BIRD 3.4.0 User's Guide

Ondrej Filip <*feela@network.cz*>, Martin Mares <*mj@ucw.cz*>, Maria Matejka <*mq@jmq.cz*>, Ondrej Zajicek <*santiago@crfreenet.org*>

This document contains user documentation for the BIRD Internet Routing Daemon project.

Contents

1 Introduction **3**
- 1.1 What is BIRD . . . 3
- 1.2 Installing BIRD . . . 4
- 1.3 Running BIRD . . . 4
- 1.4 Privileges . . . 5

2 Architecture **6**
- 2.1 Routing tables . . . 6
- 2.2 Routes and network types . . . 6
- 2.3 Protocols and channels . . . 9
- 2.4 Graceful restart . . . 10
- 2.5 MPLS . . . 10

3 Configuration **12**
- 3.1 Introduction . . . 12
- 3.2 Global options . . . 12
- 3.3 Routing table options . . . 16
- 3.4 Protocol options . . . 17
- 3.5 Channel options . . . 19
- 3.6 MPLS options . . . 21

4 Remote control **23**
- 4.1 Overview . . . 23
- 4.2 Configuration . . . 23
- 4.3 Usage . . . 23

5 Filters **27**
- 5.1 Introduction . . . 27
- 5.2 Data types . . . 28
- 5.3 Operators . . . 32
- 5.4 Control structures . . . 33
- 5.5 Route attributes . . . 34
- 5.6 Other statements . . . 36

6 Protocols **37**
- 6.1 Aggregator . . . 37
- 6.2 Babel . . . 38
- 6.3 BFD . . . 41
- 6.4 BGP . . . 45
- 6.5 BMP . . . 66
- 6.6 Bridge . . . 66
- 6.7 Device . . . 67
- 6.8 Direct . . . 68
- 6.9 EVPN . . . 69
- 6.10 Kernel . . . 71
- 6.11 L3VPN . . . 74
- 6.12 MRT . . . 76

6.13 OSPF 77
6.14 Pipe 85
6.15 RAdv 87
6.16 RIP 92
6.17 RPKI 96
6.18 Static 99

7 Conclusions **104**
7.1 Future work 104
7.2 Getting more help 104

Chapter 1: Introduction

1.1 What is BIRD

The name 'BIRD' is actually an acronym standing for 'BIRD Internet Routing Daemon'. Let's take a closer look at the meaning of the name:

BIRD: Well, we think we have already explained that. It's an acronym standing for 'BIRD Internet Routing Daemon', you remember, don't you? :-)

Internet Routing: It's a program (well, a daemon, as you are going to discover in a moment) which works as a dynamic router in an Internet type network (that is, in a network running either the IPv4 or the IPv6 protocol). Routers are devices which forward packets between interconnected networks in order to allow hosts not connected directly to the same local area network to communicate with each other. They also communicate with the other routers in the Internet to discover the topology of the network which allows them to find optimal (in terms of some metric) rules for forwarding of packets (which are called routing tables) and to adapt themselves to the changing conditions such as outages of network links, building of new connections and so on. Most of these routers are costly dedicated devices running obscure firmware which is hard to configure and not open to any changes (on the other hand, their special hardware design allows them to keep up with lots of high-speed network interfaces, better than general-purpose computer does). Fortunately, most operating systems of the UNIX family allow an ordinary computer to act as a router and forward packets belonging to the other hosts, but only according to a statically configured table.

A *Routing Daemon* is in UNIX terminology a non-interactive program running on background which does the dynamic part of Internet routing, that is it communicates with the other routers, calculates routing tables and sends them to the OS kernel which does the actual packet forwarding. There already exist other such routing daemons: routed (RIP only), GateD (non-free), Zebra and MRTD, but their capabilities are limited and they are relatively hard to configure and maintain.

BIRD is an Internet Routing Daemon designed to avoid all of these shortcomings, to support all the routing technology used in the today's Internet or planned to be used in near future and to have a clean extensible architecture allowing new routing protocols to be incorporated easily. Among other features, BIRD supports:

- both IPv4 and IPv6 protocols
- multiple routing tables
- the Border Gateway Protocol (BGPv4)
- the Routing Information Protocol (RIPv2, RIPng)
- the Open Shortest Path First protocol (OSPFv2, OSPFv3)
- the Babel Routing Protocol
- the Router Advertisements for IPv6 hosts
- a virtual protocol for exchange of routes between different routing tables on a single host
- a command-line interface allowing on-line control and inspection of status of the daemon
- soft reconfiguration (no need to use complex online commands to change the configuration, just edit the configuration file and notify BIRD to re-read it and it will smoothly switch itself to the new configuration, not disturbing routing protocols unless they are affected by the configuration changes)
- a powerful language for route filtering

BIRD has been developed at the Faculty of Math and Physics, Charles University, Prague, Czech Republic as a student project. It can be freely distributed under the terms of the GNU General Public License.

BIRD has been designed to work on all UNIX-like systems. It has been developed and tested under Linux 2.0 to 2.6, and then ported to FreeBSD, NetBSD and OpenBSD, porting to other systems (even non-UNIX ones) should be relatively easy due to its highly modular architecture.

BIRD 1.x supported either IPv4 or IPv6 protocol, but had to be compiled separately for each one. BIRD 2 supports both of them with a possibility of further extension. BIRD 2 supports Linux at least 3.16, FreeBSD 10, NetBSD 7.0, and OpenBSD 5.8. Anyway, it will probably work well also on older systems.

1.2 Installing BIRD

On a recent UNIX system with GNU development tools (GCC, binutils, m4, make) and Perl, installing BIRD should be as easy as:

```
./configure
make
make install
vi /usr/local/etc/bird.conf
bird
```

You can use `./configure --help` to get a list of configure options. The most important ones are: `--with-protocols=` to produce a slightly smaller BIRD executable by configuring out routing protocols you don't use, and `--prefix=` to install BIRD to a place different from `/usr/local`.

1.3 Running BIRD

You can pass several command-line options to bird:

`-c` *config name*
: use given configuration file instead of *prefix*`/etc/bird.conf`.

`-d`
: enable debug messages to stderr, and run bird in foreground.

`-D` *filename of debug log*
: enable debug messages to given file.

`-f`
: run bird in foreground.

`-g` *group*
: use that group ID, see the next section for details.

`-h, --help`
: display command-line options to bird.

`-l`
: look for a configuration file and a communication socket in the current working directory instead of in default system locations. However, paths specified by options `-c`, `-s` have higher priority.

`-p`
: just parse the config file and exit. Return value is zero if the config file is valid, nonzero if there are some errors.

`-P` *name of PID file*
: create a PID file with given filename.

`-R`
: apply graceful restart recovery after start.

`-s` *name of communication socket*
: use given filename for a socket for communications with the client, default is *prefix*`/var/run/bird.ctl`.

`-u` *user*
: drop privileges and use that user ID, see the next section for details.

`--version`
display bird version.

BIRD writes messages about its work to log files or syslog (according to config).

1.4 Privileges

BIRD, as a routing daemon, uses several privileged operations (like setting routing table and using raw sockets). Traditionally, BIRD is executed and runs with root privileges, which may be prone to security problems. The recommended way is to use a privilege restriction (options `-u`, `-g`). In that case BIRD is executed with root privileges, but it changes its user and group ID to an unprivileged ones, while using Linux capabilities to retain just required privileges (capabilities CAP_NET_*). Note that the control socket is created before the privileges are dropped, but the config file is read after that. The privilege restriction is not implemented in BSD port of BIRD.

An unprivileged user (as an argument to `-u` options) may be the user `nobody`, but it is suggested to use a new dedicated user account (like `bird`). The similar considerations apply for the group option, but there is one more condition – the users in the same group can use `birdc` to control BIRD.

Finally, there is a possibility to use external tools to run BIRD in an environment with restricted privileges. This may need some configuration, but it is generally easy – BIRD needs just the standard library, privileges to read the config file and create the control socket and the CAP_NET_* capabilities.

Chapter 2: Architecture

2.1 Routing tables

The heart of BIRD is a routing table. BIRD has several independent routing tables; each of them contains routes of exactly one *nettype* (see below). There are two default tables – `master4` for IPv4 routes and `master6` for IPv6 routes. Other tables must be explicitly configured.

These routing tables are not kernel forwarding tables. No forwarding is done by BIRD. If you want to forward packets using the routes in BIRD tables, you may use the Kernel protocol (see below) to synchronize them with kernel FIBs.

Every nettype defines a (kind of) primary key on routes. Every route source can supply one route for every possible primary key; new route announcement replaces the old route from the same source, keeping other routes intact. BIRD always chooses the best route for each primary key among the known routes and keeps the others as suboptimal. When the best route is retracted, BIRD re-runs the best route selection algorithm to find the current best route.

The global best route selection algorithm is (roughly) as follows:

- Preferences of the routes are compared.
- Source protocol instance preferences are compared.
- If source protocols are the same (e.g. BGP vs. BGP), the protocol's route selection algorithm is invoked.
- If source protocols are different (e.g. BGP vs. OSPF), result of the algorithm is undefined.

Usually, a routing table just chooses a selected route from a list of entries for one network. Optionally, these lists of entries are kept completely sorted (according to preference or some protocol-dependent metric). See sorted (p. 16) table option for details.

2.2 Routes and network types

BIRD works with several types of routes. Some of them are typical IP routes, others are better described as forwarding rules. We call them all routes, regardless of this difference.

Every route consists of several attributes (read more about them in the Route attributes (p. 34) section); the common for all routes are:

- IP address of router which told us about this route
- Source protocol instance
- Route preference
- Optional attributes defined by protocols

Other attributes depend on nettypes. Some of them are part of the primary key, these are marked (PK).

2.2.1 IPv4 and IPv6 routes

The traditional routes. Configuration keywords are `ipv4` and `ipv6`.

- (PK) Route destination (IP prefix together with its length)
- Route next hops (see below)

2.2.2 IPv6 source-specific routes

The IPv6 routes containing both destination and source prefix. They are used for source-specific routing (SSR), also called source-address dependent routing (SADR), see RFC 8043. Currently limited mostly to the Babel protocol. Configuration keyword is `ipv6 sadr`.

- (PK) Route destination (IP prefix together with its length)
- (PK) Route source (IP prefix together with its length)
- Route next hops (see below)

2.2.3 VPN IPv4 and IPv6 routes

Routes for IPv4 and IPv6 with VPN Route Distinguisher (RFC 4364). Configuration keywords are `vpn4` and `vpn6`.

- (PK) Route destination (IP prefix together with its length)
- (PK) Route distinguisher (according to RFC 4364)
- Route next hops

2.2.4 Route Origin Authorization for IPv4 and IPv6

These entries can be used to validate route origination of BGP routes. A ROA entry specifies prefixes which could be originated by an AS number. Their keywords are `roa4` and `roa6`.

- (PK) IP prefix together with its length
- (PK) Matching prefix maximal length
- (PK) AS number

2.2.5 Autonomous System Provider Authorization routes

These entries can be used to validate `AS_PATH` attribute of BGP routes. An ASPA entry specifies a list of valid providers for specific Autonomous System. Configuration keyword is `aspa`.

- (PK) AS number
- List of AS numbers of providers

2.2.6 Flowspec for IPv4 and IPv6

Flowspec rules are a form of firewall and traffic flow control rules distributed mostly via BGP. These rules may help the operators stop various network attacks in the beginning before eating up the whole bandwidth. Configuration keywords are `flow4` and `flow6`.

- (PK) IP prefix together with its length
- (PK) Flow definition data
- Flow action (encoded internally as BGP communities according to RFC 8955)

2.2.7 Ethernet forwarding entries

Ethernet forwarding entries are entries in bridge forwarding database. They control forwarding of ethernet frames based on destination MAC addresses. Each entry maps a MAC address (and VLAN ID) to a specific port, determining which interface should receive frames destined for that address. Optionally, next hop can be used for VXLAN encapsulation (as part of EVPN deployment). Configuration keyword is `eth`.

- (PK) MAC address
- (PK) VLAN ID
- Route interface (and next hop)

2.2.8 MPLS switching rules

MPLS routes control MPLS forwarding in the same way as IP routes control IP forwarding. MPLS-aware routing protocols produce both labeled IP routes and corresponding MPLS routes. Configuration keyword is `mpls`.

- (PK) MPLS label
- Route next hops

2.2.9 EVPN routes

EVPN routes encode control plane information for Ethernet VPN (EVPN) networks. They carry MAC address and IP address reachability information using BGP advertisements. There are several types of EVPN routes, each serves specific purposes - MAC (type 2) routes advertise MAC/IP locations, IMET (type 3) routes announce endpoints for broadcast / unknown / multicast communication. These routes enable distributed learning of endpoint locations across multiple sites in the overlay network. Configuration keyword is `evpn`.

EVPN Ethernet Auto-Discovery (EAD) routes

EVPN EAD routes advertise Ethernet segment reachability and enable multihoming redundancy mechanisms. Currently not used by BIRD.

- (PK) Ethernet Segment Identifier (ESI)
- (PK) Ethernet Tag ID (tag)
- (PK) Route Distinguisher (RD)
- MPLS label / VNI

EVPN MAC/IP Advertisement (MAC) routes

EVPN MAC routes advertise MAC addresses and optionally their associated IP addresses within an EVPN instance.

- (PK) MAC address
- (PK) IP address (optional)
- (PK) Ethernet Tag ID (tag)
- (PK) Route Distinguisher (RD)
- Ethernet Segment Identifier (ESI)
- Router IP address
- MPLS label / VNI

EVPN Inclusive Multicast Ethernet Tag (IMET) routes

EVPN IMET routes advertise endpoints for BUM (Broadcast, Unknown, Multicast) traffic forwarding.

- (PK) Ethernet Tag ID (tag)
- (PK) Route Distinguisher (RD)
- (PK) Router IP address
- MPLS label / VNI (in PMSI attribute)

EVPN Ethernet Segment (ES) routes

EVPN ES routes advertise Ethernet segment identifiers and designated forwarder election information for multihoming. Currently not used by BIRD.

- (PK) Ethernet Segment Identifier (ESI)
- (PK) Route Distinguisher (RD)
- (PK) Router IP address

2.2.10 Neighbor entries

Neighbor entries represent discovered network neighbors. These are used to exchange neighbor information between protocols. For example, in automatic BGP peering scenarios, protocols responsible for router discovery (such as RAdv) announce these entries, which are then exported to a dynamic BGP instance to automatically establish peering sessions. The same type is used for both IPv4 and IPv6 neighbors. Configuration keyword is `neighbor`.

- (PK) IP address
- (PK) Interface index

2.2.11 Route next hops

This is not a nettype. The route next hop is a complex attribute common for many nettypes as you can see before. Every next hop has its assigned device (either assumed from its IP address or set explicitly). It may have also an IP address and an MPLS stack (one or both independently). Maximal MPLS stack depth is set (in compile time) to 8 labels.

Every route (when eligible to have a next hop) can have more than one next hop. In that case, every next hop has also its weight.

2.3 Protocols and channels

BIRD protocol is an abstract class of producers and consumers of the routes. Each protocol may run in multiple instances and bind on one side to route tables via channels, on the other side to specified listen sockets (BGP), interfaces (Babel, OSPF, RIP), APIs (Kernel, Direct), or nothing (Static, Pipe).

There are also two protocols that do not have any channels – BFD and Device. Both of them are kind of service for other protocols.

Each protocol is connected to a routing table through a channel. Some protocols support only one channel (OSPF, RIP), some protocols support more channels (BGP, Direct). Each channel has two filters which can accept, reject and modify the routes. An *export* filter is applied to routes passed from the routing table to the protocol, an *import* filter is applied to routes in the opposite direction.

2.4 Graceful restart

When BIRD is started after restart or crash, it repopulates routing tables in an uncoordinated manner, like after clean start. This may be impractical in some cases, because if the forwarding plane (i.e. kernel routing tables) remains intact, then its synchronization with BIRD would temporarily disrupt packet forwarding until protocols converge. Graceful restart is a mechanism that could help with this issue. Generally, it works by starting protocols and letting them repopulate routing tables while deferring route propagation until protocols acknowledge their convergence. Note that graceful restart behavior have to be configured for all relevant protocols and requires protocol-specific support (currently implemented for Kernel and BGP protocols), it is activated for particular boot by option `-R`.

Some protocols (e.g. BGP) could be restarted gracefully after both intentional outage and crash, while others (e.g. OSPF) after intentional outage only. For planned graceful restart, BIRD must be shut down by graceful restart (p. 26) command instead of regular down (p. 26) command. In this way routing neighbors are notified about planned graceful restart and routes are kept in kernel table after shutdown.

2.5 MPLS

Multiprotocol Label Switching (MPLS) is a networking technology which works below IP routing but above the link (e.g. ethernet) layer. It is described in RFC 3031.

In regular IP forwarding, the destination address of a packet is independently examined in each hop, a route with longest prefix match is selected from the routing table, and packet is processed accordingly. In general, there is no difference between border routers and internal routers w.r.t. IP forwarding.

In MPLS forwarding, when a packet enters the network, it is classified (based on destination address, ingress interface and other factors) into one of forwarding equivalence classes (FECs), then a header with a MPLS label identifying the FEC is attached to it, and the packet is forwarded. In internal routers, only the MPLS label is examined, the matching MPLS route is selected from the MPLS routing table, and the packet is processed accordingly. The specific value of MPLS label has local meaning only and may change between hops (that is why it is called label switching). When the packet leaves the network, the MPLS header is removed.

The advantage of the MPLS approach is that other factors than the destination address can be considered and used consistently in the whole network, for example IP traffic with multiple overlapping private address ranges could be mixed together, or particular paths for specific flows could be defined. Another advantage is that MPLS forwarding by internal routers can be much simpler than IP forwarding, as instead of the longest prefix match algorithm it uses simpler exact match for MPLS route selection. The disadvantage is additional complexity in signaling. For further details, see RFC 3031.

MPLS-aware routing protocols not only distribute IP routing information, but they also distribute labels. Therefore, they produce labeled routes - routes representing label switched paths (LSPs) through the MPLS domain. Such routes have IP prefix and next hop address like regular (non-labeled) routes, but they also have local MPLS label (in route attribute mpls_label (p. 35)) and outgoing MPLS label (as a part of the next hop). They are stored in regular IP routing tables.

Labeled routes are used for exchange of routing information between routing protocols and for ingress (IP -> MPLS) forwarding, but they are not directly used for MPLS forwarding. For that purpose MPLS routes (p. 8) are used. These are routes that have local MPLS label as a primary key and they are stored in the MPLS routing table.

In BIRD, the whole process generally works this way: A MPLS-aware routing protocol (say BGP) receives routing information including remote label. It produces a route with attribute mpls_policy (p. 35) specifying desired MPLS label policy (p. 22). Such route then passes the import filter (which could modify the MPLS label policy or perhaps assign a static label) and when it is accepted, a local MPLS label is selected (according to the label policy) and attached to the route, producing labeled route. When a new MPLS label is allocated, the MPLS-aware protocol automatically produces corresponding MPLS route. When all labeled routes that use specific local MPLS label are retracted, the corresponding MPLS route is retracted too.

There are three important concepts for MPLS in BIRD: MPLS domains, MPLS tables and MPLS channels. MPLS domain represents an independent label space, all MPLS-aware protocols are associated with some MPLS domain. It is responsible for label management, handling label allocation requests from MPLS-aware protocols. MPLS table is just a routing table for MPLS routes. Routers usually have one MPLS domain

and one MPLS table, with Kernel protocol to export MPLS routes into kernel FIB.

MPLS channels make protocols MPLS-aware, they are responsible for keeping track of active FECs (and corresponding allocated labels), selecting FECs / local labels for labeled routes, and maintaining correspondence between labeled routes and MPLS routes.

Note that local labels are allocated to individual MPLS-aware protocols and therefore it is not possible to share local labels between different protocols.

Chapter 3: Configuration

3.1 Introduction

BIRD is configured using a text configuration file. Upon startup, BIRD reads *prefix*`/etc/bird.conf` (unless the `-c` command line option is given). Configuration may be changed at user's request: if you modify the config file and then signal BIRD with `SIGHUP`, it will adjust to the new config. Then there's the client which allows you to talk with BIRD in an extensive way.

In the config, everything on a line after `#` or inside `/* */` is a comment, whitespace characters are treated as a single space. If there's a variable number of options, they are grouped using the { } brackets. Each option is terminated by a `;`. Configuration is case sensitive. There are two ways how to name symbols (like protocol names, filter names, constants etc.). You can either use a simple string starting with a letter (or underscore) followed by any combination of letters, numbers and underscores (e.g. `R123`, `my_filter`, `bgp5`) or you can enclose the name into apostrophes (`'`) and than you can use any combination of numbers, letters, underscores, hyphens, dots and colons (e.g. `'1:strange-name'`, `'-NAME-'`, `'cool::name'`).

In most cases where options use an argument that is a basic data type (e.g. number, string, IP address) it is possible to use a named constant (defined by define (p. 14) statement), or a constant expression enclosed in parenthesis (e.g. `(2 + 2)`). These expressions use the BIRD filter language (p. 27).

Here is an example of a simple config file. It enables synchronization of routing tables with OS kernel, learns network interfaces and runs RIP on all network interfaces found.

```
protocol kernel {
        ipv4 {
                export all;     # Default is export none
        };
        persist;                # Don't remove routes on BIRD shutdown
}

protocol device {
}

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

3.2 Global options

`include "`*filename*`";`

This statement causes inclusion of a new file. The *filename* could also be a wildcard, in that case matching files are included in alphabetic order. The maximal depth is 8. Note that this statement can be used anywhere in the config file, even inside other options, but always on the beginning of line. In the following example, the first semicolon belongs to the `include`, the second to `ipv6 table`. If the `tablename.conf` contains exactly one token (the name of the table), this construction is correct:

```
ipv6 table
include "tablename.conf";;
```

`log "`*filename*`" [`*limit* `"`*backup*`"] | fixed "`*filename*`"` *size* | `syslog [name` *name*`] | stderr | udp` *address* `[port` *port*`] all|{` *list of classes* `}`

Set logging of messages having the given class (either `all` or { `error|trace [, ...]` } etc.) into

selected destination - a file specified as a filename string (with optional log rotation information), syslog (with optional name argument), the stderr output, or as a UDP message (in RFC 3164 syslog format).

Classes are: `info`, `warning`, `error` and `fatal` for messages about local problems, `debug` for debugging messages, `trace` when you want to know what happens in the network, `remote` for messages about misbehavior of remote machines, `auth` about authentication failures, `bug` for internal BIRD bugs.

Logging directly to file supports basic log rotation – there is an optional log file limit and a backup filename, when log file reaches the limit, the current log file is renamed to the backup filename and a new log file is created. It's also possible to log to a single file behaving as a ring buffer with a fixed size.

You may specify more than one `log` line to establish logging to multiple destinations. Default: log everything to the system log, or to the debug output if debugging is enabled by `-d`/`-D` command-line option.

`debug protocols all|off|{ states|routes|filters|interfaces|events|packets [, ...] }`
Set global defaults of protocol debugging options. See debug (p. 17) in the following section. Default: off.

`debug channels all|off|{ states|routes|filters|events [, ...] }`
Set global defaults of channel debugging options. See debug (p. 19) in the channel section. Default: off.

`debug tables all|off|{ states|routes|filters|events [, ...] }`
Set global defaults of table debugging options. See debug (p. 17) in the table section. Default: off.

`debug commands` *number*
Control logging of client connections (0 for no logging, 1 for logging of connects and disconnects, 2 and higher for logging of all client commands). Default: 0.

`debug latency all|off|{ ping|wakeup|scheduling|sockets|events|timers }`
Activate tracking of internal scheduler actions. This is a developer and technical support tool for cases when internal events are missed. You should keep this off unless you know what you are doing. Default: off.

`debug latency limit` *time*
If `debug latency` is enabled, this option allows to specify a limit for elapsed time. Events exceeding the limit are logged. Default: 1 s.

`watchdog warning` *time*
Set time limit for I/O loop cycle. If one iteration took more time to complete, a warning is logged. Default: 5 s.

`watchdog timeout` *time*
Set time limit for I/O loop cycle. If the limit is breached, BIRD is killed by abort signal. The timeout has effective granularity of seconds, zero means disabled. Default: disabled (0).

`mrtdump "`*filename*`"`
Set MRTdump file name. This option must be specified to allow MRTdump feature. Default: no dump file.

`mrtdump protocols all|off|{ states|messages [, ...] }`
Set global defaults of MRTdump options. See `mrtdump` in the following section. Default: off.

`filter` *name local variables*`{` *commands* `}`
Define a filter. You can learn more about filters in the following chapter.

`function` *name* `(`*parameters*`) [ ->` *return type* `]` *local variables* `{` *commands* `}`
Define a function. You can learn more about functions in the following chapter.

`protocol rip|ospf|bgp|...` [*name* [`from` *name2*]] { *protocol options* }
Define a protocol instance called *name* (or with a name like "rip5" generated automatically if you don't specify any *name*). You can learn more about configuring protocols in their own chapters. When `from` *name2* expression is used, initial protocol options are taken from protocol or template *name2* You can run more than one instance of most protocols (like RIP or BGP). By default, no instances are configured.

`template rip|ospf|bgp|...` [*name* [`from` *name2*]] { *protocol options* }
Define a protocol template instance called *name* (or with a name like "bgp1" generated automatically if you don't specify any *name*). Protocol templates can be used to group common options when many similarly configured protocol instances are to be defined. Protocol instances (and other templates) can use templates by using `from` expression and the name of the template. At the moment templates (and `from` expression) are not implemented for OSPF protocol.

`define` *constant* `=` *expression*
Define a constant. You can use it later in every place you could use a value of the same type. Besides, there are some predefined numeric constants based on /etc/iproute2/rt_* files. A list of defined constants can be seen (together with other symbols) using 'show symbols' command.

`attribute` *type name*
Declare a custom route attribute. You can set and get it in filters like any other route attribute. This feature is intended for marking routes in import filters for export filtering purposes instead of locally assigned BGP communities which have to be deleted in export filters.

`router id` *number* | *IPv4 address*
Set BIRD's router ID. It's a 4-byte non-zero integer that should be unique within an AS. Default: the lowest IPv4 address of the router.

`router id from [-] [ "`*mask*`" ] [` *prefix* `] [, ...]`
Set BIRD's router ID based on an IPv4 address of an interface specified by an interface pattern. See interface (p. 18) section for detailed description of interface patterns with extended clauses.

`hostname "`*name*`"`
Set hostname. Default: node name as returned by 'uname -n'.

`graceful restart wait` *number*
During graceful restart recovery, BIRD waits for convergence of routing protocols. This option allows to specify a timeout for the recovery to prevent waiting indefinitely if some protocols cannot converge. Default: 240 seconds.

`timeformat route|protocol|base|log "`*format1*`" [`*limit* `"`*format2*`"]`
This option allows to specify a format of date/time used by BIRD. The first argument specifies for which purpose such format is used. `route` is a format used in 'show route' command output, `protocol` is used in 'show protocols' command output, `base` is used for other commands and `log` is used in a log file.

"*format1*" is a format string using *strftime(3)* notation (see *man strftime* for details). It is extended to support sub-second time part with variable precision (up to microseconds) using "%f" conversion code (e.g., "%T.%3f" is hh:mm:ss.sss time). *limit* and "*format2*" allow to specify the second format string for times in past deeper than *limit* seconds.

There are several shorthands: `iso long` is a ISO 8601 date/time format (YYYY-MM-DD hh:mm:ss) that can be also specified using `"%F %T"`. Similarly, `iso long ms` and `iso long us` are ISO 8601 date/time formats with millisecond or microsecond precision. `iso short` is a variant of ISO 8601 that uses just the time format (hh:mm:ss) for near times (up to 20 hours in the past) and the date format (YYYY-MM-DD) for far times. This is a shorthand for `"%T" 72000 "%F"`. And there are also `iso short ms` and `iso short us` high-precision variants of that.

By default, BIRD uses the `iso short ms` format for `route` and `protocol` times, and the `iso long ms` format for `base` and `log` times.

nettype `table` *name* `[` { *option*; `[...]` } `]`
Define a new routing table. The default routing tables `master4` and `master6` are defined implicitly, other routing tables have to be defined by this option. See the routing table configuration section (p. 16) for routing table options.

`mpls domain` *name* `[` { *option*; `[...]` } `]`
Define a new MPLS domain. MPLS domains represent independent label spaces and are responsible for MPLS label management. All MPLS-aware protocols are associated with some MPLS domain. See the MPLS configuration section (p. 21) for MPLS domain options.

`eval` *expr*
Evaluates given filter expression. It is used by the developers for testing of filters.

3.2.1 Thread setup

BIRD runs in several threads. There is one main thread, taking care about startup, shutdown, (re)configuration, CLI and several protocols which have not yet been updated to run in other threads. Then there are several thread groups, running the rest.

Default thread group is `worker`. This group runs (by default) BGP, BMP, MRT, Pipe and RPKI. Also the routing table maintenance routines run in these threads. For BFD, there is another thread group called `express`, with minimal latency, expecting all tasks to be extremely fast. The threads are started as soon as some work is required from them, therefore if you don't configure BFD, the express thread won't start.

Any configuration of thread groups must be strictly placed before any table or protocol configuration, as these depend on the thread group existence.

Thread groups are configured by writing `thread group` *name* {} blocks, which contain the following options:

`threads` *number*
Set how many threads should BIRD spawn in this thread group. Every thread can utilize one complete CPU core. You probably want to keep at least one free core for other processes. The maximum feasible thread count heavily depends on the actual workload and must be determined by testing or estimation. Default: 1

`default` *bool*
Mark this thread group as default. There must be exactly one thread group marked as default. Default default thread group: `worker`

`max latency` *time*
Set maximum latency for the thread group. The group tries to dispatch every event before this time elapses. Setting this too low causes BIRD to spend lots of time on overhead and fill logs with reports. Setting this too high causes BIRD to hold locks for long times. Do not change unless you know what you are doing. Default: 1 s for worker, 10 ms for express.

`min time` *time*
Set minimum awarded loop execution time for the thread group. This is an internal performance tuning knob which may change between minor versions. Do not change unless you know what you are doing. Default: 10 ms for worker, 1 ms for express.

`max time` *time*
Set maximum awarded loop execution time for the thread group. This is an internal performance tuning knob which may change between minor versions. Do not change unless you know what you are doing. Default: 300 ms for worker, 10 ms for express.

`wakeup time` *time*
Set maximum sleep time for the thread group. Every thread will wake up after this time even if no work is requested. This is an internal IO loop knob which may change between minor versions. Do not change unless you know what you are doing. Default: 31415 s for worker, 60 s for express.

There is also a "simple" thread group setting. If you write `threads` *number* on top level, it is equivalent to setting the worker group thread count to that number, and the express group thread count to one. This setting is deprecated and may disappear in some future version.

3.3 Routing table options

Most routing tables do not need any options and are defined without an option block, but there are still some options to tweak routing table behavior. Note that implicit tables (`master4` and `master6`) can be redefined in order to set options.

`sorted` *switch*
Usually, a routing table just chooses the selected (best) route from a list of routes for each network, while keeping remaining routes unsorted. If enabled, these lists of routes are kept completely sorted (according to preference or some protocol-dependent metric).

This is needed for some protocol features (e.g. `secondary` option of BGP protocol, which allows to accept not just a selected route, but the first route (in the sorted list) that is accepted by filters), but it is incompatible with some other features (e.g. `deterministic med` option of BGP protocol, which activates a way of choosing selected route that cannot be described using comparison and ordering). Minor advantage is that routes are shown sorted in `show route`, minor disadvantage is that it is slightly more computationally expensive. Default: off.

`trie` *switch*
BIRD routing tables are implemented with hash tables, which is efficient for exact-match lookups, but inconvenient for longest-match lookups or interval lookups (finding superprefix or subprefixes). This option activates additional trie structure that is used to accelerate these lookups, while using the hash table for exact-match lookups.

This has advantage for RPKI (p. 96) (on ROA tables), for recursive next-hops (p. 58) (on IGP tables), and is required for flowspec validation (p. 59) (on base IP tables). Another advantage is that interval results (like from `show route in ...` command) are lexicographically sorted. The disadvantage is that trie-enabled routing tables require more memory, which may be an issue especially in multi-table setups. Default: off.

`gc threshold` *number*
Specify a minimum amount of removed networks that triggers a garbage collection (GC) cycle. Default: 1000.

`gc period` *time*
Specify a period of time between consecutive GC cycles. When there is a significant amount of route withdraws, GC cycles are executed repeatedly with given period time (with some random factor). When there is just small amount of changes, GC cycles are not executed. In extensive route server setups, running GC on hundreds of full BGP routing tables can take significant amount of time, therefore they should use higher GC periods. Default: adaptive, based on number of routing tables in the configuration. From 10 s (with <= 25 routing tables) up to 600 s (with >= 1500 routing tables).

`cork threshold` *number number*
Too many pending exports may lead to memory bloating. In such cases, BIRD tries to relieve the memory pressure by pausing some routines until the queue sizes get low enough. This option allows the user to set the thresholds; first value is the low threshold (when to resume), the second one is the high threshold (when to pause). The higher is the threshold, the more memory can get used. In most cases, the defaults should work for you but if you experience memory bloating on import surges, this knob is the first to turn down. Default: 333300 1011010.

`export settle time` *time time*
Minimum and maximum settle times, respectively, for export announcements. When multiple routes are changing, this mechanism waits for the changes to settle before waking up sleeping export threads but if the changes are coming steadily, BIRD isn't waiting forever; at most the maximum time. Default values: `1 ms 100 ms`. You have to always provide both values.

`route refresh export settle time` *time time*
Minimum and maximum settle times, respectively, for export announcements (the same as above), valid when any channel is currently doing a route refresh. This serves a purpose of even more aggresive change bundling, knowing that there is some active process generating changes in a fast pace. If you don't want this feature, set this to the same values as export settle time (p. 59). Default values: `100 ms 3 s`.

`digest settle time` *time time*
Minimum and maximum settle times, respectively, for table change digests. This settle time applies to ROA table changes where a trie is generated containing all changed ROAs to automatically reload depending channels. Default values: `1 s 20 s`.

`debug all|off|{ states|routes|events [, ...] }`
Set table debugging options. Each table can write some trace messages into log with category `trace`. You can request `all` trace messages or select some types: `states` for table state changes and auxiliary processes, `routes` for auxiliary route notifications (next hop update, flowspec revalidation) and `events` for more detailed auxiliary routine debug. See also channel debugging option (p. 19). Default: off.

`thread group` *name*
Assign this table's maintenance tasks to this thread group. Default: `worker`.

3.4 Protocol options

For each protocol instance, you can configure a bunch of options. Some of them (those described in this section) are generic, some are specific to the protocol (see sections talking about the protocols).

Several options use a *switch* argument. It can be either `on`, `yes` or a numeric expression with a non-zero value for the option to be enabled or `off`, `no` or a numeric expression evaluating to zero to disable it. An empty *switch* is equivalent to `on` ("silence means agreement").

`disabled` *switch*
Disables the protocol. You can change the disable/enable status from the command line interface without needing to touch the configuration. Disabled protocols are not activated. Default: protocol is enabled.

`restart time limit` *time*
Set time limit for subsequent automatic restarts of the protocol. If the protocol hits the limit (with a restart action) before this time elapses from starting the protocol, the protocol is disabled with an error message in the config file. This doesn't apply to manual restarts or reconfiguration. Default: 5 s.

`debug all|off|{ states|routes|filters|interfaces|events|packets [, ...] }`
Set protocol debugging options. If asked, each protocol is capable of writing trace messages about its work to the log (with category `trace`). You can either request printing of `all` trace messages or only of the selected types: `states` for protocol state changes (protocol going up, down, starting, stopping etc.), `routes` for routes exchanged with the routing table, `filters` for details on route filtering, `interfaces` for interface change events sent to the protocol, `events` for events internal to the protocol and `packets` for packets sent and received by the protocol. Classes `routes` and `filters` can be also set per-channel using channel debugging option (p. 19)) Default: off.

`mrtdump all|off|{ states|messages [, ...] }`
Set protocol MRTdump flags. MRTdump is a standard binary format for logging information from routing protocols and daemons. These flags control what kind of information is logged from the protocol to the MRTdump file (which must be specified by global `mrtdump` option, see the previous section). Although these flags are similar to flags of `debug` option, their meaning is different and protocol-specific. For BGP protocol, `states` logs BGP state changes and `messages` logs received BGP messages. Other protocols does not support MRTdump yet.

`router id` *number* | *IPv4 address*
This option can be used to override global router id for a given protocol. Default: uses global router id.

`hostname "`*name*`"`
This option can be used to override global hostname for a given protocol. Default: uses global hostname.

`description "`*text*`"`
This is an optional description of the protocol. It is displayed as a part of the output of 'show protocols all' command.

`vrf "`*text*`"|default`
Associate the protocol with specific VRF. The protocol will be restricted to interfaces assigned to the VRF and will use sockets bound to the VRF. A corresponding VRF interface must exist on OS level. For kernel protocol, an appropriate table still must be explicitly selected by `table` option.

By selecting `default`, the protocol is associated with the default VRF; i.e., it will be restricted to interfaces not assigned to any regular VRF. That is different from not specifying `vrf` at all, in which case the protocol may use any interface regardless of its VRF status.

Note that for proper VRF support it is necessary to use Linux kernel version at least 4.14, older versions have limited VRF implementation. Before Linux kernel 5.0, a socket bound to a port in default VRF collide with others in regular VRFs. In BGP, this can be avoided by using strict bind (p. 50) option.

channel name `[{`*channel config*`}]`
Every channel must be explicitly stated. See the protocol-specific configuration for the list of supported channel names. See the channel configuration section (p. 19) for channel definition.

`thread group` *name*
Assign this protocol's tasks to this thread group. Ignored in protocols not yet updated for the multi-threaded execution model. Default: `worker`.

There are several options that give sense only with certain protocols:

`interface [-] [ "`*mask*`" ] [ `*prefix*` ] [, ...] [ { `*option*`; [...] } ]`
Specifies a set of interfaces on which the protocol is activated with given interface-specific options. A set of interfaces specified by one interface option is described using an interface pattern. The interface pattern consists of a sequence of clauses (separated by commas), each clause is a mask specified as a shell-like pattern. Interfaces are matched by their name.

An interface matches the pattern if it matches any of its clauses. If the clause begins with `-`, matching interfaces are excluded. Patterns are processed left-to-right, thus `interface "eth0", -"eth*", "*";` means eth0 and all non-ethernets.

Some protocols (namely OSPFv2 and Direct) support extended clauses that may contain a mask, a prefix, or both of them. An interface matches such clause if its name matches the mask (if specified) and its address matches the prefix (if specified). Extended clauses are used when the protocol handles multiple addresses on an interface independently.

An interface option can be used more times with different interface-specific options, in that case for given interface the first matching interface option is used.

This option is allowed in Babel, BFD, Device, Direct, OSPF, RAdv and RIP protocols. In OSPF protocol it is used in the `area` subsection.

Default: none.

Examples:

`interface "*" { type broadcast; };` - start the protocol on all interfaces with `type broadcast` option.

`interface "eth1", "eth4", "eth5" { type ptp; };` - start the protocol on enumerated interfaces with `type ptp` option.

`interface -192.168.1.0/24, 192.168.0.0/16;` - start the protocol on all interfaces that have address from 192.168.0.0/16, but not from 192.168.1.0/24.

`interface "eth*" 192.168.1.0/24;` - start the protocol on all ethernet interfaces that have address from 192.168.1.0/24.

`tx class|dscp` *number*
This option specifies the value of ToS/DS/Class field in IP headers of the outgoing protocol packets. This may affect how the protocol packets are processed by the network relative to the other network traffic. With `class` keyword, the value (0-255) is used for the whole ToS/Class octet (but two bits reserved for ECN are ignored). With `dscp` keyword, the value (0-63) is used just for the DS field in the octet. Default value is 0xc0 (DSCP 0x30 - CS6).

`tx priority` *number*
This option specifies the local packet priority. This may affect how the protocol packets are processed in the local TX queues. This option is Linux specific. Default value is 7 (highest priority, privileged traffic).

`password "`*password*`"` | *bytestring* `[ {` *password options* `} ]`
Specifies a password that can be used by the protocol as a shared secret key. Password option can be used more times to specify more passwords. If more passwords are specified, it is a protocol-dependent decision which one is really used. Specifying passwords does not mean that authentication is enabled, authentication can be enabled by separate, protocol-dependent `authentication` option.

A password can be specified as a string or as a sequence of hexadecimal digit pairs (bytestring (p. 28)).

This option is allowed in BFD, OSPF, RIP, and Babel protocols. BGP has also `password` option, but it is slightly different and described separately. Default: none.

Password option can contain section with some (not necessary all) password sub-options:

`id` *number*
ID of the password, (0-255). If it is not specified, BIRD will choose ID based on an order of the password item in the interface, starting from 1. For example, second password item in one interface will have default ID 2. ID 0 is allowed by BIRD, but some other implementations may not allow it. ID is used by some routing protocols to identify which password was used to authenticate protocol packets.

`generate from "`*time*`"`
The start time of the usage of the password for packet signing. The format of *time* is `YYYY-MM-DD [hh:mm:ss[.sss]]`.

`generate to "`*time*`"`
The last time of the usage of the password for packet signing.

`accept from "`*time*`"`
The start time of the usage of the password for packet verification.

`accept to "`*time*`"`
The last time of the usage of the password for packet verification.

`from "`*time*`"`
Shorthand for setting both `generate from` and `accept from`.

`to "`*time*`"`
Shorthand for setting both `generate to` and `accept to`.

`algorithm ( keyed md5 | keyed sha1 | hmac sha1 | hmac sha256 | hmac sha384 | hmac sha512 | blake2s128 | blake2s256 | blake2b256 | blake2b512 )`
The message authentication algorithm for the password when cryptographic authentication is enabled. The default value depends on the protocol. For RIP and OSPFv2 it is Keyed-MD5 (for compatibility), for OSPFv3 and Babel it is HMAC-SHA-256.

3.5 Channel options

Every channel belongs to a protocol and is configured inside its block. The minimal channel config is empty, then it uses default values. The name of the channel implies its nettype. Channel definitions can be inherited from protocol templates. Multiple definitions of the same channel are forbidden, but channels inherited from templates can be updated by new definitions.

`debug all|off|{ states|routes|filters [, ...] }`
Set channel debugging options. Like in protocol debugging (p. 17), channels are capable of writing trace messages about its work to the log (with category `trace`). You can either request printing of `all` trace messages or only of the selected types: `states` for channel state changes (channel going up, down, feeding, reloading etc.), `routes` for routes propagated through the channel, `filters` for details on route filtering, remaining debug flags are not used in channel debug. Default: off.

`table` *name*
Specify a table to which the channel is connected. Default: the first table of given nettype.

`preference` *expr*
Sets the preference of routes generated by the protocol and imported through this channel. Default: protocol dependent.

`import all` | `none` | `filter` *name* | `filter` { *filter commands* } | `where` *boolean filter expression*
Specify a filter to be used for filtering routes coming from the protocol to the routing table. `all` is for keeping all routes, `none` is for dropping all routes. Default: `all` (except for EBGP).

`export [ in` *prefix* `]` *filter*
This is similar to the `import` keyword, except that it works in the direction from the routing table to the protocol. If `in` keyword is used, only routes inside the given prefix are exported. Other routes are completely ignored (e.g. no logging and no statistics). Default: `none` (except for EBGP and L3VPN).

`import keep filtered` *switch*
Usually, if an import filter rejects a route, the route is forgotten. When this option is active, these routes are kept in the routing table, but they are hidden and not propagated to other protocols. But it is possible to show them using `show route filtered`. Note that this option does not work for the pipe protocol. Default: off.

`rpki reload` *switch*
Import or export filters may depend on route RPKI status (using `roa_check()` or `aspa_check()` operators). In contrast to other filter operators, this status for the same route may change as the content of ROA and ASPA tables changes. When this option is active, BIRD activates automatic reload of the appropriate subset of prefixes imported or exported by the channels whenever ROA and ASPA tables are updated (after a short settle time). When disabled, route reloads have to be requested manually. The option is ignored if neither `roa_check()` nor `aspa_check()` is used in channel filters. Note that for BGP channels, automatic reload requires import table (p. 58) or export table (p. 59) (for respective direction). Default: on.

`import limit` [*number* | `off ] [action warn | block | restart | disable]`
Specify an import route limit (a maximum number of routes imported from the protocol) and optionally the action to be taken when the limit is hit. Warn action just prints warning log message. Block action discards new routes coming from the protocol. Restart and disable actions shut the protocol down like appropriate commands. Disable is the default action if an action is not explicitly specified. Note that limits are reset during protocol reconfigure, reload or restart. Default: `off`.

`receive limit` [*number* | `off ] [action warn | block | restart | disable]`
Specify an receive route limit (a maximum number of routes received from the protocol and remembered). It works almost identically to `import limit` option, the only difference is that if `import keep filtered` option is active, filtered routes are counted towards the limit and blocked routes are forgotten, as the main purpose of the receive limit is to protect routing tables from overflow. Import limit, on the contrary, counts accepted routes only and routes blocked by the limit are handled like filtered routes. Default: `off`.

`export limit [` *number* | `off ] [action warn | block | restart | disable]`
Specify an export route limit, works similarly to the `import limit` option, but for the routes exported to the protocol. This option is experimental, there are some problems in details of its behavior – the number of exported routes can temporarily exceed the limit without triggering it during protocol reload, exported routes counter ignores route blocking and block action also blocks route updates of already accepted routes – and these details will probably change in the future. Default: `off`.

`export block` *number*
Set the minimum amount of routes exported at once when feeding or if ‘merge paths‘ or ‘secondary‘ is selected. This affects overall latency. Basically, when your export filters are very expensive, processing the whole block of routes may take too much time. In such cases, you may need to shrink this value to improve responsiveness. Default: `16384`.

This is a trivial example of RIP configured for IPv6 on all interfaces:

```
protocol rip ng {
        ipv6;
        interface "*";
}
```

This is a non-trivial example.

```
protocol rip ng {
        ipv6 {
                table mytable6;
                import filter { ... };
                export filter { ... };
                import limit 50;
        };
        interface "*";
}
```

And this is even more complicated example using templates.

```
template bgp {
        local 198.51.100.14 as 65000;

        ipv4 {
                table mytable4;
                import filter { ... };
                export none;
        };
        ipv6 {
                table mytable6;
                import filter { ... };
                export none;
        };
}

protocol bgp from  {
        neighbor 198.51.100.130 as 64496;

        # IPv4 channel is inherited as-is, while IPv6
        # channel is adjusted by export filter option
        ipv6 {
                export filter { ... };
        };
}
```

3.6 MPLS options

The MPLS domain definition is mandatory for a MPLS router. All MPLS channels and MPLS-aware protocols are associated with some MPLS domain (although usually implicitly with the sole one). In the MPLS domain definition you can configure details of MPLS label allocation. Currently, there is just one option, `label range`.

Note that the MPLS subsystem is experimental, it is likely that there will be some backward-incompatible changes in the future.

`label range` *name* { `start` *number*; `length` *number*; [...] }

Define a new label range, or redefine implicit label ranges `static` and `dynamic`. MPLS channels use configured label ranges for dynamic label allocation, while `static` label range is used for static label allocation. The label range definition must specify the extent of the range. By default, the range `static` is 16-1000, while the range `dynamic` is 1000-10000.

MPLS channel should be defined in each MPLS-aware protocol in addition to its regular channels. It is responsible for label allocation and for announcing MPLS routes to the MPLS routing table. Besides common channel options (p. 19), MPLS channels have some specific options:

`domain` *name*
Specify a MPLS domain to which this channel and protocol belongs. Default: The first defined MPLS domain.

`label range` *name*
Use specific label range for dynamic label allocation. Note that static labels always use the range `static`. Default: the range `dynamic`.

`label policy static|prefix|aggregate|vrf`
Label policy specifies how routes are grouped to forwarding equivalence classes (FECs) and how labels are assigned to them.

The policy `static` means no dynamic label allocation is done, and static labels must be set in import filters using the route attribute mpls_label (p. 35).

The policy `prefix` means each prefix uses separate label associated with that prefix. When a labeled route is updated, it keeps the label. This policy is appropriate for IGPs.

The policy `aggregate` means routes are grouped to FECs according to their next hops (including next hop labels), and one label is used for all routes in the same FEC. When a labeled route is updated, it may change next hop, change FEC and therefore change label. This policy is appropriate for BGP.

The policy `vrf` is only valid in L3VPN protocols. It uses one label for all routes from a VRF, while replacing the original next hop with lookup in the VRF.

Default: `prefix`.

This is a trivial example of MPLS setup:

```
mpls domain mdom {
        label range bgprange { start 2000; length 1000; };
}

mpls table mtab;

protocol static {
        ipv6;
        mpls;

        route 2001:db8:1:1/64 mpls 100 via 2001:db8:1:2::1/64 mpls 200;
}

protocol bgp {
        # regular channels
        ipv6 mpls { ... };
        vpn6 mpls { ... };

        # MPLS channel
        mpls {
                # domain mdom;
                # table mtab;
                label range bgprange;
                label policy aggregate;
        };

        ...
}
```

Chapter 4: Remote control

4.1 Overview

You can use the command-line client `birdc` to talk with a running BIRD. Communication is done using the appropriate UNIX domain socket. The commands can perform simple actions such as enabling/disabling of protocols, telling BIRD to show various information, telling it to show routing table filtered by filter, or asking BIRD to reconfigure. Press `?` at any time to get online help. Option `-r` can be used to enable a restricted mode of BIRD client, which allows just read-only commands (`show ...`). Option `-v` can be passed to the client, to make it dump numeric return codes along with the messages. You do not necessarily need to use `birdc` to talk to BIRD, your own applications could do that, too – the format of communication between BIRD and `birdc` is stable (see the programmer's documentation).

There is also lightweight variant of BIRD client called `birdcl`, which does not support command line editing and history and has minimal dependencies. This is useful for running BIRD in resource constrained environments, where Readline library (required for regular BIRD client) is not available.

4.2 Configuration

By default, BIRD opens `bird.ctl` UNIX domain socket and the CLI tool connects to it. If changed on the command line by the `-s` option, BIRD or the CLI tool connects there instead.

It's also possible to configure additional remote control sockets in the configuration file by `cli "name" {` *options* `};` and you can open how many sockets you wish. There are no checks whether the user configured the same socket multiple times and BIRD may behave weirdly if this happens. On shutdown, the additional sockets get removed immediately and only the main socket stays until the very end. If there are no options, the braces may be omitted.

Options:

`restrict`
: Set the socket to be restricted as if the user always sent the `restrict` command after connecting. The user may still overload the daemon by requesting insanely complex filters so you shouldn't expose this socket to public even if restricted.

`v2 attributes`
: Display the names and composition of route attributes the same way as BIRD 2 does. This is a compatibility option for easier transition from BIRD 2 to BIRD 3.

4.3 Usage

Here is a brief list of supported functions.

Note: Many commands have the *name* of the protocol instance as an argument. This argument can be omitted if there exists only a single instance.

`show status`
: Show router status, that is BIRD version, uptime and time from last reconfiguration.

`show memory`
: Show BIRD memory statistics. BIRD does not stop other computations in order to measure the used memory, hence the measured memory usage may differ from the real memory usage.

`show interfaces [summary]`
: Show the list of interfaces. For each interface, print its type, state, MTU and addresses assigned.

`show protocols [all]`
: Show list of protocol instances along with tables they are connected to and protocol status, possibly giving verbose information, if `all` is specified.

`reload bgp [in|out] [`*name*`]`
Manually request (in) or send (out) route refresh (or both) on the given BGP protocol(s).

`show ospf interface [`*name*`] ["`*interface*`"]`
Show detailed information about OSPF interfaces.

`show ospf neighbors [`*name*`] ["`*interface*`"]`
Show a list of OSPF neighbors and a state of adjacency to them.

`show ospf state [all] [`*name*`]`
Show detailed information about OSPF areas based on a content of the link-state database. It shows network topology, stub networks, aggregated networks and routers from other areas and external routes. The command shows information about reachable network nodes, use option `all` to show information about all network nodes in the link-state database.

`show ospf topology [all] [`*name*`]`
Show a topology of OSPF areas based on a content of the link-state database. It is just a stripped-down version of 'show ospf state'.

`show ospf lsadb [global | area` *id* `| link] [type` *number*`] [lsid` *id*`] [self | router` *id*`]`
`[`*name*`]`
Show contents of an OSPF LSA database. Options could be used to filter entries.

`show radv interfaces [`*name*`] ["`*interface*`"]`
Show detailed information about RAdv interfaces and associated state.

`show rip interfaces [`*name*`] ["`*interface*`"]`
Show detailed information about RIP interfaces.

`show rip neighbors [`*name*`] ["`*interface*`"]`
Show a list of RIP neighbors and associated state.

`show static [`*name*`]`
Show detailed information about static routes.

`show bfd sessions [`*name*`] [address (`*IP*`|`*prefix*`)] [(interface|dev) "`*name*`"] [ipv4|ipv6]`
`[direct|multihop] [all]`
Show information about BFD sessions. Options could be used to filter entries, or in the case of the option `all` to give verbose output.

`show symbols [table|filter|function|protocol|template|roa|`*symbol*`]`
Show the list of symbols defined in the configuration (names of protocols, routing tables etc.).

`show route [[(for|in)]` *prefix*`|for` *IP*`] [table (`*t*`|all)] [(import|export) table` *p*`.`*c*`] [filter`
f`|where` *cond*`] [(export|preexport|noexport)` *p*`] [protocol` *p*`] [(stats|count)] [`*options*`]`
Show contents of specified routing tables, that is routes, their metrics and (in case the `all` switch is given) all their attributes.

You can specify a *prefix* if you want to print routes for a specific network. If you use `for` *prefix or IP*, you'll get the entry which will be used for forwarding of packets to the given destination. Finally, if you use `in` *prefix*, you get all prefixes covered by the given prefix. By default, all routes for each network are printed with the selected one at the top, unless `primary` is given in which case only the selected route is shown.

The `show route` command can process one or multiple routing tables. The set of selected tables is determined on three levels: First, tables can be explicitly selected by `table` switch, which could be used multiple times, all tables are specified by `table all`. Second, tables can be implicitly selected by channels or protocols that are arguments of several other switches (e.g., `export`, `protocol`). Last, the set of default tables is used: `master4`, `master6` and each first table of any other network type.

There are internal tables when `(import|export) table` options are used for some channels. They can be selected explicitly with `(import|export) table` switch, specifying protocol *p* and channel name *c*.

You can also ask for printing only routes processed and accepted by a given filter (`filter` *name* or `filter` { *filter* } or matching a given condition (`where` *condition*).

The `export`, `preexport` and `noexport` switches ask for printing of routes that are exported to the specified protocol or channel. With `preexport`, the export filter of the channel is skipped. With `noexport`, routes rejected by the export filter are printed instead. Note that routes not exported for other reasons (e.g. secondary routes or routes imported from that protocol) are not printed even with `noexport`. These switches also imply that associated routing tables are selected instead of default ones.

You can also select just routes added by a specific protocol. `protocol` *p*. This switch also implies that associated routing tables are selected instead of default ones.

If BIRD is configured to keep filtered routes (see `import keep filtered` option), you can show them instead of routes by using `filtered` switch.

The `stats` switch requests showing of route statistics (the number of networks, number of routes before and after filtering). If you use `count` instead, only the statistics will be printed.

`mrt dump table` *name*|"*pattern*" `to` "*filename*" `[filter` *f*|`where` *c*`]`
Dump content of a routing table to a specified file in MRT table dump format. See MRT protocol (p. 76) for details.

`configure [soft] ["`*config file*`"] [timeout [`*number*`]]`
Reload configuration from a given file. BIRD will smoothly switch itself to the new configuration, protocols are reconfigured if possible, restarted otherwise. Changes in filters usually lead to restart of affected protocols.

The previous configuration is saved and the user can switch back to it with configure undo (p. 25) command. The old saved configuration is released (even if the reconfiguration attempt fails due to e.g. a syntax error).

If `soft` option is used, changes in filters does not cause BIRD to restart affected protocols, therefore already accepted routes (according to old filters) would be still propagated, but new routes would be processed according to the new filters.

If `timeout` option is used, config timer is activated. The new configuration could be either confirmed using `configure confirm` command, or it will be reverted to the old one when the config timer expires. This is useful for cases when reconfiguration breaks current routing and a router becomes inaccessible for an administrator. The config timeout expiration is equivalent to `configure undo` command. The timeout duration could be specified, default is 300 s.

`configure confirm`
Deactivate the config undo timer and therefore confirm the current configuration.

`configure undo`
Undo the last configuration change and smoothly switch back to the previous (stored) configuration. If the last configuration change was soft, the undo change is also soft. There is only one level of undo, but in some specific cases when several reconfiguration requests are given immediately in a row and the intermediate ones are skipped then the undo also skips them back.

`configure check ["`*config file*`"]`
Read and parse given config file, but do not use it. useful for checking syntactic and some semantic validity of an config file.

`enable|disable|restart` *name*|"*pattern*"|`all`
Enable, disable or restart a given protocol instance, instances matching the *pattern* or `all` instances.

`reload filters [in|out] (`*name*|"*pattern*"|`all) [partial prefix]`
Reload a given protocol instance, that means re-import routes from the protocol instance and re-export preferred routes to the instance. If `in` or `out` options are used, the command is restricted to one direction (re-import or re-export).

This command is useful if appropriate filters have changed but the protocol instance was not restarted (or reloaded), therefore it still propagates the old set of routes. For example when `configure soft` command was used to change filters.

If `partial prefix` option is used, only corresponding routes are reloaded. Protocol BGP does partial reload only if it has import table enabled, otherwise partial reload for BGP is refused.

Re-export always succeeds, but re-import is protocol-dependent and might fail (for example, if BGP neighbor does not support route-refresh extension). In that case, re-export is also skipped. Note that for the pipe protocol, both directions are always reloaded together (`in` or `out` options are ignored in that case).

`timeformat "`*format1*`" [`*limit* `"`*format2*`"]`
Override format of date/time used by BIRD in this CLI session.

Meaning of "*format1*", *limit*, and "*format2*" is the same as in the timeformat (p. 14) configuration option. Also, the same `iso ...` shorthands may be used.

`down`
Shut BIRD down.

`graceful restart`
Shut BIRD down for graceful restart. See graceful restart (p. 10) section for details.

`debug` *protocol*|*pattern*|`all all`|`off`|{ `states`|`routes`|`filters`|`events`|`packets [, ...]` }
Control protocol debugging.

`dump resources`|`sockets`|`ao keys`|`events`|`interfaces`|`neighbors`|`attributes`|`attribute stats`|`routes`|`protocols "`*file*`"`
Creates the given file (it must not exist) and dumps contents of internal data structures there. By sending SIGUSR1, you get all of these concatenated to `bird.dump` in the current directory. The file is only readable for the user running the daemon. The format of dump files is internal and could change in the future without any notice.

`echo all`|`off`|{ *list of log classes* } `[` *buffer-size* `]`
Control echoing of log messages to the command-line output. See log option (p. 12) for a list of log classes.

`eval` *expr*
Evaluate given expression.

Chapter 5: Filters

5.1 Introduction

BIRD contains a simple programming language. (No, it can't yet read mail :-). There are two objects in this language: filters and functions. Filters are interpreted by BIRD core when a route is being passed between protocols and routing tables. The filter language contains control structures such as if's and switches, but it allows no loops. An example of a filter using many features can be found in `filter/test.conf`.

Filter gets the route, looks at its attributes and modifies some of them if it wishes. At the end, it decides whether to pass the changed route through (using `accept`) or whether to `reject` it. A simple filter looks like this:

```
filter not_too_far
{
        int var;
        if defined( rip_metric ) then
                var = rip_metric;
        else {
                var = 1;
                rip_metric = 1;
        }
        if rip_metric > 10 then
                reject "RIP metric is too big";
        else
                accept "ok";
}
```

As you can see, a filter has a header, a list of local variables, and a body. The header consists of the `filter` keyword followed by a (unique) name of filter. The list of local variables consists of *type name*`;` pairs where each pair declares one local variable. The body consists of { *statements* }. Each *statement* is terminated by a `;`. You can group several statements to a single compound statement by using braces ({ *statements* }) which is useful if you want to make a bigger block of code conditional.

BIRD supports functions, so that you don not have to repeat the same blocks of code over and over. Functions can have zero or more parameters and they can have local variables. If the function returns value, then you should always specify its return type. Direct recursion is possible. Function definitions look like this:

```
function name() -> int
{
        int local_variable;
        int another_variable = 5;
        return 42;
}

function with_parameters(int parameter) -> pair
{
        print parameter;
        return (1, 2);
}
```

Like in C programming language, variables are declared inside function body, either at the beginning, or mixed with other statements. Declarations may contain initialization. You can also declare variables in nested blocks, such variables have scope restricted to such block. There is a deprecated syntax to declare variables after the `function` line, but before the first {. Functions are called like in C: `name(); with_parameters(5);`. Function may return values using the `return` *[expr]* command. Returning a value exits from current function (this is similar to C).

Filters are defined in a way similar to functions except they cannot have explicit parameters and cannot return. They get a route table entry as an implicit parameter, it is also passed automatically to any functions called. The filter must terminate with either `accept` or `reject` statement. If there is a runtime error in filter, the route is rejected.

A nice trick to debug filters is to use `show route filter` *name* from the command line client. An example session might look like:

```
pavel@bug:~/bird$ ./birdc -s bird.ctl
BIRD 0.0.0 ready.
bird> show route
10.0.0.0/8         dev eth0 [direct1 23:21] (240)
195.113.30.2/32    dev tunl1 [direct1 23:21] (240)
127.0.0.0/8        dev lo [direct1 23:21] (240)
bird> show route ?
show route [<prefix>] [table <t>] [filter <f>] [all] [primary]...
bird> show route filter { if 127.0.0.5 ~ net then accept; }
127.0.0.0/8        dev lo [direct1 23:21] (240)
bird>
```

5.2 Data types

Each variable and each value has certain type. Booleans, integers and enums are incompatible with each other (that is to prevent you from shooting oneself in the foot).

`bool`
: This is a boolean type, it can have only two values, `true` and `false`. Boolean is the only type you can use in `if` statements.

`int`
: This is a general integer type. It is an unsigned 32bit type; i.e., you can expect it to store values from 0 to 4294967295. Overflows are not checked. You can use `0x1234` syntax to write hexadecimal values.

`pair`
: This is a pair of two short integers. Each component can have values from 0 to 65535. Literals of this type are written as `(1234,5678)`. The same syntax can also be used to construct a pair from two arbitrary integer expressions (for example `(1+2,a)`).

  Operators `.asn` and `.data` can be used to extract corresponding components of a pair: `(`*asn*`,` *data*`)`.

`quad`
: This is a dotted quad of numbers used to represent router IDs (and others). Each component can have a value from 0 to 255. Literals of this type are written like IPv4 addresses.

`string`
: This is a string of characters. There are no ways to modify strings in filters. You can pass them between functions, assign them to variables of type `string`, print such variables, use standard string comparison operations (e.g. `=, !=,` $<$, $>$, $<=$, $>=$), and concatenate two strings with `append(`*A*`,` *B*`)` function or `++` operator.

  String literals are written as `"This is a string constant"`. Additionally matching (`~, !~`) operators could be used to match a string value against a shell pattern (represented also as a string).

`bytestring`
: This is a sequence of arbitrary bytes. There are no ways to modify bytestrings in filters. You can pass them between functions, assign them to variables of type `bytestring`, print such values, compare bytestings (`=, !=`), and concatenate two bytestrings with `append(`*A*`,` *B*`)` function or `++` operator.

  Bytestring literals are written as a sequence of hexadecimal digit pairs, optionally colon-separated. A bytestring specified this way must be either at least 16 bytes (32 digits) long, or prefixed by the `hex:` prefix: `01:23:45:67:89:ab:cd:ef:01:23:45:67:89:ab:cd:ef`, `0123456789abcdef0123456789abcdef`, `hex:`, `hex:12:34:56`, `hex:12345678`.

A bytestring can be made from a hex string using `from_hex()` function. Source strings can use any number of dots, colons, hyphens and spaces as byte separators: `from_hex(" 12.34 56:78 ab-cd-ef ")`.

`ip`

This type can hold a single IP address. The IPv4 addresses are stored as IPv4-Mapped IPv6 addresses so one data type for both of them is used. Whether the address is IPv4 or not may be checked by `.is_v4` which returns a `bool`. IP addresses are written in the standard notation (`10.20.30.40` or `fec0:3:4::1`). You can apply special operator `.mask(`*number*`)` on values of type ip. It masks out all but first *number* bits from the IP address. So `1.2.3.4.mask(8) = 1.0.0.0` is true.

`mac`

This type can hold a single MAC address. MAC addresses are written in the usual notation (e.g. `62:68:7f:d9:c6:ec`).

`prefix`

This type can hold a network prefix consisting of IP address, prefix length and several other values. This is the key in route tables.

Prefixes may be of several types, which can be determined by the special operator `.type`. The type may be:

`NET_IP4` and `NET_IP6` prefixes hold an IP prefix. The literals are written as *ipaddress*`/`*pxlen*. There are two special operators on these: `.ip` which extracts the IP address from the pair, and `.len`, which separates prefix length from the pair. So `1.2.0.0/16.len = 16` is true.

`NET_IP6_SADR` nettype holds both destination and source IPv6 prefix. The literals are written as *ipaddress*`/`*pxlen* `from` *ipaddress*`/`*pxlen*, where the first part is the destination prefix and the second art is the source prefix. They support the same operators as IP prefixes, but just for the destination part. They also support `.src` and `.dst` operators to get respective parts of the address as separate `NET_IP6` values.

`NET_VPN4` and `NET_VPN6` prefixes hold an IP prefix with VPN Route Distinguisher (RFC 4364). They support the same special operators as IP prefixes, and also `.rd` which extracts the Route Distinguisher. Their literals are written as *rd* *ipprefix*

`NET_ROA4` and `NET_ROA6` prefixes hold an IP prefix range together with an ASN. They support the same special operators as IP prefixes, and also `.maxlen` which extracts maximal prefix length, and `.asn` which extracts the ASN.

`NET_ASPA` prefixes hold an ASN, which can be extracted by `.asn` operator.

`NET_FLOW4` and `NET_FLOW6` hold an IP prefix together with a flowspec rule. Filters currently do not support much flowspec parsing, only `.src` and `.dst` operators to get source and destination parts of the flowspec as separate `NET_IP4` / `NET_IP6` values.

`NET_ETH` prefixes holds a MAC address and a VLAN ID, which can be extracted by operators `.mac` and `.vlan_id`, respectively.

`NET_MPLS` holds a single MPLS label and its handling is currently not implemented.

`NET_EVPN` is a complex nettype that holds reachability information for Ethernet VPN. It has several subtypes (`NET_EVPN_EAD`, `NET_EVPN_MAC`, `NET_EVPN_IMET`, `NET_EVPN_ES`), which can be determined by the operator `.evpn_type`. Fields and related operators are: Route Distinguisher (`.rd`), MAC address (`.mac`), IP address (`.ip`), EVPN tag (`.evpn_tag`), Ethernet Segment Identifier (`.evpn_esi`), router IP (`.router_ip`). Note that only fields that are a part of primary key for given route subtype can be accessed by these operators (e.g. `.evpn_esi` can be used for `NET_EVPN_ES`, but not for `NET_EVPN_MAC`, where ESI is stored in evpn_esi (p. 35) attribute), remaining attributes are stored outside prefix as other route attributes. See EVPN routes (p. 8) for description of specific subtypes.

`NET_NEIGHBOR` holds a single IP address and an interface index. The IP address can be accessed by operator `.ip`, while the access to the interface index is not implemented.

`rd`

This is a route distinguisher according to RFC 4364. There are three kinds of RDs: *asn*`:`*32bit int*, *asn4*`:`*16bit int* and *IPv4 address*`:`*32bit int*

`ec`

This is a specialized type used to represent BGP extended community values. It is essentially a 64bit value, literals of this type are usually written as (*kind*, *key*, *value*), where `kind` is a kind of extended community (e.g. `rt` / `ro` for a route target / route origin communities), the format and possible values of `key` and `value` are usually integers, but it depends on the used kind. Similarly to pairs, ECs can be constructed using expressions for `key` and `value` parts, (e.g. `(ro, myas, 3*10)`, where `myas` is an integer variable).

`lc`

This is a specialized type used to represent BGP large community values. It is essentially a triplet of 32bit values, where the first value is reserved for the AS number of the issuer, while meaning of remaining parts is defined by the issuer. Literals of this type are written as `(123, 456, 789)`, with any integer values. Similarly to pairs, LCs can be constructed using expressions for its parts, (e.g. `(myas, 10+20, 3*10)`, where `myas` is an integer variable).

Operators `.asn`, `.data1`, and `.data2` can be used to extract corresponding components of LCs: (*asn*, *data1*, *data2*).

`int|pair|quad|ip|prefix|ec|lc|rd|enum set`

Filters recognize several types of sets. Sets are similar to strings: you can pass them around but you cannot modify them. Literals of type `int set` look like `[ 1, 2, 5..7 ]`. As you can see, both simple values and ranges are permitted in sets.

For pair sets, expressions like `(123,*)` can be used to denote ranges (in that case `(123,0)..(123,65535)`). You can also use `(123,5..100)` for range `(123,5)..(123,100)`. You can also use `*` and `a..b` expressions in the first part of a pair, note that such expressions are translated to a set of intervals, which may be memory intensive. E.g. `(*,4..20)` is translated to `(0,4..20)`, `(1,4..20)`, `(2,4..20)`, ... `(65535, 4..20)`.

EC sets use similar expressions like pair sets, e.g. `(rt, 123, 10..20)` or `(ro, 123, *)`. Expressions requiring the translation (like `(rt, *, 3)`) are not allowed (as they usually have 4B range for ASNs).

Also LC sets use similar expressions like pair sets. You can use ranges and wildcards, but if one field uses that, more specific (later) fields must be wildcards. E.g., `(10, 20..30, *)` or `(10, 20, 30..40)` is valid, while `(10, *, 20..30)` or `(10, 20..30, 40)` is not valid.

You can also use named constants or compound expressions for non-prefix set values. However, it must be possible to evaluate these expressions before daemon boots. So you can use only constants inside them. Also, in case of compound expressions, they require parentheses around them. E.g.

```
define one=1;
define myas=64500;

int set odds = [ one, (2+1), (6-one), (2*2*2-1), 9, 11 ];
pair set ps = [ (1,one+one), (3,4)..(4,8), (5,*), (6,3..6), (7..9,*) ];
ec set es = [ (rt, myas, *), (rt, myas+2, 0..16*16*16-1) ];
```

Sets of prefixes are special: their literals does not allow ranges, but allows prefix patterns that are written as *ipaddress*`/`*pxlen*{*low*`,`*high*}. Prefix *ip1*`/`*len1* matches prefix pattern *ip2*`/`*len2*{*l*`,`*h*} if the first `min(len1, len2)` bits of `ip1` and `ip2` are identical and `l <= len1 <= h`. A valid prefix pattern has to satisfy `low <= high`, but `pxlen` is not constrained by `low` or `high`. Obviously, a prefix matches a prefix set literal if it matches any prefix pattern in the prefix set literal.

There are also two shorthands for prefix patterns: *address*`/`*len*`+` is a shorthand for *address*`/`*len*{*len*`,`*maxlen*} (where *maxlen* is 32 for IPv4 and 128 for IPv6), that means network prefix *address*`/`*len* and all its subnets. *address*`/`*len*`-` is a shorthand for *address*`/`*len*{`0`,*len*}, that means network prefix *address*`/`*len* and all its supernets (network prefixes that contain it).

For example, `[ 1.0.0.0/8, 2.0.0.0/8+, 3.0.0.0/8-, 4.0.0.0/8{16,24} ]` matches prefix `1.0.0.0/8`, all subprefixes of `2.0.0.0/8`, all superprefixes of `3.0.0.0/8` and prefixes `4.X.X.X` whose prefix length is 16 to 24. `[ 0.0.0.0/0{20,24} ]` matches all prefixes (regardless of IP address) whose prefix length is 20 to 24, `[ 1.2.3.4/32- ]` matches any prefix that contains IP address `1.2.3.4`. `1.2.0.0/16 ~ [ 1.0.0.0/8{15,17} ]` is true, but `1.0.0.0/16 ~ [ 1.0.0.0/8- ]` is false.

Cisco-style patterns like `10.0.0.0/8 ge 16 le 24` can be expressed in BIRD as `10.0.0.0/8{16,24}`, `192.168.0.0/16 le 24` as `192.168.0.0/16{16,24}` and `192.168.0.0/16 ge 24` as `192.168.0.0/16{24,32}`.

It is not possible to mix IPv4 and IPv6 prefixes in a prefix set. It is currently possible to mix IPv4 and IPv6 addresses in an ip set, but that behavior may change between versions without any warning; don't do it unless you are more than sure what you are doing. (Really, don't do it.)

`enum`

Enumeration types are fixed sets of possibilities. You can't define your own variables of such type, but some route attributes are of enumeration type. Enumeration types are incompatible with each other.

`bgppath`

BGP path is a list of autonomous system numbers. You can't write literals of this type. There are several special operators on bgppaths:

P`.first` returns the first ASN (the neighbor ASN) in path P.

P`.last` returns the last ASN (the source ASN) in path P.

P`.last_nonaggregated` returns the last ASN in the non-aggregated part of the path P.

Both `first` and `last` return zero if there is no appropriate ASN, for example if the path contains an AS set element as the first (or the last) part. If the path ends with an AS set, `last_nonaggregated` may be used to get last ASN before any AS set.

P`.len` returns the length of path P.

P`.empty` makes the path P empty. Can't be used as a value, always modifies the object.

P`.prepend(`A`)` prepends ASN A to path P and returns the result.

P`.delete(`A`)` deletes all instances of ASN A from from path P and returns the result. A may also be an integer set, in that case the operator deletes all ASNs from path P that are also members of set A.

P`.filter(`A`)` deletes all ASNs from path P that are not members of integer set A, and returns the result. I.e., `filter` do the same as `delete` with inverted set A.

Methods `prepend`, `delete` and `filter` keep the original object intact as long as you use the result in any way. You can also write e.g. P`.prepend(`A`);` as a standalone statement. This variant does modify the original object with the result of the operation.

`bgpmask`

BGP masks are patterns used for BGP path matching (using `path ~ [= 2 3 5 * =]` syntax). The masks resemble wildcard patterns as used by UNIX shells. Autonomous system numbers match themselves, `*` matches any (even empty) sequence of arbitrary AS numbers and `?` matches one arbitrary AS number. For example, if `bgp_path` is 4 3 2 1, then: `bgp_path ~ [= * 4 3 * =]` is true, but `bgp_path ~ [= * 4 5 * =]` is false. There is also `+` operator which matches one or multiple instances of previous expression, e.g. `[= 1 2+ 3 =]` matches both path 1 2 3 and path 1 2 2 2 3, but not 1 3 nor 1 2 4 3. Note that while `*` and `?` are wildcard-style operators, `+` is regex-style operator.

BGP mask expressions can also contain integer expressions enclosed in parenthesis and integer variables, for example `[= * 4 (1+2) a =]`. You can also use ranges (e.g. `[= * 3..5 2 100..200 * =]`) and sets (e.g. `[= 1 2 [3, 5, 7] * =]`).

`clist`

Clist is similar to a set, except that unlike other sets, it can be modified. The type is used for community list (a set of pairs) and for cluster list (a set of quads). There exist no literals of this type. There are special operators on clists:

C`.len` returns the length of clist C.

C`.empty` makes the list C empty. Can't be used as a value, always modifies the object.

C`.add(`P`)` adds pair (or quad) P to clist C and returns the result. If item P is already in clist C, it does nothing. P may also be a clist, in that case all its members are added; i.e., it works as clist union.

C`.delete(`P`)` deletes pair (or quad) P from clist C and returns the result. If clist C does not contain item P, it does nothing. P may also be a pair (or quad) set, in that case the operator deletes all items

from clist C that are also members of set P. Moreover, P may also be a clist, which works analogously; i.e., it works as clist difference.

C`.filter(`P`)` deletes all items from clist C that are not members of pair (or quad) set P, and returns the result. I.e., `filter` do the same as `delete` with inverted set P. P may also be a clist, which works analogously; i.e., it works as clist intersection.

Methods `add`, `delete` and `filter` keep the original object intact as long as you use the result in any way. You can also write e.g. P`.add(`A`);` as a standalone statement. This variant does modify the original object with the result of the operation.

C`.min` returns the minimum element of clist C.

C`.max` returns the maximum element of clist C.

Operators `.min`, `.max` can be used together with `filter` to extract the community from the specific subset of communities (e.g. localpref or prepend) without the need to check every possible value (e.g. `filter(bgp_community, [(23456, 1000..1099)]).min`).

`eclist`
: Eclist is a data type used for BGP extended community lists. Eclists are very similar to clists, but they are sets of ECs instead of pairs. The same operations (like `add`, `delete` or `~` and `!~` membership operators) can be used to modify or test eclists, with ECs instead of pairs as arguments.

`lclist`
: Lclist is a data type used for BGP large community lists. Like eclists, lclists are very similar to clists, but they are sets of LCs instead of pairs. The same operations (like `add`, `delete` or `~` and `!~` membership operators) can be used to modify or test lclists, with LCs instead of pairs as arguments.

5.3 Operators

The filter language supports common integer operators `(+,-,*,/)`, parentheses `(a*(b+c))`, comparison `(a=b, a!=b, a<b, a>=b)`.
Logical operations include unary not (`!`), and (`&&`), and or (||).
Strings and bytestrings can be concatenated with `++` operator.
Special operators include (`~`, `!~`) for "is (not) element of a set" operation - it can be used on:

- element and set of elements of the same type (returning true if element is contained in the given set)
- two strings (returning true if the first string matches a shell-like pattern stored in the second string)
- IP and prefix (returning true if IP is within the range defined by that prefix)
- prefix and prefix (returning true if the first prefix is more specific than the second one)
- bgppath and bgpmask (returning true if the path matches the mask)
- number and bgppath (returning true if the number is in the path)
- bgppath and int (number) set (returning true if any ASN from the path is in the set)
- pair/quad and clist (returning true if the pair/quad is element of the clist)
- clist and pair/quad set (returning true if there is an element of the clist that is also a member of the pair/quad set).

There are also operators related to RPKI infrastructure used to run RFC 6483 route origin validation and (draft) AS path validation.

- `roa_check(`*table*`)` checks the current route in the specified ROA table and returns `ROA_UNKNOWN`, `ROA_INVALID` or `ROA_VALID`, if the validation result is unknown, invalid, or valid, respectively. The result is valid if there is a matching ROA, it is invalid if there is either matching ROA with a different ASN, or any covering ROA with shorter maximal prefix length.

- `roa_check(`*table*`, `*prefix*`, `*asn*`)` is an explicit version of the ROA check if the user for whatever reason needs to check a different prefix or different ASN than the default one. The equivalent call of the short variant is `roa_check(`*table*`, net, bgp_path.last)` and it is faster to call the short variant.
- `aspa_check_downstream(`*table*`)` checks the current route in the specified ASPA table and returns `ASPA_UNKNOWN`, `ASPA_INVALID`, or `ASPA_VALID` if the validation result is unknown, invalid, or valid, respectively. The result is valid if there is a full coverage of matching ASPA records according to the Algorithm for Downstream Paths by the (draft). This operator is not present if BGP is not compiled in.
- `aspa_check_upstream(`*table*`)` checks the current route in the specified ASPA table as the former operator, but it applies the (stricter) Algorithm for Upstream Paths by the (draft). This operator is not present if BGP is not compiled in.
- `aspa_check(`*table*`, `*path*`, `*is_upstream*`)` is an explicit version of the former two ASPA check operators. The equivalent of `aspa_check_downstream` is `aspa_check(`*table*`, bgp_path, false)` and for `aspa_check_upstream` it is `aspa_check(`*table*`, bgp_path, true)`. Note: the ASPA check does not include the local ASN in the AS path. Also, `ASPA_INVALID` is returned for an empty AS path or for AS path containing `CONFED_SET` or `CONFED_SEQUENCE` blocks, as the (draft) stipulates.

The following example checks for ROA and ASPA on routes from a customer:

```
roa6 table r6;
aspa table at;
attribute int valid_roa;
attribute int valid_aspa;

filter customer_check {
  case roa_check(r6) {
    ROA_INVALID: reject "Invalid ROA";
    ROA_VALID: valid_roa = 1;
  }

  case aspa_check_upstream(at) {
    ASPA_INVALID: reject "Invalid ASPA";
    ASPA_VALID: valid_aspa = 1;
  }

  accept;
}
```

5.4 Control structures

Filters support several control structures: conditions, for loops and case switches.

Syntax of a condition is: `if` *boolean expression* `then` *commandT*`;` `else` *commandF*`;` and you can use { *command1*`;` *command2*`;` *...* } instead of either command. The `else` clause may be omitted. If the *boolean expression* is true, *commandT* is executed, otherwise *commandF* is executed.

For loops allow to iterate over elements in compound data like BGP paths or community lists. The syntax is: `for [` *type* `]` *variable* `in` *expr* `do` *command*`;` and you can also use compound command like in conditions. The expression is evaluated to a compound data, then for each element from such data the command is executed with the item assigned to the variable. A variable may be an existing one (when just name is used) or a locally defined (when type and name is used). In both cases, it must have the same type as elements.

The `case` is similar to case from Pascal. Syntax is `case` *expr* { `else:` | *set_body_expr* `/:` *statement* `;` `[... ]` }. The expression after `case` can be of any type that could be a member of a set, while the *set_body_expr* before `:` can be anything (constants, intervals, expressions) that could be a part of a set literal. One exception is prefix type, which can be used in sets bud not in `case` structure. Multiple commands must be grouped by {}. If *expr* matches one of the `:` clauses, the statement or block after it is executed. If *expr* matches neither of the `:` clauses, the statement or block after `else:` is executed.

Here is example that uses `if` and `case` structures:

```
if 1234 = i then printn "."; else {
        print "not 1234";
        print "You need {} around multiple commands";
}

for int asn in bgp_path do {
        printn "ASN: ", asn;
        if asn < 65536 then print " (2B)"; else print " (4B)";
}

case arg1 {
        2: { print "two"; print "Multiple commands must brace themselves."; }
        3 .. 5: print "three to five";
        else: print "something else";
}
```

5.5 Route attributes

A filter is implicitly passed a route, and it can access its attributes just like it accesses variables. There are common route attributes, protocol-specific route attributes and custom route attributes. Most common attributes are mandatory (always defined), while remaining are optional. Attempts to access undefined attribute result in a runtime error; you can check if an attribute is defined by using the `defined(` *attribute* `)` operator. One notable exception to this rule are attributes of bgppath and *clist types, where undefined value is regarded as empty bgppath/*clist for most purposes.

Attributes can be defined by just setting them in filters. Custom attributes have to be first declared by attribute (p. 14) global option. You can also undefine optional attribute back to non-existence by using the `unset(` *attribute* `)` operator.

Common route attributes are:

prefix `net`
: The network prefix or anything else the route is talking about. The primary key of the routing table. Read-only. (See the chapter about routes (p. 6).)

int `preference`
: Preference of the route.

ip `from`
: The router which the route has originated from.

ip `gw`
: Next hop packets routed using this route should be forwarded to.

string `proto`
: The name of the protocol which the route has been imported from. Read-only.

enum `source`
: what protocol has told me about this route. Possible values: `RTS_STATIC`, `RTS_INHERIT`, `RTS_DEVICE`, `RTS_RIP`, `RTS_OSPF`, `RTS_OSPF_IA`, `RTS_OSPF_EXT1`, `RTS_OSPF_EXT2`, `RTS_BGP`, `RTS_PIPE`, `RTS_BABEL`.

enum `dest`
: Type of destination the packets should be sent to (`RTD_ROUTER` for forwarding to a neighboring router, `RTD_DEVICE` for routing to a directly-connected network, `RTD_MULTIPATH` for multipath destinations, `RTD_BLACKHOLE` for packets to be silently discarded, `RTD_UNREACHABLE`, `RTD_PROHIBIT` for packets that should be returned with ICMP host unreachable / ICMP administratively prohibited messages). Can be changed, but only to `RTD_BLACKHOLE`, `RTD_UNREACHABLE` or `RTD_PROHIBIT`.

string `ifname`
Name of the outgoing interface. Sink routes (like blackhole, unreachable or prohibit) and multipath routes have no interface associated with them, so `ifname` returns an empty string for such routes. Setting it would also change route to a direct one (remove gateway).

int `ifindex`
Index of the outgoing interface. System wide index of the interface. May be used for interface matching, however indexes might change on interface creation/removal. Zero is returned for routes with undefined outgoing interfaces. Read-only.

int `weight`
Multipath weight of route next hops. Valid values are 1-256. Reading returns the weight of the first next hop, setting it sets weights of all next hops to the specified value. Therefore, this attribute is not much useful for manipulating individual next hops of an ECMP route, but can be used in BGP multipath setup to set weights of individual routes that are merged to one ECMP route during export to the Kernel protocol (with active marge paths (p. 72) option).

int `gw_mpls`
Outgoing MPLS label attached to route (i.e., incoming MPLS label on the next hop router for this label-switched path). Reading returns the label value and setting it sets it to the start of the label stack. Setting implicit-NULL label (3) disables the MPLS label stack. Only the first next hop and only one label in the label stack supported right now. This is experimental option, will be likely changed in the future to handle full MPLS label stack.

int `igp_metric`
The optional attribute that can be used to specify a distance to the network for routes that do not have a native protocol metric attribute (like `ospf_metric1` for OSPF routes). It is used to fill in local_metric (p. 35) attribute when recursive nexthops are resolved, so that BGP can compare internal distances to boundary routers (see below).

int `local_metric`
Attribute storing the route's metric to the resolved recursive nexthop, mainly used by BGP best route selection. Also known as "interior cost" in RFC 4271. It's the copy of the underlying protocol's metric, e.g. `ospf_metric1`, or the generic `igp_metric`. For recursive nexthops, setting the `local_metric` attribute in the import filters is futile, as the nexthop resolver overwrites it. Note: In versions 3.0.x and 3.1.x, the `igp_metric` attribute was confusingly used for this purpose.

int `mpls_label`
Local MPLS label attached to the route. This attribute is produced by MPLS-aware protocols for labeled routes. It can also be set in import filters to assign static labels, but that also requires static MPLS label policy.

enum `mpls_policy`
For MPLS-aware protocols, this attribute defines which MPLS label policy (p. 22) will be used for the route. It can be set in import filters to change it on per-route basis. Valid values are `MPLS_POLICY_NONE` (no label), `MPLS_POLICY_STATIC` (static label), `MPLS_POLICY_PREFIX` (per-prefix label), `MPLS_POLICY_AGGREGATE` (aggregated label), and `MPLS_POLICY_VRF` (per-VRF label). See MPLS label policy (p. 22) for details.

int `mpls_class`
When MPLS label policy (p. 22) is set to `aggregate`, it may be useful to apply more fine-grained aggregation than just one based on next hops. When routes have different value of this attribute, they will not be aggregated under one local label even if they have the same next hops.

bytestring `evpn_esi`
EVPN Ethernet Segment Identifier (ESI). Some EVPN route (p. 8) types (e.g. EVPN EAD routes) have ESI as a part of primary key (p. 29). Others (e.g. EVPN MAC routes) have ESI as this optional route attribute. BIRD does not currently use this attribute, but can pass it as a BGP route reflector.

Protocol-specific route attributes are described in the corresponding protocol sections.

5.6 Other statements

The following statements are available:

variable `=` *expr*
: Set variable (or route attribute) to a given value.

`accept`|`reject` `[` *expr* `]`
: Accept or reject the route, possibly printing *expr*.

`return` *expr*
: Return *expr* from the current function, the function ends at this point.

`print`|`printn` *expr* `[`, *expr...*`]`
: Prints given expressions; useful mainly while debugging filters. The `printn` variant does not terminate the line.

Chapter 6: Protocols

6.1 Aggregator

6.1.1 Introduction

The Aggregator protocol explicitly merges routes by the given rules. There are four phases of aggregation. First routes are filtered, then sorted into buckets, then buckets are merged and finally the results are filtered once again. Aggregating an already aggregated route is forbidden.

This is an experimental protocol, use with caution.

6.1.2 Configuration

`table` *table*
: The table from which routes are exported to get aggregated.

`export` *...*
: A standard channel's `export` clause, defining which routes are accepted into aggregation.

`aggregate on` *expr* | *attribute* `[`*, ...*`]`
: All the given filter expressions and route attributes are evaluated for each route. Then routes are sorted into buckets where *all* values are the same. Note: due to performance reasons, all filter expressions must return a compact type, e.g. integer, a BGP (standard, extended, large) community or an IP address. If you need to compare e.g. modified AS Paths in the aggregation rule, you can define a custom route attribute and set this attribute in the export filter. For now, it's mandatory to say `net` here, we can't merge prefixes yet.

`merge by` { *filter code* }
: The given filter code has an extra symbol defined: `routes`. By iterating over `routes`, you get all the routes in the bucket and you can construct your new route. All attributes selected in `aggregate on` are already set to the common values. For now, it's not possible to use a named filter here. You have to finalize the route by calling `accept`.

`import` *...*
: Filter applied to the route after `merge by`. Here you can use a named filter.

`peer table` *table*
: The table to which aggregated routes are imported. It may be the same table as `table`.

6.1.3 Example

```
protocol aggregator {
  table master6;
  export where defined(bgp_path);
  /* Merge all routes with the same AS Path length */
  aggregate on net, bgp_path.len;
  merge by {
    for route r in routes do {
      if ! defined(bgp_path) then { bgp_path = r.bgp_path }
      bgp_community = bgp_community.add(r.bgp_community);
    }
    accept;
  };
  import all;
  peer table agr_result;
}
```

6.2 Babel

6.2.1 Introduction

The Babel protocol (RFC 8966) is a loop-avoiding distance-vector routing protocol that is robust and efficient both in ordinary wired networks and in wireless mesh networks. Babel is conceptually very simple in its operation and "just works" in its default configuration, though some configuration is possible and in some cases desirable.

The Babel protocol is dual stack; i.e., it can carry both IPv4 and IPv6 routes over the same IPv6 transport. For sending and receiving Babel packets, only a link-local IPv6 address is needed.

BIRD implements an extension for IPv6 source-specific routing (SSR or SADR), but must be configured accordingly to use it. SADR-enabled Babel router can interoperate with non-SADR Babel router, but the later would ignore routes with specific (non-zero) source prefix.

6.2.2 Configuration

The Babel protocol support both IPv4 and IPv6 channels; both can be configured simultaneously. It can also be configured with IPv6 SADR (p. 7) channel instead of regular IPv6 channel, in such case SADR support is enabled. Babel supports no global configuration options apart from those common to all other protocols, but supports the following per-interface configuration options:

```
protocol babel [<name>] {
        ipv4 { <channel config> };
        ipv6 [sadr] { <channel config> };
        randomize router id <switch>;
        interface <interface pattern> {
                type wired|wireless|tunnel;
                rxcost <number>;
                limit <number>;
                hello interval <time>;
                update interval <time>;
                port <number>;
                tx class|dscp <number>;
                tx priority <number>;
                rx buffer <number>;
                tx length <number>;
                check link <switch>;
                next hop ipv4 <address>;
                next hop ipv6 <address>;
                next hop prefer native|ipv6;
                extended next hop <switch>;
                rtt cost <number>;
                rtt min <time>;
                rtt max <time>;
                rtt decay <number>;
                send timestamps <switch>;
                authentication none|mac [permissive];
                password "<text>";
                password "<text>" {
                        id <number>;
                        generate from "<date>";
                        generate to "<date>";
                        accept from "<date>";
                        accept to "<date>";
                        from "<date>";
                        to "<date>";
                        algorithm ( hmac sha1 | hmac sha256 | hmac sha384 |
```

```
                hmac sha512 | blake2s128 | blake2s256 | blake2b256 | blake2b512 );
                        };
                };
        }
```

`ipv4` | `ipv6 [sadr]` *channel config*
The supported channels are IPv4, IPv6, and IPv6 SADR.

`randomize router id` *switch*
If enabled, Bird will randomize the top 32 bits of its router ID whenever the protocol instance starts up. If a Babel node restarts, it loses its sequence number, which can cause its routes to be rejected by peers until the state is cleared out by other nodes in the network (which can take on the order of minutes). Enabling this option causes Bird to pick a random router ID every time it starts up, which avoids this problem at the cost of not having stable router IDs in the network. Default: no.

`type wired|wireless|tunnel`
This option specifies the interface type: Wired, wireless or tunnel. On wired interfaces a neighbor is considered unreachable after a small number of Hello packets are lost, as described by `limit` option. On wireless interfaces the ETX link quality estimation technique is used to compute the metrics of routes discovered over this interface. This technique will gradually degrade the metric of routes when packets are lost rather than the more binary up/down mechanism of wired type links. A tunnel is like a wired interface, but turns on RTT-based metrics with a default cost of 96. Default: `wired`.

`rxcost` *number*
This option specifies the nominal RX cost of the interface. The effective neighbor costs for route metrics will be computed from this value with a mechanism determined by the interface `type`. Note that in contrast to other routing protocols like RIP or OSPF, the `rxcost` specifies the cost of RX instead of TX, so it affects primarily neighbors' route selection and not local route selection. Default: 96 for wired interfaces, 256 for wireless.

`limit` *number*
BIRD keeps track of received Hello messages from each neighbor to establish neighbor reachability. For wired type interfaces, this option specifies how many of last 16 hellos have to be correctly received in order to neighbor is assumed to be up. The option is ignored on wireless type interfaces, where gradual cost degradation is used instead of sharp limit. Default: 12.

`hello interval` *time* `s|ms`
Interval at which periodic Hello messages are sent on this interface, with time units. Default: 4 seconds.

`update interval` *time* `s|ms`
Interval at which periodic (full) updates are sent, with time units. Default: 4 times the hello interval.

`port` *number*
This option selects an UDP port to operate on. The default is to operate on port 6696 as specified in the Babel RFC.

`tx class|dscp|priority` *number*
These options specify the ToS/DiffServ/Traffic class/Priority of the outgoing Babel packets. See tx class (p. 18) common option for detailed description.

`rx buffer` *number*
This option specifies the size of buffers used for packet processing. The buffer size should be bigger than maximal size of received packets. The default value is the interface MTU, and the value will be clamped to a minimum of 512 bytes + IP packet overhead.

`tx length` *number*
This option specifies the maximum length of generated Babel packets. To avoid IP fragmentation, it should not exceed the interface MTU value. The default value is the interface MTU value, and the value will be clamped to a minimum of 512 bytes + IP packet overhead.

`check link` *switch*
If set, the hardware link state (as reported by OS) is taken into consideration. When the link disappears (e.g. an ethernet cable is unplugged), neighbors are immediately considered unreachable and all routes received from them are withdrawn. It is possible that some hardware drivers or platforms do not implement this feature. Default: yes.

`next hop ipv4` *address*
Set the IPv4 next hop address advertised for (IPv4) routes advertised on this interface. Default: the preferred IPv4 address of the interface.

`next hop ipv6` *address*
Set the IPv6 next hop address advertised for routes advertised on this interface. If not set, the same link-local address that is used as the source for Babel packets will be used. In normal operation, it should not be necessary to set this option.

`next hop prefer native|ipv6`
By default, BIRD prefers to advertise IPv4 routes with an IPv4 next hop address, using an IPv6 next hop address only when IPv4 addresses are absent from the interface. When set to `ipv6`, BIRD will advertise IPv4 routes with an IPv6 next hop address even when IPv4 addresses are present on the interface (assuming the option extended next hop (p. 40) is enabled). Default: native.

`extended next hop` *switch*
Specify whether BIRD should allow IPv4 routes with an IPv6 next hop, as described in RFC 9229. Note that when both IPv4 and IPv6 next hops are available, the option next hop prefer (p. 40) controls which one is advertised. Default: yes.

`rtt cost` *number*
The RTT-based cost that will be applied to all routes from each neighbour based on the measured RTT to that neighbour. If this value is set, timestamps will be included in generated Babel Hello and IHU messages, and (if the neighbours also have timestamps enabled), the RTT to each neighbour will be computed. An additional cost is added to a neighbour if its RTT is above the rtt min (p. 40) value configured on the interface. The added cost scales linearly from 0 up to the RTT cost configured in this option; the full cost is applied if the neighbour RTT reaches the RTT configured in the rtt max (p. 40) option (and for all RTTs above this value). Default: 0 (disabled), except for tunnel interfaces, where it is 96.

`rtt min` *time* `s|ms`
The minimum RTT above which the RTT cost will start to be applied (scaling linearly from zero up to the full cost). Default: 10 ms

`rtt max` *time* `s|ms`
The maximum RTT above which the full RTT cost will start be applied. Default: 120 ms

`rtt decay` *number*
The decay factor used for the exponentional moving average of the RTT samples from each neighbour, in units of 1/256. Higher values discards old RTT samples faster. Must be between 1 and 256. Default: 42

`send timestamps` *switch*
Whether to send the timestamps used for RTT calculation on this interface. Sending the timestamps enables peers to calculate an RTT to this node, even if no RTT cost is applied to the route metrics. Default: yes.

`authentication none|mac [permissive]`
Selects authentication method to be used. `none` means that packets are not authenticated at all, `mac` means MAC authentication is performed as described in RFC 8967. If MAC authentication is selected, the `permissive` suffix can be used to select an operation mode where outgoing packets are signed, but incoming packets will be accepted even if they fail authentication. This can be useful for incremental deployment of MAC authentication across a network. If MAC authentication is selected, a key must be specified with the `password` configuration option. Default: none.

`password "`*text*`"`
Specifies a password used for authentication. See the password (p. 19) common option for a detailed description. The Babel protocol will only accept HMAC-based algorithms or one of the Blake algorithms, and the length of the supplied password string must match the key size used by the selected algorithm.

6.2.3 Attributes

Babel defines just one attribute: the internal babel metric of the route. It is exposed as the `babel_metric` attribute and has range from 1 to infinity (65535).

6.2.4 Example

```
protocol babel {
        interface "eth*" {
                type wired;
        };
        interface "wlan0", "wlan1" {
                type wireless;
                hello interval 1;
                rxcost 512;
        };
        interface "tap0";

        # This matches the default of babeld: redistribute all addresses
        # configured on local interfaces, plus re-distribute all routes received
        # from other babel peers.

        ipv4 {
                export where (source = RTS_DEVICE) || (source = RTS_BABEL);
        };
        ipv6 {
                export where (source = RTS_DEVICE) || (source = RTS_BABEL);
        };
}
```

6.2.5 Known issues

When retracting a route, Babel generates an unreachable route for a little while (according to RFC). The interaction of this behavior with other protocols is not well tested and strange things may happen.

6.3 BFD

6.3.1 Introduction

Bidirectional Forwarding Detection (BFD) is not a routing protocol itself, it is an independent tool providing liveness and failure detection. Routing protocols like OSPF and BGP use integrated periodic "hello" messages to monitor liveness of neighbors, but detection times of these mechanisms are high (e.g. 40 seconds by default in OSPF, could be set down to several seconds). BFD offers universal, fast and low-overhead mechanism for failure detection, which could be attached to any routing protocol in an advisory role.

BFD consists of mostly independent BFD sessions. Each session monitors an unicast bidirectional path between two BFD-enabled routers. This is done by periodically sending control packets in both directions. BFD does not handle neighbor discovery, BFD sessions are created on demand by request of other protocols (like OSPF or BGP), which supply appropriate information like IP addresses and associated interfaces. When a session changes its state, these protocols are notified and act accordingly (e.g. break an OSPF adjacency when the BFD session went down).

BIRD implements basic BFD behavior as defined in RFC 5880 (some advanced features like the echo mode are not implemented), IP transport for BFD as defined in RFC 5881 and RFC 5883 and interaction with client protocols as defined in RFC 5882.

BFD packets are sent with a dynamic source port number. Linux systems use by default a bit different dynamic port range than the IANA approved one (49152-65535). If you experience problems with compatibility, please adjust `/proc/sys/net/ipv4/ip_local_port_range`.

6.3.2 Configuration

BFD configuration consists mainly of multiple definitions of interfaces. Most BFD config options are session specific. When a new session is requested and dynamically created, it is configured from one of these definitions. For sessions to directly connected neighbors, `interface` definitions are chosen based on the interface associated with the session, while `multihop` definition is used for multihop sessions. If no definition is relevant, the session is just created with the default configuration. Therefore, an empty BFD configuration is often sufficient.

Note that to use BFD for other protocols like OSPF or BGP, these protocols also have to be configured to request BFD sessions, usually by `bfd` option. In BGP case, it is also possible to specify per-peer BFD session options (e.g. rx/tx intervals) as a part of the `bfd` option.

A BFD instance not associated with any VRF handles session requests from all other protocols, even ones associated with a VRF. Such setup would work for single-hop BFD sessions if `net.ipv4.udp_l3mdev_accept` sysctl is enabled, but does not currently work for multihop sessions. Another approach is to configure multiple BFD instances, one for each VRF (including the default VRF). Each BFD instance associated with a VRF (regular or default) only handles session requests from protocols in the same VRF.

Some of BFD session options require *time* value, which has to be specified with the appropriate unit: *number* `s|ms|us`. Although microseconds are allowed as units, practical minimum values are usually in order of tens of milliseconds.

Beware, all BFD instances pick up requests and sessions asynchronously, and any instance can pick up any matching request, regardless of the order in the configuration file. There may be a future update, allowing for strict matching, yet for now, we do not have such an option.

```
protocol bfd [<name>] {
        accept [ipv4|ipv6] [direct|multihop];
        strict bind <switch>;
        zero udp6 checksum rx <switch>;
        interface <interface pattern> {
                interval <time>;
                min rx interval <time>;
                min tx interval <time>;
                idle tx interval <time>;
                multiplier <number>;
                passive <switch>;
                authentication none;
                authentication simple;
                authentication [meticulous] keyed md5|sha1;
                password "<text>";
                password "<text>" {
                        id <number>;
                        generate from "<date>";
                        generate to "<date>";
                        accept from "<date>";
                        accept to "<date>";
                        from "<date>";
                        to "<date>";
                };
        };
        multihop {
                interval <time>;
```

```
                        min rx interval <time>;
                        min tx interval <time>;
                        idle tx interval <time>;
                        multiplier <number>;
                        passive <switch>;
                };
                neighbor <ip> [dev "<interface>"] [local <ip>] [multihop <switch>];
        }
```

`accept [ipv4|ipv6] [direct|multihop]`
A BFD protocol instance accepts (by default) all BFD session requests (with regard to VRF restrictions, see above). This option controls whether IPv4 / IPv6 and direct / multihop session requests are accepted (and which listening sockets are opened). It can be used, for example, to configure separate BFD protocol instances for IPv4 and for IPv6 sessions.

`strict bind` *switch*
Specify whether each BFD interface should use a separate listening socket bound to its local address, or just use a shared listening socket accepting all addresses. Binding to a specific address could be useful in cases like running multiple BIRD instances on a machine, each handling a different set of interfaces. Default: disabled.

`zero udp6 checksum rx` *switch*
UDP checksum computation is optional in IPv4 while it is mandatory in IPv6. Some BFD implementations send UDP datagrams with zero (blank) checksum even in IPv6 case. This option configures BFD listening sockets to accept such datagrams. It is available only on platforms that support the relevant socket option (e.g. `UDP_NO_CHECK6_RX` on Linux). Default: disabled.

`interface` *pattern* `[, ...]` { *options* }
Interface definitions allow to specify options for sessions associated with such interfaces and also may contain interface specific options. See interface (p. 18) common option for a detailed description of interface patterns. Note that contrary to the behavior of `interface` definitions of other protocols, BFD protocol would accept sessions (in default configuration) even on interfaces not covered by such definitions.

`multihop` { *options* }
Multihop definitions allow to specify options for multihop BFD sessions, in the same manner as `interface` definitions are used for directly connected sessions. Currently only one such definition (for all multihop sessions) could be used.

`neighbor` *ip* `[dev "`*interface*`"] [local` *ip*`] [multihop` *switch*`]`
BFD sessions are usually created on demand as requested by other protocols (like OSPF or BGP). This option allows to explicitly add a BFD session to the specified neighbor regardless of such requests.

The session is identified by the IP address of the neighbor, with optional specification of used interface and local IP. By default the neighbor must be directly connected, unless the session is configured as multihop. Note that local IP must be specified for multihop sessions.

Session specific options (part of `interface` and `multihop` definitions):

`interval` *time*
BFD ensures availability of the forwarding path associated with the session by periodically sending BFD control packets in both directions. The rate of such packets is controlled by two options, `min rx interval` and `min tx interval` (see below). This option is just a shorthand to set both of these options together.

`min rx interval` *time*
This option specifies the minimum RX interval, which is announced to the neighbor and used there to limit the neighbor's rate of generated BFD control packets. Default: 10 ms.

`min tx interval` *time*
This option specifies the desired TX interval, which controls the rate of generated BFD control packets

(together with `min rx interval` announced by the neighbor). Note that this value is used only if the BFD session is up, otherwise the value of `idle tx interval` is used instead. Default: 100 ms.

`idle tx interval` *time*
In order to limit unnecessary traffic in cases where a neighbor is not available or not running BFD, the rate of generated BFD control packets is lower when the BFD session is not up. This option specifies the desired TX interval in such cases instead of `min tx interval`. Default: 1 s.

`multiplier` *number*
Failure detection time for BFD sessions is based on established rate of BFD control packets (`min rx/tx interval`) multiplied by this multiplier, which is essentially (ignoring jitter) a number of missed packets after which the session is declared down. Note that rates and multipliers could be different in each direction of a BFD session. Default: 5.

`passive` *switch*
Generally, both BFD session endpoints try to establish the session by sending control packets to the other side. This option allows to enable passive mode, which means that the router does not send BFD packets until it has received one from the other side. Default: disabled.

`authentication none`
No passwords are sent in BFD packets. This is the default value.

`authentication simple`
Every packet carries 16 bytes of password. Received packets lacking this password are ignored. This authentication mechanism is very weak.

`authentication [meticulous] keyed md5|sha1`
An authentication code is appended to each packet. The cryptographic algorithm is keyed MD5 or keyed SHA-1. Note that the algorithm is common for all keys (on one interface), in contrast to OSPF or RIP, where it is a per-key option. Passwords (keys) are not sent open via network.

The `meticulous` variant means that cryptographic sequence numbers are increased for each sent packet, while in the basic variant they are increased about once per second. Generally, the `meticulous` variant offers better resistance to replay attacks but may require more computation.

`password "`*text*`"`
Specifies a password used for authentication. See password (p. 19) common option for detailed description. Note that password option `algorithm` is not available in BFD protocol. The algorithm is selected by `authentication` option for all passwords.

6.3.3 Example

```
protocol bfd {
        interface "eth*" {
                min rx interval 20 ms;
                min tx interval 50 ms;
                idle tx interval 300 ms;
        };
        interface "gre*" {
                interval 200 ms;
                multiplier 10;
                passive;
        };
        multihop {
                interval 200 ms;
                multiplier 10;
        };

        neighbor 192.168.1.10;
        neighbor 192.168.2.2 dev "eth2";
```

```
        neighbor 192.168.10.1 local 192.168.1.1 multihop;
}
```

6.4 BGP

The Border Gateway Protocol is the routing protocol used for backbone level routing in the today's Internet. Contrary to other protocols, its convergence does not rely on all routers following the same rules for route selection, making it possible to implement any routing policy at any router in the network, the only restriction being that if a router advertises a route, it must accept and forward packets according to it.

BGP works in terms of autonomous systems (often abbreviated as AS). Each AS is a part of the network with common management and common routing policy. It is identified by a unique 16-bit number (ASN). Routers within each AS usually exchange AS-internal routing information with each other using an interior gateway protocol (IGP, such as OSPF or RIP). Boundary routers at the border of the AS communicate global (inter-AS) network reachability information with their neighbors in the neighboring AS'es via exterior BGP (eBGP) and redistribute received information to other routers in the AS via interior BGP (iBGP).

Each BGP router sends to its neighbors updates of the parts of its routing table it wishes to export along with complete path information (a list of AS'es the packet will travel through if it uses the particular route) in order to avoid routing loops.

6.4.1 Supported standards

- RFC 4271 – Border Gateway Protocol 4 (BGP)
- RFC 1997 – BGP Communities Attribute
- RFC 2385 – Protection of BGP Sessions via TCP MD5 Signature
- RFC 2545 – Use of BGP Multiprotocol Extensions for IPv6
- RFC 2918 – Route Refresh Capability
- RFC 3107 – Carrying Label Information in BGP
- RFC 4360 – BGP Extended Communities Attribute
- RFC 4364 – BGP/MPLS IPv4 Virtual Private Networks
- RFC 4456 – BGP Route Reflection
- RFC 4486 – Subcodes for BGP Cease Notification Message
- RFC 4659 – BGP/MPLS IPv6 Virtual Private Networks
- RFC 4724 – Graceful Restart Mechanism for BGP
- RFC 4760 – Multiprotocol extensions for BGP
- RFC 4798 – Connecting IPv6 Islands over IPv4 MPLS
- RFC 5065 – AS confederations for BGP
- RFC 5082 – Generalized TTL Security Mechanism
- RFC 5492 – Capabilities Advertisement with BGP
- RFC 8955 – Dissemination of Flow Specification Rules for IPv4
- RFC 8956 – Dissemination of Flow Specification Rules for IPv6
- RFC 5668 – 4-Octet AS Specific BGP Extended Community
- RFC 5925 – TCP Authentication Option

- RFC 6286 – AS-Wide Unique BGP Identifier
- RFC 6608 – Subcodes for BGP Finite State Machine Error
- RFC 6793 – BGP Support for 4-Octet AS Numbers
- RFC 7311 – Accumulated IGP Metric Attribute for BGP
- RFC 7313 – Enhanced Route Refresh Capability for BGP
- RFC 7606 – Revised Error Handling for BGP UPDATE Messages
- RFC 7911 – Advertisement of Multiple Paths in BGP
- RFC 7947 – Internet Exchange BGP Route Server
- RFC 8092 – BGP Large Communities Attribute
- RFC 8212 – Default EBGP Route Propagation Behavior without Policies
- RFC 8654 – Extended Message Support for BGP
- RFC 8950 – Advertising IPv4 NLRI with an IPv6 Next Hop
- RFC 9003 – Extended BGP Administrative Shutdown Communication
- RFC 9072 – Extended Optional Parameters Length for BGP OPEN Message
- RFC 9117 – Revised Validation Procedure for BGP Flow Specifications
- RFC 9234 – Route Leak Prevention and Detection Using Roles
- RFC 9494 – Long-Lived Graceful Restart for BGP
- RFC 9687 – Send Hold Timer

6.4.2 Route selection rules

BGP doesn't have any simple metric, so the rules for selection of an optimal route among multiple BGP routes with the same preference are a bit more complex and they are implemented according to the following algorithm. It starts the first rule, if there are more "best" routes, then it uses the second rule to choose among them and so on.

- Prefer route with the highest Local Preference attribute.
- Prefer route with the shortest AS path.
- Prefer IGP origin over EGP and EGP origin over incomplete.
- Prefer the lowest value of the Multiple Exit Discriminator.
- Prefer routes received via eBGP over ones received via iBGP.
- Prefer routes with lower internal distance (p. 35) to a boundary router.
- Prefer the route with the lowest value of router ID of the advertising router.

6.4.3 IGP routing table

BGP is mainly concerned with global network reachability and with routes to other autonomous systems. When such routes are redistributed to routers in the AS via BGP, they contain IP addresses of a boundary routers (in route attribute NEXT_HOP). BGP depends on existing IGP routing table with AS-internal routes to determine immediate next hops for routes and to know their internal distances to boundary routers for the purpose of BGP route selection. In BIRD, there is usually one routing table used for both IGP routes and BGP routes.

6.4.4 Protocol configuration

Each instance of the BGP corresponds to one neighboring router. This allows to set routing policy and all the other parameters differently for each neighbor using the following configuration parameters:

```
protocol bgp [<name>] {
        ipv4|ipv6|... {
                <channel-options>
                mandatory <switch>;
                next hop keep <switch>|ibgp|ebgp;
                next hop self <switch>|ibgp|ebgp;
                next hop address <ip>;
                next hop prefer global;
                link local next hop format native|single|double;
                gateway direct|recursive;
                igp table <name>;
                import table <switch>;
                export table <switch>;
                secondary <switch>;
                validate <switch>;
                base table <name>;
                extended next hop <switch>;
                require extended next hop <switch>;
                add paths <switch>|rx|tx;
                require add paths <switch>;
                ignore otc <switch>;
                aigp <switch>|originate;
                cost <number>;
                graceful restart <switch>;
                long lived graceful restart <switch>;
                long lived stale time <number>;
                min long lived stale time <number>;
                max long lived stale time <number>;
        };
        local [<ip>] [port <number>] [as <number>];
        neighbor [<ip> | range <prefix>] [onlink] [port <number>] [as <number>] [internal|e
        interface "<text>";
        interface range <interface pattern>;
        onlink <switch>;
        direct;
        multihop [<number>];
        source address <ip>;
        dynamic name "<text>";
        dynamic name digits <number>;
        strict bind <switch>;
        free bind <switch>;
        check link <switch>;
        bfd <switch>|graceful| { <bfd-options> };
        ttl security <switch>;
        authentication none|md5|ao;
        password "<text>";
        keys {
                key {
                        id <number>;
                        send id <number>;
                        recv id <number>;
                        secret "<text>"|<bytestring>;
                        algorithm ( hmac md5 | hmac sha1 | hmac sha224 | hmac sha256 |
```

```
                                hmac sha384 | hmac sha512 | cmac aes128 );
                    preferred;
                    deprecated;
            };
            ...
};
setkey <switch>;
passive <switch>;
confederation <number>;
confederation member <switch>;
rr client <switch>;
rr cluster id <number>|<IPv4 address>;
rs client <switch>;
allow bgp_local_pref <switch>;
allow bgp_med <switch>;
allow local as [<number>];
allow as sets <switch>;
enforce first as <switch>;
enable route refresh <switch>;
require route refresh <switch>;
enable enhanced route refresh <switch>;
require enhanced route refresh <switch>;
graceful restart <switch>|aware;
graceful restart time <number>;
min graceful restart time <number>;
max graceful restart time <number>;
require graceful restart <switch>;
long lived graceful restart <switch>|aware;
long lived stale time <number>;
min long lived stale time <number>;
max long lived stale time <number>;
require long lived graceful restart <switch>;
interpret communities <switch>;
enable as4 <switch>;
require as4 <switch>;
enable extended messages <switch>;
require extended messages <switch>;
capabilities <switch>;
advertise hostname <switch>;
require hostname <switch>;
disable after error <switch>;
disable after cease <switch>|<set-of-flags>;
hold time <number>;
min hold time <number>;
startup hold time <number>;
keepalive time <number>;
min keepalive time <number>;
send hold time <number>;
connect delay time <number>;
connect retry time <number>;
error wait time <number>, <number>;
error forget time <number>;
path metric <switch>;
med metric <switch>;
deterministic med <switch>;
igp metric <switch>;
prefer older <switch>;
```

```
        default bgp_med <number>;
        default bgp_local_pref <number>;
        local role <role-name>;
        require roles <switch>;
    }
```

`local` [*ip*] [`port` *number*] [`as` *number*]
: Define which AS we are part of. (Note that contrary to other IP routers, BIRD is able to act as a router located in multiple AS'es simultaneously, but in such cases you need to tweak the BGP paths manually in the filters to get consistent behavior.) Optional `ip` argument specifies a source address, equivalent to the `source address` option (see below). Optional `port` argument specifies the local BGP port instead of standard port 179. The parameter may be used multiple times with different sub-options (e.g., both `local 10.0.0.1 as 65000;` and `local 10.0.0.1; local as 65000;` are valid). This parameter is mandatory.

`neighbor` [*ip* | `range` *prefix*] [`port` *number*] [`as` *number*] [`internal`|`external`]
: Define neighboring router this instance will be talking to and what AS it is located in. In case the neighbor is in the same AS as we are, we automatically switch to IBGP. Alternatively, it is possible to specify just `internal` or `external` instead of AS number, in that case either local AS number, or any external AS number is accepted. Optionally, the remote port may also be specified. Like `local` parameter, this parameter may also be used multiple times with different sub-options. This parameter is mandatory.

  It is possible to specify network prefix (with `range` keyword) instead of explicit neighbor IP address. This enables dynamic BGP behavior, where the BGP instance listens on BGP port, but new BGP instances are spawned for incoming BGP connections (if source address matches the network prefix). It is possible to mix regular BGP instances with dynamic BGP instances and have multiple dynamic BGP instances with different ranges. These spawned dynamic BGP instances share the parent configuration, therefore by reconfiguring the parent protocol forces reconfiguration of the spawned protocols.

  When the neighbor range is changed, all the spawned dynamic instances shut down. Reconfiguration clears all dynamic instances which were previously disabled by the `disable` CLI command. This may re-enable connection of some clients otherwise blocked by the disabled instance.

`interface "`*text*`"`
: Define interface we should use for link-local BGP IPv6 sessions. Interface can also be specified as a part of `neighbor address` (e.g., `neighbor fe80::1234%eth0 as 65000;`). The option may also be used for non link-local sessions when it is necessary to explicitly specify an interface, but only for direct (not multihop) sessions.

`interface range` *interface pattern*
: Set interface pattern to which the connection will be bound. This is mostly useful with the `neighbor range` option and either link-local addresses or with the `onlink` option where it's not known up front which interface the connection comes on but it needs to stay there.

  This option requires `strict bind` to be on and creates a separate listening socket for every single interface matching the pattern. If the local address is set, it also requires this exact address to be set on that interface to create a listening socket.

  Also see `strict bind` and `free bind`.

`onlink` *switch*
: For a direct neighbor, the BGP session starts immediately without waiting for the neighbor's address to appear on any interface. This option requires an interface to be configured. Next hops of all routes from this session also have the `onlink` attribute.

  This option may generally lead to weird behavior without other configuration in place. One may e.g. need to insert a working route for the given neighbor manually to allow for ACKs from the incoming connection to be routed back correctly. That route may also need to be announced via IGP, or `next hop self` in iBGP may be needed.

  When trying setups with no neighbor route in containerized environments, we got some results with `strict bind`. As of Linux 6.12, we can't recommend running that setup though.

  Onlink behavior may also be specified inside the `neighbor` option. Default: disabled.

`direct`
Specify that the neighbor is directly connected. The IP address of the neighbor must be from a directly reachable IP range (i.e. associated with one of your router's interfaces), otherwise the BGP session wouldn't start but it would wait for such interface to appear. The alternative is the `multihop` option. Default: enabled for eBGP.

`multihop` [*number*]
Configure multihop BGP session to a neighbor that isn't directly connected. Accurately, this option should be used if the configured neighbor IP address does not match with any local network subnets. Such IP address have to be reachable through system routing table. The alternative is the `direct` option. For multihop BGP it is recommended to explicitly configure the source address to have it stable. Optional `number` argument can be used to specify the number of hops (used for TTL). Note that the number of networks (edges) in a path is counted; i.e., if two BGP speakers are separated by one router, the number of hops is 2. Default: enabled for iBGP.

`source address` *ip*
Define local address we should use as a source address for the BGP session. Default: the address of the local end of the interface our neighbor is connected to.

`dynamic name` "*text*"
Define common prefix of names used for new BGP instances spawned when dynamic BGP behavior is active. Actual names also contain numeric index to distinguish individual instances. Default: "dynbgp".

`dynamic name digits` *number*
Define minimum number of digits for index in names of spawned dynamic BGP instances. E.g., if set to 2, then the first name would be "dynbgp01". Default: 0.

`strict bind` *switch*
Specify whether BGP listening socket should be bound to a specific local address (the same as the `source address`) and associated interface or to all addresses. Binding to a specific address could be useful in cases like running multiple BIRD instances on a machine, each using its IP address.

Note that listening sockets bound to a specific address and to all addresses collide. Also listening sockets bound to a specific interface and to all interfaces may behave weirdly. Therefore, all BGP protocols (of the same address family and using the same local port) should have set `strict bind`, or none of them, and in the same way, all should have a `interface` or `interface range` or none of them.

Default: disabled.

`free bind` *switch*
Use IP_FREEBIND socket option for the listening socket, which allows binding to an IP address not (yet) assigned to an interface. Note that all BGP instances that share a listening socket should have the same value of the `free bind` option.

If `interface range` is set together with a local address and `free bind` as well, it creates a free-bind listening socket for every interface regardless of the assigned address. This is an experimental feature.

Default: disabled.

`check link` *switch*
BGP could use hardware link state into consideration. If enabled, BIRD tracks the link state of the associated interface and when link disappears (e.g. an ethernet cable is unplugged), the BGP session is immediately shut down. Note that this option cannot be used with multihop BGP. Default: enabled for direct BGP, disabled otherwise.

`bfd` *switch*|`graceful`| { *options* }
BGP could use BFD protocol as an advisory mechanism for neighbor liveness and failure detection. If enabled, BIRD setups a BFD session for the BGP neighbor and tracks its liveness by it. This has an advantage of an order of magnitude lower detection times in case of failure. When a neighbor failure is detected, the BGP session is restarted. Optionally, it can be configured (by `graceful` argument) to trigger graceful restart instead of regular restart. It is also possible to specify section with per-peer BFD session options instead of just the switch argument. All BFD session-specific options are allowed

here. Note that BFD protocol also has to be configured, see BFD (p. 41) section for details. Default: disabled.

`ttl security` *switch*
Use GTSM (RFC 5082 - the generalized TTL security mechanism). GTSM protects against spoofed packets by ignoring received packets with a smaller than expected TTL. To work properly, GTSM have to be enabled on both sides of a BGP session. If both `ttl security` and `multihop` options are enabled, `multihop` option should specify proper hop value to compute expected TTL. Kernel support required: Linux: 2.6.34+ (IPv4), 2.6.35+ (IPv6), BSD: since long ago, IPv4 only. Note that full (ICMP protection, for example) RFC 5082 support is provided by Linux only. Default: disabled.

`authentication none|md5|ao`
Selects authentication method to be used. `none` means that the BGP session is not authenticated at all. `md5` means that the TCP MD5 authentication of BGP sessions (RFC 2385) is used, in that case the option password (p. 51) is used to specify the (single) password. Finally, `ao` means to use TCP Authentication Option (TCP-AO, RFC 5925), allowing multiple keys and different cryptographic algorithms. These are specified using the option keys (p. 51). Note that TCP-AO authentication is not supported on dynamic BGP sessions. Default: none.

`password "`*text*`"`
Use this password for MD5 authentication of BGP sessions (RFC 2385). When used on BSD systems, see also `setkey` option below. Default: no authentication.

`setkey` *switch*
On BSD systems, keys for TCP MD5 authentication are stored in the global SA/SP database, which can be accessed by external utilities (e.g. setkey(8)). BIRD configures security associations in the SA/SP database automatically based on `password` options (see above), this option allows to disable automatic updates by BIRD when manual configuration by external utilities is preferred. Note that automatic SA/SP database updates are currently implemented only for FreeBSD. Passwords have to be set manually by an external utility on NetBSD and OpenBSD. Default: enabled (ignored on non-FreeBSD).

`keys { key { [...] }; [...] }`
Define a set of cryptographic keys that are used for TCP-AO authentication of BGP sessions (RFC 5925). Each key has a configuration block with its own sub-options ([`send` | `recv`] `id`, `secret`, `algorithm`, `preferred`, `deprecated`).

TCP-AO key has two IDs - for outgoing and incoming direction (Send / Recv ID). Among keys on one protocol all Send IDs must be unique and all Recv IDs must be unique. They must be in range 0-255 and they can be set independently with key options `send id` and `recv id`, or together with option `id`. Note that specifying these IDs is mandatory.

Of course, TCP-AO key contains a shared secret key. It is specified by the option `secret` as a text string or as a sequence of hexadecimal digit pairs (bytestring (p. 28)).

Used cryptographic algorithm can be specified for each key with the option `algorithm`. Possible values are: `hmac md5`, `hmac sha1`, `hmac sha224`, `hmac sha256`, `hmac sha384`, `hmac sha512`, and `cmac aes128`. Default value is `hmac sha1`.

When multiple keys are available, BIRD selects one to advertise as RNext key (the key it prefers to be used to sign incoming traffic). Keys marked as `preferred` are selected before unmarked keys, while keys marked as `deprecated` are never selected (but still could be used when the other side asks for them). Therefore, there must be always at least one non-deprecated key.

Currently, only the selected key is used during the initial handshake of session establishment (and therefore must be known by the other side). This may change in the future.

It is possible to add, remove, or modify keys during reconfiguration without breaking the BGP session. The recommended way is to refrain from removing a key that is in active use (as reported by `Current key` and `RNext key` in `show protocols all`), instead marking the key as `deprecated` on both sides of the session. It is possible to remove an active key directly, BIRD would forcibly switch to another key (as long as there is a non-deprecated key that is not added, removed or modified during this reconfiguration). This is not recommended as it skips the proper key change mechanism and may switch to a key that is not available to the other side.

Modification of existing keys (except of marking them `preferred` or `deprecated`) is equivalent to removing and then adding them, with the same issues related to removing of active keys.

`passive` *switch*
Standard BGP behavior is both initiating outgoing connections and accepting incoming connections. In passive mode, outgoing connections are not initiated. Default: off.

`confederation` *number*
BGP confederations (RFC 5065) are collections of autonomous systems that act as one entity to external systems, represented by one confederation identifier (instead of AS numbers). This option allows to enable BGP confederation behavior and to specify the local confederation identifier. When BGP confederations are used, all BGP speakers that are members of the BGP confederation should have the same confederation identifier configured. Default: 0 (no confederation).

`confederation member` *switch*
When BGP confederations are used, this option allows to specify whether the BGP neighbor is a member of the same confederation as the local BGP speaker. The option is unnecessary (and ignored) for IBGP sessions, as the same AS number implies the same confederation. Default: no.

`rr client` *switch*
Be a route reflector and treat the neighbor as a route reflection client. Default: disabled.

`rr cluster id` *number* | *IPv4 address*
Route reflectors use cluster id to avoid route reflection loops. When there is one route reflector in a cluster it usually uses its router id as a cluster id, but when there are more route reflectors in a cluster, these need to be configured (using this option) to use a common cluster id. Clients in a cluster need not know their cluster id and this option is not allowed for them. Default: the same as router id.

`rs client` *switch*
Be a route server and treat the neighbor as a route server client. A route server is used as a replacement for full mesh EBGP routing in Internet exchange points in a similar way to route reflectors used in IBGP routing. BIRD does not implement obsoleted RFC 1863, but uses ad-hoc implementation, which behaves like plain EBGP but reduces modifications to advertised route attributes to be transparent (for example does not prepend its AS number to AS PATH attribute and keeps MED attribute). Default: disabled.

`allow bgp_local_pref` *switch*
Standard BGP implementations do not send the Local Preference attribute to EBGP neighbors and ignore this attribute if received from EBGP neighbors, as per RFC 4271. When this option is enabled on an EBGP session, this attribute will be sent to and accepted from the peer, which is useful for example if you have a setup like in RFC 7938. The option does not affect IBGP sessions. Default: off.

`allow bgp_med` *switch*
Standard BGP implementations do not propagate the MULTI_EXIT_DESC attribute unless it is configured locally. When this option is enabled on an EBGP session, this attribute will be sent to the peer regardless, which is useful for example if you have a setup like in RFC 7938. The option does not affect IBGP sessions. Default: off.

`allow local as` [*number*]
BGP prevents routing loops by rejecting received routes with the local AS number in the AS path. This option allows to loose or disable the check. Optional `number` argument can be used to specify the maximum number of local ASNs in the AS path that is allowed for received routes. When the option is used without the argument, the check is completely disabled and you should ensure loop-free behavior by some other means. Default: 0 (no local AS number allowed).

`allow as sets` *switch*
Historically, AS path attribute received with BGP routes may have contained not only sequences of AS numbers, but also sets of AS numbers. These rarely used artifacts were results of inter-AS route aggregation. AS sets are deprecated (RFC 6472, RFC 9774) and AS paths containing them (or AS confed sets) are considered malformed. Corresponding BGP updates are therefore treated as withdraws. When this option is enabled, then received AS paths containing these deprecated AS sets or AS confed sets are accepted. Default: off (since 2.18 and 3.2).

`enforce first as` *switch*
Routes received from an EBGP neighbor are generally expected to have the first (leftmost) AS number in their AS path equal to the neighbor AS number. This is not enforced by default as there are legitimate cases where it is not true, e.g. connections to route servers. When this option is enabled, routes with non-matching first AS number are rejected and corresponding updates are treated as withdraws. The option is valid on EBGP sessions only. Default: off.

`enable route refresh` *switch*
After the initial route exchange, BGP protocol uses incremental updates to keep BGP speakers synchronized. Sometimes (e.g., if BGP speaker changes its import filter, or if there is suspicion of inconsistency) it is necessary to do a new complete route exchange. BGP protocol extension Route Refresh (RFC 2918) allows BGP speaker to request re-advertisement of all routes from its neighbor. This option specifies whether BIRD advertises this capability and supports related procedures. Note that even when disabled, BIRD can send route refresh requests. Disabling Route Refresh also disables Enhanced Route Refresh. Default: on.

`require route refresh` *switch*
If enabled, the BGP Route Refresh capability (RFC 2918) must be announced by the BGP neighbor, otherwise the BGP session will not be established. Default: off.

`enable enhanced route refresh` *switch*
BGP protocol extension Enhanced Route Refresh (RFC 7313) specifies explicit begin and end for Route Refresh (see previous option), therefore the receiver can remove stale routes that were not advertised during the exchange. This option specifies whether BIRD advertises this capability and supports related procedures. Default: on.

`require enhanced route refresh` *switch*
If enabled, the BGP Enhanced Route Refresh capability (RFC 7313) must be announced by the BGP neighbor, otherwise the BGP session will not be established. Default: off.

`graceful restart` *switch*|`aware`
When a BGP speaker restarts or crashes, neighbors will discard all received paths from the speaker, which disrupts packet forwarding even when the forwarding plane of the speaker remains intact. RFC 4724 specifies an optional graceful restart mechanism to alleviate this issue. This option controls the mechanism. It has three states: Disabled, when no support is provided. Aware, when the graceful restart support is announced and the support for restarting neighbors is provided, but no local graceful restart is allowed (i.e. receiving-only role). Enabled, when the full graceful restart support is provided (i.e. both restarting and receiving role). Restarting role could be also configured per-channel. Note that proper support for local graceful restart requires also configuration of other protocols. Default: aware.

`graceful restart time` *number*
The restart time is announced in the BGP Graceful Restart capability and specifies how long the neighbor would wait for the BGP session to re-establish after a restart before deleting stale routes. Default: 120 seconds.

`min graceful restart time` *number*
The lower bound for the graceful restart time to override the value received in the BGP Graceful Restart capability announced by the neighbor. Default: no lower bound.

`max graceful restart time` *number*
The upper bound for the graceful restart time to override the value received in the BGP Graceful Restart capability announced by the neighbor. Default: no upper bound.

`require graceful restart` *switch*
If enabled, the BGP Graceful Restart capability (RFC 4724) must be announced by the BGP neighbor, otherwise the BGP session will not be established. Default: off.

`long lived graceful restart` *switch*|`aware`
The long-lived graceful restart is an extension of the traditional BGP graceful restart (p. 53), where stale routes are kept even after the restart time (p. 53) expires for additional long-lived stale time, but

they are marked with the LLGR_STALE community, depreferenced, and withdrawn from routers not supporting LLGR. Like traditional BGP graceful restart, it has three states: disabled, aware (receiving-only), and enabled. Note that long-lived graceful restart requires at least aware level of traditional BGP graceful restart. Default: aware, unless graceful restart is disabled.

`long lived stale time` *number*
The long-lived stale time is announced in the BGP Long-lived Graceful Restart capability and specifies how long the neighbor would keep stale routes depreferenced during long-lived graceful restart until either the session is re-stablished and synchronized or the stale time expires and routes are removed. Default: 3600 seconds.

`min long lived stale time` *number*
The lower bound for the long-lived stale time to override the value received in the BGP Long-lived Graceful Restart capability announced by the neighbor. Default: no lower bound.

`max long lived stale time` *number*
The upper bound for the long-lived stale time to override the value received in the BGP Long-lived Graceful Restart capability announced by the neighbor. Default: no upper bound.

`require long lived graceful restart` *switch*
If enabled, the BGP Long-lived Graceful Restart capability (RFC 9494) must be announced by the BGP neighbor, otherwise the BGP session will not be established. Default: off.

`interpret communities` *switch*
RFC 1997 demands that BGP speaker should process well-known communities like no-export (65535, 65281) or no-advertise (65535, 65282). For example, received route carrying a no-advertise community should not be advertised to any of its neighbors. If this option is enabled (which is by default), BIRD has such behavior automatically (it is evaluated when a route is exported to the BGP protocol just before the export filter). Otherwise, this integrated processing of well-known communities is disabled. In that case, similar behavior can be implemented in the export filter. Default: on.

`enable as4` *switch*
BGP protocol was designed to use 2B AS numbers and was extended later to allow 4B AS number. BIRD supports 4B AS extension, but by disabling this option it can be persuaded not to advertise it and to maintain old-style sessions with its neighbors. This might be useful for circumventing bugs in neighbor's implementation of 4B AS extension. Even when disabled (off), BIRD behaves internally as AS4-aware BGP router. Default: on.

`require as4` *switch*
If enabled, the BGP 4B AS number capability (RFC 6793) must be announced by the BGP neighbor, otherwise the BGP session will not be established. Default: off.

`enable extended messages` *switch*
The BGP protocol uses maximum message length of 4096 bytes. This option provides an extension (RFC 8654) to allow extended messages with length up to 65535 bytes. Default: off.

`require extended messages` *switch*
If enabled, the BGP Extended Message capability (RFC 8654) must be announced by the BGP neighbor, otherwise the BGP session will not be established. Default: off.

`capabilities` *switch*
Use capability advertisement to advertise optional capabilities. This is standard behavior for newer BGP implementations, but there might be some older BGP implementations that reject such connection attempts. When disabled (off), features that request it (4B AS support) are also disabled. Default: on, with automatic fallback to off when received capability-related error.

`advertise hostname` *switch*
Advertise the hostname capability along with the hostname. Default: off.

`require hostname` *switch*
If enabled, the hostname capability must be announced by the BGP neighbor, otherwise the BGP session negotiation fails. Default: off.

`disable after error` *switch*
When an error is encountered (either locally or by the other side), disable the instance automatically and wait for an administrator to fix the problem manually. Default: off.

`disable after cease` *switch*|*set-of-flags*
When a Cease notification is received, disable the instance automatically and wait for an administrator to fix the problem manually. When used with *switch* argument, it means handle every Cease subtype with the exception of `connection collision`. Default: off.

The *set-of-flags* allows to narrow down relevant Cease subtypes. The syntax is {*flag* `[, ...]` }, where flags are: `cease`, `prefix limit hit`, `administrative shutdown`, `peer deconfigured`, `administrative reset`, `connection rejected`, `configuration change`, `connection collision`, `out of resources`.

`hold time` *number*
Time in seconds to wait for a Keepalive message from the other side before considering the connection stale. The effective value is negotiated during session establishment and it is a minimum of this configured value and the value proposed by the peer. The zero value has a special meaning, signifying that no keepalives are used. Default: 240 seconds.

`min hold time` *number*
Minimum value of the hold time that is accepted during session negotiation. If the peer proposes a lower value, the session is rejected with error. Default: none.

`startup hold time` *number*
Value of the hold timer used before the routers have a chance to exchange open messages and agree on the real value. Default: 240 seconds.

`keepalive time` *number*
Delay in seconds between sending of two consecutive Keepalive messages. The effective value depends on the negotiated hold time, as it is scaled to maintain proportion between the keepalive time and the hold time. Default: One third of the hold time.

`min keepalive time` *number*
Minimum value of the keepalive time that is accepted during session negotiation. If the proposed hold time would lead to a lower value of the keepalive time, the session is rejected with error. Default: none.

`send hold time` *number*
Maximum time in seconds betweeen successfull transmissions of BGP messages. Send hold timer drops the session if the neighbor is sending keepalives, but does not receive our messages, causing the TCP connection to stall. This may happen due to malfunctioning or overwhelmed neighbor. See RFC 9687 for more details.

Like the option `keepalive time`, the effective value depends on the negotiated hold time, as it is scaled to maintain proportion between the send hold time and the keepalive time. If it is set to zero, the timer is disabled. Default: double of the hold timer limit.

The option `disable rx` is intended only for testing this feature and should not be used anywhere else. It discards received messages and disables the hold timer.

`connect delay time` *number*
Delay in seconds between protocol startup and the first attempt to connect. Default: 5 seconds.

`connect retry time` *number*
Time in seconds to wait before retrying a failed attempt to connect. Default: 120 seconds.

`error wait time` *number*`,` *number*
Minimum and maximum delay in seconds between a protocol failure (either local or reported by the peer) and automatic restart. Does not apply when `disable after error` is configured. If consecutive errors happen, the delay is increased exponentially until it reaches the maximum. Default: 60, 300.

`error forget time` *number*
Maximum time in seconds between two protocol failures to treat them as a error sequence which makes `error wait time` increase exponentially. Default: 300 seconds.

`path metric` *switch*
Enable comparison of path lengths when deciding which BGP route is the best one. Default: on.

`med metric` *switch*
Enable comparison of MED attributes (during best route selection) even between routes received from different ASes. This may be useful if all MED attributes contain some consistent metric, perhaps enforced in import filters of AS boundary routers. If this option is disabled, MED attributes are compared only if routes are received from the same AS (which is the standard behavior). Default: off.

`deterministic med` *switch*
BGP route selection algorithm is often viewed as a comparison between individual routes (e.g. if a new route appears and is better than the current best one, it is chosen as the new best one). But the proper route selection, as specified by RFC 4271, cannot be fully implemented in that way. The problem is mainly in handling the MED attribute. BIRD, by default, uses an simplification based on individual route comparison, which in some cases may lead to temporally dependent behavior (i.e. the selection is dependent on the order in which routes appeared). This option enables a different (and slower) algorithm implementing proper RFC 4271 route selection, which is deterministic. Alternative way how to get deterministic behavior is to use `med metric` option. This option is incompatible with sorted tables (p. 6). Default: off.

`igp metric` *switch*
Enable comparison of internal distances to boundary routers during best route selection. Default: on.

`prefer older` *switch*
Standard route selection algorithm breaks ties by comparing router IDs. This changes the behavior to prefer older routes (when both are external and from different peer). For details, see RFC 5004. Default: off.

`default bgp_med` *number*
Value of the Multiple Exit Discriminator to be used during route selection when the MED attribute is missing. Default: 0.

`default bgp_local_pref` *number*
A default value for the Local Preference attribute. It is used when a new Local Preference attribute is attached to a route by the BGP protocol itself (for example, if a route is received through eBGP and therefore does not have such attribute). Default: 100 (0 in pre-1.2.0 versions of BIRD).

`local role` *role-name*
BGP roles are a mechanism for route leak prevention and automatic route filtering based on common BGP topology relationships. They are defined in RFC 9234. Instead of manually configuring filters and communities, automatic filtering is done with the help of the OTC attribute - a flag for routes that should be sent only to customers. The same attribute is also used to automatically detect and filter route leaks created by third parties.

This option is valid for EBGP sessions, but it is not recommended to be used within AS confederations (which would require manual filtering of `bgp_otc` attribute on confederation boundaries).

Possible *role-name* values are: `provider`, `rs_server`, `rs_client`, `customer` and `peer`. Default: No local role assigned.

`require roles` *switch*
If this option is set, the BGP roles must be defined on both sides, otherwise the session will not be established. This behavior is defined in RFC 9234 as "strict mode" and is used to enforce corresponding configuration at your conterpart side. Default: disabled.

`tx size warning` *number*
If this option is set, the bgp protocol prints warning when it consumes at least given size of memory. This includes also the memory consumed by any export table. Zero means disabled. Default: disabled.

6.4.5 Channel configuration

BGP supports several AFIs and SAFIs over one connection. Every AFI/SAFI announced to the peer corresponds to one channel. The table of supported AFI/SAFIs together with their appropriate channels follows.

Channel name	Table nettype	IGP table allowed	AFI	SAFI
ipv4	ipv4	ipv4 and ipv6	1	1
ipv6	ipv6	ipv4 and ipv6	2	1
ipv4 multicast	ipv4	ipv4 and ipv6	1	2
ipv6 multicast	ipv6	ipv4 and ipv6	2	2
ipv4 mpls	ipv4	ipv4 and ipv6	1	4
ipv6 mpls	ipv6	ipv4 and ipv6	2	4
vpn4 mpls	vpn4	ipv4 and ipv6	1	128
vpn6 mpls	vpn6	ipv4 and ipv6	2	128
vpn4 multicast	vpn4	ipv4 and ipv6	1	129
vpn6 multicast	vpn6	ipv4 and ipv6	2	129
flow4	flow4	—	1	133
flow6	flow6	—	2	133
evpn	evpn	ipv4 and ipv6	25	70
neighbors	neighbor	—	—	—

The BGP protocol can be configured as MPLS-aware (by defining both AFI/SAFI channels and the MPLS channel). In such case the BGP protocol assigns labels to routes imported from MPLS-aware SAFIs (i.e. `ipvX mpls` and `vpnX mpls`) and automatically announces corresponding MPLS route for each labeled route. As BGP generally processes a large amount of routes, it is suggested to set MPLS label policy to `aggregate`.

Note that even BGP instances without MPLS channel and without local MPLS configuration can still propagate third-party MPLS labels, e.g. as route reflectors, they just will not assign local labels to imported routes and will not announce MPLS routes for local MPLS forwarding.

Due to RFC 8212, external BGP protocol requires explicit configuration of import and export policies (in contrast to other protocols, where default policies of `import all` and `export none` are used in absence of explicit configuration). Note that blanket policies like `all` or `none` can still be used in explicit configuration.

The `neighbors` channel is a special channel used for BGP unnumbered automatic peering. Unlike traditional AFI/SAFI channels that exchange routing information, the `neighbors` channel is used to dynamically trigger the creation of new BGP sessions. When present in a dynamic BGP configuration, exporting neighbor objects (typically from RAdv with router discovery enabled) to this channel will automatically spawn new BGP sessions for each discovered neighbor. The lifetime of these automatic sessions is tied to the presence of neighbor objects in the neighbor routing table, withdrawing an object will trigger deprecation of the corresponding BGP session. This channel is only allowed in dynamic BGP instances (those configured with `neighbor range`).

BGP channels have additional config options (together with the common ones):

`mandatory` *switch*
: When local and neighbor sets of configured AFI/SAFI pairs differ, capability negotiation ensures that a common subset is used. For mandatory channels their associated AFI/SAFI must be negotiated (i.e., also announced by the neighbor), otherwise BGP session negotiation fails with *'Required capability missing'* error. Regardless, at least one AFI/SAFI must be negotiated in order to BGP session be successfully established. Default: off.

`next hop keep` *switch*|`ibgp`|`ebgp`
: Do not modify the Next Hop attribute and advertise the current one unchanged even in cases where our own local address should be used instead. This is necessary when the BGP speaker does not forward network traffic (route servers and some route reflectors) and also can be useful in some other cases (e.g. multihop EBGP sessions). Can be enabled for all routes, or just for routes received from IBGP / EBGP neighbors. Default: disabled for regular BGP, enabled for route servers, `ibgp` for route reflectors.

`next hop self` *switch*|`ibgp`|`ebgp`
Always advertise our own local address as a next hop, even in cases where the current Next Hop attribute should be used unchanged. This is sometimes used for routes propagated from EBGP to IBGP when IGP routing does not cover inter-AS links, therefore IP addreses of EBGP neighbors are not resolvable through IGP. Can be enabled for all routes, or just for routes received from IBGP / EBGP neighbors. Default: disabled.

`next hop address` *ip*
Specify which address to use when our own local address should be announced in the Next Hop attribute. Default: the source address of the BGP session (if acceptable), or the preferred address of an associated interface.

`next hop prefer global|local`
For IPv6 routes, the Next Hop attribute may contain both a global IP address and a link-local IP address. For IBGP sessions, the global IP address is resolved (gateway recursive (p. 58)) through an IGP routing table (igp table (p. 58)) to get an immediate next hop. If the resulting IGP route is a direct route (i.e., the next hop is a direct neighbor), then the link-local IP address from the Next Hop attribute is used as the immediate next hop. This option changes it to prefer global or link-local IP address, respectively. Note that even with this option set to global, route may end with a link-local immediate next hop when the IGP route has one.

When set to `local`, prefer link-local IPv6 addresses over global addresses for immediate next hops of received routes. When set to `global`, prefer global IPv6 address to link-local IPv6 address for immediate next hops of received routes. Default: global in direct mode (EBGP), local in recursive mode (IBGP).

`link local next hop format native|single|double`
For IPv6 routes, BGP assumes that the Next Hop attribute contains a global IPv6 address (in the first position) and an optional link-local IPv6 address (in the second position): [*global*, *link-local*]. When a BGP session is established using just link-local addresses, there may be no global IPv6 address for the next hop. BGP implementations differ on how to encode such next hops. BIRD `native` format is to send [zero, *link-local*], `single` format is [*link-local*], `double` format is [*link-local*, *link-local*]. BIRD accepts all these variants when decoding received routes, but this option controls which one it uses to encode such next hops. Default: native.

`gateway direct|recursive`
For received routes, their `gw` (immediate next hop) attribute is computed from received `bgp_next_hop` attribute. This option specifies how it is computed. Direct mode means that the IP address from `bgp_next_hop` is used and must be directly reachable. Recursive mode means that the gateway is computed by an IGP routing table lookup for the IP address from `bgp_next_hop`. Note that there is just one level of indirection in recursive mode - the route obtained by the lookup must not be recursive itself, to prevent mutually recursive routes.

Recursive mode is the behavior specified by the BGP standard. Direct mode is simpler, does not require any routes in a routing table, and was used in older versions of BIRD, but does not handle well nontrivial iBGP setups and multihop. Recursive mode is incompatible with sorted tables (p. 6). Default: `direct` for direct sessions, `recursive` for multihop sessions.

`igp table` *name*
Specifies a table that is used as an IGP routing table. The type of this table must be as allowed in the table above. This option is allowed once for every allowed table type. Default: the same as the main table the channel is connected to (if eligible).

`import table` *switch*
A BGP import table contains all received routes from given BGP neighbor, before application of import filters. It is also called *Adj-RIB-In* in BGP terminology. BIRD BGP by default operates without import tables, in which case received routes are just processed by import filters, accepted ones are stored in the master table, and the rest is forgotten. Enabling `import table` allows to store unprocessed routes, which can be examined later by `show route`, and can be used to reconfigure import filters without full route refresh. Default: off.

`export table` *switch*
A BGP export table contains all routes sent to given BGP neighbor, after application of export filters. It is also called *Adj-RIB-Out* in BGP terminology. BIRD BGP by default operates without export tables, in which case routes from master table are just processed by export filters and then announced by BGP. Enabling `export table` allows to store routes after export filter processing, so they can be examined later by `show route`, and can be used to eliminate unnecessary updates or withdraws. Default: off.

`export settle time` *time time*
Minimum and maximum settle times, respectively, for announcements from export table to external readers. These values don't apply for regular TX, just for side channel exports. You will probably never need to change these values. Default values: `10 ms 100 ms`. You have to always provide both values.

`secondary` *switch*
Usually, if an export filter rejects a selected route, no other route is propagated for that network. This option allows to try the next route in order until one that is accepted is found or all routes for that network are rejected. This can be used for route servers that need to propagate different tables to each client but do not want to have these tables explicitly (to conserve memory). This option requires that the connected routing table is sorted (p. 6). Default: off.

`validate` *switch*
Apply flowspec validation procedure as described in RFC 8955 section 6 and RFC 9117. The Validation procedure enforces that only routers in the forwarding path for a network can originate flowspec rules for that network. The validation procedure should be used for EBGP to prevent injection of malicious flowspec rules from outside, but it should also be used for IBGP to ensure that selected flowspec rules are consistent with selected IP routes. The validation procedure uses an IP routing table (base table (p. 59), see below) against which flowspec rules are validated. This option is limited to flowspec channels. Default: off (for compatibility reasons).

Note that currently the flowspec validation does not work reliably together with import table (p. 58) option enabled on flowspec channels.

`base table` *name*
Specifies an IP table used for the flowspec validation procedure. The table must have enabled `trie` option, otherwise the validation procedure would not work. The type of the table must be `ipv4` for `flow4` channels and `ipv6` for `flow6` channels. This option is limited to flowspec channels. Default: the main table of the `ipv4` / `ipv6` channel of the same BGP instance, or the `master4` / `master6` table if there is no such channel.

`extended next hop` *switch*
BGP expects that announced next hops have the same address family as associated network prefixes. This option provides an extension to use IPv4 next hops with IPv6 prefixes and vice versa. For IPv4 / VPNv4 channels, the behavior is controlled by the Extended Next Hop Encoding capability, as described in RFC 8950. For IPv6 / VPNv6 channels, just IPv4-mapped IPv6 addresses are used, as described in RFC 4798 and RFC 4659. Default: off.

`require extended next hop` *switch*
If enabled, the BGP Extended Next Hop Encoding capability (RFC 8950) must be announced by the BGP neighbor, otherwise the BGP session will not be established. Note that this option is relevant just for IPv4 / VPNv4 channels, as IPv6 / VPNv6 channels use a different mechanism not signalled by a capability. Default: off.

`add paths` *switch*|`rx`|`tx`
Standard BGP can propagate only one path (route) per destination network (usually the selected one). This option controls the ADD-PATH protocol extension, which allows to advertise any number of paths to a destination. Note that to be active, ADD-PATH has to be enabled on both sides of the BGP session, but it could be enabled separately for RX and TX direction. When active, all available routes accepted by the export filter are advertised to the neighbor. Default: off.

`require add paths` *switch*
If enabled, the BGP ADD-PATH capability (RFC 7911) must be announced by the BGP neighbor, otherwise the BGP session will not be established. Announced directions in the capability must be compatible with locally configured directions. E.g., If `add path tx` is configured locally, then the neighbor capability must announce RX. Default: off.

`ignore otc` *switch*
When BGP roles (p. 56) are set, BIRD automatically drops leaked routes, as mandated by RFC 9234. This option allows the channel to accept routes with offending bgp_otc attribute (p. 64), so that the user may do additional processing by the filter engine before rejecting. Specifically, this option may be useful when accompanied with keeping the filtered routes (p. 20) for inspection.

The sole existence of this option knowingly violates the requirements stipulated in RFC 9234, Section 5, last sentence. Do not set this option unless you really know what you are doing. Default: off.

`aigp` *switch*|`originate`
The BGP protocol does not use a common metric like other routing protocols, instead it uses a set of criteria for route selection consisting both overall AS path length and a distance to the nearest AS boundary router. Assuming that metrics of different autonomous systems are incomparable, once a route is propagated from an AS to a next one, the distance in the old AS does not matter.

The AIGP extension (RFC 7311) allows to propagate accumulated IGP metric (in the AIGP attribute) through both IBGP and EBGP links, computing total distance through multiple autonomous systems (assuming they use comparable IGP metric). The total AIGP metric is compared in the route selection process just after Local Preference comparison (and before AS path length comparison).

This option controls whether AIGP attribute propagation is allowed on the session. Optionally, it can be set to `originate`, which not only allows AIGP attribute propagation, but also new AIGP attributes are automatically attached to non-BGP routes with valid IGP metric (e.g. `ospf_metric1`) as they are exported to the BGP session. Default: enabled for IBGP (and intra-confederation EBGP), disabled for regular EBGP.

`cost` *number*
When BGP gateway mode (p. 58) is `recursive` (mainly multihop IBGP sessions), then the distance to BGP next hop is based on underlying IGP metric. This option specifies the distance to BGP next hop for BGP sessions in direct gateway mode (mainly direct EBGP sessions).

`graceful restart` *switch*
Although BGP graceful restart is configured mainly by protocol-wide options (p. 53), it is possible to configure restarting role per AFI/SAFI pair by this channel option. The option is ignored if graceful restart is disabled by protocol-wide option. Default: off in aware mode, on in full mode.

`long lived graceful restart` *switch*
BGP long-lived graceful restart is configured mainly by protocol-wide options (p. 53), but the restarting role can be set per AFI/SAFI pair by this channel option. The option is ignored if long-lived graceful restart is disabled by protocol-wide option. Default: off in aware mode, on in full mode.

`long lived stale time` *number*
Like previous graceful restart channel options, this option allows to set long lived stale time (p. 54) per AFI/SAFI pair instead of per protocol. Default: set by protocol-wide option.

`min long lived stale time` *number*
Like previous graceful restart channel options, this option allows to set min long lived stale time (p. 54) per AFI/SAFI pair instead of per protocol. Default: set by protocol-wide option.

`max long lived stale time` *number*
Like previous graceful restart channel options, this option allows to set max long lived stale time (p. 54) per AFI/SAFI pair instead of per protocol. Default: set by protocol-wide option.

6.4.6 Reconfiguration

Running the `configure` command with a changed configuration file will trigger a reconfiguration. This may cause a reload of the affected channel or a restart of the BGP protocol. A change in options that just affect route processing generally causes a reload, while a change in setting of BGP capabilities or other properties negotiated during session establishment always leads to a restart.

With the `configure soft` command, configuration changes do not trigger a reload, but instead just log a message about the change, suggesting manual reload of the channel. However, changes demanding restart still trigger a restart of the protocol. See configure (p. 25) and reload (p. 25) commands for more details. In the rest of the section, we assume the usage of the `configure` command without the `soft` option.

Changes in these channel options cause a reload of the channel in the import direction (re-import of routes from the BGP neighbor):

- `import`
- `preference`
- `gateway`
- `next hop prefer`
- `aigp`
- `cost`
- `ignore otc`

Note: Reload in the import direction requires the `route refresh` capability negotiated, otherwise it will trigger a restart. In case of import filter change, routes will get reloaded from `import table` instead, if enabled. None of these applies for export direction.

Changes in these channel options cause a reload of the channel in the export direction (re-export of routes to the BGP neighbor):

- `export`
- `next hop self`
- `next hop keep`
- `link local next hop format`
- `aigp`
- `aigp originate`

Protocol options that cause a restart when changed:

- `router id`
- `hostname`
- `vrf`

All BGP-specific protocol options cause a restart when changed **except** for these:

- `bfd`
- `check link`
- `require route refresh`
- `require enhanced route refresh`
- `require AS4`

- require extended messages
- require hostname
- require graceful restart
- require long lived graceful restart
- interface range if the already existing option is updated to another value.

Channel options that cause a restart when changed:

- table
- secondary
- validate
- graceful restart
- long lived graceful restart
- long lived stale time
- next hop address
- extended next hop
- add paths
- import table
- export table
- igp table
- base table

Channel options that may cause a restart (when changed to a value incompatible with the current protocol state):

- mandatory
- min long lived stale time
- max long lived stale time

All these configuration changes apply also to the spawned dynamic BGP sessions and therefore reconfiguring the parent protocol may lead to shutdown of some or all of the dynamic BGP sessions.

6.4.7 Attributes

BGP defines several route attributes. Some of them (those marked with 'I' in the table below) are available on internal BGP connections only, some of them (marked with 'O') are optional.

bgppath bgp_path
: Sequence of AS numbers describing the AS path the packet will travel through when forwarded according to the particular route. In case of internal BGP it doesn't contain the number of the local AS.

int bgp_local_pref [I]
: Local preference value used for selection among multiple BGP routes (see the selection rules above). It's used as an additional metric which is propagated through the whole local AS.

`int bgp_med [O]`
The Multiple Exit Discriminator of the route is an optional attribute which is used on external (inter-AS) links to convey to an adjacent AS the optimal entry point into the local AS. The received attribute is also propagated over internal BGP links. The attribute value is zeroed when a route is exported to an external BGP instance to ensure that the attribute received from a neighboring AS is not propagated to other neighboring ASes. A new value might be set in the export filter of an external BGP instance. See RFC 4451 for further discussion of BGP MED attribute.

`enum bgp_origin`
Origin of the route: either `ORIGIN_IGP` if the route has originated in an interior routing protocol or `ORIGIN_EGP` if it's been imported from the `EGP` protocol (nowadays it seems to be obsolete) or `ORIGIN_INCOMPLETE` if the origin is unknown.

`ip bgp_next_hop`
Next hop to be used for forwarding of packets to this destination. On internal BGP connections, it's an address of the originating router if it's inside the local AS or a boundary router the packet will leave the AS through if it's an exterior route, so each BGP speaker within the AS has a chance to use the shortest interior path possible to this point.

`void bgp_atomic_aggr [O]`
This is an optional attribute which carries no value, but the sole presence of which indicates that the route has been aggregated from multiple routes by some router on the path from the originator.

`void bgp_aggregator [O]`
This is an optional attribute specifying AS number and IP address of the BGP router that created the route by aggregating multiple BGP routes. Currently, the attribute is not accessible from filters.

`clist bgp_community [O]`
List of community values associated with the route. Each such value is a pair (represented as a `pair` data type inside the filters) of 16-bit integers, the first of them containing the number of the AS which defines the community and the second one being a per-AS identifier. There are lots of uses of the community mechanism, but generally they are used to carry policy information like "don't export to USA peers". As each AS can define its own routing policy, it also has a complete freedom about which community attributes it defines and what will their semantics be.

`eclist bgp_ext_community [O]`
List of extended community values associated with the route. Extended communities have similar usage as plain communities, but they have an extended range (to allow 4B ASNs) and a nontrivial structure with a type field. Individual community values are represented using an `ec` data type inside the filters.

`lclist bgp_large_community [O]`
List of large community values associated with the route. Large BGP communities is another variant of communities, but contrary to extended communities they behave very much the same way as regular communities, just larger – they are uniform untyped triplets of 32bit numbers. Individual community values are represented using an `lc` data type inside the filters.

`quad bgp_originator_id [I, O]`
This attribute is created by the route reflector when reflecting the route and contains the router ID of the originator of the route in the local AS.

`clist bgp_cluster_list [I, O]`
This attribute contains a list of cluster IDs of route reflectors. Each route reflector prepends its cluster ID when reflecting the route.

`void bgp_pmsi_tunnel [O]`
The PMSI (P-Multicast Service Interface) tunnel BGP attribute is carried by EVPN IMET routes to describe the multicast tunnel information that carries BUM (Broadcast, Unknown, Multicast) traffic. It specifies the tunnel type (such as PIM-SSM, PIM-ASM, or Ingress Replication), the tunnel identifier, and the MPLS/VNI label for a specific EVPN instance. Currently, the attribute is not accessible from filters.

`void bgp_aigp [O]`
This attribute contains accumulated IGP metric, which is a total distance to the destination through multiple autonomous systems. Currently, the attribute is not accessible from filters.

`int bgp_otc [O]`
This attribute is defined in RFC 9234. OTC is a flag that marks routes that should be sent only to customers. If local role (p. 56) is configured it is set and checked automatically.

The equivalent filter code for the OTC check to pass the route in the provider or route server role, is: `!defined(bgp_otc)`

The equivalent filter code for the OTC check to pass the route in the peer role, is: `(!defined(bgp_otc) || bgp_otc == their_asn)` where `their_asn` stands for the configured AS number of the peer.

For attributes unknown by BIRD, the user can assign a name (on top level) to an attribute by its number. This defined name can be used then to get, set (as a bytestring, transitive) or unset the given attribute even though BIRD knows nothing about it.

Note that it is not possible to define an attribute with the same number as one known by BIRD, therefore use of this statement carries a risk of incompatibility with future BIRD versions.

`attribute bgp` *number* `bytestring` *name*`;`

6.4.8 Example

```
protocol bgp {
        local 198.51.100.14 as 65000;        # Use a private AS number
        neighbor 198.51.100.130 as 64496;    # Our neighbor ...
        multihop;                            # ... which is connected indirectly
        authentication ao;                   # We use TCP-AO authentication
        keys {
                key {
                        id 0;
                        secret "hello321";
                        algorithm hmac sha256;
                        preferred;
                };
                key {
                        send id 2;
                        recv id 1;
                        secret "bye123";
                        algorithm cmac aes128;
                };
        };
        ipv4 {
                export filter {                      # We use non-trivial export rules
                        if source = RTS_STATIC then { # Export only static routes
                                # Assign our community
                                bgp_community.add((65000,64501));
                                # Artificially increase path length
                                # by advertising local AS number twice
                                if bgp_path ~ [= 65000 =] then
                                        bgp_path.prepend(65000);
                                accept;
                        }
                        reject;
                };
                import all;
                next hop self; # advertise this router as next hop
                igp table myigptable4; # IGP table for routes with IPv4 nexthops
                igp table myigptable6; # IGP table for routes with IPv6 nexthops
```

```
        };
        ipv6 {
                export filter mylargefilter; # We use a named filter
                import all;
                missing lladdr self;
                igp table myigptable4; # IGP table for routes with IPv4 nexthops
                igp table myigptable6; # IGP table for routes with IPv6 nexthops
        };
        ipv4 multicast {
                import all;
                export filter someotherfilter;
                table mymulticasttable4; # Another IPv4 table, dedicated for multicast
                igp table myigptable4;
        };
}
```

Example configuration for BGP unnumbered automatic peering using router discovery:

```
# Table for discovered peers
neighbor table peers;

# RAdv protocol with router discovery enabled
protocol radv {
        neighbors { table peers; };     # Export discovered peers to this table

        interface "eth*" {
                # Normal RAdv configuration options...
                router discovery yes;   # Enable reception of ICMPv6 RAs
        };
}

# Dynamic BGP with neighbor channel for automatic peering
protocol bgp bgp_root {
        local as 65000;
        neighbor range fe80::/64 external;      # Accept link-local peers
        dynamic name "bgp_auto_";       # Name spawned sessions with this prefix
        dynamic name digits 2;          # Number of digits in name after prefix

        ipv4 {
                export all;
                import all;
                extended next hop;      # Allow IPv6 next hops for IPv4 routes
        };

        ipv6 {
                export all;
                import all;
        };

        neighbors {
                table peers;            # Import peer discovery objects from this table
                export all;             # Allow all discovered peers to trigger sessions
        };
}
```

6.5 BMP

The BGP Monitoring Protocol is used for monitoring BGP sessions and obtaining routing table data. The current implementation in BIRD is a preliminary release with a limited feature set, it will be subject to significant changes in the future.

The implementation supports monitoring protocol state changes, pre-policy routes (in BGP import tables (p. 58)) and post-policy routes (in regular routing tables). All BGP protocols are monitored automatically.

6.5.1 Configuration (incomplete)

`tx buffer limit` *number*
: How much data we are going to queue before we call the session stuck and restart it, in megabytes. Default value: 1024 (effectively 1 gigabyte).

6.5.2 Example

```
protocol bmp {
        # The monitoring station to connect to
        station address ip 198.51.100.10 port 1790;

        # Monitor received routes (in import table)
        monitoring rib in pre_policy;

        # Monitor accepted routes (passed import filters)
        monitoring rib in post_policy;

        # Allow only 64M of pending data
        tx buffer limit 64;
}
```

6.6 Bridge

The Bridge protocol is responsible for synchronization of BIRD's ethernet table with a forwarding database of a bridge device in the OS kernel. Similarly to the Kernel (p. 71) protocol, it sends all ethernet table updates to the OS kernel, listens to notifications from the OS kernel and from time to time it scans the OS bridge forwarding database to see whether some routes have changed.

The Bridge protocol support not only regular forwarding entries to local interfaces, but also forwarding entries with VXLAN encapsulation to remote targets. Due to peculiarities of Linux OS networking stack, VXLAN interface bound to the bridge interface has its own forwarding database. The Bridge protocol in BIRD is responsible for managing both.

Currently, the bridge protocol is limited to Linux.

6.6.1 Configuration

The Bridge protocol uses one ethernet channel. There are some options:

`bridge device "`*name*`"`
: The name of bridge device that is managed by the Bridge protocol instance. Mandatory.

`vlan filtering` *switch*
: Specify whether the bridge is VLAN-aware or not. Allows export and import of ethernet forwarding entries with defined VLAN ID. The OS bridge device must be created with `vlan_filtering 1` in order to be VLAN-aware. Default: off.

`scan time` *time*
: Time between two consecutive scans of the forwarding database of the bridge device. Default: 60 s.

6.6.2 Example

```
eth table ethtab0;

protocol static {
        eth { table ethtab0; };

        # Locally reachable MAC address
        route eth 12:20:30:40:50:60 via "eth0";

        # Remotely reachable MAC address through VXLAN (dst 10.1.1.100 vni 100);
        route eth 12:20:30:40:50:80 via 10.1.1.100 dev "vxlan0" mpls 100;
}

protocol bridge {
        eth { table ethtab0; export all; };

        bridge device "br0";
        scan time 20 s;
}
```

6.6.3 OS setup

The user is responsible to prepare bridge and (optionally) VXLAN interfaces on OS level. Here is an example how to setup simple bridge with VXLAN interface that uses fixed VNI:

```
ip link add name $BRIDGE type bridge
ip link set $BRIDGE up

ip link add name $TUNNEL type vxlan id $VNI local $ROUTER_ADDR dstport 4789 nolearning
ip link set $TUNNEL master $BRIDGE
bridge link set dev $TUNNEL learning off
ip link set $TUNNEL up
```

And here is an example how to setup VLAN-aware bridge with VXLAN interface that allows multiple VNIs and dynamic mapping between VNIs and VLANs:

```
ip link add name $BRIDGE type bridge vlan_filtering 1
ip link set $BRIDGE up

ip link add name $TUNNEL type vxlan local $ROUTER_ADDR dstport 4789 nolearning external
ip link set $TUNNEL master $BRIDGE
bridge link set dev $TUNNEL learning off
bridge link set dev $TUNNEL vlan_tunnel on
ip link set $TUNNEL up
```

When VLANs are defined by EVPN protocol, The membership of the VXLAN interface in VLANs is managed by BIRD, while user is responsible for setting VLANs for other (physical) interfaces in the bridge.

6.7 Device

The Device protocol is not a real routing protocol. It doesn't generate any routes and it only serves as a module for getting information about network interfaces from the kernel. This protocol supports no channel.

Except for very unusual circumstances, you probably should include this protocol in the configuration since almost all other protocols require network interfaces to be defined for them to work with.

6.7.1 Configuration

`scan time` *number*
Time in seconds between two scans of the network interface list. On systems where we are notified about interface status changes asynchronously (such as newer versions of Linux), we need to scan the list only in order to avoid confusion by lost notification messages, so the default time is set to a large value.

`interface` *pattern* `[, ...]`
By default, the Device protocol handles all interfaces without any configuration. Interface definitions allow to specify optional parameters for specific interfaces. See interface (p. 18) common option for detailed description. Currently only one interface option is available:

`preferred` *ip*
If a network interface has more than one IP address, BIRD chooses one of them as a preferred one. Preferred IP address is used as source address for packets or announced next hop by routing protocols. Precisely, BIRD chooses one preferred IPv4 address, one preferred IPv6 address and one preferred link-local IPv6 address. By default, BIRD chooses the first found IP address as the preferred one.

This option allows to specify which IP address should be preferred. May be used multiple times for different address classes (IPv4, IPv6, IPv6 link-local). In all cases, an address marked by operating system as secondary cannot be chosen as the primary one.

As the Device protocol doesn't generate any routes, it cannot have any attributes. Example configuration looks like this:

```
protocol device {
        scan time 10;           # Scan the interfaces often
        interface "eth0" {
                preferred 192.168.1.1;
                preferred 2001:db8:1:10::1;
        };
}
```

6.8 Direct

The Direct protocol is a simple generator of device routes for all the directly connected networks according to the list of interfaces provided by the kernel via the Device protocol. The Direct protocol supports both IPv4 and IPv6 channels; both can be configured simultaneously. It can also be configured with IPv6 SADR (p. 7) channel instead of regular IPv6 channel in order to be used together with SADR-enabled Babel protocol.

The question is whether it is a good idea to have such device routes in BIRD routing table. OS kernel usually handles device routes for directly connected networks by itself so we don't need (and don't want) to export these routes to the kernel protocol. OSPF protocol creates device routes for its interfaces itself and BGP protocol is usually used for exporting aggregate routes. But the Direct protocol is necessary for distance-vector protocols like RIP or Babel to announce local networks.

There are just few configuration options for the Direct protocol:

`interface` *pattern* `[, ...]`
By default, the Direct protocol will generate device routes for all the interfaces available. If you want to restrict it to some subset of interfaces or addresses (e.g. if you're using multiple routing tables for policy routing and some of the policy domains don't contain all interfaces), just use this clause. See interface (p. 18) common option for detailed description. The Direct protocol uses extended interface clauses.

`check link` *switch*
If enabled, a hardware link state (reported by OS) is taken into consideration. Routes for directly connected networks are generated only if link up is reported and they are withdrawn when link disappears (e.g., an ethernet cable is unplugged). Default value is no.

`host` *switch*
If enabled, for each address on the interface (including those marked secondary), a host route is created with the prefix length set to maximum. Such a route should never end up in the kernel forwarding table, and the user should take proper filtering actions to keep it safely away. Default: no.

`prefix` *switch*
If enabled, a regular device route is created for the whole prefix. Default: yes.

Direct device routes don't contain any specific attributes.

Example config might look like this:

```
protocol direct {
        ipv4;
        ipv6;
        interface -"arc*", "*";         # Exclude the ARCnets
}
```

6.9 EVPN

6.9.1 Introduction

The EVPN protocol is a component for Ethernet VPNs (RFC 7432) with VXLAN overlay (RFC 8365) and implements policies defined there. It serves as a translator between BGP EVPN routes and ethernet forwarding entries.

In BGP EVPNs, route distribution is controlled by Route Targets (RT). EVPN instances are associated with one or more RTs. Routes are also associated with one or more RTs, which are encoded as route target extended communities in bgp_ext_community (p. 63). A route is then imported into each EVPN instance that shares an associated Route Target. The EVPN protocol implements this mechanism through mandatory `import target` and `export target` protocol options.

One EVPN instance can consist of multiple broadcast domains (VLANs), these are identified by EVPN tags on BGP EVPN side and are mapped to VLAN IDs (VID) on ethernet side.

The current EVPN implementation in BIRD is a preliminary release limited to basic handling of MAC and IMET EVPN routes, it will be subject to changes in the future.

6.9.2 Configuration

The EVPN protocol instance always must have two channels: `evpn` and `eth`. For convenience, the default export filter in these channels is `all`, as the primary way to control import and export of routes is through protocol options `import target` and `export target`. If custom filters are used, note that the export filter of the input channel is applied before the route translation, while the import filter of the output channel is applied after that.

Note that outside of BIRD, it is necessary to configure bridge and VXLAN interface on Linux OS level. See Bridge (p. 66) protocol for details. When VLANs are used, EVPN and Bridge protocols in BIRD are responsible for configuring VLANs on the bridge interface for the VXLAN interface, while user is responsible for setting VLANs for other (physical) interfaces in the bridge.

```
protocol evpn [<name>] {
        route distinguisher <rd>;
        rd <rd>;
        import target <ec> | <ec-set>
        export target <ec> | <ec-set>
        route target <ec> | <ec-set>
        vni <number>;
        vid <number>;
        tag <number>;
        encapsulation vxlan {
```

```
                tunnel device "<name>";
                router address <ip>;
        };
        vlan <number> {
                range <number>;
                vni <number>;
                vid <number>;
        };
}
```

`route distinguisher` *rd*
: The route distinguisher that is attached to routes in the export direction. Mandatory.

`rd` *rd*
: A shorthand for the option `route distinguisher`.

`import target` *ec*|*ec-set*|`none`|`all`
: Route target extended communities specifying which routes should be imported. Either one community or a set. A route is imported if there is non-empty intersection between extended communities of the route and the import target of the EVPN protocol. Mandatory.

`export target` *ec*|*ec-set*|`none`
: Route target extended communities that are attached to the route in the export direction. Either one community or a set. Other route target extended communities are removed. Mandatory.

`route target` *ec*|*ec-set*|`none`
: A shorthand for both `import target` and `export target`.

`encapsulation` *type* { *options* }
: The encapsulation block specifies how ethernet frames are encapsulated when forwarded through overlay network. Currently, the only supported encapsulation type is `vxlan`.

`tunnel device "`*name*`"`
: The name of VXLAN tunnel device that is used for encapsulation. Mandatory.

`router address` *ip*
: The IP address used for encapsulated traffic. Can be usually autodetected from the VXLAN device.

`tag` *number*
: EVPN tag is used in BGP EVPN to identify specific broadcast domain. Must be unique per EVPN instance. Can be used as a protocol option to specify non-zero tag for the EVPN instance. Default: 0

`vni` *number*
: VNI label is used in dataplane encapsulation to identify specific broadcast domain. Must be unique per PE router. Can be used both as a protocol option (to specify VNI when the EVPN instance is one broadcast domain) or in `vlan` block (to specify VNI for each VLAN). Mandatory as a protocol option (unless autodetected from the VXLAN device).

`vid` *number*
: VLAN ID is used on ethernet to identify specific ethernet VLAN. Must be unique per ethernet network. Can be used both as a protocol option (to specify VID when the EVPN instance is one broadcast domain) or in `vlan` block (to specify VID for each VLAN). Default: 0 (untagged) as a protocol option.

`vlan` *number* { *options* }
: Defines a separate broadcast domain (VLAN) in an EVPN instance. Each VLAN has its own EVPN tag, encapsulation VNI, and ethernet VID, all initialized to the same value (the `vlan` argument), but VNI and VID can be changed by options in `vlan` block. When used with `range` option, it can define an interval of VLANs instead of just one VLAN.

`range` *number*
: Extends `vlan` block to define an interval of VLANs instead of just one VLAN. Defined VLANs have consecutive values of EVPN tags, VNIs, and VIDs. See example (p. 71).

6.9.3 Attributes

The EVPN protocol does not define any route attributes, but uses several BGP attributes for EVPN routes.

6.9.4 Example

Here is an example of EVPN setup with one VPN and BGP uplink. Locally reachable MAC addresses are learned by the Bridge protocol instance `bridge0` as ethernet forwarding entries and stored in the table `ethtab0`, then translated by the EVPN protocol instance `evpn0` and stored in global EVPN table `evpntab`. Routes can also be exchanged through BGP with different sites hosting that VPN. Remotelly reachable MAC addresses are received by BGP as EVPN routes, translated by `evpn0` to ethernet forwarding entries directed at `vxlan0` interface and then exported to the kernel by `bridge0`. Forwarding of VPN traffic through the network is handled by VXLAN.

```
evpn table evpntab;

# Exchange EVPN routes through BGP
protocol bgp {
        evpn { table evpntab; import all; export all; };
        local 10.0.0.1 as 10;
        neighbor 10.0.0.2 as 10;
}

# EVPN Instance 0
eth table ethtab0;

protocol bridge bridge0 {
        eth { table ethtab0; export all; };

        bridge device "br0";
        vlan filtering;
}

protocol evpn evpn0 {
        eth { table ethtab0; };
        evpn { table evpntab; };

        rd 10:12;
        route target (rt, 10, 10);

        encapsulation vxlan {
                tunnel device "vxlan0";
        };

        # VNI for untagged traffic
        vni 100;

        # VLANs 20-29, VNIs 120-129
        vlan 20 { range 10; vni 120; };
}
```

6.10 Kernel

The Kernel protocol is not a real routing protocol. Instead of communicating with other routers in the network, it performs synchronization of BIRD's routing tables with the OS kernel. Basically, it sends all routing table updates to the kernel and from time to time it scans the kernel tables to see whether some routes have disappeared (for example due to unnoticed up/down transition of an interface) or whether an

'alien' route has been added by someone else (depending on the `learn` switch, such routes are either ignored or accepted to our table).

Note that routes created by OS kernel itself, namely direct routes representing IP subnets of associated interfaces, are imported only with `learn all` enabled.

If your OS supports only a single routing table, you can configure only one instance of the Kernel protocol. If it supports multiple tables (in order to allow policy routing; such an OS is for example Linux), you can run as many instances as you want, but each of them must be connected to a different BIRD routing table and to a different kernel table.

Because the kernel protocol is partially integrated with the connected routing table, there are two limitations - it is not possible to connect more kernel protocols to the same routing table and changing route destination (gateway) in an export filter of a kernel protocol does not work. Both limitations can be overcome using another routing table and the pipe protocol.

The Kernel protocol supports both IPv4 and IPv6 channels; only one channel can be configured in each protocol instance. On Linux, it also supports IPv6 SADR (p. 7) and MPLS (p. 8) channels.

6.10.1 Configuration

`persist` *switch*
: Tell BIRD to leave all its routes in the routing tables when it exits (instead of cleaning them up).

`scan time` *number*
: Time in seconds between two consecutive scans of the kernel routing table.

`learn` *switch*|`all`
: Enable learning of routes added to the kernel routing tables by other routing daemons or by the system administrator. This is possible only on systems which support identification of route authorship. By default, routes created by kernel (marked as "proto kernel") are not imported. Use `learn all` option to import even these routes.

`kernel table` *number*
: Select which kernel table should this particular instance of the Kernel protocol work with. Available only on systems supporting multiple routing tables.

`metric` *number*
: (Linux) Use specified value as a kernel metric (priority) for all routes sent to the kernel. When multiple routes for the same network are in the kernel routing table, the Linux kernel chooses one with lower metric. Also, routes with different metrics do not clash with each other, therefore using dedicated metric value is a reliable way to avoid overwriting routes from other sources (e.g. kernel device routes). Metric 0 has a special meaning of undefined metric, in which either OS default is used, or per-route metric can be set using `krt_metric` attribute. Default: 32.

`graceful restart` *switch*
: Participate in graceful restart recovery. If this option is enabled and a graceful restart recovery is active, the Kernel protocol will defer synchronization of routing tables until the end of the recovery. Note that import of kernel routes to BIRD is not affected.

`merge paths` *switch* `[limit` *number*`]`
: Usually, only best routes are exported to the kernel protocol. With path merging enabled, both best routes and equivalent non-best routes are merged during export to generate one ECMP (equal-cost multipath) route for each network. This is useful e.g. for BGP multipath. Note that best routes are still pivotal for route export (responsible for most properties of resulting ECMP routes), while exported non-best routes are responsible just for additional multipath next hops. This option also allows to specify a limit on maximal number of nexthops in one route. By default, multipath merging is disabled. If enabled, default value of the limit is 16.

`netlink rx buffer` *number*
: (Linux) Set kernel receive buffer size (in bytes) for the netlink socket. The default value is OS-dependent (from the `/proc/sys/net/core/rmem_default` file), If you get some "Kernel dropped some netlink message ..." warnings, you may increase this value.

6.10.2 Attributes

The Kernel protocol defines several attributes. These attributes are translated to appropriate system (and OS-specific) route attributes. We support these attributes:

`int krt_source`
The original source of the imported kernel route. The value is system-dependent. On Linux, it is a value of the protocol field of the route. See /etc/iproute2/rt_protos for common values. On BSD, it is based on STATIC and PROTOx flags. The attribute is read-only.

`int krt_metric`
(Linux) The kernel metric of the route. When multiple same routes are in a kernel routing table, the Linux kernel chooses one with lower metric. Note that preferred way to set kernel metric is to use protocol option `metric`, unless per-route metric values are needed.

`ip krt_prefsrc`
(Linux) The preferred source address. Used in source address selection for outgoing packets. Has to be one of the IP addresses of the router.

`int krt_realm`
(Linux) The realm of the route. Can be used for traffic classification.

`int krt_scope`
(Linux IPv4) The scope of the route. Valid values are 0-254, although Linux kernel may reject some values depending on route type and nexthop. It is supposed to represent 'indirectness' of the route, where nexthops of routes are resolved through routes with a higher scope, but in current kernels anything below *link* (253) is treated as *global* (0). When not present, global scope is implied for all routes except device routes, where link scope is used by default.

In Linux, there is also a plenty of obscure route attributes mostly focused on tuning TCP performance of local connections. BIRD supports most of these attributes, see Linux or iproute2 documentation for their meaning. Attributes `krt_lock_*` and `krt_feature_*` have type bool, `krt_congctl` has type string, others have type int. Supported attributes are:
`krt_mtu`, `krt_lock_mtu`, `krt_window`, `krt_lock_window`, `krt_rtt`, `krt_lock_rtt`, `krt_rttvar`, `krt_lock_rttvar`, `krt_ssthresh`, `krt_lock_ssthresh`, `krt_cwnd`, `krt_lock_cwnd`, `krt_advmss`, `krt_lock_advmss`, `krt_reordering`, `krt_lock_reordering`, `krt_hoplimit`, `krt_lock_hoplimit`, `krt_rto_min`, `krt_lock_rto_min`, `krt_initcwnd`, `krt_lock_initcwnd`, `krt_initrwnd`, `krt_lock_initrwnd`, `krt_quickack`, `krt_lock_quickack`, `krt_congctl`, `krt_lock_congctl`, `krt_fastopen_no_cookie`, `krt_lock_fastopen_no_cookie`, `krt_feature_ecn`, `krt_feature_allfrag`

6.10.3 Example

A simple configuration can look this way:

```
protocol kernel {
        export all;
}
```

Or for a system with two routing tables:

```
protocol kernel {               # Primary routing table
        learn;                  # Learn alien routes from the kernel
        persist;                # Do not remove routes on bird shutdown
        scan time 10;           # Scan kernel routing table every 10 seconds
        ipv4 {
                import all;
                export all;
        };
}
```

```
protocol kernel {               # Secondary routing table
        kernel table 100;
        ipv4 {
                table auxtable;
                export all;
        };
}
```

6.11 L3VPN

6.11.1 Introduction

The L3VPN protocol serves as a translator between IP routes and VPN routes. It is a component for BGP/MPLS IP VPNs (RFC 4364) and implements policies defined there. In import direction (VPN -> IP), VPN routes matching import target specification are stripped of route distinguisher and MPLS labels and announced as IP routes, In export direction (IP -> VPN), IP routes are expanded with specific route distinguisher, export target communities and MPLS label and announced as labeled VPN routes. Unlike the Pipe protocol, the L3VPN protocol propagates just the best route for each network.

In BGP/MPLS IP VPNs, route distribution is controlled by Route Targets (RT). VRFs are associated with one or more RTs. Routes are also associated with one or more RTs, which are encoded as route target extended communities in bgp_ext_community (p. 63). A route is then imported into each VRF that shares an associated Route Target. The L3VPN protocol implements this mechanism through mandatory `import target` and `export target` protocol options.

6.11.2 Configuration

L3VPN configuration consists of a few mandatory options and multiple channel definitions. For convenience, the default export filter in L3VPN channels is `all`, as the primary way to control import and export of routes is through protocol options `import target` and `export target`. If custom filters are used, note that the export filter of the input channel is applied before the route translation, while the import filter of the output channel is applied after that.

In contrast to the Pipe protocol, the L3VPN protocol can handle both IPv4 and IPv6 routes in one instance, also both IP side and VPN side are represented as separate channels, although that may change in the future. The L3VPN is always MPLS-aware protocol, therefore a MPLS channel is mandatory. Altogether, L3VPN could have up to 5 channels: `ipv4`, `ipv6`, `vpn4`, `vpn6`, and `mpls`.

`route distinguisher` *rd*
: The route distinguisher that is attached to routes in the export direction. Mandatory.

`rd` *rd*
: A shorthand for the option `route distinguisher`.

`import target` *ec*|*ec-set*|`none`|`all`
: Route target extended communities specifying which routes should be imported. Either one community or a set. A route is imported if there is non-empty intersection between extended communities of the route and the import target of the L3VPN protocol. Mandatory.

`export target` *ec*|*ec-set*|`none`
: Route target extended communities that are attached to the route in the export direction. Either one community or a set. Other route target extended communities are removed. Mandatory.

`route target` *ec*|*ec-set*|`none`
: A shorthand for both `import target` and `export target`.

6.11.3 Attributes

The L3VPN protocol does not define any route attributes.

6.11.4 Example

Here is an example of L3VPN setup with one VPN and BGP uplink. IP routes learned from a customer in the VPN are stored in `vrf0vX` tables, which are mapped to kernel VRF vrf0. Routes can also be exchanged through BGP with different sites hosting that VPN. Forwarding of VPN traffic through the network is handled by MPLS.

Omitted from the example are some routing protocol to exchange routes with the customer and some sort of MPLS-aware IGP to resolve next hops for BGP VPN routes.

```
# MPLS basics
mpls domain mdom;
mpls table  mtab;

protocol kernel krt_mpls {
        mpls { table mtab; export all; };
}

vpn4 table vpntab4;
vpn6 table vpntab6;

# Exchange VPN routes through BGP
protocol bgp {
        vpn4 { table vpntab4; import all; export all; };
        vpn6 { table vpntab6; import all; export all; };
        mpls { label policy aggregate; };
        local 10.0.0.1 as 10;
        neighbor 10.0.0.2 as 10;
}

# VRF 0
ipv4 table vrf0v4;
ipv6 table vrf0v6;

protocol kernel kernel0v4 {
        vrf "vrf0";
        ipv4 { table vrf0v4; export all; };
        kernel table 100;
}

protocol kernel kernel0v6 {
        vrf "vrf0";
        ipv6 { table vrf0v6; export all; };
        kernel table 100;
}

protocol l3vpn l3vpn0 {
        vrf "vrf0";
        ipv4 { table vrf0v4; };
        ipv6 { table vrf0v6; };
        vpn4 { table vpntab4; };
        vpn6 { table vpntab6; };
        mpls { label policy vrf; };

        rd 10:12;
        import target [(rt, 10, 32..40)];
        export target [(rt, 10, 30), (rt, 10, 31)];
}
```

6.12 MRT

6.12.1 Introduction

The MRT protocol is a component responsible for handling the Multi-Threaded Routing Toolkit (MRT) routing information export format, which is mainly used for collecting and analyzing of routing information from BGP routers. The MRT protocol can be configured to do periodic dumps of routing tables, created MRT files can be analyzed later by other tools. Independent MRT table dumps can also be requested from BIRD client. There is also a feature to save incoming BGP messages in MRT files, but it is controlled by mrtdump (p. 17) options independently of MRT protocol, although that might change in the future.

BIRD implements the main MRT format specification as defined in RFC 6396 and the ADD_PATH extension (RFC 8050).

6.12.2 Configuration

MRT configuration consists of several statements describing routing table dumps. Multiple independent periodic dumps can be done as multiple MRT protocol instances. The MRT protocol does not use channels. There are two mandatory statements: `filename` and `period`.

The behavior can be modified by following configuration parameters:

`table` *name* | "*pattern*"
: Specify a routing table (or a set of routing tables described by a wildcard pattern) that are to be dumped by the MRT protocol instance. Default: the master table.

`filter` { *filter commands* }
: The MRT protocol allows to specify a filter that is applied to routes as they are dumped. Rejected routes are ignored and not saved to the MRT dump file. Default: no filter.

`where` *filter expression*
: An alternative way to specify a filter for the MRT protocol.

`filename` "*filename*"
: Specify a filename for MRT dump files. The filename may contain time format sequences with *strftime(3)* notation (see *man strftime* for details), there is also a sequence "%N" that is expanded to the name of dumped table. Therefore, each periodic dump of each table can be saved to a different file. Mandatory, see example below.

`period` *number*
: Specify the time interval (in seconds) between periodic dumps. Mandatory.

`always add path` *switch*
: The MRT format uses special records (specified in RFC 8050) for routes received using BGP ADD_PATH extension to keep Path ID, while other routes use regular records. This has advantage of better compatibility with tools that do not know special records, but it loses information about which route is the best route. When this option is enabled, both ADD_PATH and non-ADD_PATH routes are stored in ADD_PATH records and order of routes for network is preserved. Default: disabled.

6.12.3 Example

```
protocol mrt {
        table "tab*";
        where source = RTS_BGP;
        filename "/var/log/bird/%N_%F_%T.mrt";
        period 300;
}
```

6.13 OSPF

6.13.1 Introduction

Open Shortest Path First (OSPF) is a quite complex interior gateway protocol. The current IPv4 version (OSPFv2) is defined in RFC 2328 and the current IPv6 version (OSPFv3) is defined in RFC 5340 It's a link state (a.k.a. shortest path first) protocol – each router maintains a database describing the autonomous system's topology. Each participating router has an identical copy of the database and all routers run the same algorithm calculating a shortest path tree with themselves as a root. OSPF chooses the least cost path as the best path.

In OSPF, the autonomous system can be split to several areas in order to reduce the amount of resources consumed for exchanging the routing information and to protect the other areas from incorrect routing data. Topology of the area is hidden to the rest of the autonomous system.

Another very important feature of OSPF is that it can keep routing information from other protocols (like Static or BGP) in its link state database as external routes. Each external route can be tagged by the advertising router, making it possible to pass additional information between routers on the boundary of the autonomous system.

OSPF quickly detects topological changes in the autonomous system (such as router interface failures) and calculates new loop-free routes after a short period of convergence. Only a minimal amount of routing traffic is involved.

Each router participating in OSPF routing periodically sends Hello messages to all its interfaces. This allows neighbors to be discovered dynamically. Then the neighbors exchange theirs parts of the link state database and keep it identical by flooding updates. The flooding process is reliable and ensures that each router detects all changes.

6.13.2 Configuration

First, the desired OSPF version can be specified by using `ospf v2` or `ospf v3` as a protocol type. By default, OSPFv2 is used. In the main part of configuration, there can be multiple definitions of OSPF areas, each with a different id. These definitions includes many other switches and multiple definitions of interfaces. Definition of interface may contain many switches and constant definitions and list of neighbors on nonbroadcast networks.

OSPFv2 needs one IPv4 channel. OSPFv3 needs either one IPv6 channel, or one IPv4 channel (RFC 5838). Therefore, it is possible to use OSPFv3 for both IPv4 and Pv6 routing, but it is necessary to have two protocol instances anyway. If no channel is configured, appropriate channel is defined with default parameters.

```
protocol ospf [v2|v3] <name> {
        rfc1583compat <switch>;
        rfc5838 <switch>;
        instance id <number>;
        stub router <switch>;
        tick <number>;
        ecmp <switch> [limit <number>];
        merge external <switch>;
        graceful restart <switch>|aware;
        graceful restart time <number>;
        area <id> {
                stub;
                nssa;
                summary <switch>;
                default nssa <switch>;
                default cost <number>;
                default cost2 <number>;
                translator <switch>;
                translator stability <number>;

                networks {
```

```
        <prefix>;
        <prefix> hidden;
};
external {
        <prefix>;
        <prefix> hidden;
        <prefix> tag <number>;
};
stubnet <prefix>;
stubnet <prefix> {
        hidden <switch>;
        summary <switch>;
        cost <number>;
};
interface <interface pattern> [instance <number>] {
        cost <number>;
        stub <switch>;
        hello <number>;
        poll <number>;
        retransmit <number>;
        priority <number>;
        wait <number>;
        dead count <number>;
        dead <number>;
        secondary <switch>;
        rx buffer [normal|large|<number>];
        tx length <number>;
        type [broadcast|bcast|pointopoint|ptp|
                nonbroadcast|nbma|pointomultipoint|ptmp];
        link lsa suppression <switch>;
        strict nonbroadcast <switch>;
        real broadcast <switch>;
        ptp netmask <switch>;
        ptp address <switch>;
        check link <switch>;
        bfd <switch>;
        ecmp weight <number>;
        ttl security [<switch>; | tx only]
        tx class|dscp <number>;
        tx priority <number>;
        authentication none|simple|cryptographic;
        password "<text>";
        password "<text>" {
                id <number>;
                generate from "<date>";
                generate to "<date>";
                accept from "<date>";
                accept to "<date>";
                from "<date>";
                to "<date>";
                algorithm ( keyed md5 | keyed sha1 | hmac sha1 | hmac sha25
        };
        neighbors {
                <ip>;
                <ip> eligible;
        };
};
```

```
                        virtual link <id> [instance <number>] {
                                hello <number>;
                                retransmit <number>;
                                wait <number>;
                                dead count <number>;
                                dead <number>;
                                authentication none|simple|cryptographic;
                                password "<text>";
                                password "<text>" {
                                        id <number>;
                                        generate from "<date>";
                                        generate to "<date>";
                                        accept from "<date>";
                                        accept to "<date>";
                                        from "<date>";
                                        to "<date>";
                                        algorithm ( keyed md5 | keyed sha1 | hmac sha1 | hmac sha25
                                };
                        };
                };
        }
```

`rfc1583compat` *switch*
This option controls compatibility of routing table calculation with RFC 1583. Default value is no.

`rfc5838` *switch*
Basic OSPFv3 is limited to IPv6 unicast routing. The RFC 5838 extension defines support for more address families (IPv4, IPv6, both unicast and multicast). The extension is enabled by default, but can be disabled if necessary, as it restricts the range of available instance IDs. Default value is yes.

`instance id` *number*
When multiple OSPF protocol instances are active on the same links, they should use different instance IDs to distinguish their packets. Although it could be done on per-interface basis, it is often preferred to set one instance ID to whole OSPF domain/topology (e.g., when multiple instances are used to represent separate logical topologies on the same physical network). This option specifies the instance ID for all interfaces of the OSPF instance, but can be overridden by `interface` option. Default value is 0 unless OSPFv3-AF extended address families are used, see RFC 5838 for that case.

`stub router` *switch*
This option configures the router to be a stub router, i.e., a router that participates in the OSPF topology but does not allow transit traffic. In OSPFv2, this is implemented by advertising maximum metric for outgoing links. In OSPFv3, the stub router behavior is announced by clearing the R-bit in the router LSA. See RFC 6987 for details. Default value is no.

`tick` *number*
The routing table calculation and clean-up of areas' databases is not performed when a single link state change arrives. To lower the CPU utilization, it's processed later at periodical intervals of *number* seconds. The default value is 1.

`ecmp` *switch* `[limit` *number*`]`
This option specifies whether OSPF is allowed to generate ECMP (equal-cost multipath) routes. Such routes are used when there are several directions to the destination, each with the same (computed) cost. This option also allows to specify a limit on maximum number of nexthops in one route. By default, ECMP is enabled if supported by Kernel. Default value of the limit is 16.

`merge external` *switch*
This option specifies whether OSPF should merge external routes from different routers/LSAs for the same destination. When enabled together with `ecmp`, equal-cost external routes will be combined to multipath routes in the same way as regular routes. When disabled, external routes from different LSAs are treated as separate even if they represents the same destination. Default value is no.

`graceful restart` *switch*|`aware`
When an OSPF instance is restarted, neighbors break adjacencies and recalculate their routing tables, which disrupts packet forwarding even when the forwarding plane of the restarting router remains intact. RFC 3623 specifies a graceful restart mechanism to alleviate this issue. For OSPF graceful restart, restarting router originates Grace-LSAs, announcing intent to do graceful restart. Neighbors receiving these LSAs enter helper mode, in which they ignore breakdown of adjacencies, behave as if nothing is happening and keep old routes. When adjacencies are reestablished, the restarting router flushes Grace-LSAs and graceful restart is ended.

This option controls the graceful restart mechanism. It has three states: Disabled, when no support is provided. Aware, when graceful restart helper mode is supported, but no local graceful restart is allowed (i.e. helper-only role). Enabled, when the full graceful restart support is provided (i.e. both restarting and helper role). Note that proper support for local graceful restart requires also configuration of other protocols. Default: aware.

`graceful restart time` *number*
The restart time is announced in the Grace-LSA and specifies how long neighbors should wait for proper end of the graceful restart before exiting helper mode prematurely. Default: 120 seconds.

`area` *id*
This defines an OSPF area with given area ID (an integer or an IPv4 address, similarly to a router ID). The most important area is the backbone (ID 0) to which every other area must be connected.

`stub`
This option configures the area to be a stub area. External routes are not flooded into stub areas. Also summary LSAs can be limited in stub areas (see option `summary`). By default, the area is not a stub area.

`nssa`
This option configures the area to be a NSSA (Not-So-Stubby Area). NSSA is a variant of a stub area which allows a limited way of external route propagation. Global external routes are not propagated into a NSSA, but an external route can be imported into NSSA as a (area-wide) NSSA-LSA (and possibly translated and/or aggregated on area boundary). By default, the area is not NSSA.

`summary` *switch*
This option controls propagation of summary LSAs into stub or NSSA areas. If enabled, summary LSAs are propagated as usual, otherwise just the default summary route (0.0.0.0/0) is propagated (this is sometimes called totally stubby area). If a stub area has more area boundary routers, propagating summary LSAs could lead to more efficient routing at the cost of larger link state database. Default value is no.

`default nssa` *switch*
When `summary` option is enabled, default summary route is no longer propagated to the NSSA. In that case, this option allows to originate default route as NSSA-LSA to the NSSA. Default value is no.

`default cost` *number*
This option controls the cost of a default route propagated to stub and NSSA areas. Default value is 1000.

`default cost2` *number*
When a default route is originated as NSSA-LSA, its cost can use either type 1 or type 2 metric. This option allows to specify the cost of a default route in type 2 metric. By default, type 1 metric (option `default cost`) is used.

`translator` *switch*
This option controls translation of NSSA-LSAs into external LSAs. By default, one translator per NSSA is automatically elected from area boundary routers. If enabled, this area boundary router would unconditionally translate all NSSA-LSAs regardless of translator election. Default value is no.

`translator stability` *number*
This option controls the translator stability interval (in seconds). When the new translator is elected, the old one keeps translating until the interval is over. Default value is 40.

`networks` { *set* }
Definition of area IP ranges. This is used in summary LSA origination. Hidden networks are not propagated into other areas.

`external` { *set* }
Definition of external area IP ranges for NSSAs. This is used for NSSA-LSA translation. Hidden networks are not translated into external LSAs. Networks can have configured route tag.

`stubnet` *prefix* { *options* }
Stub networks are networks that are not transit networks between OSPF routers. They are also propagated through an OSPF area as a part of a link state database. By default, BIRD generates a stub network record for each primary network address on each OSPF interface that does not have any OSPF neighbors, and also for each non-primary network address on each OSPF interface. This option allows to alter a set of stub networks propagated by this router.

Each instance of this option adds a stub network with given network prefix to the set of propagated stub network, unless option `hidden` is used. It also suppresses default stub networks for given network prefix. When option `summary` is used, also default stub networks that are subnetworks of given stub network are suppressed. This might be used, for example, to aggregate generated stub networks.

`interface` *pattern* [`instance` *number*]
Defines that the specified interfaces belong to the area being defined. See interface (p. 18) common option for detailed description. In OSPFv2, extended interface clauses are used, because each network prefix is handled as a separate virtual interface.

You can specify alternative instance ID for the interface definition, therefore it is possible to have several instances of that interface with different options or even in different areas. For OSPFv2, instance ID support is an extension (RFC 6549) and is supposed to be set per-protocol. For OSPFv3, it is an integral feature.

`virtual link` *id* [`instance` *number*]
Virtual link to router with the router id. Virtual link acts as a point-to-point interface belonging to backbone. The actual area is used as a transport area. This item cannot be in the backbone. Like with `interface` option, you could also use several virtual links to one destination with different instance IDs.

`cost` *number*
Specifies output cost (metric) of an interface. Default value is 10.

`stub` *switch*
If set to interface it does not listen to any packet and does not send any hello. Default value is no.

`hello` *number*
Specifies interval in seconds between sending of Hello messages. Beware, all routers on the same network need to have the same hello interval. Default value is 10.

`poll` *number*
Specifies interval in seconds between sending of Hello messages for some neighbors on NBMA network. Default value is 20.

`retransmit` *number*
Specifies interval in seconds between retransmissions of unacknowledged updates. Default value is 5.

`transmit delay` *number*
Specifies estimated transmission delay of link state updates send over the interface. The value is added to LSA age of LSAs propagated through it. Default value is 1.

`priority` *number*
On every multiple access network (e.g., the Ethernet) Designated Router and Backup Designated router are elected. These routers have some special functions in the flooding process. Higher priority increases preferences in this election. Routers with priority 0 are not eligible. Default value is 1.

`wait` *number*
After start, router waits for the specified number of seconds between starting election and building adjacency. Default value is 4**hello*.

`dead count` *number*
When the router does not receive any messages from a neighbor in *dead count***hello* seconds, it will consider the neighbor down.

`dead` *number*
When the router does not receive any messages from a neighbor in *dead* seconds, it will consider the neighbor down. If both directives `dead count` and `dead` are used, `dead` has precedence.

`rx buffer` *number*
This option allows to specify the size of buffers used for packet processing. The buffer size should be bigger than maximal size of any packets. By default, buffers are dynamically resized as needed, but a fixed value could be specified. Value `large` means maximal allowed packet size - 65535.

`tx length` *number*
Transmitted OSPF messages that contain large amount of information are segmented to separate OSPF packets to avoid IP fragmentation. This option specifies the soft ceiling for the length of generated OSPF packets. Default value is the MTU of the network interface. Note that larger OSPF packets may still be generated if underlying OSPF messages cannot be splitted (e.g. when one large LSA is propagated).

`type broadcast|bcast`
BIRD detects a type of a connected network automatically, but sometimes it's convenient to force use of a different type manually. On broadcast networks (like ethernet), flooding and Hello messages are sent using multicasts (a single packet for all the neighbors). A designated router is elected and it is responsible for synchronizing the link-state databases and originating network LSAs. This network type cannot be used on physically NBMA networks and on unnumbered networks (networks without proper IP prefix).

`type pointopoint|ptp`
Point-to-point networks connect just 2 routers together. No election is performed and no network LSA is originated, which makes it simpler and faster to establish. This network type is useful not only for physically PtP ifaces (like PPP or tunnels), but also for broadcast networks used as PtP links. This network type cannot be used on physically NBMA networks.

`type nonbroadcast|nbma`
On NBMA networks, the packets are sent to each neighbor separately because of lack of multicast capabilities. Like on broadcast networks, a designated router is elected, which plays a central role in propagation of LSAs. This network type cannot be used on unnumbered networks.

`type pointomultipoint|ptmp`
This is another network type designed to handle NBMA networks. In this case the NBMA network is treated as a collection of PtP links. This is useful if not every pair of routers on the NBMA network has direct communication, or if the NBMA network is used as an (possibly unnumbered) PtP link.

`link lsa suppression` *switch*
In OSPFv3, link LSAs are generated for each link, announcing link-local IPv6 address of the router to its local neighbors. These are useless on PtP or PtMP networks and this option allows to suppress the link LSA origination for such interfaces. The option is ignored on other than PtP or PtMP interfaces. Default value is no.

`strict nonbroadcast` *switch*
If set, don't send hello to any undefined neighbor. This switch is ignored on other than NBMA or PtMP interfaces. Default value is no.

`real broadcast` *switch*
In `type broadcast` or `type ptp` network configuration, OSPF packets are sent as IP multicast packets. This option changes the behavior to using old-fashioned IP broadcast packets. This may be useful as

a workaround if IP multicast for some reason does not work or does not work reliably. This is a non-standard option and probably is not interoperable with other OSPF implementations. Default value is no.

`ptp netmask` *switch*
In `type ptp` network configurations, OSPFv2 implementations should ignore received netmask field in hello packets and should send hello packets with zero netmask field on unnumbered PtP links. But some OSPFv2 implementations perform netmask checking even for PtP links.

This option specifies whether real netmask will be used in hello packets on `type ptp` interfaces. You should ignore this option unless you meet some compatibility problems related to this issue. Default value is no for unnumbered PtP links, yes otherwise.

`ptp address` *switch*
In `type ptp` network configurations, OSPFv2 implementations should use IP address for regular PtP links and interface id for unnumbered PtP links in data field of link description records in router LSA. This data field has only local meaning for PtP links, but some broken OSPFv2 implementations assume there is an IP address and use it as a next hop in SPF calculations. Note that interface id for unnumbered PtP links is necessary when graceful restart is enabled to distinguish PtP links with the same local IP address.

This option specifies whether an IP address will be used in data field for `type ptp` interfaces, it is ignored for other interfaces. You should ignore this option unless you meet some compatibility problems related to this issue. Default value is no for unnumbered PtP links when graceful restart is enabled, yes otherwise.

`check link` *switch*
If set, a hardware link state (reported by OS) is taken into consideration. When a link disappears (e.g. an ethernet cable is unplugged), neighbors are immediately considered unreachable and only the address of the iface (instead of whole network prefix) is propagated. It is possible that some hardware drivers or platforms do not implement this feature. Default value is yes.

`bfd` *switch*
OSPF could use BFD protocol as an advisory mechanism for neighbor liveness and failure detection. If enabled, BIRD setups a BFD session for each OSPF neighbor and tracks its liveness by it. This has an advantage of an order of magnitude lower detection times in case of failure. Note that BFD protocol also has to be configured, see BFD (p. 41) section for details. Default value is no.

`ttl security` [*switch* | `tx only`]
TTL security is a feature that protects routing protocols from remote spoofed packets by using TTL 255 instead of TTL 1 for protocol packets destined to neighbors. Because TTL is decremented when packets are forwarded, it is non-trivial to spoof packets with TTL 255 from remote locations. Note that this option would interfere with OSPF virtual links.

If this option is enabled, the router will send OSPF packets with TTL 255 and drop received packets with TTL less than 255. If this option si set to `tx only`, TTL 255 is used for sent packets, but is not checked for received packets. Default value is no.

`tx class|dscp|priority` *number*
These options specify the ToS/DiffServ/Traffic class/Priority of the outgoing OSPF packets. See tx class (p. 18) common option for detailed description.

`ecmp weight` *number*
When ECMP (multipath) routes are allowed, this value specifies a relative weight used for nexthops going through the iface. Allowed values are 1-256. Default value is 1.

`authentication none`
No passwords are sent in OSPF packets. This is the default value.

`authentication simple`
Every packet carries 8 bytes of password. Received packets lacking this password are ignored. This authentication mechanism is very weak. This option is not available in OSPFv3.

`authentication cryptographic`
An authentication code is appended to every packet. The specific cryptographic algorithm is selected by option `algorithm` for each key. The default cryptographic algorithm for OSPFv2 keys is Keyed-MD5 and for OSPFv3 keys is HMAC-SHA-256. Passwords are not sent open via network, so this mechanism is quite secure. Packets can still be read by an attacker.

`password "`*text*`"`
Specifies a password used for authentication. See password (p. 19) common option for detailed description.

`neighbors` { *set* }
A set of neighbors to which Hello messages on NBMA or PtMP networks are to be sent. For NBMA networks, some of them could be marked as eligible. In OSPFv3, link-local addresses should be used, using global ones is possible, but it is nonstandard and might be problematic. And definitely, link-local and global addresses should not be mixed.

6.13.3 Attributes

OSPF defines four route attributes. Each internal route has a `metric`.

Metric is ranging from 1 to infinity (65535). External routes use `metric type 1` or `metric type 2`. A `metric of type 1` is comparable with internal `metric`, a `metric of type 2` is always longer than any `metric of type 1` or any `internal metric`. `Internal metric` or `metric of type 1` is stored in attribute `ospf_metric1`, `metric type 2` is stored in attribute `ospf_metric2`.

When both metrics are specified then `metric of type 2` is used. This is relevant e.g. when a type 2 external route is propagated from one OSPF domain to another and `ospf_metric1` is an internal distance to the original ASBR, while `ospf_metric2` stores the type 2 metric. Note that in such cases if `ospf_metric1` is non-zero then `ospf_metric2` is increased by one to ensure monotonicity of metric, as internal distance is reset to zero when an external route is announced.

Each external route can also carry attribute `ospf_tag` which is a 32-bit integer which is used when exporting routes to other protocols; otherwise, it doesn't affect routing inside the OSPF domain at all. The fourth attribute `ospf_router_id` is a router ID of the router advertising that route / network. This attribute is read-only. Default is `ospf_metric2 = 10000` and `ospf_tag = 0`.

6.13.4 Example

```
protocol ospf MyOSPF {
        ipv4 {
                export filter {
                        if source = RTS_BGP then {
                                ospf_metric1 = 100;
                                accept;
                        }
                        reject;
                };
        };
        area 0.0.0.0 {
                interface "eth*" {
                        cost 11;
                        hello 15;
                        priority 100;
                        retransmit 7;
                        authentication simple;
                        password "aaa";
                };
                interface "ppp*" {
                        cost 100;
                        authentication cryptographic;
```

```
                        password "abc" {
                                id 1;
                                generate to "2023-04-22 11:00:06";
                                accept from "2021-01-17 12:01:05";
                                algorithm hmac sha384;
                        };
                        password "def" {
                                id 2;
                                generate to "2025-07-22";
                                accept from "2021-02-22";
                                algorithm hmac sha512;
                        };
                };
                interface "arc0" {
                        cost 10;
                        stub yes;
                };
                interface "arc1";
        };
        area 120 {
                stub yes;
                networks {
                        172.16.1.0/24;
                        172.16.2.0/24 hidden;
                };
                interface "-arc0" , "arc*" {
                        type nonbroadcast;
                        authentication none;
                        strict nonbroadcast yes;
                        wait 120;
                        poll 40;
                        dead count 8;
                        neighbors {
                                192.168.120.1 eligible;
                                192.168.120.2;
                                192.168.120.10;
                        };
                };
        };
}
```

6.14 Pipe

6.14.1 Introduction

The Pipe protocol serves as a link between two routing tables, allowing routes to be passed from a table declared as primary (i.e., the one the pipe is connected to using the `table` configuration keyword) to the secondary one (declared using `peer table`) and vice versa, depending on what's allowed by the filters. Export filters control export of routes from the primary table to the secondary one, import filters control the opposite direction. Both tables must be of the same nettype.

The Pipe protocol retransmits all routes from one table to the other table, retaining their original source and attributes. If import and export filters are set to accept, then both tables would have the same content.

The primary use of multiple routing tables and the Pipe protocol is for policy routing, where handling of a single packet doesn't depend only on its destination address, but also on its source address, source interface, protocol type and other similar parameters. In many systems (Linux being a good example), the kernel allows to enforce routing policies by defining routing rules which choose one of several routing tables to be

used for a packet according to its parameters. Setting of these rules is outside the scope of BIRD's work (on Linux, you can use the `ip` command), but you can create several routing tables in BIRD, connect them to the kernel ones, use filters to control which routes appear in which tables and also you can employ the Pipe protocol for exporting a selected subset of one table to another one.

6.14.2 Configuration

Essentially, the Pipe protocol is just a channel connected to a table on both sides. Therefore, the configuration block for `protocol pipe` shall directly include standard channel config options; see the example below.

`peer table` *table*
Defines secondary routing table to connect to. The primary one is selected by the `table` keyword.

`max generation` *expr*
Sets maximal generation of route that may pass through this pipe. The generation value is increased by one by each pipe on its path. Not meeting this requirement causes an error message complaining about an overpiped route. If you have long chains of pipes, you probably want to raise this value; anyway the default of 16 should be enough for even most strange uses. Maximum is 254.

6.14.3 Attributes

The Pipe protocol doesn't define any route attributes.

6.14.4 Example

Let's consider a router which serves as a boundary router of two different autonomous systems, each of them connected to a subset of interfaces of the router, having its own exterior connectivity and wishing to use the other AS as a backup connectivity in case of outage of its own exterior line.

Probably the simplest solution to this situation is to use two routing tables (we'll call them `as1` and `as2`) and set up kernel routing rules, so that packets having arrived from interfaces belonging to the first AS will be routed according to `as1` and similarly for the second AS. Thus we have split our router to two logical routers, each one acting on its own routing table, having its own routing protocols on its own interfaces. In order to use the other AS's routes for backup purposes, we can pass the routes between the tables through a Pipe protocol while decreasing their preferences and correcting their BGP paths to reflect the AS boundary crossing.

```
ipv4 table as1;                         # Define the tables
ipv4 table as2;

protocol kernel kern1 {                 # Synchronize them with the kernel
        ipv4 { table as1; export all; };
        kernel table 1;
}

protocol kernel kern2 {
        ipv4 { table as2; export all; };
        kernel table 2;
}

protocol bgp bgp1 {                     # The outside connections
        ipv4 { table as1; import all; export all; };
        local as 1;
        neighbor 192.168.0.1 as 1001;
}

protocol bgp bgp2 {
        ipv4 { table as2; import all; export all; };
```

```
        local as 2;
        neighbor 10.0.0.1 as 1002;
}

protocol pipe {                         # The Pipe
        table as1;
        peer table as2;
        export filter {
                if net ~ [ 1.0.0.0/8+] then {   # Only AS1 networks
                        if preference>10 then preference = preference-10;
                        if source=RTS_BGP then bgp_path.prepend(1);
                        accept;
                }
                reject;
        };
        import filter {
                if net ~ [ 2.0.0.0/8+] then {   # Only AS2 networks
                        if preference>10 then preference = preference-10;
                        if source=RTS_BGP then bgp_path.prepend(2);
                        accept;
                }
                reject;
        };
}
```

6.15 RAdv

6.15.1 Introduction

The RAdv protocol is an implementation of Router Advertisements, which are used in the IPv6 stateless autoconfiguration. IPv6 routers send (in irregular time intervals or as an answer to a request) advertisement packets to connected networks. These packets contain basic information about a local network (e.g. a list of network prefixes), which allows network hosts to autoconfigure network addresses and choose a default route. BIRD implements router behavior as defined in RFC 4861, router preferences and specific routes (RFC 4191), and DNS extensions (RFC 6106).

The RAdv protocols supports the IPv6 channel (for the trigger prefix (p. 88) and more specific route propagation (p. 88)) and the `neighbors` channel (for the router discovery (p. 90)).

6.15.2 Configuration

There are several classes of definitions in RAdv configuration – interface definitions, prefix definitions and DNS definitions:

`interface` *pattern* `[, ...]` { *options* }
: Interface definitions specify a set of interfaces on which the protocol is activated and contain interface specific options. See interface (p. 18) common options for detailed description.

`prefix` *prefix* { *options* }
: Prefix definitions allow to modify a list of advertised prefixes. By default, the advertised prefixes are the same as the network prefixes assigned to the interface. For each network prefix, the matching prefix definition is found and its options are used. If no matching prefix definition is found, the prefix is used with default options.

  Prefix definitions can be either global or interface-specific. The second ones are part of interface options. The prefix definition matching is done in the first-match style, when interface-specific definitions are processed before global definitions. As expected, the prefix definition is matching if the network prefix is a subnet of the prefix in prefix definition.

`rdnss` { *options* }
RDNSS definitions allow to specify a list of advertised recursive DNS servers together with their options. As options are seldom necessary, there is also a short variant `rdnss` *address* that just specifies one DNS server. Multiple definitions are cumulative. RDNSS definitions may also be interface-specific when used inside interface options. By default, interface uses both global and interface-specific options, but that can be changed by `rdnss local` option.

`dnssl` { *options* }
DNSSL definitions allow to specify a list of advertised DNS search domains together with their options. Like `rdnss` above, multiple definitions are cumulative, they can be used also as interface-specific options and there is a short variant `dnssl` *domain* that just specifies one DNS search domain.

`custom option type` *number* `value` *bytestring*
Custom option definitions allow to define an arbitrary option to advertise. You need to specify the option type number and the binary payload of the option. The length field is calculated automatically. Like `rdnss` above, multiple definitions are cumulative, they can be used also as interface-specific options.

The following example advertises PREF64 option (RFC 8781) with prefix `2001:db8:a:b::/96` and the lifetime of `1 hour`:

```
custom option type 38 value hex:0e:10:20:01:0d:b8:00:0a:00:0b:00:00:00:00;
```

`trigger` *prefix*
RAdv protocol could be configured to change its behavior based on availability of routes. When this option is used, the protocol waits in suppressed state until a *trigger route* (for the specified network) is exported to the protocol, the protocol also returns to suppressed state if the *trigger route* disappears. Note that route export depends on specified export filter, as usual. This option could be used, e.g., for handling failover in multihoming scenarios.

During suppressed state, router advertisements are generated, but with some fields zeroed. Exact behavior depends on which fields are zeroed, this can be configured by `sensitive` option for appropriate fields. By default, just `default lifetime` (also called `router lifetime`) is zeroed, which means hosts cannot use the router as a default router. `preferred lifetime` and `valid lifetime` could also be configured as `sensitive` for a prefix, which would cause autoconfigured IPs to be deprecated or even removed.

`propagate routes` *switch*
This option controls propagation of more specific routes, as defined in RFC 4191. If enabled, all routes exported to the RAdv protocol, with the exception of the trigger prefix, are added to advertisments as additional options. The lifetime and preference of advertised routes can be set individually by `ra_lifetime` and `ra_preference` route attributes, or per interface by `route lifetime` and `route preference` options. Default: disabled.

Note that the RFC discourages from sending more than 17 routes and recommends the routes to be configured manually.

Interface specific options:

`max ra interval` *expr*
Unsolicited router advertisements are sent in irregular time intervals. This option specifies the maximum length of these intervals, in seconds. Valid values are 4-1800. Default: 600

`min ra interval` *expr*
This option specifies the minimum length of that intervals, in seconds. Must be at least 3 and at most 3/4 * `max ra interval`. Default: about 1/3 * `max ra interval`.

`min delay` *expr*
The minimum delay between two consecutive router advertisements, in seconds. Default: 3

`solicited ra unicast` *switch*
Solicited router advertisements are usually sent to all-nodes multicast group like unsolicited ones, but the router can be configured to send them as unicast directly to soliciting nodes instead. This is especially useful on wireless networks (see RFC 7772). Default: no

`managed` *switch*
This option specifies whether hosts should use DHCPv6 for IP address configuration. Default: no

`other config` *switch*
This option specifies whether hosts should use DHCPv6 to receive other configuration information. Default: no

`link mtu` *expr*
This option specifies which value of MTU should be used by hosts. 0 means unspecified. Default: 0

`reachable time` *expr*
This option specifies the time (in milliseconds) how long hosts should assume a neighbor is reachable (from the last confirmation). Maximum is 3600000, 0 means unspecified. Default 0.

`retrans timer` *expr*
This option specifies the time (in milliseconds) how long hosts should wait before retransmitting Neighbor Solicitation messages. 0 means unspecified. Default 0.

`current hop limit` *expr*
This option specifies which value of Hop Limit should be used by hosts. Valid values are 0-255, 0 means unspecified. Default: 64

`default lifetime` *expr* `[sensitive` *switch*`]`
This option specifies the time (in seconds) how long (since the receipt of RA) hosts may use the router as a default router. 0 means do not use as a default router. For `sensitive` option, see trigger (p. 88). Default: 3 * `max ra interval`, `sensitive` yes.

`default preference low|medium|high`
This option specifies the Default Router Preference value to advertise to hosts. Default: medium.

`route lifetime` *expr* `[sensitive` *switch*`]`
This option specifies the default value of advertised lifetime for specific routes; i.e., the time (in seconds) for how long (since the receipt of RA) hosts should consider these routes valid. A special value 0xffffffff represents infinity. The lifetime can be overriden on a per route basis by the ra_lifetime (p. 91) route attribute. Default: 3 * `max ra interval`, `sensitive` no.

For the `sensitive` option, see trigger (p. 88). If `sensitive` is enabled, even the routes with the `ra_lifetime` attribute become sensitive to the trigger.

`route preference low|medium|high`
This option specifies the default value of advertised route preference for specific routes. The value can be overriden on a per route basis by the ra_preference (p. 91) route attribute. Default: medium.

`prefix linger time` *expr*
When a prefix or a route disappears, it is advertised for some time with zero lifetime, to inform clients it is no longer valid. This option specifies the time (in seconds) for how long prefixes are advertised that way. Default: 3 * `max ra interval`.

`route linger time` *expr*
When a prefix or a route disappears, it is advertised for some time with zero lifetime, to inform clients it is no longer valid. This option specifies the time (in seconds) for how long routes are advertised that way. Default: 3 * `max ra interval`.

`rdnss local` *switch*
Use only local (interface-specific) RDNSS definitions for this interface. Otherwise, both global and local definitions are used. Could also be used to disable RDNSS for given interface if no local definitons are specified. Default: no.

`dnssl local` *switch*
Use only local DNSSL definitions for this interface. See `rdnss local` option above. Default: no.

`custom option local` *switch*
Use only local custom option definitions for this interface. See `rdnss local` option above. Default: no.

`router discovery` *switch*
Enable reception and processing of incoming ICMPv6 Router Advertisement messages. When enabled, the RAdv protocol listens for RA messages on configured interfaces and creates route-like objects in a peers routing table based on detected advertisements. These objects contain information about discovered routers and their lifetimes are managed by the Router Lifetime field in the RA messages. When an RA expires, the corresponding object is automatically removed from the peers table. Currently, this feature is primarily used in conjunction with BGP unnumbered automatic peering to automatically discover BGP neighbors. Default: no.

Prefix specific options

`skip` *switch*
This option allows to specify that given prefix should not be advertised. This is useful for making exceptions from a default policy of advertising all prefixes. Note that for withdrawing an already advertised prefix it is more useful to advertise it with zero valid lifetime. Default: no

`onlink` *switch*
This option specifies whether hosts may use the advertised prefix for onlink determination. Default: yes

`autonomous` *switch*
This option specifies whether hosts may use the advertised prefix for stateless autoconfiguration. Default: yes

`pd preferred`*switch*
This option specifies whether hosts should use prefix delegation in DHCPv6 instead of using stateless address autoconfiguration (SLAAC). Default: no.

`valid lifetime` *expr* `[sensitive` *switch*`]`
This option specifies the time (in seconds) how long (after the receipt of RA) the prefix information is valid, i.e., autoconfigured IP addresses can be assigned and hosts with that IP addresses are considered directly reachable. 0 means the prefix is no longer valid. For `sensitive` option, see trigger (p. 88). Default: 86400 (1 day), `sensitive` no.

`preferred lifetime` *expr* `[sensitive` *switch*`]`
This option specifies the time (in seconds) how long (after the receipt of RA) IP addresses generated from the prefix using stateless autoconfiguration remain preferred. For `sensitive` option, see trigger (p. 88). Default: 14400 (4 hours), `sensitive` no.

RDNSS specific options:

`ns` *address*
This option specifies one recursive DNS server. Can be used multiple times for multiple servers. It is mandatory to have at least one `ns` option in `rdnss` definition.

`lifetime [mult]` *expr*
This option specifies the time how long the RDNSS information may be used by clients after the receipt of RA. It is expressed either in seconds or (when `mult` is used) in multiples of `max ra interval`. Note that RDNSS information is also invalidated when `default lifetime` expires. 0 means these addresses are no longer valid DNS servers. Default: 3 * `max ra interval`.

DNSSL specific options:

`domain` *address*
This option specifies one DNS search domain. Can be used multiple times for multiple domains. It is mandatory to have at least one `domain` option in `dnssl` definition.

`lifetime [mult]` *expr*
This option specifies the time how long the DNSSL information may be used by clients after the receipt of RA. Details are the same as for RDNSS `lifetime` option above. Default: 3 * `max ra interval`.

6.15.3 Attributes

RAdv defines two route attributes:

`enum ra_preference`
: The preference of the route. The value can be *RA_PREF_LOW*, *RA_PREF_MEDIUM* or *RA_PREF_HIGH*. If the attribute is not set, the route preference (p. 89) option is used.

`int ra_lifetime`
: The advertised lifetime of the route, in seconds. The special value of 0xffffffff represents infinity. If the attribute is not set, the route lifetime (p. 89) option is used.

6.15.4 Example

```
ipv6 table radv_routes;                 # Manually configured routes go here

protocol static {
        ipv6 { table radv_routes; };

        route 2001:0DB8:4000::/48 unreachable;
        route 2001:0DB8:4010::/48 unreachable;

        route 2001:0DB8:4020::/48 unreachable {
                ra_preference = RA_PREF_HIGH;
                ra_lifetime = 3600;
        };
}

protocol radv {
        propagate routes yes;           # Propagate the routes from the radv_routes table
        ipv6 { table radv_routes; export all; };

        interface "eth2" {
                max ra interval 5;      # Fast failover with more routers
                managed yes;            # Using DHCPv6 on eth2
                prefix ::/0 {
                        autonomous off; # So do not autoconfigure any IP
                };
        };

        interface "eth*";               # No need for any other options

        prefix 2001:0DB8:1234::/48 {
                preferred lifetime 0;   # Deprecated address range
        };

        prefix 2001:0DB8:2000::/48 {
                autonomous off;         # Do not autoconfigure
        };

        rdnss 2001:0DB8:1234::10;       # Short form of RDNSS

        rdnss {
                lifetime mult 10;
                ns 2001:0DB8:1234::11;
                ns 2001:0DB8:1234::12;
        };
```

```
        dnssl {
                lifetime 3600;
                domain "abc.com";
                domain "xyz.com";
        };
}
```

6.16 RIP

6.16.1 Introduction

The RIP protocol (also sometimes called Rest In Pieces) is a simple protocol, where each router broadcasts (to all its neighbors) distances to all networks it can reach. When a router hears distance to another network, it increments it and broadcasts it back. Broadcasts are done in regular intervals. Therefore, if some network goes unreachable, routers keep telling each other that its distance is the original distance plus 1 (actually, plus interface metric, which is usually one). After some time, the distance reaches infinity (that's 15 in RIP) and all routers know that network is unreachable. RIP tries to minimize situations where counting to infinity is necessary, because it is slow. Due to infinity being 16, you can't use RIP on networks where maximal distance is higher than 15 hosts.

BIRD supports RIPv1 (RFC 1058), RIPv2 (RFC 2453), RIPng (RFC 2080), Triggered RIP for demand circuits (RFC 2091), and RIP cryptographic authentication (RFC 4822).

RIP is a very simple protocol, and it has a lot of shortcomings. Slow convergence, big network load and inability to handle larger networks makes it pretty much obsolete. It is still usable on very small networks.

6.16.2 Configuration

RIP configuration consists mainly of common protocol options and interface definitions, most RIP options are interface specific. RIPng (RIP for IPv6) protocol instance can be configured by using `rip ng` instead of just `rip` as a protocol type.

RIP needs one IPv4 channel. RIPng needs one IPv6 channel. If no channel is configured, appropriate channel is defined with default parameters.

```
protocol rip [ng] [<name>] {
        infinity <number>;
        ecmp <switch> [limit <number>];
        interface <interface pattern> {
                metric <number>;
                mode multicast|broadcast;
                passive <switch>;
                address <ip>;
                port <number>;
                version 1|2;
                split horizon <switch>;
                poison reverse <switch>;
                demand circuit <switch>;
                check zero <switch>;
                update time <number>;
                timeout time <number>;
                garbage time <number>;
                ecmp weight <number>;
                ttl security <switch>; | tx only;
                tx class|dscp <number>;
                tx priority <number>;
                rx buffer <number>;
                tx length <number>;
                check link <switch>;
```

```
                        authentication none|plaintext|cryptographic;
                        password "<text>";
                        password "<text>" {
                                id <number>;
                                generate from "<date>";
                                generate to "<date>";
                                accept from "<date>";
                                accept to "<date>";
                                from "<date>";
                                to "<date>";
                                algorithm ( keyed md5 | keyed sha1 | hmac sha1 |
                                        hmac sha256 | hmac sha384 | hmac sha512 );
                        };
                };
        }
```

`infinity` *number*
Selects the distance of infinity. Bigger values will make protocol convergence even slower. The default value is 16.

`ecmp` *switch* `[limit` *number*`]`
This option specifies whether RIP is allowed to generate ECMP (equal-cost multipath) routes. Such routes are used when there are several directions to the destination, each with the same (computed) cost. This option also allows to specify a limit on maximum number of nexthops in one route. By default, ECMP is enabled if supported by Kernel. Default value of the limit is 16.

`interface` *pattern* `[,` *...*`]` { *options* }
Interface definitions specify a set of interfaces on which the protocol is activated and contain interface specific options. See interface (p. 18) common options for detailed description.

Interface specific options:

`metric` *number*
This option specifies the metric of the interface. When a route is received from the interface, its metric is increased by this value before further processing. Valid values are 1-255, but values higher than infinity has no further meaning. Default: 1.

`mode multicast|broadcast`
This option selects the mode for RIP to use on the interface. The default is multicast mode for RIPv2 and broadcast mode for RIPv1. RIPng always uses the multicast mode.

`passive` *switch*
Passive interfaces receive routing updates but do not transmit any messages. Default: no.

`address` *ip*
This option specifies a destination address used for multicast or broadcast messages, the default is the official RIP (224.0.0.9) or RIPng (ff02::9) multicast address, or an appropriate broadcast address in the broadcast mode.

`port` *number*
This option selects an UDP port to operate on, the default is the official RIP (520) or RIPng (521) port.

`version 1|2`
This option selects the version of RIP used on the interface. For RIPv1, automatic subnet aggregation is not implemented, only classful network routes and host routes are propagated. Note that BIRD allows RIPv1 to be configured with features that are defined for RIPv2 only, like authentication or using multicast sockets. The default is RIPv2 for IPv4 RIP, the option is not supported for RIPng, as no further versions are defined.

`version only` *switch*
Regardless of RIP version configured for the interface, BIRD accepts incoming packets of any RIP version. This option restrict accepted packets to the configured version. Default: no.

`split horizon` *switch*
Split horizon is a scheme for preventing routing loops. When split horizon is active, routes are not regularly propagated back to the interface from which they were received. They are either not propagated back at all (plain split horizon) or propagated back with an infinity metric (split horizon with poisoned reverse). Therefore, other routers on the interface will not consider the router as a part of an independent path to the destination of the route. Default: yes.

`poison reverse` *switch*
When split horizon is active, this option specifies whether the poisoned reverse variant (propagating routes back with an infinity metric) is used. The poisoned reverse has some advantages in faster convergence, but uses more network traffic. Default: yes.

`demand circuit` *switch*
Regular RIP sends periodic full updates on an interface. There is the Triggered RIP extension for demand circuits (RFC 2091), which removes periodic updates and introduces update acknowledgments. When enabled, there is no RIP communication in steady-state network. Note that in order to work, it must be enabled on both sides. As there are no hello packets, it depends on hardware link state to detect neighbor failures. Also, it is designed for PtP links and it does not work properly with multiple RIP neighbors on an interface. Default: no.

`check zero` *switch*
Received RIPv1 packets with non-zero values in reserved fields should be discarded. This option specifies whether the check is performed or such packets are just processed as usual. Default: yes.

`update time` *number*
Specifies the number of seconds between periodic updates. A lower number will mean faster convergence but bigger network load. Default: 30.

`timeout time` *number*
Specifies the time interval (in seconds) between the last received route announcement and the route expiration. After that, the network is considered unreachable, but still is propagated with infinity distance. Default: 180.

`garbage time` *number*
Specifies the time interval (in seconds) between the route expiration and the removal of the unreachable network entry. The garbage interval, when a route with infinity metric is propagated, is used for both internal (after expiration) and external (after withdrawal) routes. Default: 120.

`ecmp weight` *number*
When ECMP (multipath) routes are allowed, this value specifies a relative weight used for nexthops going through the iface. Valid values are 1-256. Default value is 1.

`authentication none|plaintext|cryptographic`
Selects authentication method to be used. `none` means that packets are not authenticated at all, `plaintext` means that a plaintext password is embedded into each packet, and `cryptographic` means that packets are authenticated using some cryptographic hash function selected by option `algorithm` for each key. The default cryptographic algorithm for RIP keys is Keyed-MD5. If you set authentication to not-none, it is a good idea to add `password` section. Default: none.

`password "`*text*`"`
Specifies a password used for authentication. See password (p. 19) common option for detailed description.

`ttl security [`*switch* | `tx only]`
TTL security is a feature that protects routing protocols from remote spoofed packets by using TTL 255 instead of TTL 1 for protocol packets destined to neighbors. Because TTL is decremented when packets are forwarded, it is non-trivial to spoof packets with TTL 255 from remote locations.

If this option is enabled, the router will send RIP packets with TTL 255 and drop received packets with TTL less than 255. If this option si set to `tx only`, TTL 255 is used for sent packets, but is not checked for received packets. Such setting does not offer protection, but offers compatibility with neighbors regardless of whether they use ttl security.

For RIPng, TTL security is a standard behavior (required by RFC 2080) and therefore default value is yes. For IPv4 RIP, default value is no.

`tx class|dscp|priority` *number*
These options specify the ToS/DiffServ/Traffic class/Priority of the outgoing RIP packets. See tx class (p. 18) common option for detailed description.

`rx buffer` *number*
This option specifies the size of buffers used for packet processing. The buffer size should be bigger than maximal size of received packets. The default value is 532 for IPv4 RIP and interface MTU value for RIPng.

`tx length` *number*
This option specifies the maximum length of generated RIP packets. To avoid IP fragmentation, it should not exceed the interface MTU value. The default value is 532 for IPv4 RIP and interface MTU value for RIPng.

`check link` *switch*
If set, the hardware link state (as reported by OS) is taken into consideration. When the link disappears (e.g. an ethernet cable is unplugged), neighbors are immediately considered unreachable and all routes received from them are withdrawn. It is possible that some hardware drivers or platforms do not implement this feature. Default: yes.

6.16.3 Attributes

RIP defines two route attributes:

`int rip_metric`
RIP metric of the route (ranging from 0 to `infinity`). When routes from different RIP instances are available and all of them have the same preference, BIRD prefers the route with lowest `rip_metric`. When a non-RIP route is exported to RIP, the default metric is 1.

`int rip_tag`
RIP route tag: a 16-bit number which can be used to carry additional information with the route (for example, an originating AS number in case of external routes). When a non-RIP route is exported to RIP, the default tag is 0.

6.16.4 Example

```
protocol rip {
        ipv4 {
                import all;
                export all;
        };
        interface "eth*" {
                metric 2;
                port 1520;
                mode multicast;
                update time 12;
                timeout time 60;
                authentication cryptographic;
                password "secret" { algorithm hmac sha256; };
        };
}
```

6.17 RPKI

6.17.1 Introduction

The Resource Public Key Infrastructure (RPKI) is mechanism for origin validation of BGP routes (RFC 6480). BIRD supports only so-called RPKI-based origin validation. There is implemented RPKI to Router (RPKI-RTR) protocol (RFC 6810). It uses some of the RPKI data to allow a router to verify that the autonomous system announcing an IP address prefix is in fact authorized to do so. This is not crypto checked so can be violated. But it should prevent the vast majority of accidental hijackings on the Internet today, e.g. the famous Pakistani accidental announcement of YouTube's address space.

The RPKI-RTR protocol receives and maintains a set of ROAs from a cache server (also called validator). You can validate routes (RFC 6483, RFC 6811) using function `roa_check()` in filter and set it as import filter at the BGP protocol. BIRD offers crude automatic re-validating of affected routes after RPKI update, see option rpki reload (p. 20). Or you can use a BIRD client command `reload in` *bgp_protocol_name* for manual call of revalidation of all routes.

The same protocol, since version 2, also receives and maintains a set of ASPAs. You can then validate AS paths using function `aspa_check()` in (import) filters.

6.17.2 Supported transports

- Unprotected transport over TCP uses a port 323. The cache server and BIRD router should be on the same trusted and controlled network for security reasons.
- SSHv2 encrypted transport connection uses the normal SSH port 22.

6.17.3 Configuration overview

We currently support just one cache server per protocol. However you can define more RPKI protocols generally.

```
protocol rpki [<name>] {
        roa4 { table <tab>; };
        roa6 { table <tab>; };
        aspa { table <tab>; };
        remote <ip> | "<domain>" [port <number>];
        port <number>;
        local address <ip>;
        refresh [keep] <number>;
        retry [keep] <number>;
        expire [keep] <number>;
        ignore max length <switch>;
        min version <number>;
        max version <number>;
        transport tcp {
                authentication none|md5;
                password "<text>";
        };
        transport ssh {
                bird private key "</path/to/id_rsa>";
                remote public key "</path/to/known_host>";
                user "<name>";
        };
}
```

Alse note that you have to specify the ROA and ASPA channels. If you want to import only IPv4 prefixes you have to specify only roa4 channel. Similarly with IPv6 prefixes only. If you want to fetch both IPv4 and even IPv6 ROAs you have to specify both channels.

RPKI protocol options

`remote` *ip* | "*hostname*" `[port` *number*`]`
Specifies a destination address of the cache server. Can be specified by an IP address or by full domain name string. Only one cache can be specified per protocol. This option is required.

`port` *number*
Specifies the port number. The default port number is 323 for transport without any encryption and 22 for transport with SSH encryption.

`local address` *ip*
Define local address we should use as a source address for the RTR session.

`refresh [keep]` *number*
Time period in seconds. Tells how long to wait before next attempting to poll the cache using a Serial Query or a Reset Query packet. Must be lower than 86400 seconds (one day). Too low value can caused a false positive detection of network connection problems. A keyword `keep` suppresses updating this value by a cache server. Default: 3600 seconds

`retry [keep]` *number*
Time period in seconds between a failed Serial/Reset Query and a next attempt. Maximum allowed value is 7200 seconds (two hours). Too low value can caused a false positive detection of network connection problems. A keyword `keep` suppresses updating this value by a cache server. Default: 600 seconds

`expire [keep]` *number*
Time period in seconds. Received records are deleted if the client was unable to successfully refresh data for this time period. Must be in range from 600 seconds (ten minutes) to 172800 seconds (two days). A keyword `keep` suppresses updating this value by a cache server. Default: 7200 seconds

`ignore max length` *switch*
Ignore received max length in ROA records and use max value (32 or 128) instead. This may be useful for implementing loose RPKI check for blackholes. Default: disabled.

`min version` *number*
Minimal allowed version of the RTR protocol. BIRD will refuse to downgrade a connection below this version and drop the session instead. Default: 0

`max version` *number*
Maximal allowed version of the RTR protocol. BIRD will start with this version. Use this option if sending version 2 to your cache causes problems. Default: 2

`transport tcp` { *TCP transport options...* }
Transport over TCP, it's the default transport. Cannot be combined with a SSH transport. Default: TCP, no authentication.

`transport ssh` { *SSH transport options...* }
It enables a SSHv2 transport encryption. Cannot be combined with a TCP transport. Default: off

TCP transport options

`authentication none|md5`
Select authentication method to be used. `none` means no authentication, `md5` is TCP-MD5 authentication (RFC 2385). Default: no authentication.

`password "`*text*`"`
Use this password for TCP-MD5 authentication of the RPKI-To-Router session.

SSH transport options

`bird private key "`*/path/to/id_rsa*`"`
A path to the BIRD's private SSH key for authentication. It can be a *id_rsa* file.

`remote public key "`*/path/to/known_host*`"`
A path to the cache's public SSH key for verification identity of the cache server. It could be a path to *known_host* file.

`user "`*name*`"`
A SSH user name for authentication. This option is required.

6.17.4 Examples

BGP origin validation

Policy: Don't import `ROA_INVALID` routes.

```
roa4 table r4;
roa6 table r6;

protocol rpki {
        debug all;

        roa4 { table r4; };
        roa6 { table r6; };

        # Please, do not use rpki-validator.realmv6.org in production
        remote "rpki-validator.realmv6.org" port 8282;

        retry keep 5;
        refresh keep 30;
        expire 600;
}

filter peer_in_v4 {
        if (roa_check(r4, net, bgp_path.last) = ROA_INVALID) then
        {
                print "Ignore RPKI invalid ", net, " for ASN ", bgp_path.last;
                reject;
        }
        accept;
}

protocol bgp {
        debug all;
        local as 65000;
        neighbor 192.168.2.1 as 65001;
        ipv4 {
                import filter peer_in_v4;
                export none;
        };
}
```

SSHv2 transport encryption

```
roa4 table r4;
roa6 table r6;

protocol rpki {
        debug all;

        roa4 { table r4; };
        roa6 { table r6; };

        remote 127.0.0.1 port 2345;
        transport ssh {
                bird private key "/home/birdgeek/.ssh/id_rsa";
                remote public key "/home/birdgeek/.ssh/known_hosts";
                user "birdgeek";
        };

        # Default interval values
}
```

6.18 Static

The Static protocol doesn't communicate with other routers in the network, but instead it allows you to define routes manually. This is often used for specifying how to forward packets to parts of the network which don't use dynamic routing at all and also for defining sink routes (i.e., those telling to return packets as undeliverable if they are in your IP block, you don't have any specific destination for them and you don't want to send them out through the default route to prevent routing loops).

There are three classes of definitions in Static protocol configuration – global options, static route definitions, and per-route options. Usually, the definition of the protocol contains mainly a list of static routes. Static routes have no specific attributes, but igp_metric (p. 35) attribute is used to compare static routes with the same preference.

The list of static routes may contain multiple routes for the same network (usually, but not necessary, distinquished by `preference` or `igp_metric`), but only routes of the same network type are allowed, as the static protocol has just one channel. E.g., to have both IPv4 and IPv6 static routes, define two static protocols, each with appropriate routes and channel.

The Static protocol can be configured as MPLS-aware (by defining both the primary channel and MPLS channel). In that case the Static protocol assigns labels to IP routes and automatically announces corresponding MPLS route for each labeled route.

Global options:

`check link` *switch*
: If set, hardware link states of network interfaces are taken into consideration. When link disappears (e.g. ethernet cable is unplugged), static routes directing to that interface are removed. It is possible that some hardware drivers or platforms do not implement this feature. Default: off.

`igp table` *name*
: Specifies a table that is used for route table lookups of recursive routes. Default: the same table as the protocol is connected to.

Route definitions (each may also contain a block of per-route options):

6.18.1 Regular routes; MPLS switching rules

There exist several types of routes; keep in mind that *prefix* syntax is dependent on network type (p. 29).

`route` *prefix* `[mpls` *number*`] via` *ip*|*"interface"* `[`*per-nexthop options*`] [via ...]`
: Regular routes may bear one or more next hops (p. 9). Every next hop is preceded by `via` and configured as shown.

When the Static protocol is MPLS-aware, the optional `mpls` statement after *prefix* specifies a static label for the labeled route, instead of using dynamically allocated label.

`route` *prefix* `[mpls` *number*`] recursive` *ip* `[mpls` *number*`[/`*number*`[/`*number*`[...]]]]`
Recursive nexthop resolves the given IP in the configured IGP table and uses that route's next hop. The MPLS stacks are concatenated; on top is the IGP's nexthop stack and on bottom is this route's stack.

`route` *prefix* `[mpls` *number*`] blackhole|unreachable|prohibit`
Special routes specifying to silently drop the packet, return it as unreachable or return it as administratively prohibited. First two targets are also known as `drop` and `reject`.

When the particular destination is not available (the interface is down or the next hop of the route is not a neighbor at the moment), Static just uninstalls the route from the table it is connected to and adds it again as soon as the destination becomes adjacent again.

Per-nexthop options

There are several options that in a case of multipath route are per-nexthop (i.e., they can be used multiple times for a route, one time for each nexthop). Syntactically, they are not separate options but just parts of `route` statement after each `via` statement, not separated by semicolons. E.g., statement `route 10.0.0.0/8 via 192.0.2.1 bfd weight 1 via 192.0.2.2 weight 2;` describes a route with two nexthops, the first nexthop has two per-nexthop options (`bfd` and `weight 1`), the second nexthop has just `weight 2`.

`bfd` *switch*
The Static protocol could use BFD protocol for next hop liveness detection. If enabled, a BFD session to the route next hop is created and the static route is BFD-controlled – the static route is announced only if the next hop liveness is confirmed by BFD. If the BFD session fails, the static route (or just the affected nexthop from multiple ones) is removed. Note that this is a bit different compared to other protocols, which may use BFD as an advisory mechanism for fast failure detection but ignore it if a BFD session is not even established. Note that BFD protocol also has to be configured, see BFD (p. 41) section for details. Default value is no.

`dev` *text*
The outgoing interface associated with the nexthop. Useful for link-local nexthop addresses or when multiple interfaces use the same network prefix. By default, the outgoing interface is resolved from the nexthop address.

`mpls` *number*`[/`*number*`[/`*number*`[...]]]`
MPLS labels that should be pushed to packets forwarded by the route. The option could be used for both IP routes (on MPLS ingress routers) and MPLS switching rules (on MPLS transit routers). Default value is no labels.

`onlink` *switch*
Onlink flag means that the specified nexthop is accessible on the (specified) interface regardless of IP prefixes of the interface. The interface must be attached to nexthop IP address using link-local-scope format (e.g. `192.0.2.1%eth0`). Default value is no.

`weight` *switch*
For multipath routes, this value specifies a relative weight of the nexthop. Allowed values are 1-256. Default value is 1.

6.18.2 Route Origin Authorization

The ROA config is just `route` *prefix* `max` *int* `as` *int* with no nexthop.

6.18.3 Autonomous System Provider Authorization

The ASPA config is `route aspa` *int* `providers` *int* `[,` *int* `...]` with no nexthop. The first ASN is client and the following are a list of providers. For a transit, you can also write `route aspa` *int* `transit` to get the no-provider ASPA.

6.18.4 Flowspec Network Type

The flow specification are rules for routers and firewalls for filtering purpose. It is described by RFC 8955 and RFC 8956. There are 3 types of arguments: *inet4* or *inet6* prefixes, numeric matching expressions and bitmask matching expressions.

Numeric matching is a matching sequence of numbers and ranges separeted by a commas (`,`) (e.g. `10,20,30`). Ranges can be written using double dots `..` notation (e.g. `80..90,120..124`). An alternative notation are sequence of one or more pairs of relational operators and values separated by logical operators `&&` or ||. Allowed relational operators are `=`, `!=`, <, `<=`, >, `>=`, `true` and `false`.

Bitmask matching is written using *value*`/`*mask* or `!`*value*`/`*mask* pairs. It means that `(`*data* `&` *mask*`)` is or is not equal to *value*. It is also possible to use multiple value/mask pairs connected by logical operators `&&` or ||. Note that for negated matches, value must be either zero or equal to bitmask (e.g. `!0x0/0xf` or `!0xf/0xf`, but not `!0x3/0xf`).

IPv4 Flowspec

`dst` *inet4*
: Set a matching destination prefix (e.g. `dst 192.168.0.0/16`).

`src` *inet4*
: Set a matching source prefix (e.g. `src 10.0.0.0/8`).

`proto` *numbers-match*
: Set a matching IP protocol numbers (e.g. `proto 6`).

`port` *numbers-match*
: Set a matching source or destination TCP/UDP port numbers (e.g. `port 1..1023,1194,3306`).

`dport` *numbers-match*
: Set a matching destination port numbers (e.g. `dport 49151`).

`sport` *numbers-match*
: Set a matching source port numbers (e.g. `sport = 0`).

`icmp type` *numbers-match*
: Set a matching type field number of an ICMP packet (e.g. `icmp type 3`)

`icmp code` *numbers-match*
: Set a matching code field number of an ICMP packet (e.g. `icmp code 1`)

`tcp flags` *bitmask-match*
: Set a matching bitmask for TCP header flags (aka control bits) (e.g. `tcp flags 0x03/0x0f;`). The maximum length of mask is 12 bits (0xfff).

`length` *numbers-match*
: Set a matching packet length (e.g. `length` > `1500`)

`dscp` *numbers-match*
: Set a matching DiffServ Code Point number (e.g. `dscp 8..15`).

`fragment` *fragmentation-type*
: Set a matching type of packet fragmentation. Allowed fragmentation types are `dont_fragment`, `is_fragment`, `first_fragment`, `last_fragment` (e.g. `fragment is_fragment && !dont_fragment`).

```
protocol static {
        flow4;

        route flow4 {
                dst 10.0.0.0/8;
                port > 24 && < 30 || 40..50,60..70,80 && >= 90;
                tcp flags 0x03/0x0f;
                length > 1024;
                dscp = 63;
                fragment dont_fragment, is_fragment || !first_fragment;
        };
}
```

Differences for IPv6 Flowspec

Flowspec IPv6 are same as Flowspec IPv4 with a few exceptions.

- Prefixes *inet6* can be specified not only with prefix length, but with prefix `offset` *number* too (e.g. `::1234:5678:9800:0000/101 offset 64`). Offset means to don't care of *number* first bits.
- In IPv6 packets, there is a matching the last next header value for a matching IP protocol number (e.g. `next header 6`).
- It is not possible to set `dont_fragment` as a type of packet fragmentation.

`dst` *inet6* `[offset` *number*`]`
Set a matching destination IPv6 prefix (e.g. `dst ::1c77:3769:27ad:a11a/128 offset 64`).

`src` *inet6* `[offset` *number*`]`
Set a matching source IPv6 prefix (e.g. `src fe80::/64`).

`next header` *numbers-match*
Set a matching IP protocol numbers (e.g. `next header != 6`).

`label` *numbers-match*
Set numbers for matching the 20-bit Flow Label field in IPv6 packets (e.g. `label != 1234`).

```
protocol static {
        flow6 { table myflow6; };

        route flow6 {
                dst fec0:1122:3344:5566:7788:99aa:bbcc:ddee/128;
                src 0000:0000:0000:0001:1234:5678:9800:0000/101 offset 63;
                next header = 23;
                sport > 24 && < 30 || = 40 || 50,60,70..80;
                dport = 50;
                tcp flags 0x03/0x0f && !0/0xff || 0x33/0x33;
                fragment !is_fragment || !first_fragment;
                label > 1111 && != 1234;
        };
}
```

6.18.5 Per-route options

filter expression
This is a special option that allows filter expressions to be configured on per-route basis. Can be used multiple times. These expressions are evaluated when the route is originated, similarly to the import filter of the static protocol. This is especially useful for configuring route attributes, e.g., `ospf_metric1 = 100;` for a route that will be exported to the OSPF protocol.

6.18.6 Example static configs

```
protocol static {
        ipv4 { table testable; };       # Connect to a non-default routing table
        check link;                     # Advertise routes only if link is up
        route 0.0.0.0/0 via 198.51.100.130; # Default route
        route 10.0.0.0/8                # Multipath route
                via 198.51.100.10 weight 2
                via 198.51.100.20 bfd   # BFD-controlled next hop
                via 192.0.2.1;
        route 203.0.113.0/24 blackhole; # Sink route
        route 10.2.0.0/24 via "arc0";   # Direct route
        route 10.2.2.0/24 via 192.0.2.1 dev "eth0" onlink; # Route with both nexthop and if
        route 192.168.10.0/24 via 198.51.100.100 {
                ospf_metric1 = 20;      # Set extended attribute
        };
        route 192.168.11.0/24 via 198.51.100.100 {
                ospf_metric2 = 100;     # Set extended attribute
                ospf_tag = 2;           # Set extended attribute
        };
        route 192.168.12.0/24 via 198.51.100.100 {
                bgp_community.add((65535, 65281));      # Set extended BGP attribute
                bgp_large_community.add((64512, 1, 1)); # Set extended BGP attribute
        };
}

protocol static {
        ipv6;                                           # Channel is mandatory
        route 2001:db8:10::/48 via 2001:db8:1::1;       # Route with global nexthop
        route 2001:db8:20::/48 via fe80::10%eth0;       # Route with link-local nexthop
        route 2001:db8:30::/48 via fe80::20%'eth1.60';  # Iface with non-alphanumeric chara
        route 2001:db8:40::/48 via fe80::30 dev "eth1"; # Another link-local nexthop
        route 2001:db8:50::/48 via "eth2";              # Direct route to eth2
        route 2001:db8::/32 unreachable;                # Unreachable route
        route ::/0 via 2001:db8:1::1 bfd;               # BFD-controlled default route
}
```

Chapter 7: Conclusions

7.1 Future work

Although BIRD supports all the commonly used routing protocols, there are still some features which would surely deserve to be implemented in future versions of BIRD:

- Opaque LSA's
- Route aggregation and flap dampening
- Multicast routing protocols
- Ports to other systems

7.2 Getting more help

If you use BIRD, you're welcome to join the bird-users mailing list (bird-users@network.cz) where you can share your experiences with the other users and consult your problems with the authors. To subscribe to the list, visit https://bird.nic.cz/lists/bird-users/. The home page of BIRD can be found at http://bird.nic.cz/.

BIRD is a relatively young system and it probably contains some bugs. You can report any problems to the bird-users list and the authors will be glad to solve them, but before you do so, please make sure you have read the available documentation and that you are running the latest version (available at https://bird.nic.cz/get-bird). (Of course, a patch which fixes the bug is always welcome as an attachment.)

If you want to understand what is going inside, Internet standards are a good and interesting reading. You can get them from ftp.rfc-editor.org.

Good luck and happy routing!