`, exporting every route from BIRD's routing table `master4` to the kernel. Next to it is the device protocol, set to scan every 10 seconds. Then we run the RIP protocol, filtering no route out. That means RIP will get all routes read from kernel, exchange them with its neighbors, and put new routes into the main table. It will communicate on all available interfaces.

Let's try something more interesting, omitting the kernel and device protocol:

```
protocol rip {
  debug packets;
  interface "eth*";
  ipv4 {
    import where rip_metric < 10;
    export all;
   };
}

protocol ospf {
  area 0 {
    interface "wlan*";
  };
  ipv4 { import all; export all; };
}

protocol static {
  route 10.42.0.0/16 via 10.0.0.42;
  route 10.6.6.6 blackhole;
  route 192.168.1.0/24 recursive 192.168.0.42;
}
```

Using natural language, these directives are easily readable and self-explanatory. There are multiple levels of messages in the log, and especially the level `trace` has categories; `debug packets` will turn on debug messages about packets for the RIP protocol. The RIP will run only on interfaces with name starting with "eth", and its routes will be taken into account only when having RIP metric under 10.

OSPF configuration defines the backbone area, spanning over the device's wlan interfaces. The static protocol defines some routes, we can see an usual route having next hop, a blackhole route, which means every packet choosing this route will get silently dropped, and a recursive route, where the next hop is the same as the next hop for IP `192.168.0.42`.

3.6 The internals

The overall architecture can be called event-driven. When you implement a new protocol, you register hooks to be called when there is an interface status change, route update, protocol is about to be shut down, reconfigured, and so on. The most interesting hooks can be registered on sockets and timers.

BIRD sockets are a thin layer over traditional Unix sockets. They try to hide incompatibilities between different Unixes by providing system-dependent implementation. But the main benefit of them is simplified receiving and sending packets using hooks.

What might be surprising, BIRD is single-threaded. It has only one IO loop that dispatches these hooks. That means you do not have to mess with locking in the common code, but it comes with a few drawbacks. In the loop, there is

a `select` syscall waiting on all sockets. Whenever there are some data waiting, BIRD calls a `rx_hook` on that socket.

Another very common usage pattern in network protocols are timers. You often need to send a message every ten seconds, or let something timeout. It would be very inefficient to have three threads sleeping for every network prefix, so BIRD introduces internal timers. Timers have one hook and a time when it should be called. BIRD does not promise an exact time, but it ensures the hook would not be called sooner. The timeout for the `select` in the loop is set to the time until the first timer should be fired.

When doing some computations that can take a long time, for example going through a complete routing table, you cannot do it in single hook as there can be other, possibly time sensitive, protocols waiting for their time. A mechanism called events was made for this purpose. An event can be scheduled, and its hook will run at the end of current IO loop cycle. It is recommended to split such long-running pieces of code into several event calls, simulating multiple threads running with less overhead than actually having them.

Because there are computations in the IO loop that can take longer time, it is not possible to have precise timers. For that reason, internal timers had the precision of one second. When a timer is delayed by two seconds, it does not matter much when it was set to ten seconds. But the same delay introduced to a few-millisecond timer can be critical.

Traversing over a routing table spread over more iterations of the main IO loop brings several problems. When any event is called in-between, you have to deal with added entries, removed entries (especially the current one being removed), and complete rehashes of the table. Routing table is stored in clever hash table called FIB, *Forwarding Information Base*. This table follows a special property making elements keep their relative order when the table is rehashed. Furthermore, every node keeps a list of iterators, and when the node is removed, all its iterators are moved to the next element.

3.6.1 Resources

Common software problem, dangerous especially for daemons, is memory leakage. When writing network software, one often needs to allocate temporary structures for holding additional information tied to an interface, neighbor or a network prefix. It can happen and often happens that you lose all handles to allocated memory and you cannot deallocate it. This forgotten memory accumulates over time and memory usage of your daemon grows. When it overgrows the limited memory in embedded devices, your daemon is killed.

BIRD tries to fight (not only) this problem with dedicated mechanisms for memory allocations. There are several resource pools from which resources are allocated. You can then deallocate either the resources one by one, or the whole pool at once. For example, every protocol instance has its own pool, which is freed when the instance is shut down.

This mechanism has more advantages. Every resource type can have its destructor specified, so it can remove itself from lists, free related memory and so.

To give an example, timers has their own resource type, and when a timer is freed, it removes itself from the list of upcoming timers. Or, when a socket is closed, it deallocates its buffers.

Resource pools offer multiple memory allocation strategies, making them more efficient. BIRD offers memory blocks (the usual strategy similar to malloc/free), linear memory pool and Slabs. Linear memory pools are handy when creating a temporary structure, which will be freed as a whole after using. Slabs are efficient in managing a lot of fixed-size blocks, e.g., the FIB nodes.

3.6.2 System dependent parts

A routing daemon, whether we want or not, does have to communicate tightly with the kernel. The BIRD is written with all Unix-like operating systems in mind, and it was originally developed at least for Linux and BSD. However, the development of last years is strongly focused on Linux only.

To avoid having a lot of compile-time checks and conditionally compiled blocks, BIRD has a source code subtree, which is linked according to the target operating system. This subtree is called `sysdep`. There are parts which are special for Linux or BSD, but also parts common for all Unixes. The creators of BIRD wanted to make porting to different operating systems as simple as possible.

When you look in the sysdep code, you will find for example functions to read system time, to communicate with the kernel or the socket abstraction layer, but also the whole implementation of the main IO loop. If you look into the forbidden lands of the kernel protocol, you will find arcane magic.

3.7 The user interface

To communicate with the running daemon, the BIRD opens a control socket, and offers a well-defined protocol which you can use to control the running daemon. To make it actually usable, the BIRD is provided with a shell-like utility, `birdc`. You can run commands to examine the routing tables, dump protocol information or to change the configuration on-the-fly by disabling and enabling protocol instances. The client needs to open a connection to the control socket, and therefore you usually have to run it with the root privileges.

When you update the configuration, it is possible to reload it to the running daemon without losing routing tables in the memory. BIRD tries to avoid restarting as much as possible. There are mechanisms how to reconfigure a running protocol, how to restart it without losing its routes, or how to restart only those protocols that changed too much.

4. Multicast in the BIRD

When we designed the multicast routing features into BIRD, we tried to respect local customs. We have separated the synchronization of routing tables with the kernel, which is system-dependent, into a protocol named *mkernel*. Another standalone part is the IGMP protocol, which is independent on the particular multicast routing protocol used. The PIM protocol itself could be further divided into parts, but it would require the network administrator to understand the internal structure, making it significantly harder to configure.

We decided, for now, to support IPv4 only. Supporting IPv6 would require adding another two protocols, because the kernel interface is separated and the IGMP is IPv4-specific. For IPv6 there is a protocol called Multicast Listener Discovery (MLD), and though they share common principles with IGMP, they differ not only in the address length. The PIM protocol is designed with IPv6 in mind.

4.1 The IGMP

Our implementation of IGMP is straightforward. It resides in `/proto/igmp/` and it is split into three files: `igmp.h`, `igmp.c` and `packets.c`.

In the last, one, we can find a packet processing code. As there are only two packet types in IGMPv2, this code looks simpler than similar code in other protocols. Because there can be only one IGMP control socket which receives all the packets, we have decided to open the control socket in the mkernel protocol and emulate packet receiving for IGMP instances.

Because IGMP needs to send packets themselves, we decided to dedicate a special type of BIRD sockets for them – `SK_IGMP`. When a protocols opens a socket of this type, an actual raw IP socket with protocol `IPPROTO_IGMP` is opened under the hood. But this socket is used for sending packets only, its read buffer is not allocated and the socket itself is not added to the main IO loop. Instead, depending on whether it is specific to one interface or not, it is added ("registered") to one of the internal lists in the mkernel protocol. Mkernel has its IGMP control socket opened, and it calls `rx_hook` for every registered socket. It temporarily substitutes these protocols' read buffer with the shared one to avoid copying the packet data, so read hooks should not modify the packet.

As in IGMPv2, every packet consists of a common header only, a simple `struct igmp_pkt` is fitted onto the received data and read. Similarly, when sending a packet, this structure is filled and sent as a sequence of bytes.

The IGMP instance keeps a state structure `struct igmp_iface` for every interface known. It contains a socket for sending multicast packets there and holds the state machine with all the timers. This state machine ensures there is only one querying router on every interface. Next, there is a hash table of joined groups.

There are four states for every group according to the standard. One of them, "no members" is implicit by not having any state information for this group.

An existing group state for a group with no members may exist only temporarily. In the other states, `struct igmp_grp` exists for such group with all the timers needed. Internal state changes do not affect routing.

The IGMP produces "multicast requests" – tuples of (S, G, I) with the meaning that there is a host which joined (S, G) on interface I. These requests are stored in a table common for multiple protocols. This way protocols communicate, and it allows us to provide a PIM implementation independent on the IGMP, and vice versa.

These tables are just a special kind of routing tables. This matches the paradigm of the BIRD made of routing tables and protocols, and it comes with surprising benefits. For this purpose, internal routing tables were generalized to use `struct net_addr` as a key. This structure can be inherited and its methods adapted – as in case of `net_addr_mreq`. While the most used `net_addr` stores a network prefix (hence its name), `net_addr_mreq` stores group address and interface. When a "route" exists for (G, I), then the routing protocol should deliver the traffic for G to the interface I. These multicast requests do not have their kernel counterpart.

One of the benefits of having this information in routing table is that you can reuse BIRD's powerful mechanisms to mangle routes. You can for example set these request statically by adding a static route. Or you can exclude one specific group using route filters. You can have more request tables and run independent instances of the IGMP connected to different multicast routing protocols. Or, when it does make sense, you can even use the BIRD to manage one group with different routing protocols on different interfaces.

4.2 The PIM

As already said, we have implemented only one multicast routing protocol to the BIRD, that is PIM in its bidirectional variant. This variant is simpler than the others, but still needs to hold a lot of state in a way different from unicast routing protocols. Because of that, a great portion of PIM code only manages its internal state. Despite implementing only this variant, we have thought of other PIM variants and the internal structure is as future-proof as possible.

You can find the PIM implementation in `/proto/pim/`. It is split into four parts: `pim.c`, `pim.h`, `packets.c` and `df.c`. Let us first look at the header file. There are lots of structure definitions for holding the internal state.

`pim_proto`
: This is the instance of the protocol.

`pim_iface`
: For every interface PIM knows, we need to open a socket. Information needed to send hello packets is stored here. These are stored in a list `pim_proto->ifaces`. Even the kernel keeps all interfaces in a list, so this should be safe and fast enough. Moreover, in every socket there is a pointer to this structure, so we do not need to traverse the list on every packet received.

`pim_rp`
: Keeps the shared tree information. All instances of this structure are stored in `pim_proto->rp_states`, which is a hash table indexed by the RPA. The protocol design assumes a lot of groups share a RP, so there should be only a few of these. There is no way how to abandon a RP tree, so this state is never deleted.

  RP state caches its upstream and metric towards RP, so we do not have to search in the MRIB often.

`pim_rp_iface`
: For every shared RP tree and every interface there is a DF elected, so we keep an instance for every member of the cartesian product of `pim_iface` and `pim_rp`. All of them are stored in a hash table in `pim_proto`, because the DF election is bursty and fast, so we want to find these state structures quickly.

`pim_grp`, `pim_grp_iface`
: The PIM join/prune mechanism maintains a group state over the RP tree. We need to remember which downstream branches have joined, and keep a timer to join upstream. We keep the upstream state (`pim_grp`) only when we're joined upstream, and the downstream state (`pim_grp_iface`) only when there's a joined router downstream.

  The `pim_grp` instances are kept in a hash table in `pim_proto`, but the downstream states are kept in a list inside the upstream state. Iterating over this list must be cheap anyway, because we need to forward every single packet to all these interfaces.

`pim_neigh`
: We maintain a direct-neighbor cache. For every neighbor, we need to remember the flags it announced in its hello message, and its generation identifier.

Other structures are either too trivial to explain or just temporary to share information among PIM compile units.

The PIM itself is split into three logical units, which share a common state. The join/prune mechanism, the DF election mechanism and the forwarding logic. You can see how the units communicate on the figure 4.1.

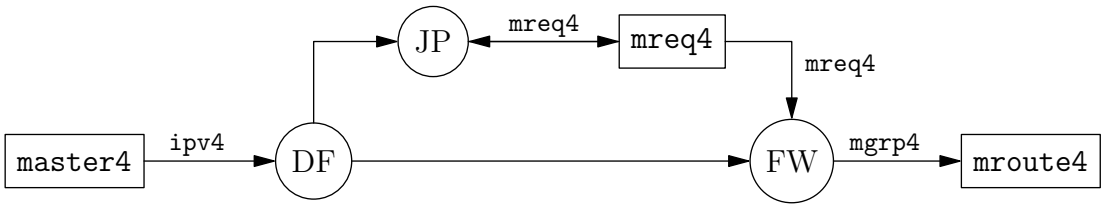


Figure 4.1: A scheme of the PIM protocol

4.2.1 The join/prune mechanism

When the routers distribute traffic over the RP tree, they need to know which branches do have receivers for a group G. The join/prune mechanism is connected to the multicast requests table, and uses the channel in both ways. Everything starts, when a hosts joins a group by sending an IGMP membership report. The first-hop router notes it, and adds an entry to the request table. From there it is propagated to the join/prune mechanism of an active PIM instance.

As it is the first join for a group G, the `pim_grp` structure for G is created. Existence of the upstream state results in sending a Join to the *upstream neighbor* – the Designated Forwarder on the upstream interface. When the upstream neighbor receives a join on downstream, it creates the downstream state, where it keeps an expiry timer. Then it adds an entry to the multicast request table. The entry is then, being a route, announced back to the PIM. Because the PIM does not make a difference between different multicast requests, it is essentially the same situation as on the downstream router.

This way the mechanism is split into two layers – downstream, receiving PIM joins, and upstream, propagating any joins to the upstream neighbor. Of course, looping the simple request through the BIRD's routing tables introduces a small delay, but you can alter its behavior using filters.

The main domain of the BIRD is its ability to run multiple routing protocols at the same time. While routing an unicast packet through multiple autonomous systems with different routing protocols is possible, well defined and widely used, it is not so easy with multicast. One way how to make it work is through this request sharing.

An interesting implementation detail is that BIRD can announce routes in two complementary ways. Either you can let it announce any route (`RA_ANY`), then it will call the `rt_notify` hook with the old and new route version, or you can ask for optimal route updates only (`RA_OPTIMAL`), which will announce the old and new optimal route for a network prefix. Having the optimal mode on the request table will result in only three possible combinations of the old and new route. When the old one is `NULL`, and a new one exists, this is a new request and we need to create the downstream state. When the old one exists and the new one is `NULL`, the last request for this group and interface vanished, and it is safe to stop forwarding. When both the old and new route exists, we do not have to do anything, because we already consider this interface joined.

4.2.2 The DF election mechanism

On every interface and for every RPA known, an election takes place. The election is based on the routers shouting their metric, and whoever shouts the most is the winner. The mechanism is rather complicated to avoid routing loops. Because of it, a state machine with a lot of transition edges is defined. The behavior of this machine is separated from the main code into the file `/proto/pim/df.c`.

The majority of transition edges are based on an incoming packet. Every packet has a metric attached, there are four types of packets and four states of the state machine. To handle those 32 cases, there is a huge `switch` statement,

which uses macros to combine all three variables into one number, and a few other macros to keep the switch as small as possible while retaining readability.

Whenever a packet comes, there is a quick lookup in the hash table to find the state structure `pim_rp_iface`. Then the `pim_df_message` method is called. Responses are sent immediately.

This part required a notable change in BIRD's IO loop. Previously, the BIRD's timers had a precision of one second only. The reasons for this were explained in the section 3.6. The DF election must be fast, its messages are spaced by 100 ms by default, so the BIRD timers were not applicable. There already is a protocol which is more time sensitive, the BFD. It solves this problem by having a separate IO loop. But BFD does not alter the routing tables, and BIRD internals are not thread-safe, so having a separate loop for PIM is not an option.

Because complete rewrite of the main loop is planned, we have decided only to temporarily hack the existing timers. It had to be done carefully not to break existing protocols. The solution was to add another field to the timer structure, keeping a `btime`. That is an already used BIRD time abstraction, with defined constants for most used time fractions. Because existing protocols sometimes use the original field, we have kept it there. Then, we've added new methods to work with these timers with a microsecond precision: `tm_start_btime`, `tm_remains_btime` and `tm_start_min`/`max_btime`. The old respective ones were changed to convert seconds to `btime` and call the new methods. This way all the timers have the `btime` precision, but the old API was kept the same, not breaking anything.

4.2.3 The forwarding logic

Having the RP tree built and knowing which branches we want to forward traffic to, the forwarding logic is pretty straightforward. But first, let us explain how we represent multicast routes. As in case of the multicast request, multicast routes are another special kind of "networks", and they have their routing tables. The network type is called `mgrp4` or `mgrp6`, depending on the IP version.

Multicast routes are slightly different from the unicast ones. In unicast routing, the most common route has a next hop router, only the last-hop routers have a direct device route. As multicasts are always sent directly to link, no such thing as a next-hop route exists. Furthermore, you often want to send a packet in multiple copies to several interfaces. We have discussed a bit how to control multicast routing in Linux in section 1.4.1, but we want to keep the internal tables system-independent.

Every interface, which you want to use for multicast routing, must be assigned an index. Those reflect the VIF indices, but the implementation does not depend on them and can be changed any time. In order to assign an index, you must enable the interface for the mkernel protocol. If you don't, packets won't be forwarded, but you will not be warned. This is the expected behavior, because you probably do not want to forward traffic there, when you have explicitly disabled the interface for the mkernel.

From the theoretical point of view, the most generic representation for a multicast routing table could be a mapping of $(S, G, I) \rightarrow O$, having a set of outgoing interfaces O for every source S, group G and input interface I. This is however too generic and space-consuming, so we have made it simpler. First, there is no need to store a source yet, because we do not support source-specific multicast routing. Without the restriction to one source, there are often more incoming interfaces, for which the outgoing set is almost identical (typically having a common superset, from which the incoming interface is removed). Such routes can be merged, keeping a set of incoming interfaces. The outgoing set is often a subset of the incoming set. There could possibly be two disjoint sets of input interfaces, for which the output interfaces are different, but that is something we do not want to allow.

The final representation of multicast routing table we decided to implement is a mapping of $G \rightarrow (\text{iifs}, \text{oifs})$, having a set of feasible incoming interfaces and set of outgoing ones for every group. When a packet comes on interface $I \in \text{iifs}$ for a group G, forward it to every $O \in \text{oifs}, O \neq I$. These sets are represented as bitmaps in the route. The bits are indexed by the interface index assigned by mkernel.

Having a multicast routing table, the forwarding logic of PIM is only a simple filtering aggregator of multicast requests. Whenever a routing change occurs, it constructs a new route for G, having all interfaces where the router is acting DF in incoming interfaces, and every interface for which a multicast requests exists and the router is an acting DF in outgoing interfaces. Such a route is then announced to the routing table. When the group state is removed, a `NULL` route is announced.

4.3 The mkernel protocol

The protocol itself is hidden in the system-dependent source subtree, in the file `/sysdep/unix/mkrt.c` and corresponding header file. We have tried to keep all inconveniences in the kernel design there. Because we need to reflect how kernel thinks of multicast routing, we were restricted in designing this protocol.

First of all, you cannot run more than one instance of this protocol. This is subject to change, because it is possible to have more than one kernel multicast routing table on some operating systems. Even when multiple tables are involved, there must be exactly one control socket opened. One task of the mkernel is to internally redistribute IGMP packets received there.

Another thing mkernel does is assigning the interface multicast indices. It uses a simple algorithm, keeping a bitmap of used indices, and assigning the first one available. Using this identifier, a VIF with the same index is created. Obviously, to assign an index, you have to pass this interface to the protocol. Because zero is a valid index too, when the index is assigned to a `struct iface`, a flag `IF_VIFI_ASSIGNED` is added, too.

The main purpose of this protocol is the synchronization of the routing table. If you look at the overview of kernel routes in the first chapter, you won't find a type similar to those we're using. To make matters worse, none of them is

usable for our purposes. (S, G) routes obviously work only when we know the source, and $(*, G)$ routes cannot limit the incoming interfaces per group. Ignoring this limitation would cause routing loops even in common scenarios.

When the kernel does not find a matching MFC entry, it calls the userspace daemon to resolve the situation. In the PIM-SM mode, this often results in encapsulating the packet, but not in PIM-BIDIR. The so-called *upcall* consists of sending an artificial IGMP packet with a packet type unused in the IGMP standard. The constants are defined in `<linux/mroute.h>`. Three upcalls can be found in the linux kernel source code:

`IGMPMSG_NOCACHE`	There is an MFC miss. This is the only upcall we need to listen to.
`IGMPMSG_WRONGVIF`	An MFC entry exists, but different incoming interface was expected. This upcall is sent only in one specific situation when switching the RP and the source-specific tree in PIM-SM. In other situations, this upcall is suppressed.
`IGMPMSG_WHOLEPKT`	An MFC miss occurred, but the kernel is configured to pass us the whole incoming packet for encapsulation.

When an MFC miss occurs, the packet is enqueued in the kernel. If the MFC entry is added fast enough, it will be resolved and forwarded afterwards. Because these packets are ordinary IGMP packets, but it does not make sense to process them in the IGMP protocol, we are forced to partially parse the IGMP packet in the mkernel protocol. If it is one of these upcalls, resolve it now, otherwise pass it to the IGMP protocol(s).

While resolving an MFC entry, we look up the route for the group, check if the incoming interface is in the iifs, and add a route specifically for this source. Because (S, G) routes use a RPF check, we can control both iifs and oifs. The main problem in this method is reflecting changes in routing. When a route changes in BIRD, we are announced the group, but we need to change all the MFC entries for specific sources we have added. That means we have to remember what sources we've added and delete everything on every route change.

Also, without exposing the group to RPA mapping, we cannot route groups without an upstream state in PIM. This results in an inconvenient but reasonable restriction – every source must join the group it sends packets to.

5. Multicast in the BIRD – configuration

The multicast routing features are part of the BIRD, and as such they are configurable through the BIRD configuration mechanism. We also tried to make the configuration fit well into the existing scheme. Configuration option names are chosen to reflect the variable names used in the standards.

To use the multicast routing features in the BIRD, you must enable at least two protocols: PIM and mkernel. When any of the router's links have hosts connected, you probably want to enable also the IGMP. Though you have to include these protocols in your configuration, all the protocol-specific options have reasonable default values suggested by the protocols standards, and you do not need to include them at all. This chapter covers the complete configuration of all three protocols.

5.1 The IGMP

The IGMP listens for hosts' multicast reports, and fills the request table in the BIRD. The default values are well suited for the majority of use cases, but you still need to include the IGMP configuration section to make IGMP run. Do it so, whenever a router is connected to a link with local hosts. So, the most common configuration will be the following:

```
protocol igmp { }
```

All the configurable variables have the same names as in the IGMP standard, RFC 2236 [4]. Intervals are specified by time expressions to avoid unit problems. Have a look at a configuration example:

```
protocol igmp {
  interface "eth*" {
    robustness 2;
    query interval 125 s;
    query response interval 10 s;
    startup query count 2;
    startup query interval 31 s;
    last member query count 2;
    last member query interval 1 s;
  };
  mreq4 { import all; };
}
```

All configuration variables are interface-specific. Let us go through their meaning:

`robustness` r (integer)
: Loss of less than r packets cannot cause the protocol to have inconsistent

states. Increasing r will increase stability on lossy links, but groups will be left more slowly.

Default value: 2

`query interval` t (time)
: A general query will be sent once per t. Decreasing t will cause more control traffic, but faster convergence on lossy links.

Default value: 125 seconds

`query response interval` t (time)
: Hosts have to send their report until t passes. Larger values spread the burst of control traffic after a general query.

Default value: 10 seconds

`startup query count` c (integer)
`startup query interval` t (time)
: When starting up, a loss of a query is critical, because it causes undetected delay before starting to forward traffic. First c queries are sent in shorter interval t.

Default values: `robustness` messages, $\frac{1}{4}$ the `query interval`

`last member query count` c (integer)
`last member query interval` t (time)
: After receiving a leave, c group-specific queries are sent to the group, once per t. After time $c \cdot t$, the router assumes there are no group members. Decreasing these will cause groups to be left faster.

Default values: `robustness` messages, 1 second

There is also an implicit channel configuration for the multicast request table. An IGMP instance can have only one channel of type `mreq4` configured. The defaults filter settings on a channel are well suited here, so you can omit the the channel configuration completely. The IGMP rejects all route updates.

5.2 The PIM protocol

PIM tries to satisfy the multicast requests by joining the interface into a distribution tree. To use PIM to manage a group, you must configure its RP address. You can specify a common RPA for a range of groups using a CIDR notation:

```
protocol pim {
  group 224.42.0.0/16 {
    rpa 10.0.0.1;
  };
}
```

In order to function properly, all routers must be configured to use the same RPA for a group. When choosing a RPA, bear in mind that every multicast packet

will have to be transferred to the link where RPA belongs. If you have only one fixed source, the best choice for RPA is the source address. Otherwise you can choose some address belonging to the backbone of your network.

Other options for PIM are for fine-tuning the constants for specific interfaces (slow or lossy). An example follows:

```
protocol pim {
  interface "*" {
    hello period 30 s;
    hello delay 5 s;
    hello holdtime 105 s;

    election robustness 3;

    override interval 3 s;
    joinprune period 60 s;
    joinprune holdtime 60 s;
  };
}
```

`hello period` t (time)
: A PIM hello packet is sent once per t. PIM does not use BFD, and it depends on hello packets to detect a neighbor failure.

  The default value is 30 seconds.

`hello holdtime` t (time)
: This value is announced to neighbors. After a period of t without receiving a hello packet from this router, neighbors assume this router is dead.

  If omitted, its value is calculated as $3.5 \cdot$ hello period.

`hello delay` t (time)
: A maximum value for the random delay before sending a first hello. It prevents bursts of control traffic when the link becomes up.

  The default value is 5 seconds.

`election robustness` r (integer)
: The minimum number of packets that must be lost to break the DF election mechanism. Every message is transmitted r times. The larger the r, the slower the election will be, but it will be more prone to packet loss.

  The default value is 3.

  The delays between election messages are not configurable, their values are specified in the standard.

`override interval` t (time)
: When a prune is received, other routers on a link interested in a traffic for a group G must send their overriding join during t. After a delay of t, this router assumes there are no other routers it and will stop forwarding G to this link.

Note that this value can be changed in the runtime through options in hello messages.

The default is 3 seconds.

`joinprune period` t (time)
A Join message for every group will be sent once per t, unless an other router is sending them.

The default value is 60 seconds.

`joinprune holdtime` t (time)
This value is transmitted to the upstream neighbor. After a period of t, the neighbor assumes we have pruned the group, but the Prune packet was lost.

If not present, the value is set to $3.5 \cdot$ `joinprune period`.

The channel configuration of PIM is more complicated than of other protocols, because PIM is connected to three routing tables of different types. The basic setting (not the default though):

```
protocol pim {
  ipv4  { import none; export all;  };
  mreq4 { import all;  export all;  };
  mgrp4 { import all;  export none; };
}
```

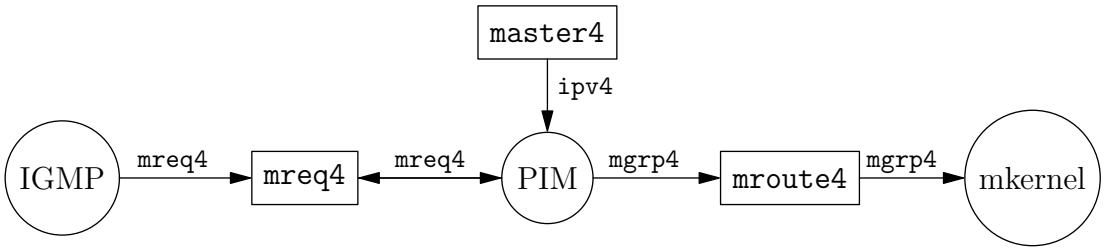


Figure 5.1: A scheme of channels connecting routing tables with protocols

The multicast route channel (`mgrp4`) has the default setting, causing multicast routes to be propagated into kernel, and no routes to be announced back. It does not make much sense to learn multicast routes from other protocols.

If you leave the default `export none` setting for the multicast request channel `mreq4`, PIM will only forward traffic upstream. Reasons for the decision to loop PIM internal requests through the multicast requests table can be found in section 4.2.1.

The last channel, `ipv4`, is providing the network topology, i.e. the MRIB, to the routing protocol. The topology is read-only for PIM, no routes are imported back to the routing table. If you do not supply any topology to PIM (for example by leaving the export filter on its default value `none`), PIM will not know its upstream interface and therefore cannot join the shared tree. You can see the scheme of all the channels on the figure 5.1.

By changing the MRIB channel, you can unleash the full power of BIRD. To give an example, imagine you are running a system of networks running BGP. As a metric, you are using the link reliability, having the BIRD select the most reliable route as the best one. Now you would like to use multicast for video conferences. For transferring video, there are more important properties of the connection than reliability – bandwidth, low and stable latency. In BIRD, this is very easy. Just create an internal routing table, run another routing protocol (either a different one, or BGP with different settings) and supply its topology to PIM as the MRIB. Unicast is still routed the same, but multicasts will use a different path.

When the PIM is electing the designated forwarder, it advertises the IGP metric, and the one with the best metric becomes the forwarder. You can alter the metric exported to PIM to any value you want, so you do not even have to run two separate routing protocols.

5.3 The mkernel protocol

There is nothing you can configure in the mkernel protocol directly. You only need to include the configuration section to enable the multicast routing features.

```
protocol mkernel {
  mgrp4 { export all; };
}
```

You must still configure the only channel mkernel uses. By default, it is connected to the routing table `mroute4`. As mkernel never announces any route, the default filter settings are pretty much useless. You can see the most basic configuration in the last example.

5.4 Dump output

You can use the BIRD client to examine protocol behavior. Let us show some actual output and walk through its meaning. This output was gathered by connecting to a running BIRD with the configuration presented in this chapter. The only difference was running an OSPF protocol to map the network topology. Unrelated protocols were omitted from the output.

```
birdc> dump protocols
  protocol mkernel1 state UP
    TABLE mroute4
    VIFs as in bitmaps:
      veth-br1 veth-r1 d1
    (S,G) entries in MFC in kernel:
      (10.3.3.1, 224.42.42.42, veth-br1) -> 111 110
```

First you can see the mkernel protocol running. It is connected to the default table `mroute4`, with both filters set to accept all. Then mkernel lists interfaces which were assigned an index for multicast routing. Follows a list of (S,G) entries that were inserted in the kernel. Every entry has three fields: a source address, a group address and the incoming interface. After the arrow there are two bitmaps from the original route, a set of incoming and outgoing interfaces. Every interface from the list of VIFs has either a zero or one, and they are in the same order.

```
protocol igmp1 state UP
  TABLE mreq4
  Output filter: REJECT
  Interface d1 is up, querier
  Interface veth-r1 is up, other querier present
  Interface veth-br1 is up, querier
```

The IGMP dumps only its interfaces. An IGMP router can be either the querier on the link, or just passively listening because other querier is present. That happened on the interface veth-r1.

```
protocol pim1 state UP
  TABLE master4
  Input filter: REJECT
  TABLE mreq4
  TABLE mroute4
  Output filter: REJECT
  Interface d1, up, holdtime 105, gen ID 413907797
  Interface veth-r1, up, holdtime 105, gen ID 957644869
    Neighbor 10.10.1.1 UP BIDIR
  Interface veth-br1, up, holdtime 105, gen ID 3449771873
    Neighbor 10.3.1.4 UP BIDIR
    Neighbor 10.3.1.2
  Group 224.42.42.42, RP 10.1.0.42,
    PIM joins: veth-br1; joined on: veth-br1
  Group 224.0.0.2, RP 10.1.0.42,
    PIM joins: veth-br1; joined on: veth-r1 veth-br1
  Group 224.0.0.5, RP 10.1.0.42,
    PIM joins: veth-br1; joined on: veth-r1 veth-br1
  Group 224.0.0.6, RP 10.1.0.42,
    PIM joins: veth-br1; joined on: veth-br1 veth-r1
  Group 224.0.0.13, RP 10.1.0.42,
    PIM joins: veth-br1; joined on: veth-r1 veth-br1
  Group 224.0.0.22, RP 10.1.0.42,
    PIM joins: veth-br1; joined on: veth-r1 veth-br1
  RP 10.1.0.42, RPF veth-r1
    iface veth-br1, DF state: Winner, DF: ::
    iface veth-r1, DF state: Lose, DF: 10.10.1.1
    iface d1, DF state: Winner, DF: ::
```

The PIM protocol has significantly more to output. First, PIM enabled interfaces are listed with their settings and neighbors known there. You can see from the

flags that some neighbors are BIDIR-capable. There are packets coming from the neighbor `10.3.1.2`, but we haven't seen a hello from it yet. Then there is a list of groups with existing state. For every group, you can see its RPA, interfaces which joined the group downstream by PIM Join mechanism, and all interfaces for which a multicast request exists. Finally, for every RP tree this router is connected to, there is a RP section. You can see the upstream as the RPF, then the DF election state for every interface.

6. Testing

An important part of development is testing. It is hard to test a network protocol, because a lot of code depends on the network state and temporary conditions. Also, especially for testing routing protocols, you need multiple networks, and run more than one instance of the routing daemon.

There are several tools that can virtualize the network topology. Most of them are based on virtual machines or containers. This is useful for configuration or performance testing, but not that much for development, so we have created a lightweight testing framework based on Linux network namespaces to suit our needs.

While developing the multicast routing features, we continuously tested it in various testing environments, targeting the concrete feature being developed. The IGMP implementation is tested against Linux kernel, which works as the IGMP host.

It is more complicated with the PIM protocol, because we haven't found any software solution doing the bidirectional variant of PIM. There are routers from Cisco or Juniper which are able to manage a group in bidirectional mode, but we haven't been provided any to test our implementation against it. The network analyzer Wireshark is able to parse even bidirectional PIM packets, and we've examined the packets our implementation sends whether they are constructed correctly. Of course, the PIM implementation in BIRDs works correctly against itself.

6.1 Netlab

Our testing framework is called the *netlab*. It is published in its separate repository on Github.[1] It is inspired by the testing workflow of the BIRD team members. They were already using network namespaces with a collection of scripts. We have decided to clean up the scripts and give them an interface. New ideas to simplify the development process came soon after.

To use this framework, you must clone the repository into the root of the BIRD repository, or change the path to BIRD inside the main executable of netlab. Then, testing the BIRD, even in complex setups, is as simple as:

```
$ sudo netlab/netlab
[netlab] # net_up
[netlab] # start
[netlab] # log r1
[netlab] # restart r1
[netlab] # stop
[netlab] # net_down
```

[1] https://github.com/Aearsis/netlab

You can describe the testing network using a simple declarative language. For example, a setup running two BIRDs connected directly by an Ethernet cable:

```
NETLAB_CFG="$NETLAB_BASE/cfg"

netlab_node r1
netlab_node r2
if_veth r1 r2 10.10.1
```

It is very simple to add bridges, dummy interfaces representing subnets, or to create a namespace where to run other programs sending and receiving multicast packets. This is the network we've done the most of the testing on:

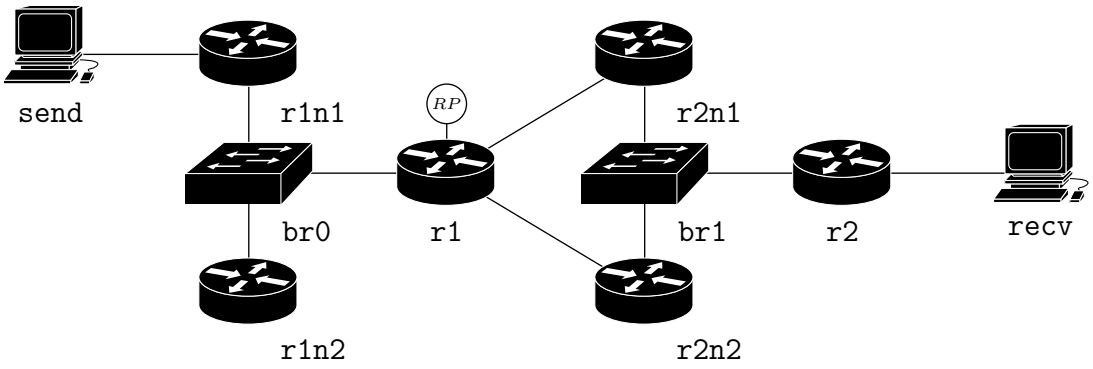


Figure 6.1: The network scheme we tested on

Moreover, every node has a dummy subnet, which is not drawn in the scheme. This is a very simple network, but you can test a lot of things there. The DF election is interesting at the `br1`, because `r2n1` and `r2n2` have the same upstream metric. If you send packets from the `send` node, you should see a traffic on the `br0`, but not on the dummy subnet at `r1n2`.

Currently, the netlab uses deterministic names. When the node name is `node`, then a namespace `netlab-node` will be created. When you define an Ethernet pair from `r1` to `r2`, then it will create an interface `veth-r2` in the `netlab-r1` namespace, and `veth-r1` in the `netlab-r2` namespace. This naming convention is very convenient, you can see immediately to which node is the interface connected.

6.2 Testing multicast

To actually test the multicast forwarding, we have used simple Python scripts.

```
#!/usr/bin/env python
import socket,struct,time

def setup(ga):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM,
    socket.IPPROTO_UDP)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

```
    sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL,
    32)
    mreq = struct.pack("4sl", socket.inet_aton(ga),
    socket.INADDR_ANY)
    sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP,
    mreq)
    return sock

def send(ga, port, msg):
    sock = setup(ga)
    bmsg = msg.encode('us-ascii')
    while 1:
        sock.sendto(bmsg, (ga, port))
        time.sleep(0.1)

def recv(ga, port):
    sock = setup(ga)
    sock.bind((ga, port))
    while 1:
            data, addr = sock.recvfrom(1024)
            print(data.decode('us-ascii'))

# send.py
send('224.42.42.42', 42000, "What hath god wrought!");

# recv.py
recv('224.42.42.42', 42000)
```

For example, running the send script in the `send` namespace in the setup above, the script joins a group and then starts multicasting a lot of messages. If you run a packet sniffer, for example `tcpdump`, you should see the traffic traveling upstream, stopping at the RPL. When you run the receiving script in the `recv` namespace, messages should start coming. You can use the netlab for this purpose:

```
[netlab] # shell send
[netlab send] # ./send.py &
[netlab send] # disown && exit
[netlab] # shell r1
[netlab r1] # tcpdump -i veth-br0
```

When testing, please be aware that PIM needs a path to the RP to work. In the testing environment an OSPF instance is taking care of this, and it can take a minute to initialize.

Conclusion

We have succeeded in implementing the multicast routing features in the BIRD. Because the protocol is fairly complex, it deserves to be more tested and, what is more important, reviewed by the BIRD team. They were too busy to deep-study my implementation continuously. But merging the PIM into BIRD is on the plan.

The source code for the whole project is available as a Git repository, located at the CZ.NIC GitLab:

`https://gitlab.labs.nic.cz/Aearsis/bird.git`

In the repository, you can find my work on the branch `pim`. Or, the current snapshot is attached as a patchset attached to the electronic version of the thesis.

There was a considerable amount of obstacles during the work. First, I had to learn how the BIRD itself works. It is not a small project, and there are lots of internal details one must know before starting to contribute in a major way. Also, I have decided to change the existing BIRD code as little as possible and write my code in the spirit of the existing one, to follow the coding practices and used design patterns. That made me study the code for even longer time, but hopefully it will pay off when merging my code to the main development branch.

Next, there were problems with undocumented code in Linux kernel. Looking back, when we know where to look for the issues, they seem trivial. But they blocked the work for a while. Namely, all the surprises around the IGMP control socket, details about how MFC entries work, or the Linux bridge doing IGMP snooping by default.

Naturally, there are many ways how to expand this project. The first direction should be support for IPv6. It should be fairly simple, because the internal structures are prepared for it. Also, the PIM itself should be able to run on IPv6 out of the box, but this behavior is not tested. One just needs to implement the MLD protocol (IPv6 equivalent of the IGMP), and the IPv6 kernel route synchronization. In the kernel, the v6 API looks like a copy of the v4, on which a few text substitutions were applied.

Next, it would be great to implement another modes of PIM, especially the PIM-SM mode. Again, the design of the PIM protocol is prepared for it. One would need to add the different types of packets PIM-SM uses and prepend source address to the internal structures. It will probably not be that easy, but we do not see any fundamental problem now.

For the bravest, the part that needs reimplementation the most is the Linux kernel itself. The implementation of multicast forwarding is copied from BSD to stay compatible with the mrouted daemon, and it reflects how mrouted works. The API should better follow the structure of the protocols instead of the structure of a particular routing daemon. Also, the multicast input, IGMP processing and other related chunks of code looks more like a hack than a systematic solution. An obvious solution is to use netlink, through which the rest of routing is already controlled.

Bibliography

[1] S.E. Deering and D.R. Cheriton. Host groups: A multicast extension to the Internet Protocol. RFC 966, December 1985. Obsoleted by RFC 988.

[2] S.E. Deering. Host extensions for IP multicasting. RFC 1112 (INTERNET STANDARD), August 1989. Updated by RFC 2236.

[3] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376 (Proposed Standard), October 2002. Updated by RFC 4604.

[4] W. Fenner. Internet Group Management Protocol, Version 2. RFC 2236 (Proposed Standard), November 1997. Updated by RFC 3376.

[5] D. Plummer. Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC 826 (INTERNET STANDARD), November 1982. Updated by RFCs 5227, 5494.

[6] M. Christensen, K. Kimball, and F. Solensky. Considerations for Internet Group Management Protocol (IGMP) and Multicast Listener Discovery (MLD) Snooping Switches. RFC 4541 (Informational), May 2006.

[7] Linux kernel source tree. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/`. Version 4.5, released 2016-03-14.

[8] M. Handley, I. Kouvelas, T. Speakman, and L. Vicisano. Bidirectional Protocol Independent Multicast (BIDIR-PIM). RFC 5015 (Proposed Standard), October 2007.

[9] The BIRD Internet Routing Daemon Project. `http://bird.network.cz`.

List of Abbreviations

BFD	Bidirectional Forwarding Detection, page 17
BGP	Border Gateway Protocol, page 16
BGP	Routing Information Protocol, page 16
BIRD	BIRD Internet Routing Daemon, page 3
DF	Designated Forwarder, page 13
DR	Designated Router, page 11
IGMP	Internet Group Management Protocol, page 4
IGP	Interior Gateway Protocol, page 33
IP	Internet Protocol, page 3
MAC	Media Access Control, page 6
MFC	Multicast Forwarding Cache, page 9
MLD	Multicast Listener Discovery, page 22
MRIB	Multicast Routing Information Base, page 11
OSPF	Open Shortest Path First, page 16
PIM	Protocol Independent Multicast, page 11
PIM-BIDIR	PIM Bidirectional Mode, page 13
PIM-DM	PIM Dense Mode, page 13
PIM-SM	PIM Sparse Mode, page 13
RAdv	Router Advertisement, page 17
RP	Rendezvous-point, page 11
RPA	Rendezvous-point address, page 11
RPF	Reverse Path Forwarding, page 11
RPL	Rendezvous-point link, page 11
VIF	Virtual Interface, page 8