BIRD Internet Routing Daemon

Ondřej Zajíček
CZ.NIC

Abstract

We introduce BIRD Internet Routing Daemon, a Linux routing software. We present the overview of BIRD project, its basic concepts and design decisions, common applications, supported protocols and and examples of usage. We also discuss pitfalls in userspace/kernel interfaces encountered during BIRD development.

Introduction

BIRD overview

BIRD Internet Routing Daemon is a routing daemon; i.e., a software responsible for managing kernel packet forwarding tables. It is a free implementation of several well known and common routing and router-supplemental protocols, namely RIP, RIPng, OSPFv2, OSPFv3, BGP, BFD, and NDP/RA.

BIRD supports IPv4 and IPv6 address families, Linux kernel and several BSD variants (tested on FreeBSD, NetBSD and OpenBSD). BIRD consists of **bird** daemon and **birdc** interactive CLI client used for supervision.

BIRD started as a student project at the Faculty of Math and Physics, Charles University, Prague in 1999. After the project was finished, the development mostly ceased. Since 2008, BIRD is again in active development, sponsored by CZ.NIC.

BIRD is a free / open source software, freely distributed under GNU General Public License.

BIRD features

BIRD has several distinctive features compared to alternative routing daemons:

First, BIRD has native support for multiple protocol instances and multiple routing tables. This was one of original design considerations.

Second, BIRD has programmable route filters by internal scripting language with a familiar syntax, instead of usual access control lists. This allows greater expressive power when configuring route distribution.

Third, BIRD uses clear and structured config files for its configuration. It has automatic runtime reconfiguration – when the config file is changed and reconfiguration is requested, BIRD automatically applies necessary changes without disrupting other routing protocol sessions.

Fourth, BIRD has rather extensive documentation, both user and programmer one.

Typical applications

The basic application is using Linux system as a software router, where Linux kernel serves as a *data plane*, while BIRD serves as a *control plane*, communicating with other routers in the network, discovering the network topology and computing the routing table. Such application works well unless forwarded data rates are too high, requiring hardware routing solutions.

There are also applications where data traffic is not forwarded, system traffic is limited to control traffic and local data traffic. These are especially suitable for routing software like BIRD. Several examples: monitoring tools for OSPF networks, BGP route reflectors, fail-over solutions for servers.

One particular example is using BIRD as BGP route server in internet exchange points. The purpose of BGP route servers in IXPs is to exchange, filter and distribute routing information between BGP border routers of IXP clients to eliminate the need for configuring a BGP session between each pair of clients. Data traffic is exchanged directly between border routers bypassing the route server. Such task requires a flexible and efficient control plane as usually separate routing table for each client is used and extensive route filters based on RIR databases are applied. BIRD is very popular in European IXP community, according to Euro-IX 2014 report [1], 64 % of members' route servers run BIRD.

Concepts

There are few basic conceptual objects in BIRD: routes, protocols, tables and filters.

Routes

Routes are basic objects managed by BIRD. They are generated by protocols and stored in routing tables. Like routes in kernel forwarding table, they have a destination network prefix, an associated interface and the target (a gateway or a specific action like *unreachable*). In addition to this, they have a list of associated attributes – the responsible protocol, the route preference and some protocol-specific attributes (OSPF metrics, multiple kinds of BGP attributes).

[1] https://www.euro-ix.net/documents/1467-euro-ix-route-servers-stats-pdf

Protocols

Protocol objects represent instances of routing protocols (BGP, OSPF, RIP), or other route sources/sinks (**static**, **kernel**, **direct**). Protocols have name, type, operating state (*start*, *up*, *stop*, *down*), associated routing table (usually one, but in some cases multiple), configuration, statistics and internal state.

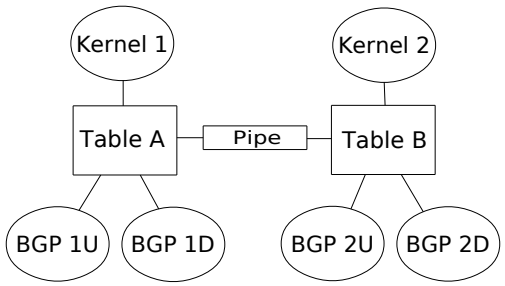
Protocols generate routes, propagate them to associated routing table and receive routes from it. It is possible to have multiple instances of one protocol type (e.g. OSPF) as separate protocol objects with different names (e.g. *ospf1*, *ospf2*). Protocols could be independently enabled, disabled, restarted or reconfigured. Protocols could be examined using `show protocols` command.

Tables

Routing tables are data tables where routes are accumulated. They are userspace analogy of kernel packet forwarding tables. In network engineering terminology, they are called RIB (*routing information base*), while kernel forwarding tables are called FIB (*forwarding information base*).

A route is generated by a protocol, then imported to an associated routing table, where it is stored. Routing tables store one route per destination and source protocol. Therefore, it is possible to have multiple routes for each destination (from different protocols) concurrently. For each destination, a preferred route is selected (based on route preferences and protocol metrics) and then exported to associated protocols.

BIRD supports any number of routing tables. There is a default one (named *master*), others have to be defined in configuration. Note that BIRD routing tables are not automatically synchronized with kernel forwarding tables. There is a **kernel** protocol serving to that purpose – when a route is exported to the **kernel** protocol, it is propagated to a (configurable) kernel forwarding table, and vice versa. Therefore, some BIRD routing tables may be synchronized with kernel forwarding tables, while others may be just internal to BIRD. There is also a **pipe** protocol to exchange routes between two routing tables.



Filters

Filters stand between protocols and tables. When a route is imported from a protocol to a table, it passes through an import filter. Likewise, when a route is exported from a table to a protocol, it passes through an export filter. Such filters may modify, reject or accept routes. A filter is written in a scripting language, which may access and examine all route attributes. Named filters are defined as top-level objects but used in **import** or **export** options of a protocol. Filters can also be defined using **where** expression. There are two keywords **all** and **none** that can be used as accept-all or reject-all filters. Note that default is **import all** and **export none**.

```
define martians = [ 10.0.0.0/8+,
  172.16.0.0/12+,   192.168.0.0/16+,
  169.254.0.0/16+,  224.0.0.0/4+,
  240.0.0.0/4+,     0.0.0.0/32- ];

filter bgp_in {
  if net ~ martians then reject;
  if bgp_path.first != 1234 then reject;
  if bgp_path.len > 64 then reject;

  if net ~ 128.66.0.0/16+
  then bgp_local_pref = 500;
  else bgp_local_pref = 100;

  bgp_med = 0;
  accept;
}

protocol bgp {
  import filter bgp_in;
  export where source = RTS_BGP;

  local 192.168.1.1 as 65100;
  neighbor 192.168.1.2 as 65200;
}
```

Usage

BIRD is mainly configured through its config file, usually `/etc/bird.conf` or `/etc/bird/bird.conf`. After start, BIRD setups a unix socket, usually `/var/run/bird.ctl`, which is used by **birdc** interactive client for supervision. There are no specific access control mechanisms for the client, access is controlled just by file permissions of the socket. BIRD client is an interactive shell which uses *GNU Readline* for CLI. It contains interactive help (accessible by '?'). The most important commands are:

- `show route [all]...`
- `show protocols [all]`
- `show interfaces`
- `show ospf...`
- `enable | disable | restart` *proto*
- `configure [timeout | undo | confirm]`
- `down`

Compared to e.g. Quagga, where each protocol is a separate process, in BIRD all protocols are handled by one process. Note that in current BIRD, IPv4 and IPv6 are handled by two separate daemons, **bird** for IPv4 and **bird6** for IPv6. Likewise, the client for IPv6 daemon is named **birdc6**. There is also a lightweight client, **birdcl**, that does not depend on *GNU Readline*.

Basic setup

Basic configuration is pretty simple. Global options are usually used just to specify *router ID* and logging. Protocol **device** is necessary for enumeration of network interfaces. Protocol **kernel** is used for synchronization of BIRD routing tables and kernel forwarding tables. Protocol **static** allows to specify a set of static routes.

```
router id 192.168.1.1;
log syslog all;

protocol device {
}

protocol kernel {
  export all;
  scan time 10;
}

protocol static {
  route 192.168.10.0/24 via 192.168.1.2;
  route 192.168.20.0/24 unreachable;
}
```

OSPF – Open Shortest Path First

OSPF is a popular link-state routing protocol for internal networks. Each router monitors reachability of its neighbors, the local network topology is packed to LSAs (Link State Advertisements), distributed to neighbors and flooded through the network. Therefore, every router gets a complete map of the network and computes shortest paths to all destinations.

BIRD implements OSPFv2 for IPv4 (RFC 2328), OSPFv3 for IPv6 (RFC 5340), and NSSA areas (RFC 3101).

A protocol instance represents a complete OSPF domain possibly with multiple areas and active interfaces. Multiple instances are possible but usually not necessary. It maintains OSPF topology, when it changes, OSPF routing table is calculated and imported to BIRD routing table. When a route is exported to an OSPF protocol, it is included in its topology as an *external LSA*. Device routes for OSPF interfaces are handled internally and generated automatically.

```
protocol ospf {
  # export all static routes to OSPF
  export where source = RTS_STATIC;

  area 0 {
    interface "eth0" {
      cost 5; hello 5; wait 10; dead 25;
    };
    interface "eth*" {
      cost 100; type pointopoint;
    };
  };
}
```

BGP – Border Gateway Protocol

BGP is the standard protocol for internet routing. A router receives paths to reachable destinations from its neighbors. It chooses preferred paths by path lengths and local policy. Preferred paths are used as a routes for forwarding and also possibly propagated to other neighbors. Forwarded routes contain many additional attributes, mainly *AS_PATH* describing the path to the destination as a list of autonomous systems.

BIRD implements BGPv4 (RFC 4271), multiprotocol BGP (RFC 4760) for IPv6 (RFC 2545) and multiple BGP extensions:

- RFC 1997 – communities attribute
- RFC 2385 – MD5 password authentication
- RFC 3392 – capability negotiation
- RFC 4360 – extended communities attribute
- RFC 4456 – route reflectors
- RFC 4724 – graceful restart
- RFC 4893 – 4B AS numbers
- RFC 5668 – 4B AS numbers in extended communities

A BGP protocol instance in BIRD represents one BGP session. Therefore, it is expected to setup multiple BGP protocols, one for each BGP neighbor. A BGP implementation in BIRD is conceptually simple, it does not maintain much internal state, it is essentially a pipe to the neighbor. All exported routes (from the BIRD routing table) are sent to the neighbor, all routes received from the neighbor are imported to the BIRD routing table. BGP path attributes are accessible from import and export filters, BGP path decision process is handled in BIRD routing tables as a part of preferred route selection.

An example of configuration is skipped as it is contained in the example for *Filters*.

BFD – Bidirectional Forwarding Detection

BFD is a protocol for neighbor reachability and liveness testing. It is a supplementary protocol to OSPF, BGP and others. Although these protocols have internal neighbor liveness testing, they have timers with 1-second granularity and their reaction time is usually in tens of seconds, while BFD reaction time is tens to hundreds of milliseconds.

BIRD implements BFD (RFC 5880) for IPv4 and IPv6, both the single hop variant (RFC 5881), and the multihop variant (RFC 5883).

```
protocol bfd {
  interface "eth*" {
    interval 50 ms;
    multiplier 4;
  };
}

protocol bgp {
  . . .

  local 192.168.1.1 as 65100;
  neighbor 192.168.1.2 as 65200;
  bfd;
}
```

NDP/RA – IPv6 router advertisements

IPv6 routers use router advertisement packets from Neighbor Discovery Protocol (NDP) to announce their presence on the network to hosts. Hosts then use it for IPv6 stateless address

autoconfiguration. On Linux, this is usually handled by The Router Advertisement Daemon (*radvd*). BIRD could also generate router advertisement packets by protocol **radv**. It can learn announced address prefixes from the interface configuration. It also support RDNSS a DNSSL extensions for DNS autoconfiguration on hosts. There is also a support for dynamic IPv6 router advertisements – advertisements triggered by availability of a configured route in BIRD routing table.

```
protocol radv {
  interface "eth*";
  rdnss 2001:0DB8:1234::10;
  dnssl "domain.cz";
  trigger 2000::/3;
}
```

Pitfalls

During BIRD development, we encountered several pitfalls in kernel networking API.

Sockets API

Although the Sockets API is well-behaved and mature interface for simple TCP connections, there are plenty issues when it is used for multicast with UDP or raw sockets. The first issue is whether to use one global socket, or one socket per interface. Although the first option seems natural, there are often hidden limits for the number of joined multicast groups per socket while memberships are specific per interface, therefore such limits are encountered on systems with many network interfaces. For this and other reasons, it seems that the second option is generally better.

Another issue is that there is no universal reliable way to specify the source address and the destination interface. The syscall *bind()* is useless in this case, as it has multiple unrelated effects (like filtering incoming messages). We currently use IP_PKTINFO on Linux, IP_SENDSRCADDR on BSD for UDP and IP_HDRINCL on BSD for raw sockets. Hopefully, for IPv6, IPV6_PKTINFO works well in all cases. Also note that on Linux, socket option SO_BINDTOIFACE is very useful for our style of usage of sockets; unfortunately, no such option is available on BSD.

Ephemeral Source Port Selection

When TCP or UDP communication is initiated without explicitly specifying the source port, one is automatically allocated. Such port is called an ephemeral port. According to IANA and RFC 6335, range 49152–65535 should be used for ephemeral ports. Linux uses by default range 32768–61000, tunable by `net.ipv4.ip_local_port_range`. FreeBSD uses by default range 10000–65535, also tunable. In FreeBSD, there is a socket option IP_PORTRANGE_HIGH that forces OS to use the proper ephemeral port range. Unfortunately, there is no such option in Linux. Why even care for ephemeral port selection? Some BFD implementations reject received packets with source port< 49152.

Netlink and FIB

FIB behavior and its management using the Netlink interface is a source of multiple pitfalls:

The first issue is behavior of multipath routes where there is no consistency between IPv4 and IPv6. An IPv4 multipath route is one route with several next hop fields, while IPv6 multipath route is a set of routes with the same destination network. There is a compatibility layer for IPv6 routes, but it does not behave consistently (a route with multiple next hops could be entered, but it is scanned back as a multiple separate routes).

The second issue are missing RTM_DELROUTE notifications when routes are removed due to shutdown of related interfaces. Although there is a reason for that (to not flood Netlink with many RTM_DELROUTE notifications), it is unexpected and inconsistent behavior and the Netlink was flooded with RTM_NEWROUTE messages when these routes were added.

The third issue is the default limit for IPv6 FIBs, 4096 routes (`net.ipv6.route.max_size`), which is too small for common usages and completely unexpected by users (as there is no such limit for IPv4 FIBs).