BIRD Programmer's Documentation

Ondrej Filip <*feela@network.cz*>, Pavel Machek <*pavel@ucw.cz*>, Martin Mares <*mj@ucw.cz*>, Ondrej Zajicek <*santiago@crfreenet.org*>

This document contains programmer's documentation for the BIRD Internet Routing Daemon project.

Contents

Chapter 1: BIRD Design

1.1 Introduction

This document describes the internal workings of BIRD, its architecture, design decisions and rationale behind them. It also contains documentation on all the essential components of the system and their interfaces.

Routing daemons are complicated things which need to act in real time to complex sequences of external events, respond correctly even to the most erroneous behavior of their environment and still handle enormous amount of data with reasonable speed. Due to all of this, their design is very tricky as one needs to carefully balance between efficiency, stability and (last, but not least) simplicity of the program and it would be possible to write literally hundreds of pages about all of these issues. In accordance to the famous quote of Anton Chekhov "Shortness is a sister of talent", we've tried to write a much shorter document highlighting the most important stuff and leaving the boring technical details better explained by the program source itself together with comments contained therein.

1.2 Design goals

When planning the architecture of BIRD, we've taken a close look at the other existing routing daemons and also at some of the operating systems used on dedicated routers, gathered all important features and added lots of new ones to overcome their shortcomings and to better match the requirements of routing in today's Internet: IPv6, policy routing, route filtering and so on. From this planning, the following set of design goals has arisen:

- *Support all the standard routing protocols and make it easy to add new ones.* This leads to modularity and clean separation between the core and the protocols.
- *Support both IPv4 and IPv6 in the same source tree, re-using most of the code.* This leads to abstraction of IP addresses and operations on them.
- *Minimize OS dependent code to make porting as easy as possible.* Unfortunately, such code cannot be avoided at all as the details of communication with the IP stack differ from OS to OS and they often vary even between different versions of the same OS. But we can isolate such code in special modules and do the porting by changing or replacing just these modules. Also, don't rely on specific features of various operating systems, but be able to make use of them if they are available.
- *Allow multiple routing tables.* Easily solvable by abstracting out routing tables and the corresponding operations.
- *Offer powerful route filtering.* There already were several attempts to incorporate route filters to a dynamic router, but most of them have used simple sequences of filtering rules which were very inflexible and hard to use for non-trivial filters. We've decided to employ a simple loop-free programming language having access to all the route attributes and being able to modify the most of them.
- *Support easy configuration and re-configuration.* Most routers use a simple configuration language designed ad hoc with no structure at all and allow online changes of configuration by using their command-line interface, thus any complex re-configurations are hard to achieve without replacing the configuration file and restarting the whole router. We've decided to use a more general approach: to have a configuration defined in a context-free language with blocks and nesting, to perform all configuration changes by editing the configuration file, but to be able to read the new configuration and smoothly adapt to it without disturbing parts of the routing process which are not affected by the change.
- *Be able to be controlled online.* In addition to the online reconfiguration, a routing daemon should be able to communicate with the user and with many other programs (primarily scripts used for network maintenance) in order to make it possible to inspect contents of routing tables, status of all routing protocols and also to control their behavior (disable, enable or reset a protocol without restarting all

the others). To achieve this, we implement a simple command-line protocol based on those used by FTP and SMTP (that is textual commands and textual replies accompanied by a numeric code which makes them both readable to a human and easy to recognize in software).

- *Respond to all events in real time.* A typical solution to this problem is to use lots of threads to separate the workings of all the routing protocols and also of the user interface parts and to hope that the scheduler will assign time to them in a fair enough manner. This is surely a good solution, but we have resisted the temptation and preferred to avoid the overhead of threading and the large number of locks involved and preferred a event driven architecture with our own scheduling of events. An unpleasant consequence of such an approach is that long lasting tasks must be split to more parts linked by special events or timers to make the CPU available for other tasks as well.

1.3 Architecture

The requirements set above have lead to a simple modular architecture containing the following types of modules:

Core modules
implement the core functions of BIRD: taking care of routing tables, keeping protocol status, interacting with the user using the Command-Line Interface (to be called CLI in the rest of this document) etc.

Library modules
form a large set of various library functions implementing several data abstractions, utility functions and also functions which are a part of the standard libraries on some systems, but missing on other ones.

Resource management modules
take care of resources, their allocation and automatic freeing when the module having requested shuts itself down.

Configuration modules
are fragments of lexical analyzer, grammar rules and the corresponding snippets of C code. For each group of code modules (core, each protocol, filters) there exist a configuration module taking care of all the related configuration stuff.

The filter
implements the route filtering language.

Protocol modules
implement the individual routing protocols.

System-dependent modules
implement the interface between BIRD and specific operating systems.

The client
is a simple program providing an easy, though friendly interface to the CLI.

1.4 Implementation

BIRD has been written in GNU C. We've considered using C++, but we've preferred the simplicity and straightforward nature of C which gives us fine control over all implementation details and on the other hand enough instruments to build the abstractions we need.

The modules are statically linked to produce a single executable file (except for the client which stands on its own).

The building process is controlled by a set of Makefiles for GNU Make, intermixed with several Perl and shell scripts.

The initial configuration of the daemon, detection of system features and selection of the right modules to include for the particular OS and the set of protocols the user has chosen is performed by a configure script generated by GNU Autoconf.

The parser of the configuration is generated by the GNU Bison.

The documentation is generated using `SGMLtools` with our own DTD and mapping rules which produce both an online version in HTML and a neatly formatted one for printing (first converted from SGML to LaTeX and then processed by TeX and `dvips` to get a PostScript file).

The comments from C sources which form a part of the programmer's documentation are extracted using a modified version of the `kernel-doc` tool.

If you want to work on BIRD, it's highly recommended to configure it with a `--enable-debug` switch which enables some internal consistency checks and it also links BIRD with a memory allocation checking library if you have one (either `efence` or `dmalloc`).

Chapter 2: Core

2.1 Forwarding Information Base

FIB is a data structure designed for storage of routes indexed by their network prefixes. It supports insertion, deletion, searching by prefix, ‘routing’ (in CIDR sense, that is searching for a longest prefix matching a given IP address) and (which makes the structure very tricky to implement) asynchronous reading, that is enumerating the contents of a FIB while other modules add, modify or remove entries.

Internally, each FIB is represented as a collection of nodes of type `fib_node` indexed using a sophisticated hashing mechanism. We use two-stage hashing where we calculate a 16-bit primary hash key independent on hash table size and then we just divide the primary keys modulo table size to get a real hash key used for determining the bucket containing the node. The lists of nodes in each bucket are sorted according to the primary hash key, hence if we keep the total number of buckets to be a power of two, re-hashing of the structure keeps the relative order of the nodes.

To get the asynchronous reading consistent over node deletions, we need to keep a list of readers for each node. When a node gets deleted, its readers are automatically moved to the next node in the table.

Basic FIB operations are performed by functions defined by this module, enumerating of FIB contents is accomplished by using the *FIB_WALK()* macro or *FIB_ITERATE_START()* if you want to do it asynchronously.

For simple iteration just place the body of the loop between *FIB_WALK()* and *FIB_WALK_END()*. You can’t modify the FIB during the iteration (you can modify data in the node, but not add or remove nodes).

If you need more freedom, you can use the FIB_ITERATE_*() group of macros. First, you initialize an iterator with *FIB_ITERATE_INIT()*. Then you can put the loop body in between *FIB_ITERATE_START()* and *FIB_ITERATE_END()*. In addition, the iteration can be suspended by calling *FIB_ITERATE_PUT()*. This’ll link the iterator inside the FIB. While suspended, you may modify the FIB, exit the current function, etc. To resume the iteration, enter the loop again. You can use *FIB_ITERATE_UNLINK()* to unlink the iterator (while iteration is suspended) in cases like premature end of FIB iteration.

Note that the iterator must not be destroyed when the iteration is suspended, the FIB would then contain a pointer to invalid memory. Therefore, after each *FIB_ITERATE_INIT()* or *FIB_ITERATE_PUT()* there must be either *FIB_ITERATE_START()* or *FIB_ITERATE_UNLINK()* before the iterator is destroyed.

Function

void *fib_init* (struct fib * *f*, pool * *p*, unsigned *node_size*, unsigned *hash_order*, fib_init_func *init*) – initialize a new FIB

Arguments

struct fib * *f*
: the FIB to be initialized (the structure itself being allocated by the caller)

pool * *p*
: pool to allocate the nodes in

unsigned *node_size*
: node size to be used (each node consists of a standard header `fib_node` followed by user data)

unsigned *hash_order*
: initial hash order (a binary logarithm of hash table size), 0 to use default order (recommended)

fib_init_func *init*
: pointer a function to be called to initialize a newly created node

Description

This function initializes a newly allocated FIB and prepares it for use.

Function

void * *fib_find* (struct fib * *f*, ip_addr * *a*, int *len*) – search for FIB node by prefix

Arguments

struct fib * *f*
FIB to search in

ip_addr * *a*
pointer to IP address of the prefix

int *len*
prefix length

Description

Search for a FIB node corresponding to the given prefix, return a pointer to it or `NULL` if no such node exists.

Function

void * *fib_get* (struct fib * *f*, ip_addr * *a*, int *len*) – find or create a FIB node

Arguments

struct fib * *f*
FIB to work with

ip_addr * *a*
pointer to IP address of the prefix

int *len*
prefix length

Description

Search for a FIB node corresponding to the given prefix and return a pointer to it. If no such node exists, create it.

Function

void * *fib_route* (struct fib * *f*, ip_addr *a*, int *len*) – CIDR routing lookup

Arguments

struct fib * *f*
FIB to search in

ip_addr *a*
pointer to IP address of the prefix

int *len*
prefix length

Description

Search for a FIB node with longest prefix matching the given network, that is a node which a CIDR router would use for routing that network.

Function

void *fib_delete* (struct fib * *f*, void * *E*) – delete a FIB node

Arguments

struct fib * *f*
FIB to delete from

void * *E*
entry to delete

Description
This function removes the given entry from the FIB, taking care of all the asynchronous readers by shifting them to the next node in the canonical reading order.

Function
void *fib_free* (struct fib * *f*) – delete a FIB

Arguments

struct fib * *f*
FIB to be deleted

Description
This function deletes a FIB – it frees all memory associated with it and all its entries.

Function
void *fib_check* (struct fib * *f*) – audit a FIB

Arguments

struct fib * *f*
FIB to be checked

Description
This debugging function audits a FIB by checking its internal consistency. Use when you suspect somebody of corrupting innocent data structures.

2.2 Routing tables

Routing tables are probably the most important structures BIRD uses. They hold all the information about known networks, the associated routes and their attributes.

There are multiple routing tables (a primary one together with any number of secondary ones if requested by the configuration). Each table is basically a FIB containing entries describing the individual destination networks. For each network (represented by structure `net`), there is a one-way linked list of route entries (`rte`), the first entry on the list being the best one (i.e., the one we currently use for routing), the order of the other ones is undetermined.

The `rte` contains information specific to the route (preference, protocol metrics, time of last modification etc.) and a pointer to a `rta` structure (see the route attribute module for a precise explanation) holding the remaining route attributes which are expected to be shared by multiple routes in order to conserve memory.

Function
rte * *rte_find* (net * *net*, struct rte_src * *src*) – find a route

Arguments

net * *net*
network node

struct rte_src * *src*
route source

Description
The *rte_find()* function returns a route for destination *net* which is from route source *src*.

Function

rte * *rte_get_temp* (rta * *a*) – get a temporary `rte`

Arguments

rta * *a*
: attributes to assign to the new route (a `rta`; in case it's un-cached, *rte_update()* will create a cached copy automatically)

Description

Create a temporary `rte` and bind it with the attributes *a*. Also set route preference to the default preference set for the protocol.

Function

rte * *rte_cow_rta* (rte * *r*, linpool * *lp*) – get a private writable copy of `rte` with writable `rta`

Arguments

rte * *r*
: a route entry to be copied

linpool * *lp*
: a linpool from which to allocate `rta`

Description

rte_cow_rta() takes a `rte` and prepares it and associated `rta` for modification. There are three possibilities: First, both `rte` and `rta` are private copies, in that case they are returned unchanged. Second, `rte` is private copy, but `rta` is cached, in that case `rta` is duplicated using *rta_do_cow()*. Third, both `rte` is shared and `rta` is cached, in that case both structures are duplicated by *rte_do_cow()* and *rta_do_cow()*.
Note that in the second case, cached `rta` loses one reference, while private copy created by *rta_do_cow()* is a shallow copy sharing indirect data (eattrs, nexthops, ...) with it. To work properly, original shared `rta` should have another reference during the life of created private copy.

Result

a pointer to the new writable `rte` with writable `rta`.

Function

void *rte_announce* (rtable * *tab*, unsigned *type*, net * *net*, rte * *new*, rte * *old*, rte * *new_best*, rte * *old_best*, rte * *before_old*) – announce a routing table change

Arguments

rtable * *tab*
: table the route has been added to

unsigned *type*
: type of route announcement (RA_OPTIMAL or RA_ANY)

net * *net*
: network in question

rte * *new*
: the new route to be announced

rte * *old*
: the previous route for the same network

rte * *new_best*
: the new best route for the same network

rte * *old_best*
the previous best route for the same network

rte * *before_old*
The previous route before *old* for the same network. If *before_old* is NULL *old* was the first.

Description
This function gets a routing table update and announces it to all protocols that acccepts given type of route announcement and are connected to the same table by their announcement hooks.
Route announcement of type `RA_OPTIMAL` si generated when optimal route (in routing table *tab*) changes. In that case *old* stores the old optimal route.
Route announcement of type `RA_ANY` si generated when any route (in routing table *tab*) changes In that case *old* stores the old route from the same protocol.
For each appropriate protocol, we first call its *import_control()* hook which performs basic checks on the route (each protocol has a right to veto or force accept of the route before any filter is asked) and adds default values of attributes specific to the new protocol (metrics, tags etc.). Then it consults the protocol's export filter and if it accepts the route, the *rt_notify()* hook of the protocol gets called.

Function
void *rte_free* (rte * *e*) – delete a `rte`

Arguments

rte * *e*
`rte` to be deleted

Description
rte_free() deletes the given `rte` from the routing table it's linked to.

Function
void *rte_update2* (struct announce_hook * *ah*, net * *net*, rte * *new*, struct rte_src * *src*) – enter a new update to a routing table

Arguments

struct announce_hook * *ah*
pointer to table announce hook

net * *net*
network node

rte * *new*
a `rte` representing the new route or `NULL` for route removal.

struct rte_src * *src*
protocol originating the update

Description
This function is called by the routing protocols whenever they discover a new route or wish to update/remove an existing route. The right announcement sequence is to build route attributes first (either un-cached with *aflags* set to zero or a cached one using *rta_lookup()*; in this case please note that you need to increase the use count of the attributes yourself by calling *rta_clone()*), call *rte_get_temp()* to obtain a temporary `rte`, fill in all the appropriate data and finally submit the new `rte` by calling *rte_update()*.
src specifies the protocol that originally created the route and the meaning of protocol-dependent data of *new*. If *new* is not `NULL`, *src* have to be the same value as *new*->attrs->proto. *p* specifies the protocol that called *rte_update()*. In most cases it is the same protocol as *src*. *rte_update()* stores *p* in *new*->sender;
When *rte_update()* gets any route, it automatically validates it (checks, whether the network and next hop address are valid IP addresses and also whether a normal routing protocol doesn't try to smuggle a host or link scope route to the table), converts all protocol dependent attributes stored in the `rte` to temporary

extended attributes, consults import filters of the protocol to see if the route should be accepted and/or its attributes modified, stores the temporary attributes back to the `rte`.

Now, having a "public" version of the route, we automatically find any old route defined by the protocol *src* for network *n*, replace it by the new one (or removing it if *new* is `NULL`), recalculate the optimal route for this destination and finally broadcast the change (if any) to all routing protocols by calling *rte_announce()*.

All memory used for attribute lists and other temporary allocations is taken from a special linear pool *rte_update_pool* and freed when *rte_update()* finishes.

Function

void *rt_refresh_begin* (rtable * *t*, struct announce_hook * *ah*) – start a refresh cycle

Arguments

rtable * *t*
: related routing table

struct announce_hook * *ah*
: related announce hook

Description

This function starts a refresh cycle for given routing table and announce hook. The refresh cycle is a sequence where the protocol sends all its valid routes to the routing table (by *rte_update()*). After that, all protocol routes (more precisely routes with *ah* as *sender*) not sent during the refresh cycle but still in the table from the past are pruned. This is implemented by marking all related routes as stale by REF_STALE flag in *rt_refresh_begin()*, then marking all related stale routes with REF_DISCARD flag in *rt_refresh_end()* and then removing such routes in the prune loop.

Function

void *rt_refresh_end* (rtable * *t*, struct announce_hook * *ah*) – end a refresh cycle

Arguments

rtable * *t*
: related routing table

struct announce_hook * *ah*
: related announce hook

Description

This function starts a refresh cycle for given routing table and announce hook. See *rt_refresh_begin()* for description of refresh cycles.

Function

void *rte_dump* (rte * *e*) – dump a route

Arguments

rte * *e*
: `rte` to be dumped

Description

This functions dumps contents of a `rte` to debug output.

Function

void *rt_dump* (rtable * *t*) – dump a routing table

Arguments

rtable * *t*
: routing table to be dumped

Description

This function dumps contents of a given routing table to debug output.

Function

void *rt_dump_all* (*void*) – dump all routing tables

Description

This function dumps contents of all routing tables to debug output.

Function

void *rt_init* (*void*) – initialize routing tables

Description

This function is called during BIRD startup. It initializes the routing table module.

Function

int *rt_prune_table* (rtable * *tab*) – prune a routing table

Arguments

rtable * *tab*
a routing table for pruning

Description

This function scans the routing table *tab* and removes routes belonging to flushing protocols, discarded routes and also stale network entries, in a similar fashion like *rt_prune_loop()*. Returns 1 when all such routes are pruned. Contrary to *rt_prune_loop()*, this function is not a part of the protocol flushing loop, but it is called from *rt_event()* for just one routing table.

Note that *rt_prune_table()* and *rt_prune_loop()* share (for each table) the prune state (*prune_state*) and also the pruning iterator (*prune_fit*).

Function

int *rt_prune_loop* (*void*) – prune routing tables

Description

The prune loop scans routing tables and removes routes belonging to flushing protocols, discarded routes and also stale network entries. Returns 1 when all such routes are pruned. It is a part of the protocol flushing loop.

Function

void *rt_lock_table* (rtable * *r*) – lock a routing table

Arguments

rtable * *r*
routing table to be locked

Description

Lock a routing table, because it's in use by a protocol, preventing it from being freed when it gets undefined in a new configuration.

Function

void *rt_unlock_table* (rtable * *r*) – unlock a routing table

Arguments

rtable * *r*
routing table to be unlocked

Description

Unlock a routing table formerly locked by *rt_lock_table()*, that is decrease its use count and delete it if it's scheduled for deletion by configuration changes.

Function

void *rt_commit* (struct config * *new*, struct config * *old*) – commit new routing table configuration

Arguments

struct config * *new*
new configuration

struct config * *old*
original configuration or `NULL` if it's boot time config

Description

Scan differences between *old* and *new* configuration and modify the routing tables according to these changes. If *new* defines a previously unknown table, create it, if it omits a table existing in *old*, schedule it for deletion (it gets deleted when all protocols disconnect from it by calling *rt_unlock_table()*), if it exists in both configurations, leave it unchanged.

Function

int *rt_feed_baby* (struct proto * *p*) – advertise routes to a new protocol

Arguments

struct proto * *p*
protocol to be fed

Description

This function performs one pass of advertisement of routes to a newly initialized protocol. It's called by the protocol code as long as it has something to do. (We avoid transferring all the routes in single pass in order not to monopolize CPU time.)

Function

void *rt_feed_baby_abort* (struct proto * *p*) – abort protocol feeding

Arguments

struct proto * *p*
protocol

Description

This function is called by the protocol code when the protocol stops or ceases to exist before the last iteration of *rt_feed_baby()* has finished.

Function

net * *net_find* (rtable * *tab*, ip_addr *addr*, unsigned *len*) – find a network entry

Arguments

rtable * *tab*
a routing table

ip_addr *addr*
address of the network

unsigned *len*
length of the network prefix

Description

net_find() looks up the given network in routing table *tab* and returns a pointer to its `net` entry or `NULL` if no such network exists.

Function

net * *net_get* (rtable * *tab*, ip_addr *addr*, unsigned *len*) – obtain a network entry

Arguments

rtable * *tab*
: a routing table

ip_addr *addr*
: address of the network

unsigned *len*
: length of the network prefix

Description

net_get() looks up the given network in routing table *tab* and returns a pointer to its `net` entry. If no such entry exists, it's created.

Function

rte * *rte_cow* (rte * *r*) – copy a route for writing

Arguments

rte * *r*
: a route entry to be copied

Description

rte_cow() takes a `rte` and prepares it for modification. The exact action taken depends on the flags of the `rte` – if it's a temporary entry, it's just returned unchanged, else a new temporary entry with the same contents is created.

The primary use of this function is inside the filter machinery – when a filter wants to modify `rte` contents (to change the preference or to attach another set of attributes), it must ensure that the `rte` is not shared with anyone else (and especially that it isn't stored in any routing table).

Result

a pointer to the new writable `rte`.

2.3 Route attribute cache

Each route entry carries a set of route attributes. Several of them vary from route to route, but most attributes are usually common for a large number of routes. To conserve memory, we've decided to store only the varying ones directly in the `rte` and hold the rest in a special structure called `rta` which is shared among all the `rte`'s with these attributes.

Each `rta` contains all the static attributes of the route (i.e., those which are always present) as structure members and a list of dynamic attributes represented by a linked list of `ea_list` structures, each of them consisting of an array of `eattr`'s containing the individual attributes. An attribute can be specified more than once in the `ea_list` chain and in such case the first occurrence overrides the others. This semantics is used especially when someone (for example a filter) wishes to alter values of several dynamic attributes, but it wants to preserve the original attribute lists maintained by another module.

Each `eattr` contains an attribute identifier (split to protocol ID and per-protocol attribute ID), protocol dependent flags, a type code (consisting of several bit fields describing attribute characteristics) and either an embedded 32-bit value or a pointer to a `adata` structure holding attribute contents.

There exist two variants of `rta`'s – cached and un-cached ones. Un-cached `rta`'s can have arbitrarily complex structure of `ea_list`'s and they can be modified by any module in the route processing chain. Cached `rta`'s have their attribute lists normalized (that means at most one `ea_list` is present and its values are sorted in order to speed up searching), they are stored in a hash table to make fast lookup possible and they are provided with a use count to allow sharing.

Routing tables always contain only cached `rta`'s.

Function

struct mpnh * *mpnh_merge* (struct mpnh * *x*, struct mpnh * *y*, int *rx*, int *ry*, int *max*, linpool * *lp*) – merge nexthop lists

Arguments

struct mpnh * *x*
list 1

struct mpnh * *y*
list 2

int *rx*
reusability of list *x*

int *ry*
reusability of list *y*

int *max*
max number of nexthops

linpool * *lp*
linpool for allocating nexthops

Description

The *mpnh_merge()* function takes two nexthop lists *x* and *y* and merges them, eliminating possible duplicates. The input lists must be sorted and the result is sorted too. The number of nexthops in result is limited by *max*. New nodes are allocated from linpool *lp*.

The arguments *rx* and *ry* specify whether corresponding input lists may be consumed by the function (i.e. their nodes reused in the resulting list), in that case the caller should not access these lists after that. To eliminate issues with deallocation of these lists, the caller should use some form of bulk deallocation (e.g. stack or linpool) to free these nodes when the resulting list is no longer needed. When reusability is not set, the corresponding lists are not modified nor linked from the resulting list.

Function

eattr * *ea_find* (ea_list * *e*, unsigned *id*) – find an extended attribute

Arguments

ea_list * *e*
attribute list to search in

unsigned *id*
attribute ID to search for

Description

Given an extended attribute list, *ea_find()* searches for a first occurrence of an attribute with specified ID, returning either a pointer to its `eattr` structure or `NULL` if no such attribute exists.

Function

eattr * *ea_walk* (struct ea_walk_state * *s*, uint *id*, uint *max*) – walk through extended attributes

Arguments

struct ea_walk_state * *s*
walk state structure

uint *id*
start of attribute ID interval

uint *max*
length of attribute ID interval

Description
Given an extended attribute list, *ea_walk()* walks through the list looking for first occurrences of attributes with ID in specified interval from *id* to (*id* + *max* - 1), returning pointers to found `eattr` structures, storing its walk state in *s* for subsequent calls.
The function *ea_walk()* is supposed to be called in a loop, with initially zeroed walk state structure *s* with filled the initial extended attribute list, returning one found attribute in each call or `NULL` when no other attribute exists. The extended attribute list or the arguments should not be modified between calls. The maximum value of *max* is 128.

Function
int *ea_get_int* (ea_list * *e*, unsigned *id*, int *def*) – fetch an integer attribute

Arguments

ea_list * *e*
attribute list

unsigned *id*
attribute ID

int *def*
default value

Description
This function is a shortcut for retrieving a value of an integer attribute by calling *ea_find()* to find the attribute, extracting its value or returning a provided default if no such attribute is present.

Function
void *ea_sort* (ea_list * *e*) – sort an attribute list

Arguments

ea_list * *e*
list to be sorted

Description
This function takes a `ea_list` chain and sorts the attributes within each of its entries.
If an attribute occurs multiple times in a single `ea_list`, *ea_sort()* leaves only the first (the only significant) occurrence.

Function
unsigned *ea_scan* (ea_list * *e*) – estimate attribute list size

Arguments

ea_list * *e*
attribute list

Description
This function calculates an upper bound of the size of a given `ea_list` after merging with *ea_merge()*.

Function
void *ea_merge* (ea_list * *e*, ea_list * *t*) – merge segments of an attribute list

Arguments

ea_list * *e*
attribute list

ea_list * *t*
buffer to store the result to

Description
This function takes a possibly multi-segment attribute list and merges all of its segments to one.
The primary use of this function is for `ea_list` normalization: first call *ea_scan()* to determine how much memory will the result take, then allocate a buffer (usually using *alloca()*), merge the segments with *ea_merge()* and finally sort and prune the result by calling *ea_sort()*.

Function
int *ea_same* (ea_list * *x*, ea_list * *y*) – compare two `ea_list`'s

Arguments

ea_list * *x*
attribute list

ea_list * *y*
attribute list

Description
ea_same() compares two normalized attribute lists *x* and *y* and returns 1 if they contain the same attributes, 0 otherwise.

Function
void *ea_show* (struct cli * *c*, eattr * *e*) – print an `eattr` to CLI

Arguments

struct cli * *c*
destination CLI

eattr * *e*
attribute to be printed

Description
This function takes an extended attribute represented by its `eattr` structure and prints it to the CLI according to the type information.
If the protocol defining the attribute provides its own *get_attr()* hook, it's consulted first.

Function
void *ea_dump* (ea_list * *e*) – dump an extended attribute

Arguments

ea_list * *e*
attribute to be dumped

Description
ea_dump() dumps contents of the extended attribute given to the debug output.

Function

uint *ea_hash* (ea_list * *e*) – calculate an `ea_list` hash key

Arguments

ea_list * *e*
attribute list

Description

ea_hash() takes an extended attribute list and calculated a hopefully uniformly distributed hash value from its contents.

Function

ea_list * *ea_append* (ea_list * *to*, ea_list * *what*) – concatenate `ea_list`'s

Arguments

ea_list * *to*
destination list (can be `NULL`)

ea_list * *what*
list to be appended (can be `NULL`)

Description

This function appends the `ea_list` *what* at the end of `ea_list` *to* and returns a pointer to the resulting list.

Function

rta * *rta_lookup* (rta * *o*) – look up a `rta` in attribute cache

Arguments

rta * *o*
a un-cached `rta`

Description

rta_lookup() gets an un-cached `rta` structure and returns its cached counterpart. It starts with examining the attribute cache to see whether there exists a matching entry. If such an entry exists, it's returned and its use count is incremented, else a new entry is created with use count set to 1.

The extended attribute lists attached to the `rta` are automatically converted to the normalized form.

Function

void *rta_dump* (rta * *a*) – dump route attributes

Arguments

rta * *a*
attribute structure to dump

Description

This function takes a `rta` and dumps its contents to the debug output.

Function

void *rta_dump_all* (*void*) – dump attribute cache

Description

This function dumps the whole contents of route attribute cache to the debug output.

Function

void *rta_init* (*void*) – initialize route attribute cache

Description

This function is called during initialization of the routing table module to set up the internals of the attribute cache.

Function

rta * *rta_clone* (rta * *r*) – clone route attributes

Arguments

rta * *r*
a `rta` to be cloned

Description

rta_clone() takes a cached `rta` and returns its identical cached copy. Currently it works by just returning the original `rta` with its use count incremented.

Function

void *rta_free* (rta * *r*) – free route attributes

Arguments

rta * *r*
a `rta` to be freed

Description

If you stop using a `rta` (for example when deleting a route which uses it), you need to call *rta_free()* to notify the attribute cache the attribute is no longer in use and can be freed if you were the last user (which *rta_free()* tests by inspecting the use count).

2.4 Routing protocols

2.4.1 Introduction

The routing protocols are the bird's heart and a fine amount of code is dedicated to their management and for providing support functions to them. (-: Actually, this is the reason why the directory with sources of the core code is called `nest` :-).

When talking about protocols, one need to distinguish between *protocols* and protocol *instances*. A protocol exists exactly once, not depending on whether it's configured or not and it can have an arbitrary number of instances corresponding to its "incarnations" requested by the configuration file. Each instance is completely autonomous, has its own configuration, its own status, its own set of routes and its own set of interfaces it works on.

A protocol is represented by a `protocol` structure containing all the basic information (protocol name, default settings and pointers to most of the protocol hooks). All these structures are linked in the *protocol_list* list.

Each instance has its own `proto` structure describing all its properties: protocol type, configuration, a resource pool where all resources belonging to the instance live, various protocol attributes (take a look at the declaration of `proto` in `protocol.h`), protocol states (see below for what do they mean), connections to routing tables, filters attached to the protocol and finally a set of pointers to the rest of protocol hooks (they are the same for all instances of the protocol, but in order to avoid extra indirections when calling the hooks from the fast path, they are stored directly in `proto`). The instance is always linked in both the global instance list (*proto_list*) and a per-status list (either *active_proto_list* for running protocols, *initial_proto_list* for protocols being initialized or *flush_proto_list* when the protocol is being shut down).

The protocol hooks are described in the next chapter, for more information about configuration of protocols, please refer to the configuration chapter and also to the description of the *proto_commit* function.

2.4.2 Protocol states

As startup and shutdown of each protocol are complex processes which can be affected by lots of external events (user's actions, reconfigurations, behavior of neighboring routers etc.), we have decided to supervise them by a pair of simple state machines – the protocol state machine and a core state machine.

The *protocol state machine* corresponds to internal state of the protocol and the protocol can alter its state whenever it wants to. There are the following states:

`PS_DOWN`
: The protocol is down and waits for being woken up by calling its start() hook.

`PS_START`
: The protocol is waiting for connection with the rest of the network. It's active, it has resources allocated, but it still doesn't want any routes since it doesn't know what to do with them.

`PS_UP`
: The protocol is up and running. It communicates with the core, delivers routes to tables and wants to hear announcement about route changes.

`PS_STOP`
: The protocol has been shut down (either by being asked by the core code to do so or due to having encountered a protocol error).

Unless the protocol is in the `PS_DOWN` state, it can decide to change its state by calling the *proto_notify_state* function.

At any time, the core code can ask the protocol to shut itself down by calling its stop() hook.

The *core state machine* takes care of the core view of protocol state. The states are traversed according to changes of the protocol state machine, but sometimes the transitions are delayed if the core needs to finish some actions (for example sending of new routes to the protocol) before proceeding to the new state. There are the following core states:

`FS_HUNGRY`
: The protocol is down, it doesn't have any routes and doesn't want them.

`FS_FEEDING`
: The protocol has reached the `PS_UP` state, but we are still busy sending the initial set of routes to it.

`FS_HAPPY`
: The protocol is up and has complete routing information.

`FS_FLUSHING`
: The protocol is shutting down (it's in either `PS_STOP` or `PS_DOWN` state) and we're flushing all of its routes from the routing tables.

2.4.3 Functions of the protocol module

The protocol module provides the following functions:

Function

void * *proto_new* (struct proto_config * *c*, unsigned *size*) – create a new protocol instance

Arguments

struct proto_config * *c*
: protocol configuration

unsigned *size*
: size of protocol data structure (each protocol instance is represented by a structure starting with generic part [struct `proto`] and continued with data specific to the protocol)

Description
When a new configuration has been read in, the core code starts initializing all the protocol instances configured by calling their *init()* hooks with the corresponding instance configuration. The initialization code of the protocol is expected to create a new instance according to the configuration by calling this function and then modifying the default settings to values wanted by the protocol.

Function

struct announce_hook * *proto_add_announce_hook* (struct proto * *p*, struct rtable * *t*, struct proto_stats * *stats*) – connect protocol to a routing table

Arguments

struct proto * *p*
protocol instance

struct rtable * *t*
routing table to connect to

struct proto_stats * *stats*
per-table protocol statistics

Description
This function creates a connection between the protocol instance *p* and the routing table *t*, making the protocol hear all changes in the table.
The announce hook is linked in the protocol ahook list. Announce hooks are allocated from the routing table resource pool and when protocol accepts routes also in the table ahook list. The are linked to the table ahook list and unlinked from it depending on export_state (in *proto_want_export_up()* and *proto_want_export_down()*) and they are automatically freed after the protocol is flushed (in *proto_fell_down()*).
Unless you want to listen to multiple routing tables (as the Pipe protocol does), you needn't to worry about this function since the connection to the protocol's primary routing table is initialized automatically by the core code.

Function

struct announce_hook * *proto_find_announce_hook* (struct proto * *p*, struct rtable * *t*) – find announce hooks

Arguments

struct proto * *p*
protocol instance

struct rtable * *t*
routing table

Description
Returns pointer to announce hook or NULL

Function

void * *proto_config_new* (struct protocol * *pr*, int *class*) – create a new protocol configuration

Arguments

struct protocol * *pr*
protocol the configuration will belong to

int *class*
SYM_PROTO or SYM_TEMPLATE

Description
Whenever the configuration file says that a new instance of a routing protocol should be created, the parser calls *proto_config_new()* to create a configuration entry for this instance (a structure staring with the `proto_config` header containing all the generic items followed by protocol-specific ones). Also, the configuration entry gets added to the list of protocol instances kept in the configuration.
The function is also used to create protocol templates (when class SYM_TEMPLATE is specified), the only difference is that templates are not added to the list of protocol instances and therefore not initialized during *protos_commit()*).

Function

void *proto_copy_config* (struct proto_config * *dest*, struct proto_config * *src*) – copy a protocol configuration

Arguments

struct proto_config * *dest*
destination protocol configuration

struct proto_config * *src*
source protocol configuration

Description
Whenever a new instance of a routing protocol is created from the template, *proto_copy_config()* is called to copy a content of the source protocol configuration to the new protocol configuration. Name, class and a node in protos list of *dest* are kept intact. *copy_config()* protocol hook is used to copy protocol-specific data.

Function

void *protos_preconfig* (struct config * *c*) – pre-configuration processing

Arguments

struct config * *c*
new configuration

Description
This function calls the *preconfig()* hooks of all routing protocols available to prepare them for reading of the new configuration.

Function

void *protos_postconfig* (struct config * *c*) – post-configuration processing

Arguments

struct config * *c*
new configuration

Description
This function calls the *postconfig()* hooks of all protocol instances specified in configuration *c*. The hooks are not called for protocol templates.

Function

void *protos_commit* (struct config * *new*, struct config * *old*, int *force_reconfig*, int *type*) – commit new protocol configuration

Arguments

struct config * *new*
new configuration

struct config * *old*
: old configuration or `NULL` if it's boot time config

int *force_reconfig*
: force restart of all protocols (used for example when the router ID changes)

int *type*
: type of reconfiguration (RECONFIG_SOFT or RECONFIG_HARD)

Description
Scan differences between *old* and *new* configuration and adjust all protocol instances to conform to the new configuration.

When a protocol exists in the new configuration, but it doesn't in the original one, it's immediately started. When a collision with the other running protocol would arise, the new protocol will be temporarily stopped by the locking mechanism.

When a protocol exists in the old configuration, but it doesn't in the new one, it's shut down and deleted after the shutdown completes.

When a protocol exists in both configurations, the core decides whether it's possible to reconfigure it dynamically - it checks all the core properties of the protocol (changes in filters are ignored if type is RECONFIG_SOFT) and if they match, it asks the *reconfigure()* hook of the protocol to see if the protocol is able to switch to the new configuration. If it isn't possible, the protocol is shut down and a new instance is started with the new configuration after the shutdown is completed.

2.5 Graceful restart recovery

Graceful restart of a router is a process when the routing plane (e.g. BIRD) restarts but both the forwarding plane (e.g kernel routing table) and routing neighbors keep proper routes, and therefore uninterrupted packet forwarding is maintained.

BIRD implements graceful restart recovery by deferring export of routes to protocols until routing tables are refilled with the expected content. After start, protocols generate routes as usual, but routes are not propagated to them, until protocols report that they generated all routes. After that, graceful restart recovery is finished and the export (and the initial feed) to protocols is enabled.

When graceful restart recovery need is detected during initialization, then enabled protocols are marked with *gr_recovery* flag before start. Such protocols then decide how to proceed with graceful restart, participation is voluntary. Protocols could lock the recovery by *proto_graceful_restart_lock()* (stored in *gr_lock* flag), which means that they want to postpone the end of the recovery until they converge and then unlock it. They also could set *gr_wait* before advancing to `PS_UP`, which means that the core should defer route export to that protocol until the end of the recovery. This should be done by protocols that expect their neigbors to keep the proper routes (kernel table, BGP sessions with BGP graceful restart capability).

The graceful restart recovery is finished when either all graceful restart locks are unlocked or when graceful restart wait timer fires.

Function

void *graceful_restart_recovery* (*void*) – request initial graceful restart recovery

Graceful restart recovery
Called by the platform initialization code if the need for recovery after graceful restart is detected during boot. Have to be called before *protos_commit()*.

Function

void *graceful_restart_init* (*void*) – initialize graceful restart

Description
When graceful restart recovery was requested, the function starts an active phase of the recovery and initializes graceful restart wait timer. The function have to be called after *protos_commit()*.

Function

void *graceful_restart_done* (struct timer *t *UNUSED*) – finalize graceful restart

Arguments

struct timer *t *UNUSED*
– undescribed –

Description

When there are no locks on graceful restart, the functions finalizes the graceful restart recovery. Protocols postponing route export until the end of the recovery are awakened and the export to them is enabled. All other related state is cleared. The function is also called when the graceful restart wait timer fires (but there are still some locks).

Function

void *proto_graceful_restart_lock* (struct proto * *p*) – lock graceful restart by protocol

Arguments

struct proto * *p*
protocol instance

Description

This function allows a protocol to postpone the end of graceful restart recovery until it converges. The lock is removed when the protocol calls *proto_graceful_restart_unlock()* or when the protocol is stopped.
The function have to be called during the initial phase of graceful restart recovery and only for protocols that are part of graceful restart (i.e. their *gr_recovery* is set), which means it should be called from protocol start hooks.

Function

void *proto_graceful_restart_unlock* (struct proto * *p*) – unlock graceful restart by protocol

Arguments

struct proto * *p*
protocol instance

Description

This function unlocks a lock from *proto_graceful_restart_lock()*. It is also automatically called when the lock holding protocol went down.

Function

void *protos_dump_all* (*void*) – dump status of all protocols

Description

This function dumps status of all existing protocol instances to the debug output. It involves printing of general status information such as protocol states, its position on the protocol lists and also calling of a *dump()* hook of the protocol to print the internals.

Function

void *proto_build* (struct protocol * *p*) – make a single protocol available

Arguments

struct protocol * *p*
the protocol

Description

After the platform specific initialization code uses *protos_build()* to add all the standard protocols, it should call *proto_build()* for all platform specific protocols to inform the core that they exist.

Function

void *protos_build* (*void*) – build a protocol list

Description

This function is called during BIRD startup to insert all standard protocols to the global protocol list. Insertion of platform specific protocols (such as the kernel syncer) is in the domain of competence of the platform dependent startup code.

Function

void *proto_set_message* (struct proto * *p*, char * *msg*, int *len*) – set administrative message to protocol

Arguments

struct proto * *p*
: protocol

char * *msg*
: message

int *len*
: message length (-1 for NULL-terminated string)

Description

The function sets administrative message (string) related to protocol state change. It is called by the nest code for manual enable/disable/restart commands all routes to the protocol, and by protocol-specific code when the protocol state change is initiated by the protocol. Using NULL message clears the last message. The message string may be either NULL-terminated or with an explicit length.

Function

void *proto_request_feeding* (struct proto * *p*) – request feeding routes to the protocol

Arguments

struct proto * *p*
: given protocol

Description

Sometimes it is needed to send again all routes to the protocol. This is called feeding and can be requested by this function. This would cause protocol export state transition to ES_FEEDING (during feeding) and when completed, it will switch back to ES_READY. This function can be called even when feeding is already running, in that case it is restarted.

Function

void *proto_notify_limit* (struct announce_hook * *ah*, struct proto_limit * *l*, int *dir*, u32 *rt_count*)

Arguments

struct announce_hook * *ah*
: announce hook

struct proto_limit * *l*
: limit being hit

int *dir*
: limit direction (PLD_*)

u32 *rt_count*
: the number of routes

Description

The function is called by the route processing core when limit *l* is breached. It activates the limit and tooks appropriate action according to *l*->action.

Function

void *proto_notify_state* (struct proto * *p*, unsigned *ps*) – notify core about protocol state change

Arguments

struct proto * *p*
: protocol the state of which has changed

unsigned *ps*
: the new status

Description

Whenever a state of a protocol changes due to some event internal to the protocol (i.e., not inside a *start()* or *shutdown()* hook), it should immediately notify the core about the change by calling *proto_notify_state()* which will write the new state to the `proto` structure and take all the actions necessary to adapt to the new state. State change to PS_DOWN immediately frees resources of protocol and might execute start callback of protocol; therefore, it should be used at tail positions of protocol callbacks.

2.6 Protocol hooks

Each protocol can provide a rich set of hook functions referred to by pointers in either the `proto` or `protocol` structure. They are called by the core whenever it wants the protocol to perform some action or to notify the protocol about any change of its environment. All of the hooks can be set to `NULL` which means to ignore the change or to take a default action.

Function

void *preconfig* (struct protocol * *p*, struct config * *c*) – protocol preconfiguration

Arguments

struct protocol * *p*
: a routing protocol

struct config * *c*
: new configuration

Description

The *preconfig()* hook is called before parsing of a new configuration.

Function

void *postconfig* (struct proto_config * *c*) – instance post-configuration

Arguments

struct proto_config * *c*
: instance configuration

Description

The *postconfig()* hook is called for each configured instance after parsing of the new configuration is finished.

Function

struct proto * *init* (struct proto_config * *c*) – initialize an instance

Arguments

struct proto_config * *c*
: instance configuration

Description

The *init()* hook is called by the core to create a protocol instance according to supplied protocol configuration.

Result

a pointer to the instance created

Function

int *reconfigure* (struct proto * *p*, struct proto_config * *c*) – request instance reconfiguration

Arguments

struct proto * *p*
: an instance

struct proto_config * *c*
: new configuration

Description

The core calls the *reconfigure()* hook whenever it wants to ask the protocol for switching to a new configuration. If the reconfiguration is possible, the hook returns 1. Otherwise, it returns 0 and the core will shut down the instance and start a new one with the new configuration.
After the protocol confirms reconfiguration, it must no longer keep any references to the old configuration since the memory it's stored in can be re-used at any time.

Function

void *dump* (struct proto * *p*) – dump protocol state

Arguments

struct proto * *p*
: an instance

Description

This hook dumps the complete state of the instance to the debug output.

Function

void *dump_attrs* (rte * *e*) – dump protocol-dependent attributes

Arguments

rte * *e*
: a route entry

Description

This hook dumps all attributes in the `rte` which belong to this protocol to the debug output.

Function

int *start* (struct proto * *p*) – request instance startup

Arguments

struct proto * *p*
: protocol instance

Description

The *start()* hook is called by the core when it wishes to start the instance. Multitable protocols should lock their tables here.

Result

new protocol state

Function

int *shutdown* (struct proto * *p*) – request instance shutdown

Arguments

struct proto * *p*
protocol instance

Description

The *stop()* hook is called by the core when it wishes to shut the instance down for some reason.

Returns

new protocol state

Function

void *cleanup* (struct proto * *p*) – request instance cleanup

Arguments

struct proto * *p*
protocol instance

Description

The *cleanup()* hook is called by the core when the protocol became hungry/down, i.e. all protocol ahooks and routes are flushed. Multitable protocols should unlock their tables here.

Function

void *get_status* (struct proto * *p*, byte * *buf*) – get instance status

Arguments

struct proto * *p*
protocol instance

byte * *buf*
buffer to be filled with the status string

Description

This hook is called by the core if it wishes to obtain an brief one-line user friendly representation of the status of the instance to be printed by the <cf/show protocols/ command.

Function

void *get_route_info* (rte * *e*, byte * *buf*, ea_list * *attrs*) – get route information

Arguments

rte * *e*
a route entry

byte * *buf*
buffer to be filled with the resulting string

ea_list * *attrs*
extended attributes of the route

Description

This hook is called to fill the buffer *buf* with a brief user friendly representation of metrics of a route belonging to this protocol.

Function

int *get_attr* (eattr * *a*, byte * *buf*, int *buflen*) – get attribute information

Arguments

eattr * *a*
an extended attribute

byte * *buf*
buffer to be filled with attribute information

int *buflen*
a length of the *buf* parameter

Description
The *get_attr()* hook is called by the core to obtain a user friendly representation of an extended route attribute. It can either leave the whole conversion to the core (by returning `GA_UNKNOWN`), fill in only attribute name (and let the core format the attribute value automatically according to the type field; by returning `GA_NAME`) or doing the whole conversion (used in case the value requires extra care; return `GA_FULL`).

Function

void *if_notify* (struct proto * *p*, unsigned *flags*, struct iface * *i*) – notify instance about interface changes

Arguments

struct proto * *p*
protocol instance

unsigned *flags*
interface change flags

struct iface * *i*
the interface in question

Description
This hook is called whenever any network interface changes its status. The change is described by a combination of status bits (`IF_CHANGE_xxx`) in the *flags* parameter.

Function

void *ifa_notify* (struct proto * *p*, unsigned *flags*, struct ifa * *a*) – notify instance about interface address changes

Arguments

struct proto * *p*
protocol instance

unsigned *flags*
address change flags

struct ifa * *a*
the interface address

Description
This hook is called to notify the protocol instance about an interface acquiring or losing one of its addresses. The change is described by a combination of status bits (`IF_CHANGE_xxx`) in the *flags* parameter.

Function

void *rt_notify* (struct proto * *p*, net * *net*, rte * *new*, rte * *old*, ea_list * *attrs*) – notify instance about routing table change

Arguments

struct proto * *p*
  protocol instance

net * *net*
  a network entry

rte * *new*
  new route for the network

rte * *old*
  old route for the network

ea_list * *attrs*
  extended attributes associated with the *new* entry

Description

The *rt_notify()* hook is called to inform the protocol instance about changes in the connected routing table *table*, that is a route *old* belonging to network *net* being replaced by a new route *new* with extended attributes *attrs*. Either *new* or *old* or both can be `NULL` if the corresponding route doesn't exist.
If the type of route announcement is RA_OPTIMAL, it is an announcement of optimal route change, *new* stores the new optimal route and *old* stores the old optimal route.
If the type of route announcement is RA_ANY, it is an announcement of any route change, *new* stores the new route and *old* stores the old route from the same protocol.
p->accept_ra_types specifies which kind of route announcements protocol wants to receive.

Function

void *neigh_notify* (neighbor * *neigh*) – notify instance about neighbor status change

Arguments

neighbor * *neigh*
  a neighbor cache entry

Description

The *neigh_notify()* hook is called by the neighbor cache whenever a neighbor changes its state, that is it gets disconnected or a sticky neighbor gets connected.

Function

ea_list * *make_tmp_attrs* (rte * *e*, struct linpool * *pool*) – convert embedded attributes to temporary ones

Arguments

rte * *e*
  route entry

struct linpool * *pool*
  linear pool to allocate attribute memory in

Description

This hook is called by the routing table functions if they need to convert the protocol attributes embedded directly in the `rte` to temporary extended attributes in order to distribute them to other protocols or to filters. *make_tmp_attrs()* creates an `ea_list` in the linear pool *pool*, fills it with values of the temporary attributes and returns a pointer to it.

Function

void *store_tmp_attrs* (rte * *e*, ea_list * *attrs*) – convert temporary attributes to embedded ones

Arguments

rte * *e*
: route entry

ea_list * *attrs*
: temporary attributes to be converted

Description

This hook is an exact opposite of *make_tmp_attrs()* – it takes a list of extended attributes and converts them to attributes embedded in the `rte` corresponding to this protocol.
You must be prepared for any of the attributes being missing from the list and use default values instead.

Function

int *import_control* (struct proto * *p*, rte ** *e*, ea_list ** *attrs*, struct linpool * *pool*) – pre-filtering decisions on route import

Arguments

struct proto * *p*
: protocol instance the route is going to be imported to

rte ** *e*
: the route in question

ea_list ** *attrs*
: extended attributes of the route

struct linpool * *pool*
: linear pool for allocation of all temporary data

Description

The *import_control()* hook is called as the first step of a exporting a route from a routing table to the protocol instance. It can modify route attributes and force acceptance or rejection of the route regardless of user-specified filters. See *rte_announce()* for a complete description of the route distribution process.
The standard use of this hook is to reject routes having originated from the same instance and to set default values of the protocol's metrics.

Result

1 if the route has to be accepted, -1 if rejected and 0 if it should be passed to the filters.

Function

int *rte_recalculate* (struct rtable * *table*, struct network * *net*, struct rte * *new*, struct rte * *old*, struct rte * *old_best*) – prepare routes for comparison

Arguments

struct rtable * *table*
: a routing table

struct network * *net*
: a network entry

struct rte * *new*
: new route for the network

struct rte * *old*
old route for the network

struct rte * *old_best*
old best route for the network (may be NULL)

Description
This hook is called when a route change (from *old* to *new* for a *net* entry) is propagated to a *table*. It may be used to prepare routes for comparison by *rte_better()* in the best route selection. *new* may or may not be in *net*->routes list, *old* is not there.

Result
1 if the ordering implied by *rte_better()* changes enough that full best route calculation have to be done, 0 otherwise.

Function
int *rte_better* (rte * *new*, rte * *old*) – compare metrics of two routes

Arguments

rte * *new*
the new route

rte * *old*
the original route

Description
This hook gets called when the routing table contains two routes for the same network which have originated from different instances of a single protocol and it wants to select which one is preferred over the other one. Protocols usually decide according to route metrics.

Result
1 if *new* is better (more preferred) than *old*, 0 otherwise.

Function
int *rte_same* (rte * *e1*, rte * *e2*) – compare two routes

Arguments

rte * *e1*
route

rte * *e2*
route

Description
The *rte_same()* hook tests whether the routes *e1* and *e2* belonging to the same protocol instance have identical contents. Contents of `rta`, all the extended attributes and `rte` preference are checked by the core code, no need to take care of them here.

Result
1 if *e1* is identical to *e2*, 0 otherwise.

Function
void *rte_insert* (net * *n*, rte * *e*) – notify instance about route insertion

Arguments

net * *n*
network

rte * *e*
route

Description
This hook is called whenever a `rte` belonging to the instance is accepted for insertion to a routing table. Please avoid using this function in new protocols.

Function

void *rte_remove* (net * *n*, rte * *e*) – notify instance about route removal

Arguments

net * *n*
network

rte * *e*
route

Description
This hook is called whenever a `rte` belonging to the instance is removed from a routing table. Please avoid using this function in new protocols.

2.7 Interfaces

The interface module keeps track of all network interfaces in the system and their addresses.

Each interface is represented by an `iface` structure which carries interface capability flags (`IF_MULTIACCESS`, `IF_BROADCAST` etc.), MTU, interface name and index and finally a linked list of network prefixes assigned to the interface, each one represented by struct `ifa`.

The interface module keeps a 'soft-up' state for each `iface` which is a conjunction of link being up, the interface being of a 'sane' type and at least one IP address assigned to it.

Function

void *ifa_dump* (struct ifa * *a*) – dump interface address

Arguments

struct ifa * *a*
interface address descriptor

Description
This function dumps contents of an `ifa` to the debug output.

Function

void *if_dump* (struct iface * *i*) – dump interface

Arguments

struct iface * *i*
interface to dump

Description
This function dumps all information associated with a given network interface to the debug output.

Function

void *if_dump_all* (*void*) – dump all interfaces

Description
This function dumps information about all known network interfaces to the debug output.

Function

void *if_delete* (struct iface * *old*) – remove interface

Arguments

struct iface * *old*
interface

Description

This function is called by the low-level platform dependent code whenever it notices an interface disappears. It is just a shorthand for *if_update()*.

Function

struct iface * *if_update* (struct iface * *new*) – update interface status

Arguments

struct iface * *new*
new interface status

Description

if_update() is called by the low-level platform dependent code whenever it notices an interface change.
There exist two types of interface updates – synchronous and asynchronous ones. In the synchronous case, the low-level code calls *if_start_update()*, scans all interfaces reported by the OS, uses *if_update()* and *ifa_update()* to pass them to the core and then it finishes the update sequence by calling *if_end_update()*. When working asynchronously, the sysdep code calls *if_update()* and *ifa_update()* whenever it notices a change.
if_update() will automatically notify all other modules about the change.

Function

void *if_feed_baby* (struct proto * *p*) – advertise interfaces to a new protocol

Arguments

struct proto * *p*
protocol to feed

Description

When a new protocol starts, this function sends it a series of notifications about all existing interfaces.

Function

struct iface * *if_find_by_index* (unsigned *idx*) – find interface by ifindex

Arguments

unsigned *idx*
ifindex

Description

This function finds an `iface` structure corresponding to an interface of the given index *idx*. Returns a pointer to the structure or `NULL` if no such structure exists.

Function

struct iface * *if_find_by_name* (char * *name*) – find interface by name

Arguments

char * *name*
interface name

Description

This function finds an `iface` structure corresponding to an interface of the given name *name*. Returns a pointer to the structure or `NULL` if no such structure exists.

Function

struct ifa * *ifa_update* (struct ifa * *a*) – update interface address

Arguments

struct ifa * *a*
new interface address

Description
This function adds address information to a network interface. It's called by the platform dependent code during the interface update process described under *if_update()*.

Function

void *ifa_delete* (struct ifa * *a*) – remove interface address

Arguments

struct ifa * *a*
interface address

Description
This function removes address information from a network interface. It's called by the platform dependent code during the interface update process described under *if_update()*.

Function

void *if_init* (*void*) – initialize interface module

Description
This function is called during BIRD startup to initialize all data structures of the interface module.

2.8 Neighbor cache

Most routing protocols need to associate their internal state data with neighboring routers, check whether an address given as the next hop attribute of a route is really an address of a directly connected host and which interface is it connected through. Also, they often need to be notified when a neighbor ceases to exist or when their long awaited neighbor becomes connected. The neighbor cache is there to solve all these problems.

The neighbor cache maintains a collection of neighbor entries. Each entry represents one IP address corresponding to either our directly connected neighbor or our own end of the link (when the scope of the address is set to `SCOPE_HOST`) together with per-neighbor data belonging to a single protocol.

Active entries represent known neighbors and are stored in a hash table (to allow fast retrieval based on the IP address of the node) and two linked lists: one global and one per-interface (allowing quick processing of interface change events). Inactive entries exist only when the protocol has explicitly requested it via the `NEF_STICKY` flag because it wishes to be notified when the node will again become a neighbor. Such entries are enqueued in a special list which is walked whenever an interface changes its state to up. Neighbor entry VRF association is implied by respective protocol.

When a neighbor event occurs (a neighbor gets disconnected or a sticky inactive neighbor becomes connected), the protocol hook *neigh_notify()* is called to advertise the change.

Function

neighbor * *neigh_find* (struct proto * *p*, ip_addr * *a*, unsigned *flags*) – find or create a neighbor entry.

Arguments

struct proto * *p*
protocol which asks for the entry.

ip_addr * *a*
pointer to IP address of the node to be searched for.

unsigned *flags*
0 or `NEF_STICKY` if you want to create a sticky entry.

Description
Search the neighbor cache for a node with given IP address. If it's found, a pointer to the neighbor entry is returned. If no such entry exists and the node is directly connected on one of our active interfaces, a new entry is created and returned to the caller with protocol-dependent fields initialized to zero. If the node is not connected directly or **a* is not a valid unicast IP address, *neigh_find()* returns `NULL`.

Function
void *neigh_dump* (neighbor * *n*) – dump specified neighbor entry.

Arguments

neighbor * *n*
the entry to dump

Description
This functions dumps the contents of a given neighbor entry to debug output.

Function
void *neigh_dump_all* (*void*) – dump all neighbor entries.

Description
This function dumps the contents of the neighbor cache to debug output.

Function
void *neigh_if_up* (struct iface * *i*)

Arguments

struct iface * *i*
interface in question

Description
Tell the neighbor cache that a new interface became up.
The neighbor cache wakes up all inactive sticky neighbors with addresses belonging to prefixes of the interface *i*.

Function
void *neigh_if_down* (struct iface * *i*) – notify neighbor cache about interface down event

Arguments

struct iface * *i*
the interface in question

Description
Notify the neighbor cache that an interface has ceased to exist.
It causes all entries belonging to neighbors connected to this interface to be flushed.

Function

void *neigh_if_link* (struct iface * *i*) – notify neighbor cache about interface link change

Arguments

struct iface * *i*
: the interface in question

Description
Notify the neighbor cache that an interface changed link state. All owners of neighbor entries connected to this interface are notified.

Function

void *neigh_ifa_update* (struct ifa * *a*)

Arguments

struct ifa * *a*
: interface address in question

Description
Tell the neighbor cache that an address was added or removed.
The neighbor cache wakes up all inactive sticky neighbors with addresses belonging to prefixes of the interface belonging to *ifa* and causes all unreachable neighbors to be flushed.

Function

void *neigh_prune* (*void*) – prune neighbor cache

Description
neigh_prune() examines all neighbor entries cached and removes those corresponding to inactive protocols. It's called whenever a protocol is shut down to get rid of all its heritage.

Function

void *neigh_init* (pool * *if_pool*) – initialize the neighbor cache.

Arguments

pool * *if_pool*
: resource pool to be used for neighbor entries.

Description
This function is called during BIRD startup to initialize the neighbor cache module.

2.9 Command line interface

This module takes care of the BIRD's command-line interface (CLI). The CLI exists to provide a way to control BIRD remotely and to inspect its status. It uses a very simple textual protocol over a stream connection provided by the platform dependent code (on UNIX systems, it's a UNIX domain socket).

Each session of the CLI consists of a sequence of request and replies, slightly resembling the FTP and SMTP protocols. Requests are commands encoded as a single line of text, replies are sequences of lines starting with a four-digit code followed by either a space (if it's the last line of the reply) or a minus sign (when the reply is going to continue with the next line), the rest of the line contains a textual message semantics of which depends on the numeric code. If a reply line has the same code as the previous one and it's a continuation line, the whole prefix can be replaced by a single white space character.

Reply codes starting with 0 stand for 'action successfully completed' messages, 1 means 'table entry', 8 'runtime error' and 9 'syntax error'.

Each CLI session is internally represented by a `cli` structure and a resource pool containing all resources associated with the connection, so that it can be easily freed whenever the connection gets closed, not depending on the current state of command processing.

The CLI commands are declared as a part of the configuration grammar by using the `CF_CLI` macro. When a command is received, it is processed by the same lexical analyzer and parser as used for the configuration, but it's switched to a special mode by prepending a fake token to the text, so that it uses only the CLI command rules. Then the parser invokes an execution routine corresponding to the command, which either constructs the whole reply and returns it back or (in case it expects the reply will be long) it prints a partial reply and asks the CLI module (using the *cont* hook) to call it again when the output is transferred to the user.

The *this_cli* variable points to a `cli` structure of the session being currently parsed, but it's of course available only in command handlers not entered using the *cont* hook.

TX buffer management works as follows: At cli.tx_buf there is a list of TX buffers (struct cli_out), cli.tx_write is the buffer currently used by the producer (*cli_printf()*, *cli_alloc_out()*) and cli.tx_pos is the buffer currently used by the consumer (*cli_write()*, in system dependent code). The producer uses cli_out.wpos ptr as the current write position and the consumer uses cli_out.outpos ptr as the current read position. When the producer produces something, it calls *cli_write_trigger()*. If there is not enough space in the current buffer, the producer allocates the new one. When the consumer processes everything in the buffer queue, it calls *cli_written()*, tha frees all buffers (except the first one) and schedules cli.event .

Function

void *cli_printf* (cli * *c*, int *code*, char * *msg*, ...) – send reply to a CLI connection

Arguments

cli * *c*
CLI connection

int *code*
numeric code of the reply, negative for continuation lines

char * *msg*
a *printf()*-like formatting string.

... ...
variable arguments

Description

This function send a single line of reply to a given CLI connection. In works in all aspects like *bsprintf()* except that it automatically prepends the reply line prefix.

Please note that if the connection can be already busy sending some data in which case *cli_printf()* stores the output to a temporary buffer, so please avoid sending a large batch of replies without waiting for the buffers to be flushed.

If you want to write to the current CLI output, you can use the *cli_msg()* macro instead.

Function

void *cli_init* (*void*) – initialize the CLI module

Description

This function is called during BIRD startup to initialize the internal data structures of the CLI module.

2.10 Object locks

The lock module provides a simple mechanism for avoiding conflicts between various protocols which would like to use a single physical resource (for example a network port). It would be easy to say that such collisions can occur only when the user specifies an invalid configuration and therefore he deserves to get what he has asked for, but unfortunately they can also arise legitimately when the daemon is reconfigured and there exists (although for a short time period only) an old protocol instance being shut down and a new one willing to start up on the same interface.

The solution is very simple: when any protocol wishes to use a network port or some other non-shareable resource, it asks the core to lock it and it doesn't use the resource until it's notified that it has acquired the lock.

Object locks are represented by `object_lock` structures which are in turn a kind of resource. Lockable resources are uniquely determined by resource type (`OBJLOCK_UDP` for a UDP port etc.), IP address (usually a broadcast or multicast address the port is bound to), port number, interface and optional instance ID.

Function

struct object_lock * *olock_new* (pool * *p*) – create an object lock

Arguments

pool * *p*
resource pool to create the lock in.

Description
The *olock_new()* function creates a new resource of type `object_lock` and returns a pointer to it. After filling in the structure, the caller should call *olock_acquire()* to do the real locking.

Function

void *olock_acquire* (struct object_lock * *l*) – acquire a lock

Arguments

struct object_lock * *l*
the lock to acquire

Description
This function attempts to acquire exclusive access to the non-shareable resource described by the lock *l*. It returns immediately, but as soon as the resource becomes available, it calls the *hook()* function set up by the caller.
When you want to release the resource, just *rfree()* the lock.

Function

void *olock_init* (*void*) – initialize the object lock mechanism

Description
This function is called during BIRD startup. It initializes all the internal data structures of the lock module.

Chapter 3: Configuration

3.1 Configuration manager

Configuration of BIRD is complex, yet straightforward. There are three modules taking care of the configuration: config manager (which takes care of storage of the config information and controls switching between configs), lexical analyzer and parser.

The configuration manager stores each config as a `config` structure accompanied by a linear pool from which all information associated with the config and pointed to by the `config` structure is allocated.

There can exist up to four different configurations at one time: an active one (pointed to by *config*), configuration we are just switching from (*old_config*), one queued for the next reconfiguration (*future_config*; if there is one and the user wants to reconfigure once again, we just free the previous queued config and replace it with the new one) and finally a config being parsed (*new_config*). The stored *old_config* is also used for undo reconfiguration, which works in a similar way. Reconfiguration could also have timeout (using *config_timer*) and undo is automatically called if the new configuration is not confirmed later. The new config (*new_config*) and associated linear pool (*cfg_mem*) is non-NULL only during parsing.

Loading of new configuration is very simple: just call *config_alloc()* to get a new `config` structure, then use *config_parse()* to parse a configuration file and fill all fields of the structure and finally ask the config manager to switch to the new config by calling *config_commit()*.

CLI commands are parsed in a very similar way – there is also a stripped-down `config` structure associated with them and they are lex-ed and parsed by the same functions, only a special fake token is prepended before the command text to make the parser recognize only the rules corresponding to CLI commands.

Function

struct config * *config_alloc* (const byte * *name*) – allocate a new configuration

Arguments

const byte * *name*
name of the config

Description
This function creates new `config` structure, attaches a resource pool and a linear memory pool to it and makes it available for further use. Returns a pointer to the structure.

Function

int *config_parse* (struct config * *c*) – parse a configuration

Arguments

struct config * *c*
configuration

Description
config_parse() reads input by calling a hook function pointed to by *cf_read_hook* and parses it according to the configuration grammar. It also calls all the preconfig and postconfig hooks before, resp. after parsing.

Result
1 if the config has been parsed successfully, 0 if any error has occurred (such as anybody calling *cf_error()*) and the *err_msg* field has been set to the error message.

Function

int *cli_parse* (struct config * *c*) – parse a CLI command

Arguments

struct config * *c*
temporary config structure

Description
cli_parse() is similar to *config_parse()*, but instead of a configuration, it parses a CLI command. See the CLI module for more information.

Function
void *config_free* (struct config * *c*) – free a configuration

Arguments

struct config * *c*
configuration to be freed

Description
This function takes a `config` structure and frees all resources associated with it.

Function
int *config_commit* (struct config * *c*, int *type*, int *timeout*) – commit a configuration

Arguments

struct config * *c*
new configuration

int *type*
type of reconfiguration (RECONFIG_SOFT or RECONFIG_HARD)

int *timeout*
timeout for undo (or 0 for no timeout)

Description
When a configuration is parsed and prepared for use, the *config_commit()* function starts the process of reconfiguration. It checks whether there is already a reconfiguration in progress in which case it just queues the new config for later processing. Else it notifies all modules about the new configuration by calling their *commit()* functions which can either accept it immediately or call *config_add_obstacle()* to report that they need some time to complete the reconfiguration. After all such obstacles are removed using *config_del_obstacle()*, the old configuration is freed and everything runs according to the new one.
When *timeout* is nonzero, the undo timer is activated with given timeout. The timer is deactivated when *config_commit()*, *config_confirm()* or *config_undo()* is called.

Result
`CONF_DONE` if the configuration has been accepted immediately, `CONF_PROGRESS` if it will take some time to switch to it, `CONF_QUEUED` if it's been queued due to another reconfiguration being in progress now or `CONF_SHUTDOWN` if BIRD is in shutdown mode and no new configurations are accepted.

Function
int *config_confirm* (*void*) – confirm a commited configuration

Description
When the undo timer is activated by *config_commit()* with nonzero timeout, this function can be used to deactivate it and therefore confirm the current configuration.

Result
`CONF_CONFIRM` when the current configuration is confirmed, `CONF_NONE` when there is nothing to confirm (i.e. undo timer is not active).

Function

int *config_undo* (*void*) – undo a configuration

Description

Function *config_undo()* can be used to change the current configuration back to stored `old_config`. If no reconfiguration is running, this stored configuration is commited in the same way as a new configuration in *config_commit()*. If there is already a reconfiguration in progress and no next reconfiguration is scheduled, then the undo is scheduled for later processing as usual, but if another reconfiguration is already scheduled, then such reconfiguration is removed instead (i.e. undo is applied on the last commit that scheduled it).

Result

`CONF_DONE` if the configuration has been accepted immediately, `CONF_PROGRESS` if it will take some time to switch to it, `CONF_QUEUED` if it's been queued due to another reconfiguration being in progress now, `CONF_UNQUEUED` if a scheduled reconfiguration is removed, `CONF_NOTHING` if there is no relevant configuration to undo (the previous config request was *config_undo()* too) or `CONF_SHUTDOWN` if BIRD is in shutdown mode and no new configuration changes are accepted.

Function

void *order_shutdown* (*void*) – order BIRD shutdown

Description

This function initiates shutdown of BIRD. It's accomplished by asking for switching to an empty configuration.

Function

void *cf_error* (char * *msg*, ... *...*) – report a configuration error

Arguments

char * *msg*
printf-like format string

... *...*
variable arguments

Description

cf_error() can be called during execution of *config_parse()*, that is from the parser, a preconfig hook or a postconfig hook, to report an error in the configuration.

Function

char * *cfg_strdup* (const char * *c*) – copy a string to config memory

Arguments

const char * *c*
string to copy

Description

cfg_strdup() creates a new copy of the string in the memory pool associated with the configuration being currently parsed. It's often used when a string literal occurs in the configuration and we want to preserve it for further use.

3.2 Lexical analyzer

The lexical analyzer used for configuration files and CLI commands is generated using the `flex` tool accompanied by a couple of functions maintaining the hash tables containing information about symbols and keywords.

Each symbol is represented by a `symbol` structure containing name of the symbol, its lexical scope, symbol class (`SYM_PROTO` for a name of a protocol, `SYM_CONSTANT` for a constant etc.) and class dependent data. When an unknown symbol is encountered, it's automatically added to the symbol table with class `SYM_VOID`.

The keyword tables are generated from the grammar templates using the `gen_keywords.m4` script.

Function

void *cf_lex_unwind* (*void*) – unwind lexer state during error

Lexical analyzer

cf_lex_unwind() frees the internal state on IFS stack when the lexical analyzer is terminated by *cf_error()*.

Function

struct symbol * *cf_find_symbol* (struct config * *cfg*, byte * *c*) – find a symbol by name

Arguments

struct config * *cfg*
specificed config

byte * *c*
symbol name

Description

This functions searches the symbol table in the config *cfg* for a symbol of given name. First it examines the current scope, then the second recent one and so on until it either finds the symbol and returns a pointer to its `symbol` structure or reaches the end of the scope chain and returns `NULL` to signify no match.

Function

struct symbol * *cf_get_symbol* (byte * *c*) – get a symbol by name

Arguments

byte * *c*
symbol name

Description

This functions searches the symbol table of the currently parsed config (*new_config*) for a symbol of given name. It returns either the already existing symbol or a newly allocated undefined (`SYM_VOID`) symbol if no existing symbol is found.

Function

struct symbol * *cf_define_symbol* (struct symbol * *sym*, int *type*, void * *def*) – define meaning of a symbol

Arguments

struct symbol * *sym*
symbol to be defined

int *type*
symbol class to assign

void * *def*
class dependent data

Description
Defines new meaning of a symbol. If the symbol is an undefined one (`SYM_VOID`), it's just re-defined to the new type. If it's defined in different scope, a new symbol in current scope is created and the meaning is assigned to it. If it's already defined in the current scope, an error is reported via *cf_error()*.

Result
Pointer to the newly defined symbol. If we are in the top-level scope, it's the same *sym* as passed to the function.

Function

void *cf_lex_init* (int *is_cli*, struct config * *c*) – initialize the lexer

Arguments

int *is_cli*
: true if we're going to parse CLI command, false for configuration

struct config * *c*
: configuration structure

Description
cf_lex_init() initializes the lexical analyzer and prepares it for parsing of a new input.

Function

void *cf_push_scope* (struct symbol * *sym*) – enter new scope

Arguments

struct symbol * *sym*
: symbol representing scope name

Description
If we want to enter a new scope to process declarations inside a nested block, we can just call *cf_push_scope()* to push a new scope onto the scope stack which will cause all new symbols to be defined in this scope and all existing symbols to be sought for in all scopes stored on the stack.

Function

void *cf_pop_scope* (*void*) – leave a scope

Description
cf_pop_scope() pops the topmost scope from the scope stack, leaving all its symbols in the symbol table, but making them invisible to the rest of the config.

Function

char * *cf_symbol_class_name* (struct symbol * *sym*) – get name of a symbol class

Arguments

struct symbol * *sym*
: symbol

Description
This function returns a string representing the class of the given symbol.

3.3 Parser

Both the configuration and CLI commands are analyzed using a syntax driven parser generated by the `bison` tool from a grammar which is constructed from information gathered from grammar snippets by the `gen_parser.m4` script.

Grammar snippets are files (usually with extension `.Y`) contributed by various BIRD modules in order to provide information about syntax of their configuration and their CLI commands. Each snipped consists of several sections, each of them starting with a special keyword: `CF_HDR` for a list of `#include` directives needed by the C code, `CF_DEFINES` for a list of C declarations, `CF_DECLS` for `bison` declarations including keyword definitions specified as `CF_KEYWORDS`, `CF_GRAMMAR` for the grammar rules, `CF_CODE` for auxiliary C code and finally `CF_END` at the end of the snippet.

To create references between the snippets, it's possible to define multi-part rules by utilizing the `CF_ADDTO` macro which adds a new alternative to a multi-part rule.

CLI commands are defined using a `CF_CLI` macro. Its parameters are: the list of keywords determining the command, the list of parameters, help text for the parameters and help text for the command.

Values of `enum` filter types can be defined using `CF_ENUM` with the following parameters: name of filter type, prefix common for all literals of this type and names of all the possible values.

Chapter 4: Filters

4.1 Filters

You can find sources of the filter language in `filter/` directory. File `filter/config.Y` contains filter grammar and basically translates the source from user into a tree of `f_inst` structures. These trees are later interpreted using code in `filter/filter.c`.

A filter is represented by a tree of `f_inst` structures, one structure per "instruction". Each `f_inst` contains *code*, *aux* value which is usually the data type this instruction operates on and two generic arguments (*a1*, *a2*). Some instructions contain pointer(s) to other instructions in their (*a1*, *a2*) fields.

Filters use a `f_val` structure for their data. Each `f_val` contains type and value (types are constants prefixed with `T_`). Few of the types are special; `T_RETURN` can be or-ed with a type to indicate that return from a function or from the whole filter should be forced. Important thing about `f_val`'s is that they may be copied with a simple `=`. That's fine for all currently defined types: strings are read-only (and therefore okay), paths are copied for each operation (okay too).

Function

int *val_compare* (struct f_val *v1*, struct f_val *v2*) – compare two values

Arguments

struct f_val *v1*
first value

struct f_val *v2*
second value

Description

Compares two values and returns -1, 0, 1 on <, =, > or CMP_ERROR on error. Tree module relies on this giving consistent results so that it can be used for building balanced trees.

Function

int *val_same* (struct f_val *v1*, struct f_val *v2*) – compare two values

Arguments

struct f_val *v1*
first value

struct f_val *v2*
second value

Description

Compares two values and returns 1 if they are same and 0 if not. Comparison of values of different types is valid and returns 0.

Function

int *val_in_range* (struct f_val *v1*, struct f_val *v2*) – implement ~ operator

Arguments

struct f_val *v1*
element

struct f_val *v2*
set

Description

Checks if *v1* is element (~ operator) of *v2*.

Function

struct f_val *interpret* (struct f_inst * *what*)

Arguments

struct f_inst * *what*
: filter to interpret

Description

Interpret given tree of filter instructions. This is core function of filter system and does all the hard work.

Each instruction has 4 fields

code (which is instruction code), aux (which is extension to instruction code, typically type), arg1 and arg2 - arguments. Depending on instruction, arguments are either integers, or pointers to instruction trees. Common instructions like +, that have two expressions as arguments use TWOARGS macro to get both of them evaluated.
`f_val` structures are copied around, so there are no problems with memory managment.

Function

int *f_run* (struct filter * *filter*, struct rte ** *rte*, struct ea_list ** *tmp_attrs*, struct linpool * *tmp_pool*, int *flags*) – run a filter for a route

Arguments

struct filter * *filter*
: filter to run

struct rte ** *rte*
: route being filtered, may be modified

struct ea_list ** *tmp_attrs*
: temporary attributes, prepared by caller or generated by *f_run()*

struct linpool * *tmp_pool*
: all filter allocations go from this pool

int *flags*
: flags

Description

If filter needs to modify the route, there are several posibilities. *rte* might be read-only (with REF_COW flag), in that case rw copy is obtained by *rte_cow()* and *rte* is replaced. If *rte* is originally rw, it may be directly modified (and it is never copied).
The returned rte may reuse the (possibly cached, cloned) rta, or (if rta was modificied) contains a modified uncached rta, which uses parts allocated from *tmp_pool* and parts shared from original rta. There is one exception - if *rte* is rw but contains a cached rta and that is modified, rta in returned rte is also cached.
Ownership of cached rtas is consistent with rte, i.e. if a new rte is returned, it has its own clone of cached rta (and cached rta of read-only source rte is intact), if rte is modified in place, old cached rta is possibly freed.

Function

int *filter_same* (struct filter * *new*, struct filter * *old*) – compare two filters

Arguments

struct filter * *new*
: first filter to be compared

struct filter * *old*
: second filter to be compared, notice that this filter is damaged while comparing.

Description

Returns 1 in case filters are same, otherwise 0. If there are underlying bugs, it will rather say 0 on same filters than say 1 on different.

Function

struct f_tree * *find_tree* (struct f_tree * *t*, struct f_val *val*)

Arguments

struct f_tree * *t*
  tree to search in

struct f_val *val*
  value to find

Description

Search for given value in the tree. I relies on fact that sorted tree is populated by `f_val` structures (that can be compared by *val_compare()*). In each node of tree, either single value (then t->from==t->to) or range is present.

Both set matching and *switch()* { } construction is implemented using this function, thus both are as fast as they can be.

Function

struct f_tree * *build_tree* (struct f_tree * *from*)

Arguments

struct f_tree * *from*
  degenerated tree (linked by *tree*->left) to be transformed into form suitable for *find_tree()*

Description

Transforms degenerated tree into balanced tree.

Function

int *same_tree* (struct f_tree * *t1*, struct f_tree * *t2*)

Arguments

struct f_tree * *t1*
  first tree to be compared

struct f_tree * *t2*
  second one

Description

Compares two trees and returns 1 if they are same

4.2 Trie for prefix sets

We use a (compressed) trie to represent prefix sets. Every node in the trie represents one prefix (`addr/plen`) and `plen` also indicates the index of the bit in the address that is used to branch at the node. If we need to represent just a set of prefixes, it would be simple, but we have to represent a set of prefix patterns. Each prefix pattern consists of `ppaddr/pplen` and two integers: `low` and `high`, and a prefix `paddr/plen` matches that pattern if the first MIN(`plen`, `pplen`) bits of `paddr` and `ppaddr` are the same and `low` <= `plen` <= `high`.

We use a bitmask (`accept`) to represent accepted prefix lengths at a node. As there are 33 prefix lengths (0..32 for IPv4), but there is just one prefix of zero length in the whole trie so we have `zero` flag in `f_trie` (indicating whether the trie accepts prefix 0.0.0.0/0) as a special case, and `accept` bitmask represents accepted prefix lengths from 1 to 32.

There are two cases in prefix matching - a match when the length of the prefix is smaller that the length of the prefix pattern, (`plen` < `pplen`) and otherwise. The second case is simple - we just walk through the trie and look at every visited node whether that prefix accepts our prefix length (`plen`). The first case is tricky - we don't want to examine every descendant of a final node, so (when we create the trie) we have to propagate that information from nodes to their ascendants.

Suppose that we have two masks (M1 and M2) for a node. Mask M1 represents accepted prefix lengths by just the node and mask M2 represents accepted prefix lengths by the node or any of its descendants. Therefore M2 is a bitwise or of M1 and children's M2 and this is a maintained invariant during trie building. Basically, when we want to match a prefix, we walk through the trie, check mask M1 for our prefix length and when we came to final node, we check mask M2.

There are two differences in the real implementation. First, we use a compressed trie so there is a case that we skip our final node (if it is not in the trie) and we came to node that is either extension of our prefix, or completely out of path In the first case, we also have to check M2.

Second, we really need not to maintain two separate bitmasks. Checks for mask M1 are always larger than `applen` and we need just the first `pplen` bits of mask M2 (if trie compression hadn't been used it would suffice to know just $applen-th bit), so we have to store them together in `accept` mask - the first `pplen` bits of mask M2 and then mask M1.

There are four cases when we walk through a trie:

- we are in NULL - we are out of path (prefixes are inconsistent) - we are in the wanted (final) node (node length == `plen`) - we are beyond the end of path (node length > `plen`) - we are still on path and keep walking (node length < `plen`)

The walking code in *trie_match_prefix()* is structured according to these cases.

Function

struct f_trie * *f_new_trie* (linpool * *lp*, uint *node_size*) – allocates and returns a new empty trie

Arguments

linpool * *lp*
: linear pool to allocate items from

uint *node_size*
: node size to be used (`f_trie_node` and user data)

Function

void * *trie_add_prefix* (struct f_trie * *t*, ip_addr *px*, int *plen*, int *l*, int *h*)

Arguments

struct f_trie * *t*
: trie to add to

ip_addr *px*
: prefix address

int *plen*
: prefix length

int *l*
: prefix lower bound

int *h*
: prefix upper bound

Description

Adds prefix (prefix pattern) *px*/*plen* to trie *t*. *l* and *h* are lower and upper bounds on accepted prefix lengths, both inclusive. 0 <= l, h <= 32 (128 for IPv6).

Returns a pointer to the allocated node. The function can return a pointer to an existing node if *px* and *plen* are the same. If px/plen == 0/0 (or ::/0), a pointer to the root node is returned.

Function

int *trie_match_prefix* (struct f_trie * *t*, ip_addr *px*, int *plen*)

Arguments

struct f_trie * *t*
trie

ip_addr *px*
prefix address

int *plen*
prefix length

Description

Tries to find a matching prefix pattern in the trie such that prefix *px/plen* matches that prefix pattern. Returns 1 if there is such prefix pattern in the trie.

Function

int *trie_same* (struct f_trie * *t1*, struct f_trie * *t2*)

Arguments

struct f_trie * *t1*
first trie to be compared

struct f_trie * *t2*
second one

Description

Compares two tries and returns 1 if they are same

Function

void *trie_format* (struct f_trie * *t*, buffer * *buf*)

Arguments

struct f_trie * *t*
trie to be formatted

buffer * *buf*
destination buffer

Description

Prints the trie to the supplied buffer.

Chapter 5: Protocols

5.1 The Babel protocol

Babel (RFC6126) is a loop-avoiding distance-vector routing protocol that is robust and efficient both in ordinary wired networks and in wireless mesh networks.

The Babel protocol keeps state for each neighbour in a `babel_neighbor` struct, tracking received Hello and I Heard You (IHU) messages. A `babel_interface` struct keeps hello and update times for each interface, and a separate hello seqno is maintained for each interface.

For each prefix, Babel keeps track of both the possible routes (with next hop and router IDs), as well as the feasibility distance for each prefix and router id. The prefix itself is tracked in a `babel_entry` struct, while the possible routes for the prefix are tracked as `babel_route` entries and the feasibility distance is maintained through `babel_source` structures.

The main route selection is done in *babel_select_route()*. This is called when an entry is updated by receiving updates from the network or when modified by internal timers. It performs feasibility checks on the available routes for the prefix and selects the one with the lowest metric to be announced to the core.

Function

void *babel_announce_rte* (struct babel_proto * *p*, struct babel_entry * *e*) – announce selected route to the core

Arguments

struct babel_proto * *p*
: Babel protocol instance

struct babel_entry * *e*
: Babel route entry to announce

Description

This function announces a Babel entry to the core if it has a selected incoming path, and retracts it otherwise. If the selected entry has infinite metric, the route is announced as unreachable.

Function

void *babel_select_route* (struct babel_entry * *e*) – select best route for given route entry

Arguments

struct babel_entry * *e*
: Babel entry to select the best route for

Description

Select the best feasible route for a given prefix among the routes received from peers, and propagate it to the nest. This just selects the feasible route with the lowest metric.

If no feasible route is available for a prefix that previously had a route selected, a seqno request is sent to try to get a valid route. In the meantime, the route is marked as infeasible in the nest (to blackhole packets going to it, as per the RFC).

If no feasible route is available, and no previous route is selected, the route is removed from the nest entirely.

Function

void *babel_send_update* (struct babel_iface * *ifa*, bird_clock_t *changed*) – send route table updates

Arguments

struct babel_iface * *ifa*
: Interface to transmit on

bird_clock_t *changed*
Only send entries changed since this time

Description
This function produces update TLVs for all entries changed since the time indicated by the `changed` parameter and queues them for transmission on the selected interface. During the process, the feasibility distance for each transmitted entry is updated.

Function
void *babel_handle_update* (union babel_msg * *m*, struct babel_iface * *ifa*) – handle incoming route updates

Arguments

union babel_msg * *m*
Incoming update TLV

struct babel_iface * *ifa*
Interface the update was received on

Description
This function is called as a handler for update TLVs and handles the updating and maintenance of route entries in Babel's internal routing cache. The handling follows the actions described in the Babel RFC, and at the end of each update handling, *babel_select_route()* is called on the affected entry to optionally update the selected routes and propagate them to the core.

Function
void *babel_iface_timer* (timer * *t*) – Babel interface timer handler

Arguments

timer * *t*
Timer

Description
This function is called by the per-interface timer and triggers sending of periodic Hello's and both triggered and periodic updates. Periodic Hello's and updates are simply handled by setting the next_{hello,regular} variables on the interface, and triggering an update (and resetting the variable) whenever 'now' exceeds that value.
For triggered updates, *babel_trigger_iface_update()* will set the want_triggered field on the interface to a timestamp value. If this is set (and the next_triggered time has passed; this is a rate limiting mechanism), *babel_send_update()* will be called with this timestamp as the second parameter. This causes updates to be send consisting of only the routes that have changed since the time saved in want_triggered.
Mostly when an update is triggered, the route being modified will be set to the value of 'now' at the time of the trigger; the >= comparison for selecting which routes to send in the update will make sure this is included.

Function
void *babel_timer* (timer * *t*) – global timer hook

Arguments

timer * *t*
Timer

Description
This function is called by the global protocol instance timer and handles expiration of routes and neighbours as well as pruning of the seqno request cache.

Function

uint *babel_write_queue* (struct babel_iface * *ifa*, list * *queue*) – Write a TLV queue to a transmission buffer

Arguments

struct babel_iface * *ifa*
: Interface holding the transmission buffer

list * *queue*
: TLV queue to write (containing internal-format TLVs)

Description

This function writes a packet to the interface transmission buffer with as many TLVs from the `queue` as will fit in the buffer. It returns the number of bytes written (NOT counting the packet header). The function is called by *babel_send_queue()* and *babel_send_unicast()* to construct packets for transmission, and uses per-TLV helper functions to convert the internal-format TLVs to their wire representations.

The TLVs in the queue are freed after they are written to the buffer.

Function

void *babel_send_unicast* (union babel_msg * *msg*, struct babel_iface * *ifa*, ip_addr *dest*) – send a single TLV via unicast to a destination

Arguments

union babel_msg * *msg*
: TLV to send

struct babel_iface * *ifa*
: Interface to send via

ip_addr *dest*
: Destination of the TLV

Description

This function is used to send a single TLV via unicast to a designated receiver. This is used for replying to certain incoming requests, and for sending unicast requests to refresh routes before they expire.

Function

void *babel_enqueue* (union babel_msg * *msg*, struct babel_iface * *ifa*) – enqueue a TLV for transmission on an interface

Arguments

union babel_msg * *msg*
: TLV to enqueue (in internal TLV format)

struct babel_iface * *ifa*
: Interface to enqueue to

Description

This function is called to enqueue a TLV for subsequent transmission on an interface. The transmission event is triggered whenever a TLV is enqueued; this ensures that TLVs will be transmitted in a timely manner, but that TLVs which are enqueued in rapid succession can be transmitted together in one packet.

Function

void *babel_process_packet* (struct babel_pkt_header * *pkt*, int *len*, ip_addr *saddr*, struct babel_iface * *ifa*) – process incoming data packet

Arguments

struct babel_pkt_header * *pkt*
  Pointer to the packet data

int *len*
  Length of received packet

ip_addr *saddr*
  Address of packet sender

struct babel_iface * *ifa*
  Interface packet was received on.

Description

This function is the main processing hook of incoming Babel packets. It checks that the packet header is well-formed, then processes the TLVs contained in the packet. This is done in two passes: First all TLVs are parsed into the internal TLV format. If a TLV parser fails, processing of the rest of the packet is aborted. After the parsing step, the TLV handlers are called for each parsed TLV in order.

5.2 Bidirectional Forwarding Detection

The BFD protocol is implemented in three files: `bfd.c` containing the protocol logic and the protocol glue with BIRD core, `packets.c` handling BFD packet processing, RX, TX and protocol sockets. `io.c` then contains generic code for the event loop, threads and event sources (sockets, microsecond timers). This generic code will be merged to the main BIRD I/O code in the future.

The BFD implementation uses a separate thread with an internal event loop for handling the protocol logic, which requires high-res and low-latency timing, so it is not affected by the rest of BIRD, which has several low-granularity hooks in the main loop, uses second-based timers and cannot offer good latency. The core of BFD protocol (the code related to BFD sessions, interfaces and packets) runs in the BFD thread, while the rest (the code related to BFD requests, BFD neighbors and the protocol glue) runs in the main thread.

BFD sessions are represented by structure `bfd_session` that contains a state related to the session and two timers (TX timer for periodic packets and hold timer for session timeout). These sessions are allocated from *session_slab* and are accessible by two hash tables, *session_hash_id* (by session ID) and *session_hash_ip* (by IP addresses of neighbors). Slab and both hashes are in the main protocol structure `bfd_proto`. The protocol logic related to BFD sessions is implemented in internal functions bfd_session_*(), which are expected to be called from the context of BFD thread, and external functions *bfd_add_session()*, *bfd_remove_session()* and *bfd_reconfigure_session()*, which form an interface to the BFD core for the rest and are expected to be called from the context of main thread.

Each BFD session has an associated BFD interface, represented by structure `bfd_iface`. A BFD interface contains a socket used for TX (the one for RX is shared in `bfd_proto`), an interface configuration and reference counter. Compared to interface structures of other protocols, these structures are not created and removed based on interface notification events, but according to the needs of BFD sessions. When a new session is created, it requests a proper BFD interface by function *bfd_get_iface()*, which either finds an existing one in `iface_list` (from `bfd_proto`) or allocates a new one. When a session is removed, an associated iface is discharged by *bfd_free_iface()*.

BFD requests are the external API for the other protocols. When a protocol wants a BFD session, it calls *bfd_request_session()*, which creates a structure `bfd_request` containing approprite information and an notify hook. This structure is a resource associated with the caller's resource pool. When a BFD protocol is available, a BFD request is submitted to the protocol, an appropriate BFD session is found or created and the request is attached to the session. When a session changes state, all attached requests (and related protocols) are notified. Note that BFD requests do not depend on BFD protocol running. When the BFD protocol is stopped or removed (or not available from beginning), related BFD requests are stored in *bfd_wait_list*, where waits for a new protocol.

BFD neighbors are just a way to statically configure BFD sessions without requests from other protocol. Structures `bfd_neighbor` are part of BFD configuration (like static routes in the static protocol). BFD neighbors are handled by BFD protocol like it is a BFD client – when a BFD neighbor is ready, the protocol just creates a BFD request like any other protocol.

The protocol uses a new generic event loop (structure `birdloop`) from `io.c`, which supports sockets, timers and events like the main loop. Timers (structure `timer2`) are new microsecond based timers, while sockets and events are the same. A birdloop is associated with a thread (field *thread*) in which event hooks are executed. Most functions for setting event sources (like *sk_start()* or *tm2_start()*) must be called from the context of that thread. Birdloop allows to temporarily acquire the context of that thread for the main thread by calling *birdloop_enter()* and then *birdloop_leave()*, which also ensures mutual exclusion with all event hooks. Note that resources associated with a birdloop (like timers) should be attached to the independent resource pool, detached from the main resource tree.

There are two kinds of interaction between the BFD core (running in the BFD thread) and the rest of BFD (running in the main thread). The first kind are configuration calls from main thread to the BFD thread (like *bfd_add_session()*). These calls are synchronous and use *birdloop_enter()* mechanism for mutual exclusion. The second kind is a notification about session changes from the BFD thread to the main thread. This is done in an asynchronous way, sesions with pending notifications are linked (in the BFD thread) to *notify_list* in `bfd_proto`, and then *bfd_notify_hook()* in the main thread is activated using *bfd_notify_kick()* and a pipe. The hook then processes scheduled sessions and calls hooks from associated BFD requests. This *notify_list* (and state fields in structure `bfd_session`) is protected by a spinlock in `bfd_proto` and functions *bfd_lock_sessions() / bfd_unlock_sessions()*.

There are few data races (accessing *p*->p.debug from *TRACE()* from the BFD thread and accessing some some private fields of `bfd_session` from *bfd_show_sessions()* from the main thread, but these are harmless (i hope).

TODO: document functions and access restrictions for fields in BFD structures.

Supported standards: - RFC 5880 - main BFD standard - RFC 5881 - BFD for IP links - RFC 5882 - generic application of BFD - RFC 5883 - BFD for multihop paths

5.3 Border Gateway Protocol

The BGP protocol is implemented in three parts: `bgp.c` which takes care of the connection and most of the interface with BIRD core, `packets.c` handling both incoming and outgoing BGP packets and `attrs.c` containing functions for manipulation with BGP attribute lists.

As opposed to the other existing routing daemons, BIRD has a sophisticated core architecture which is able to keep all the information needed by BGP in the primary routing table, therefore no complex data structures like a central BGP table are needed. This increases memory footprint of a BGP router with many connections, but not too much and, which is more important, it makes BGP much easier to implement.

Each instance of BGP (corresponding to a single BGP peer) is described by a `bgp_proto` structure to which are attached individual connections represented by `bgp_connection` (usually, there exists only one connection, but during BGP session setup, there can be more of them). The connections are handled according to the BGP state machine defined in the RFC with all the timers and all the parameters configurable.

In incoming direction, we listen on the connection's socket and each time we receive some input, we pass it to *bgp_rx()*. It decodes packet headers and the markers and passes complete packets to *bgp_rx_packet()* which distributes the packet according to its type.

In outgoing direction, we gather all the routing updates and sort them to buckets (`bgp_bucket`) according to their attributes (we keep a hash table for fast comparison of `rta`'s and a `fib` which helps us to find if we already have another route for the same destination queued for sending, so that we can replace it with the new one immediately instead of sending both updates). There also exists a special bucket holding all the route withdrawals which cannot be queued anywhere else as they don't have any attributes. If we have any packet to send (due to either new routes or the connection tracking code wanting to send a Open, Keepalive or Notification message), we call *bgp_schedule_packet()* which sets the corresponding bit in a *packet_to_send* bit field in `bgp_conn` and as soon as the transmit socket buffer becomes empty, we call *bgp_fire_tx()*. It inspects state of all the packet type bits and calls the corresponding *bgp_create_xx()* functions, eventually rescheduling the same packet type if we have more data of the same type to send.

The processing of attributes consists of two functions: *bgp_decode_attrs()* for checking of the attribute blocks and translating them to the language of BIRD's extended attributes and *bgp_encode_attrs()* which does the converse. Both functions are built around a *bgp_attr_table* array describing all important characteristics of all known attributes. Unknown transitive attributes are attached to the route as `EAF_TYPE_OPAQUE` byte streams.

BGP protocol implements graceful restart in both restarting (local restart) and receiving (neighbor restart) roles. The first is handled mostly by the graceful restart code in the nest, BGP protocol just handles capabilities, sets *gr_wait* and locks graceful restart until end-of-RIB mark is received. The second is implemented by internal restart of the BGP state to `BS_IDLE` and protocol state to `PS_START`, but keeping the protocol up from the core point of view and therefore maintaining received routes. Routing table refresh cycle (*rt_refresh_begin()*, *rt_refresh_end()*) is used for removing stale routes after reestablishment of BGP session during graceful restart.

Function

int *bgp_open* (struct bgp_proto * *p*) – open a BGP instance

Arguments

struct bgp_proto * *p*
BGP instance

Description

This function allocates and configures shared BGP resources. Should be called as the last step during initialization (when lock is acquired and neighbor is ready). When error, state changed to PS_DOWN, -1 is returned and caller should return immediately.

Function

void *bgp_close* (struct bgp_proto * *p*, int *apply_md5*) – close a BGP instance

Arguments

struct bgp_proto * *p*
BGP instance

int *apply_md5*
0 to disable unsetting MD5 auth

Description

This function frees and deconfigures shared BGP resources. *apply_md5* is set to 0 when bgp_close is called as a cleanup from failed *bgp_open()*.

Function

void *bgp_start_timer* (timer * *t*, int *value*) – start a BGP timer

Arguments

timer * *t*
timer

int *value*
time to fire (0 to disable the timer)

Description

This functions calls *tm_start()* on *t* with time *value* and the amount of randomization suggested by the BGP standard. Please use it for all BGP timers.

Function

void *bgp_close_conn* (struct bgp_conn * *conn*) – close a BGP connection

Arguments

struct bgp_conn * *conn*
connection to close

Description

This function takes a connection described by the `bgp_conn` structure, closes its socket and frees all resources associated with it.

Function

void *bgp_update_startup_delay* (struct bgp_proto * *p*) – update a startup delay

Arguments

struct bgp_proto * *p*
BGP instance

Description

This function updates a startup delay that is used to postpone next BGP connect. It also handles disable_after_error and might stop BGP instance when error happened and disable_after_error is on.
It should be called when BGP protocol error happened.

Function

void *bgp_handle_graceful_restart* (struct bgp_proto * *p*) – handle detected BGP graceful restart

Arguments

struct bgp_proto * *p*
BGP instance

Description

This function is called when a BGP graceful restart of the neighbor is detected (when the TCP connection fails or when a new TCP connection appears). The function activates processing of the restart - starts routing table refresh cycle and activates BGP restart timer. The protocol state goes back to `PS_START`, but changing BGP state back to `BS_IDLE` is left for the caller.

Function

void *bgp_graceful_restart_done* (struct bgp_proto * *p*) – finish active BGP graceful restart

Arguments

struct bgp_proto * *p*
BGP instance

Description

This function is called when the active BGP graceful restart of the neighbor should be finished - either successfully (the neighbor sends all paths and reports end-of-RIB on the new session) or unsuccessfully (the neighbor does not support BGP graceful restart on the new session). The function ends routing table refresh cycle and stops BGP restart timer.

Function

void *bgp_graceful_restart_timeout* (timer * *t*) – timeout of graceful restart 'restart timer'

Arguments

timer * *t*
  timer

Description

This function is a timeout hook for *gr_timer*, implementing BGP restart time limit for reestablisment of the BGP session after the graceful restart. When fired, we just proceed with the usual protocol restart.

Function

void *bgp_refresh_begin* (struct bgp_proto * *p*) – start incoming enhanced route refresh sequence

Arguments

struct bgp_proto * *p*
  BGP instance

Description

This function is called when an incoming enhanced route refresh sequence is started by the neighbor, demarcated by the BoRR packet. The function updates the load state and starts the routing table refresh cycle. Note that graceful restart also uses routing table refresh cycle, but RFC 7313 and load states ensure that these two sequences do not overlap.

Function

void *bgp_refresh_end* (struct bgp_proto * *p*) – finish incoming enhanced route refresh sequence

Arguments

struct bgp_proto * *p*
  BGP instance

Description

This function is called when an incoming enhanced route refresh sequence is finished by the neighbor, demarcated by the EoRR packet. The function updates the load state and ends the routing table refresh cycle. Routes not received during the sequence are removed by the nest.

Function

void *bgp_connect* (struct bgp_proto * *p*) – initiate an outgoing connection

Arguments

struct bgp_proto * *p*
  BGP instance

Description

The *bgp_connect()* function creates a new `bgp_conn` and initiates a TCP connection to the peer. The rest of connection setup is governed by the BGP state machine as described in the standard.

Function

struct bgp_proto * *bgp_find_proto* (sock * *sk*) – find existing proto for incoming connection

Arguments

sock * *sk*
  TCP socket

Function

int *bgp_incoming_connection* (sock * *sk*, uint dummy *UNUSED*) – handle an incoming connection

Arguments

sock * *sk*
: TCP socket

uint dummy *UNUSED*
: – undescribed –

Description

This function serves as a socket hook for accepting of new BGP connections. It searches a BGP instance corresponding to the peer which has connected and if such an instance exists, it creates a `bgp_conn` structure, attaches it to the instance and either sends an Open message or (if there already is an active connection) it closes the new connection by sending a Notification message.

Function

void *bgp_error* (struct bgp_conn * *c*, unsigned *code*, unsigned *subcode*, byte * *data*, int *len*) – report a protocol error

Arguments

struct bgp_conn * *c*
: connection

unsigned *code*
: error code (according to the RFC)

unsigned *subcode*
: error sub-code

byte * *data*
: data to be passed in the Notification message

int *len*
: length of the data

Description

bgp_error() sends a notification packet to tell the other side that a protocol error has occurred (including the data considered erroneous if possible) and closes the connection.

Function

void *bgp_store_error* (struct bgp_proto * *p*, struct bgp_conn * *c*, u8 *class*, u32 *code*) – store last error for status report

Arguments

struct bgp_proto * *p*
: BGP instance

struct bgp_conn * *c*
: connection

u8 *class*
: error class (BE_xxx constants)

u32 *code*
: error code (class specific)

Description

bgp_store_error() decides whether given error is interesting enough and store that error to last_error variables of *p*

Function

int *bgp_fire_tx* (struct bgp_conn * *conn*) – transmit packets

Arguments

struct bgp_conn * *conn*
connection

Description

Whenever the transmit buffers of the underlying TCP connection are free and we have any packets queued for sending, the socket functions call *bgp_fire_tx()* which takes care of selecting the highest priority packet queued (Notification > Keepalive > Open > Update), assembling its header and body and sending it to the connection.

Function

void *bgp_schedule_packet* (struct bgp_conn * *conn*, int *type*) – schedule a packet for transmission

Arguments

struct bgp_conn * *conn*
connection

int *type*
packet type

Description

Schedule a packet of type *type* to be sent as soon as possible.

Function

const char * *bgp_error_dsc* (unsigned *code*, unsigned *subcode*) – return BGP error description

Arguments

unsigned *code*
BGP error code

unsigned *subcode*
BGP error subcode

Description

bgp_error_dsc() returns error description for BGP errors which might be static string or given temporary buffer.

Function

void *bgp_rx_packet* (struct bgp_conn * *conn*, byte * *pkt*, unsigned *len*) – handle a received packet

Arguments

struct bgp_conn * *conn*
BGP connection

byte * *pkt*
start of the packet

unsigned *len*
packet size

Description

bgp_rx_packet() takes a newly received packet and calls the corresponding packet handler according to the packet type.

Function

int *bgp_rx* (sock * *sk*, uint *size*) – handle received data

Arguments

sock * *sk*
socket

uint *size*
amount of data received

Description

bgp_rx() is called by the socket layer whenever new data arrive from the underlying TCP connection. It assembles the data fragments to packets, checks their headers and framing and passes complete packets to *bgp_rx_packet()*.

Function

uint *bgp_encode_attrs* (struct bgp_proto * *p*, byte * *w*, ea_list * *attrs*, int *remains*) – encode BGP attributes

Arguments

struct bgp_proto * *p*
BGP instance (or NULL)

byte * *w*
buffer

ea_list * *attrs*
a list of extended attributes

int *remains*
remaining space in the buffer

Description

The *bgp_encode_attrs()* function takes a list of extended attributes and converts it to its BGP representation (a part of an Update message).

Result

Length of the attribute block generated or -1 if not enough space.

Function

struct rta * *bgp_decode_attrs* (struct bgp_conn * *conn*, byte * *attr*, uint *len*, struct linpool * *pool*, int *mandatory*) – check and decode BGP attributes

Arguments

struct bgp_conn * *conn*
connection

byte * *attr*
start of attribute block

uint *len*
length of attribute block

struct linpool * *pool*
linear pool to make all the allocations in

int *mandatory*
1 iff presence of mandatory attributes has to be checked

Description

This function takes a BGP attribute block (a part of an Update message), checks its consistency and converts it to a list of BIRD route attributes represented by a `rta`.

5.4 Multi-Threaded Routing Toolkit (MRT) protocol

The MRT protocol is implemented in just one file: `mrt.c`. It contains of several parts: Generic functions for preparing MRT messages in a buffer, functions for MRT table dump (called from timer or CLI), functions for MRT BGP4MP dump (called from BGP), and the usual protocol glue. For the MRT table dump, the key structure is struct mrt_table_dump_state, which contains all necessary data and created when the MRT dump cycle is started for the duration of the MRT dump. The MBGP4MP dump is currently not bound to MRT protocol instance and uses the config->mrtdump_file fd.

The protocol is simple, just periodically scans routing table and export it to a file. It does not use the regular update mechanism, but a direct access in order to handle iteration through multiple routing tables. The table dump needs to dump all peers first and then use indexes to address the peers, we use a hash table (*peer_hash*) to find peer index based on BGP protocol key attributes.

One thing worth documenting is the locking. During processing, the currently processed table (*table* field in the state structure) is locked and also the explicitly named table is locked (*table_ptr* field in the state structure) if specified. Between dumps no table is locked. Also the current config is locked (by *config_add_obstacle()*) during table dumps as some data (strings, filters) are shared from the config and the running table dump may be interrupted by reconfiguration.

Supported standards: - RFC 6396 - MRT format standard - RFC 8050 - ADD_PATH extension

5.5 Open Shortest Path First (OSPF)

The OSPF protocol is quite complicated and its complex implemenation is split to many files. In `ospf.c`, you will find mainly the interface for communication with the core (e.g., reconfiguration hooks, shutdown and initialisation and so on). File `iface.c` contains the interface state machine and functions for allocation and deallocation of OSPF's interface data structures. Source `neighbor.c` includes the neighbor state machine and functions for election of Designated Router and Backup Designated router. In `packet.c`, you will find various functions for sending and receiving generic OSPF packets. There are also routines for authentication and checksumming. In `hello.c`, there are routines for sending and receiving of hello packets as well as functions for maintaining wait times and the inactivity timer. Files `lsreq.c`, `lsack.c`, `dbdes.c` contain functions for sending and receiving of link-state requests, link-state acknowledgements and database descriptions respectively. In `lsupd.c`, there are functions for sending and receiving of link-state updates and also the flooding algorithm. Source `topology.c` is a place where routines for searching LSAs in the link-state database, adding and deleting them reside, there also are functions for originating of various types of LSAs (router LSA, net LSA, external LSA). File `rt.c` contains routines for calculating the routing table. `lsalib.c` is a set of various functions for working with the LSAs (endianity conversions, calculation of checksum etc.).

One instance of the protocol is able to hold LSA databases for multiple OSPF areas, to exchange routing information between multiple neighbors and to calculate the routing tables. The core structure is `ospf_proto` to which multiple `ospf_area` and `ospf_iface` structures are connected. `ospf_proto` is also connected to `top_hash_graph` which is a dynamic hashing structure that describes the link-state database. It allows fast search, addition and deletion. Each LSA is kept in two pieces: header and body. Both of them are kept in the endianity of the CPU.

In OSPFv2 specification, it is implied that there is one IP prefix for each physical network/interface (unless it is an ptp link). But in modern systems, there might be more independent IP prefixes associated with an interface. To handle this situation, we have one `ospf_iface` for each active IP prefix (instead for each active iface); This behaves like virtual interface for the purpose of OSPF. If we receive packet, we associate it with a proper virtual interface mainly according to its source address.

OSPF keeps one socket per `ospf_iface`. This allows us (compared to one socket approach) to evade problems with a limit of multicast groups per socket and with sending multicast packets to appropriate interface in a portable way. The socket is associated with underlying physical iface and should not receive packets received on other ifaces (unfortunately, this is not true on BSD). Generally, one packet can be received by more sockets (for example, if there are more `ospf_iface` on one physical iface), therefore we explicitly filter received packets according to src/dst IP address and received iface.

Vlinks are implemented using particularly degenerate form of `ospf_iface`, which has several exceptions: it does not have its iface or socket (it copies these from 'parent' `ospf_iface`) and it is present in iface list even when down (it is not freed in *ospf_iface_down()*).

The heart beat of ospf is *ospf_disp()*. It is called at regular intervals (`ospf_proto`->tick). It is responsible for aging and flushing of LSAs in the database, updating topology information in LSAs and for routing table calculation.

To every `ospf_iface`, we connect one or more `ospf_neighbor`'s – a structure containing many timers and queues for building adjacency and for exchange of routing messages.

BIRD's OSPF implementation respects RFC2328 in every detail, but some of internal algorithms do differ. The RFC recommends making a snapshot of the link-state database when a new adjacency is forming and sending the database description packets based on the information in this snapshot. The database can be quite large in some networks, so rather we walk through a `slist` structure which allows us to continue even if the actual LSA we were working with is deleted. New LSAs are added at the tail of this `slist`.

We also do not keep a separate OSPF routing table, because the core helps us by being able to recognize when a route is updated to an identical one and it suppresses the update automatically. Due to this, we can flush all the routes we have recalculated and also those we have deleted to the core's routing table and the core will take care of the rest. This simplifies the process and conserves memory.

Supported standards: - RFC 2328 - main OSPFv2 standard - RFC 5340 - main OSPFv3 standard - RFC 3101 - OSPFv2 NSSA areas - RFC 6549 - OSPFv2 multi-instance extensions - RFC 6987 - OSPF stub router advertisement

---

Function

void *ospf_disp* (timer * *timer*) – invokes routing table calculation, aging and also *area_disp()*

Arguments

timer * *timer*
: timer usually called every *ospf_proto*->tick second, *timer*->data point to *ospf_proto*

Function

int *ospf_import_control* (struct proto * *P*, rte ** *new*, ea_list ** *attrs*, struct linpool * *pool*) – accept or reject new route from nest's routing table

Arguments

struct proto * *P*
: OSPF protocol instance

rte ** *new*
: the new route

ea_list ** *attrs*
: list of attributes

struct linpool * *pool*
: pool for allocation of attributes

Description

Its quite simple. It does not accept our own routes and leaves the decision on import to the filters.

Function

int *ospf_shutdown* (struct proto * *P*) – Finish of OSPF instance

Arguments

struct proto * *P*
: OSPF protocol instance

Description

RFC does not define any action that should be taken before router shutdown. To make my neighbors react as fast as possible, I send them hello packet with empty neighbor list. They should start their neighbor state machine with event `NEIGHBOR_1WAY`.

Function

int *ospf_reconfigure* (struct proto * *P*, struct proto_config * *c*) – reconfiguration hook

Arguments

struct proto * *P*
current instance of protocol (with old configuration)

struct proto_config * *c*
new configuration requested by user

Description

This hook tries to be a little bit intelligent. Instance of OSPF will survive change of many constants like hello interval, password change, addition or deletion of some neighbor on nonbroadcast network, cost of interface, etc.

Function

struct top_hash_entry * *ospf_install_lsa* (struct ospf_proto * *p*, struct ospf_lsa_header * *lsa*, u32 *type*, u32 *domain*, void * *body*) – install new LSA into database

Arguments

struct ospf_proto * *p*
OSPF protocol instance

struct ospf_lsa_header * *lsa*
LSA header

u32 *type*
type of LSA

u32 *domain*
domain of LSA

void * *body*
pointer to LSA body

Description

This function ensures installing new LSA received in LS update into LSA database. Old instance is replaced. Several actions are taken to detect if new routing table calculation is necessary. This is described in 13.2 of RFC 2328. This function is for received LSA only, locally originated LSAs are installed by *ospf_originate_lsa()*.
The LSA body in *body* is expected to be mb_allocated by the caller and its ownership is transferred to the LSA entry structure.

Function

void *ospf_advance_lsa* (struct ospf_proto * *p*, struct top_hash_entry * *en*, struct ospf_lsa_header * *lsa*, u32 *type*, u32 *domain*, void * *body*) – handle received unexpected self-originated LSA

Arguments

struct ospf_proto * *p*
OSPF protocol instance

struct top_hash_entry * *en*
current LSA entry or NULL

struct ospf_lsa_header * *lsa*
new LSA header

u32 *type*
type of LSA

u32 *domain*
domain of LSA

void * *body*
pointer to LSA body

Description

This function handles received unexpected self-originated LSA (*lsa*, *body*) by either advancing sequence number of the local LSA instance (*en*) and propagating it, or installing the received LSA and immediately flushing it (if there is no local LSA; i.e., *en* is NULL or MaxAge).
The LSA body in *body* is expected to be mb_allocated by the caller and its ownership is transferred to the LSA entry structure or it is freed.

Function

struct top_hash_entry * *ospf_originate_lsa* (struct ospf_proto * *p*, struct ospf_new_lsa * *lsa*) – originate new LSA

Arguments

struct ospf_proto * *p*
OSPF protocol instance

struct ospf_new_lsa * *lsa*
New LSA specification

Description

This function prepares a new LSA, installs it into the LSA database and floods it. If the new LSA cannot be originated now (because the old instance was originated within MinLSInterval, or because the LSA seqnum is currently wrapping), the origination is instead scheduled for later. If the new LSA is equivalent to the current LSA, the origination is skipped. In all cases, the corresponding LSA entry is returned. The new LSA is based on the LSA specification (*lsa*) and the LSA body from lsab buffer of *p*, which is emptied after the call. The opposite of this function is *ospf_flush_lsa()*.

Function

void *ospf_flush_lsa* (struct ospf_proto * *p*, struct top_hash_entry * *en*) – flush LSA from OSPF domain

Arguments

struct ospf_proto * *p*
OSPF protocol instance

struct top_hash_entry * *en*
LSA entry to flush

Description

This function flushes *en* from the OSPF domain by setting its age to `LSA_MAXAGE` and flooding it. That also triggers subsequent events in LSA lifecycle leading to removal of the LSA from the LSA database (e.g. the LSA content is freed when flushing is acknowledged by neighbors). The function does nothing if the LSA is already being flushed. LSA entries are not immediately removed when being flushed, the caller may assume that *en* still exists after the call. The function is the opposite of *ospf_originate_lsa()* and is supposed to do the right thing even in cases of postponed origination.

Function

void *ospf_update_lsadb* (struct ospf_proto * *p*) – update LSA database

Arguments

struct ospf_proto * *p*
OSPF protocol instance

Description

This function is periodicaly invoked from *ospf_disp()*. It does some periodic or postponed processing related to LSA entries. It originates postponed LSAs scheduled by *ospf_originate_lsa()*, It continues in flushing processes started by *ospf_flush_lsa()*. It also periodically refreshs locally originated LSAs – when the current instance is older `LSREFRESHTIME`, a new instance is originated. Finally, it also ages stored LSAs and flushes ones that reached `LSA_MAXAGE`.
The RFC 2328 says that a router should periodically check checksums of all stored LSAs to detect hardware problems. This is not implemented.

Function

void *ospf_originate_ext_lsa* (struct ospf_proto * *p*, struct ospf_area * *oa*, ort * *nf*, u8 *mode*, u32 *metric*, u32 *ebit*, ip_addr *fwaddr*, u32 *tag*, int *pbit*) – new route received from nest and filters

Arguments

struct ospf_proto * *p*
OSPF protocol instance

struct ospf_area * *oa*
ospf_area for which LSA is originated

ort * *nf*
network prefix and mask

u8 *mode*
the mode of the LSA (LSA_M_EXPORT or LSA_M_RTCALC)

u32 *metric*
the metric of a route

u32 *ebit*
E-bit for route metric (bool)

ip_addr *fwaddr*
the forwarding address

u32 *tag*
the route tag

int *pbit*
P-bit for NSSA LSAs (bool), ignored for external LSAs

Description

If I receive a message that new route is installed, I try to originate an external LSA. If *oa* is an NSSA area, NSSA-LSA is originated instead. *oa* should not be a stub area. *src* does not specify whether the LSA is external or NSSA, but it specifies the source of origination - the export from *ospf_rt_notify()*, or the NSSA-EXT translation.

Function

struct top_graph * *ospf_top_new* (struct ospf_proto *p UNUSED4 *UNUSED6*, pool * *pool*) – allocated new topology database

Arguments

struct ospf_proto *p UNUSED4 *UNUSED6*
 – undescribed –

pool * *pool*
 pool for allocation

Description
This dynamically hashed structure is used for keeping LSAs. Mainly it is used for the LSA database of the OSPF protocol, but also for LSA retransmission and request lists of OSPF neighbors.

Function
void *ospf_neigh_chstate* (struct ospf_neighbor * *n*, u8 *state*) – handles changes related to new or lod state of neighbor

Arguments

struct ospf_neighbor * *n*
 OSPF neighbor

u8 *state*
 new state

Description
Many actions have to be taken acording to a change of state of a neighbor. It starts rxmt timers, call interface state machine etc.

Function
void *ospf_neigh_sm* (struct ospf_neighbor * *n*, int *event*) – ospf neighbor state machine

Arguments

struct ospf_neighbor * *n*
 neighor

int *event*
 actual event

Description
This part implements the neighbor state machine as described in 10.3 of RFC 2328. The only difference is that state `NEIGHBOR_ATTEMPT` is not used. We discover neighbors on nonbroadcast networks in the same way as on broadcast networks. The only difference is in sending hello packets. These are sent to IPs listed in *ospf_iface*->nbma_list .

Function
void *ospf_dr_election* (struct ospf_iface * *ifa*) – (Backup) Designed Router election

Arguments

struct ospf_iface * *ifa*
 actual interface

Description
When the wait timer fires, it is time to elect (Backup) Designated Router. Structure describing me is added to this list so every electing router has the same list. Backup Designated Router is elected before Designated Router. This process is described in 9.4 of RFC 2328. The function is supposed to be called only from *ospf_iface_sm()* as a part of the interface state machine.

Function

void *ospf_iface_chstate* (struct ospf_iface * *ifa*, u8 *state*) – handle changes of interface state

Arguments

struct ospf_iface * *ifa*
OSPF interface

u8 *state*
new state

Description

Many actions must be taken according to interface state changes. New network LSAs must be originated, flushed, new multicast sockets to listen for messages for `ALLDROUTERS` have to be opened, etc.

Function

void *ospf_iface_sm* (struct ospf_iface * *ifa*, int *event*) – OSPF interface state machine

Arguments

struct ospf_iface * *ifa*
OSPF interface

int *event*
event comming to state machine

Description

This fully respects 9.3 of RFC 2328 except we have slightly different handling of `DOWN` and `LOOP` state. We remove intefaces that are `DOWN`. `DOWN` state is used when an interface is waiting for a lock. `LOOP` state is used when an interface does not have a link.

Function

int *ospf_rx_hook* (sock * *sk*, uint *len*)

Arguments

sock * *sk*
socket we received the packet.

uint *len*
size of the packet

Description

This is the entry point for messages from neighbors. Many checks (like authentication, checksums, size) are done before the packet is passed to non generic functions.

Function

int *lsa_validate* (struct ospf_lsa_header * *lsa*, u32 *lsa_type*, int *ospf2*, void * *body*) – check whether given LSA is valid

Arguments

struct ospf_lsa_header * *lsa*
LSA header

u32 *lsa_type*
one of `LSA_T_xxx`

int *ospf2*
`true` means OSPF version 2, `false` means OSPF version 3

void * *body*
pointer to LSA body

Description

Checks internal structure of given LSA body (minimal length, consistency). Returns true if valid.

Function

void *ospf_send_dbdes* (struct ospf_proto * *p*, struct ospf_neighbor * *n*) – transmit database description packet

Arguments

struct ospf_proto * *p*
OSPF protocol instance

struct ospf_neighbor * *n*
neighbor

Description

Sending of a database description packet is described in 10.8 of RFC 2328. Reception of each packet is acknowledged in the sequence number of another. When I send a packet to a neighbor I keep a copy in a buffer. If the neighbor does not reply, I don't create a new packet but just send the content of the buffer.

Function

void *ospf_rt_spf* (struct ospf_proto * *p*) – calculate internal routes

Arguments

struct ospf_proto * *p*
OSPF protocol instance

Description

Calculation of internal paths in an area is described in 16.1 of RFC 2328. It's based on Dijkstra's shortest path tree algorithms. This function is invoked from *ospf_disp()*.

5.6 Pipe

The Pipe protocol is very simple. It just connects to two routing tables using *proto_add_announce_hook()* and whenever it receives a *rt_notify()* about a change in one of the tables, it converts it to a *rte_update()* in the other one.

To avoid pipe loops, Pipe keeps a 'being updated' flag in each routing table.

A pipe has two announce hooks, the first connected to the main table, the second connected to the peer table. When a new route is announced on the main table, it gets checked by an export filter in ahook 1, and, after that, it is announced to the peer table via *rte_update()*, an import filter in ahook 2 is called. When a new route is announced in the peer table, an export filter in ahook2 and an import filter in ahook 1 are used. Oviously, there is no need in filtering the same route twice, so both import filters are set to accept, while user configured 'import' and 'export' filters are used as export filters in ahooks 2 and 1. Route limits are handled similarly, but on the import side of ahooks.

5.7 Routing Information Protocol (RIP)

The RIP protocol is implemented in two files: `rip.c` containing the protocol logic, route management and the protocol glue with BIRD core, and `packets.c` handling RIP packet processing, RX, TX and protocol sockets.

Each instance of RIP is described by a structure `rip_proto`, which contains an internal RIP routing table, a list of protocol interfaces and the main timer responsible for RIP routing table cleanup.

RIP internal routing table contains incoming and outgoing routes. For each network (represented by structure `rip_entry`) there is one outgoing route stored directly in `rip_entry` and an one-way linked list of incoming routes (structures `rip_rte`). The list contains incoming routes from different RIP neighbors, but only routes with the lowest metric are stored (i.e., all stored incoming routes have the same metric).

Note that RIP itself does not select outgoing route, that is done by the core routing table. When a new incoming route is received, it is propagated to the RIP table by *rip_update_rte()* and possibly stored in the list of incoming routes. Then the change may be propagated to the core by *rip_announce_rte()*. The core selects the best route and propagate it to RIP by *rip_rt_notify()*, which updates outgoing route part of `rip_entry` and possibly triggers route propagation by *rip_trigger_update()*.

RIP interfaces are represented by structures `rip_iface`. A RIP interface contains a per-interface socket, a list of associated neighbors, interface configuration, and state information related to scheduled interface events and running update sessions. RIP interfaces are added and removed based on core interface notifications.

There are two RIP interface events - regular updates and triggered updates. Both are managed from the RIP interface timer (*rip_iface_timer()*). Regular updates are called at fixed interval and propagate the whole routing table, while triggered updates are scheduled by *rip_trigger_update()* due to some routing table change and propagate only the routes modified since the time they were scheduled. There are also unicast-destined requested updates, but these are sent directly as a reaction to received RIP request message. The update session is started by *rip_send_table()*. There may be at most one active update session per interface, as the associated state (including the fib iterator) is stored directly in `rip_iface` structure.

RIP neighbors are represented by structures `rip_neighbor`. Compared to neighbor handling in other routing protocols, RIP does not have explicit neighbor discovery and adjacency maintenance, which makes the `rip_neighbor` related code a bit peculiar. RIP neighbors are interlinked with core neighbor structures (`neighbor`) and use core neighbor notifications to ensure that RIP neighbors are timely removed. RIP neighbors are added based on received route notifications and removed based on core neighbor and RIP interface events.

RIP neighbors are linked by RIP routes and use counter to track the number of associated routes, but when these RIP routes timeout, associated RIP neighbor is still alive (with zero counter). When RIP neighbor is removed but still has some associated routes, it is not freed, just changed to detached state (core neighbors and RIP ifaces are unlinked), then during the main timer cleanup phase the associated routes are removed and the `rip_neighbor` structure is finally freed.

Supported standards: - RFC 1058 - RIPv1 - RFC 2453 - RIPv2 - RFC 2080 - RIPng - RFC 4822 - RIP cryptographic authentication

---

Function

void *rip_announce_rte* (struct rip_proto * *p*, struct rip_entry * *en*) – announce route from RIP routing table to the core

Arguments

struct rip_proto * *p*
  RIP instance

struct rip_entry * *en*
  related network

Description

The function takes a list of incoming routes from *en*, prepare appropriate `rte` for the core and propagate it by *rte_update()*.

Function

void *rip_update_rte* (struct rip_proto * *p*, ip_addr * *prefix*, int *pxlen*, struct rip_rte * *new*) – enter a route update to RIP routing table

Arguments

struct rip_proto * *p*
  RIP instance

ip_addr * *prefix*
  network prefix

int *pxlen*
  network prefix length

struct rip_rte * *new*
 a `rip_rte` representing the new route

Description
The function is called by the RIP packet processing code whenever it receives a reachable route. The appropriate routing table entry is found and the list of incoming routes is updated. Eventually, the change is also propagated to the core by *rip_announce_rte()*. Note that for unreachable routes, *rip_withdraw_rte()* should be called instead of *rip_update_rte()*.

Function
void *rip_withdraw_rte* (struct rip_proto * *p*, ip_addr * *prefix*, int *pxlen*, struct rip_neighbor * *from*) – enter a route withdraw to RIP routing table

Arguments

struct rip_proto * *p*
 RIP instance

ip_addr * *prefix*
 network prefix

int *pxlen*
 network prefix length

struct rip_neighbor * *from*
 a `rip_neighbor` propagating the withdraw

Description
The function is called by the RIP packet processing code whenever it receives an unreachable route. The incoming route for given network from nbr *from* is removed. Eventually, the change is also propagated by *rip_announce_rte()*.

Function
void *rip_timer* (timer * *t*) – RIP main timer hook

Arguments

timer * *t*
 timer

Description
The RIP main timer is responsible for routing table maintenance. Invalid or expired routes (`rip_rte`) are removed and garbage collection of stale routing table entries (`rip_entry`) is done. Changes are propagated to core tables, route reload is also done here. Note that garbage collection uses a maximal GC time, while interfaces maintain an illusion of per-interface GC times in *rip_send_response()*.
Keeping incoming routes and the selected outgoing route are two independent functions, therefore after garbage collection some entries now considered invalid (RIP_ENTRY_DUMMY) still may have non-empty list of incoming routes, while some valid entries (representing an outgoing route) may have that list empty.
The main timer is not scheduled periodically but it uses the time of the current next event and the minimal interval of any possible event to compute the time of the next run.

Function
void *rip_iface_timer* (timer * *t*) – RIP interface timer hook

Arguments

timer * *t*
 timer

Description
RIP interface timers are responsible for scheduling both regular and triggered updates. Fixed, delay-independent period is used for regular updates, while minimal separating interval is enforced for triggered updates. The function also ensures that a new update is not started when the old one is still running.

Function

void *rip_send_table* (struct rip_proto * *p*, struct rip_iface * *ifa*, ip_addr *addr*, bird_clock_t *changed*) – RIP interface timer hook

Arguments

struct rip_proto * *p*
    RIP instance

struct rip_iface * *ifa*
    RIP interface

ip_addr *addr*
    destination IP address

bird_clock_t *changed*
    time limit for triggered updates

Description

The function activates an update session and starts sending routing update packets (using *rip_send_response()*). The session may be finished during the call or may continue in *rip_tx_hook()* until all appropriate routes are transmitted. Note that there may be at most one active update session per interface, the function will terminate the old active session before activating the new one.

5.8 Router Advertisements

The RAdv protocol is implemented in two files: `radv.c` containing the interface with BIRD core and the protocol logic and `packets.c` handling low level protocol stuff (RX, TX and packet formats). The protocol does not export any routes.

The RAdv is structured in the usual way - for each handled interface there is a structure `radv_iface` that contains a state related to that interface together with its resources (a socket, a timer). There is also a prepared RA stored in a TX buffer of the socket associated with an iface. These iface structures are created and removed according to iface events from BIRD core handled by *radv_if_notify()* callback.

The main logic of RAdv consists of two functions: *radv_iface_notify()*, which processes asynchronous events (specified by RA_EV_* codes), and *radv_timer()*, which triggers sending RAs and computes the next timeout.

The RAdv protocol could receive routes (through *radv_import_control()* and *radv_rt_notify()*), but only the configured trigger route is tracked (in `active` var). When a radv protocol is reconfigured, the connected routing table is examined (in *radv_check_active()*) to have proper `active` value in case of the specified trigger prefix was changed.

Supported standards: - RFC 4861 - main RA standard - RFC 4191 - Default Router Preferences and More-Specific Routes - RFC 6106 - DNS extensions (RDDNS, DNSSL)

5.9 Static

The Static protocol is implemented in a straightforward way. It keeps two lists of static routes: one containing interface routes and one holding the remaining ones. Interface routes are inserted and removed according to interface events received from the core via the *if_notify()* hook. Routes pointing to a neighboring router use a sticky node in the neighbor cache to be notified about gaining or losing the neighbor. Special routes like black holes or rejects are inserted all the time.

Multipath routes are tricky. Because these routes depends on several neighbors we need to integrate that to the neighbor notification handling, we use dummy static_route nodes, one for each nexthop. Therefore, a multipath route consists of a master static_route node (of dest RTD_MULTIPATH), which specifies prefix and is used in most circumstances, and a list of dummy static_route nodes (of dest RTD_NONE), which stores info about nexthops and are connected to neighbor entries and neighbor notifications. Dummy nodes are chained using mp_next, they aren't in other_routes list, and abuse some fields (masklen, if_name) for other purposes.

The only other thing worth mentioning is that when asked for reconfiguration, Static not only compares the two configurations, but it also calculates difference between the lists of static routes and it just inserts the newly added routes and removes the obsolete ones.

5.10 Direct

The Direct protocol works by converting all *ifa_notify()* events it receives to *rte_update()* calls for the corresponding network.

Chapter 6: System dependent parts

6.1 Introduction

We've tried to make BIRD as portable as possible, but unfortunately communication with the network stack differs from one OS to another, so we need at least some OS specific code. The good news is that this code is isolated in a small set of modules:

`config.h`
is a header file with configuration information, definition of the standard set of types and so on.

Startup module
controls BIRD startup. Common for a family of OS's (e.g., for all Unices).

Logging module
manages the system logs. [per OS family]

IO module
gives an implementation of sockets, timers and the global event queue. [per OS family]

KRT module
implements the Kernel and Device protocols. This is the most arcane part of the system dependent stuff and some functions differ even between various releases of a single OS.

6.2 Logging

The Logging module offers a simple set of functions for writing messages to system logs and to the debug output. Message classes used by this module are described in `birdlib.h` and also in the user's manual.

Function

void *log_commit* (int *class*, buffer * *buf*) – commit a log message

Arguments

int *class*
message class information (`L_DEBUG` to `L_BUG`, see `lib/birdlib.h`)

buffer * *buf*
message to write

Description

This function writes a message prepared in the log buffer to the log file (as specified in the configuration). The log buffer is reset after that. The log message is a full line, *log_commit()* terminates it.
The message class is an integer, not a first char of a string like in *log()*, so it should be written like *L_INFO.

Function

void *log_msg* (const char * *msg*, ... *...*) – log a message

Arguments

const char * *msg*
printf-like formatting string with message class information prepended (`L_DEBUG` to `L_BUG`, see `lib/birdlib.h`)

... *...*
variable arguments

Description

This function formats a message according to the format string *msg* and writes it to the corresponding log file (as specified in the configuration). Please note that the message is automatically formatted as a full line, no need to include `\n` inside. It is essentially a sequence of *log_reset()*, *logn()* and *log_commit()*.

Function

void *bug* (const char * *msg*, ...) – report an internal error

Arguments

const char * *msg*
a printf-like error message

... ...
variable arguments

Description
This function logs an internal error and aborts execution of the program.

Function

void *die* (const char * *msg*, ...) – report a fatal error

Arguments

const char * *msg*
a printf-like error message

... ...
variable arguments

Description
This function logs a fatal error and aborts execution of the program.

Function

void *debug* (const char * *msg*, ...) – write to debug output

Arguments

const char * *msg*
a printf-like message

... ...
variable arguments

Description
This function formats the message *msg* and prints it out to the debugging output. No newline character is appended.

6.3 Kernel synchronization

This system dependent module implements the Kernel and Device protocol, that is synchronization of interface lists and routing tables with the OS kernel.

The whole kernel synchronization is a bit messy and touches some internals of the routing table engine, because routing table maintenance is a typical example of the proverbial compatibility between different Unices and we want to keep the overhead of our KRT business as low as possible and avoid maintaining a local routing table copy.

The kernel syncer can work in three different modes (according to system config header): Either with a single routing table and single KRT protocol [traditional UNIX] or with many routing tables and separate KRT protocols for all of them or with many routing tables, but every scan including all tables, so we start separate KRT protocols which cooperate with each other [Linux]. In this case, we keep only a single scan timer.

We use FIB node flags in the routing table to keep track of route synchronization status. We also attach temporary `rte`'s to the routing table, but it cannot do any harm to the rest of BIRD since table synchronization is an atomic process.

When starting up, we cheat by looking if there is another KRT instance to be initialized later and performing table scan only once for all the instances.

The code uses OS-dependent parts for kernel updates and scans. These parts are in more specific sysdep directories (e.g. sysdep/linux) in functions krt_sys_* and kif_sys_* (and some others like *krt_replace_rte()*) and krt-sys.h header file. This is also used for platform specific protocol options and route attributes.

There was also an old code that used traditional UNIX ioctls for these tasks. It was unmaintained and later removed. For reference, see sysdep/krt-* files in commit 396dfa9042305f62da1f56589c4b98fac57fc2f6

Chapter 7: Library functions

7.1 IP addresses

BIRD uses its own abstraction of IP address in order to share the same code for both IPv4 and IPv6. IP addresses are represented as entities of type `ip_addr` which are never to be treated as numbers and instead they must be manipulated using the following functions and macros.

Function

char * *ip_scope_text* (uint *scope*) – get textual representation of address scope

Arguments

uint *scope*
scope (`SCOPE_xxx`)

Description
Returns a pointer to a textual name of the scope given.

Function

int *ipa_equal* (ip_addr x, ip_addr y) – compare two IP addresses for equality

Arguments

ip_addr x
IP address

ip_addr y
IP address

Description
ipa_equal() returns 1 if x and y represent the same IP address, else 0.

Function

int *ipa_nonzero* (ip_addr x) – test if an IP address is defined

Arguments

ip_addr x
IP address

Description
ipa_nonzero returns 1 if x is a defined IP address (not all bits are zero), else 0.
The undefined all-zero address is reachable as a `IPA_NONE` macro.

Function

ip_addr *ipa_and* (ip_addr x, ip_addr y) – compute bitwise and of two IP addresses

Arguments

ip_addr x
IP address

ip_addr y
IP address

Description
This function returns a bitwise and of x and y. It's primarily used for network masking.

Function

ip_addr *ipa_or* (ip_addr x, ip_addr y) – compute bitwise or of two IP addresses

Arguments

ip_addr x
IP address

ip_addr y
IP address

Description

This function returns a bitwise or of x and y.

Function

ip_addr *ipa_xor* (ip_addr x, ip_addr y) – compute bitwise xor of two IP addresses

Arguments

ip_addr x
IP address

ip_addr y
IP address

Description

This function returns a bitwise xor of x and y.

Function

ip_addr *ipa_not* (ip_addr x) – compute bitwise negation of two IP addresses

Arguments

ip_addr x
IP address

Description

This function returns a bitwise negation of x.

Function

ip_addr *ipa_mkmask* (int x) – create a netmask

Arguments

int x
prefix length

Description

This function returns an `ip_addr` corresponding of a netmask of an address prefix of size x.

Function

int *ipa_masklen* (ip_addr x) – calculate netmask length

Arguments

ip_addr x
IP address

Description

This function checks whether x represents a valid netmask and returns the size of the associate network prefix or -1 for invalid mask.

Function

int *ipa_hash* (ip_addr x) – hash IP addresses

Arguments

ip_addr x
: IP address

Description

ipa_hash() returns a 16-bit hash value of the IP address x.

Function

void *ipa_hton* (ip_addr x) – convert IP address to network order

Arguments

ip_addr x
: IP address

Description

Converts the IP address x to the network byte order.
Beware, this is a macro and it alters the argument!

Function

void *ipa_ntoh* (ip_addr x) – convert IP address to host order

Arguments

ip_addr x
: IP address

Description

Converts the IP address x from the network byte order.
Beware, this is a macro and it alters the argument!

Function

int *ipa_classify* (ip_addr x) – classify an IP address

Arguments

ip_addr x
: IP address

Description

ipa_classify() returns an address class of x, that is a bitwise or of address type (`IADDR_INVALID`, `IADDR_HOST`, `IADDR_BROADCAST`, `IADDR_MULTICAST`) with address scope (`SCOPE_HOST` to `SCOPE_UNIVERSE`) or -1 (`IADDR_INVALID`) for an invalid address.

Function

ip4_addr *ip4_class_mask* (ip4_addr x) – guess netmask according to address class

Arguments

ip4_addr x
: IPv4 address

Description

This function (available in IPv4 version only) returns a network mask according to the address class of x. Although classful addressing is nowadays obsolete, there still live routing protocols transferring no prefix lengths nor netmasks and this function could be useful to them.

Function

u32 *ipa_from_u32* (ip_addr *x*) – convert IPv4 address to an integer

Arguments

ip_addr *x*
IP address

Description
This function takes an IPv4 address and returns its numeric representation.

Function

ip_addr *ipa_to_u32* (u32 *x*) – convert integer to IPv4 address

Arguments

u32 *x*
a 32-bit integer

Description
ipa_to_u32() takes a numeric representation of an IPv4 address and converts it to the corresponding `ip_addr`.

Function

int *ipa_compare* (ip_addr *x*, ip_addr *y*) – compare two IP addresses for order

Arguments

ip_addr *x*
IP address

ip_addr *y*
IP address

Description
The *ipa_compare()* function takes two IP addresses and returns -1 if x is less than y in canonical ordering (lexicographical order of the bit strings), 1 if x is greater than y and 0 if they are the same.

Function

ip_addr *ipa_build6* (u32 *a1*, u32 *a2*, u32 *a3*, u32 *a4*) – build an IPv6 address from parts

Arguments

u32 *a1*
part #1

u32 *a2*
part #2

u32 *a3*
part #3

u32 *a4*
part #4

Description
ipa_build() takes *a1* to *a4* and assembles them to a single IPv6 address. It's used for example when a protocol wants to bind its socket to a hard-wired multicast address.

Function

char * *ip_ntop* (ip_addr *a*, char * *buf*) – convert IP address to textual representation

Arguments

ip_addr *a*
IP address

char * *buf*
buffer of size at least `STD_ADDRESS_P_LENGTH`

Description

This function takes an IP address and creates its textual representation for presenting to the user.

Function

char * *ip_ntox* (ip_addr *a*, char * *buf*) – convert IP address to hexadecimal representation

Arguments

ip_addr *a*
IP address

char * *buf*
buffer of size at least `STD_ADDRESS_P_LENGTH`

Description

This function takes an IP address and creates its hexadecimal textual representation. Primary use: debugging dumps.

Function

int *ip_pton* (char * *a*, ip_addr * *o*) – parse textual representation of IP address

Arguments

char * *a*
textual representation

ip_addr * *o*
where to put the resulting address

Description

This function parses a textual IP address representation and stores the decoded address to a variable pointed to by *o*. Returns 0 if a parse error has occurred, else 0.

7.2 Linked lists

The BIRD library provides a set of functions for operating on linked lists. The lists are internally represented as standard doubly linked lists with synthetic head and tail which makes all the basic operations run in constant time and contain no extra end-of-list checks. Each list is described by a `list` structure, nodes can have any format as long as they start with a `node` structure. If you want your nodes to belong to multiple lists at once, you can embed multiple `node` structures in them and use the *SKIP_BACK()* macro to calculate a pointer to the start of the structure from a `node` pointer, but beware of obscurity.

There also exist safe linked lists (`slist`, `snode` and all functions being prefixed with `s_`) which support asynchronous walking very similar to that used in the `fib` structure.

Function

LIST_INLINE void *add_tail* (list * *l*, node * *n*) – append a node to a list

Arguments

list * *l*
  linked list

node * *n*
  list node

Description

add_tail() takes a node *n* and appends it at the end of the list *l*.

Function

LIST_INLINE void *add_head* (list * *l*, node * *n*) – prepend a node to a list

Arguments

list * *l*
  linked list

node * *n*
  list node

Description

add_head() takes a node *n* and prepends it at the start of the list *l*.

Function

LIST_INLINE void *insert_node* (node * *n*, node * *after*) – insert a node to a list

Arguments

node * *n*
  a new list node

node * *after*
  a node of a list

Description

Inserts a node *n* to a linked list after an already inserted node *after*.

Function

LIST_INLINE void *rem_node* (node * *n*) – remove a node from a list

Arguments

node * *n*
  node to be removed

Description

Removes a node *n* from the list it's linked in. Afterwards, node *n* is cleared.

Function

LIST_INLINE void *replace_node* (node * *old*, node * *new*) – replace a node in a list with another one

Arguments

node * *old*
node to be removed

node * *new*
node to be inserted

Description

Replaces node *old* in the list it's linked in with node *new*. Node *old* may be a copy of the original node, which is not accessed through the list. The function could be called with *old* == *new*, which just fixes neighbors' pointers in the case that the node was reallocated.

Function

LIST_INLINE void *init_list* (list * *l*) – create an empty list

Arguments

list * *l*
list

Description

init_list() takes a `list` structure and initializes its fields, so that it represents an empty list.

Function

LIST_INLINE void *add_tail_list* (list * *to*, list * *l*) – concatenate two lists

Arguments

list * *to*
destination list

list * *l*
source list

Description

This function appends all elements of the list *l* to the list *to* in constant time.

7.3 Miscellaneous functions.

Function

int *ipsum_verify* (void * *frag*, uint *len*, ...) – verify an IP checksum

Arguments

void * *frag*
first packet fragment

uint *len*
length in bytes

... ...
variable arguments

Description

This function verifies whether a given fragmented packet has correct one's complement checksum as used by the IP protocol.
It uses all the clever tricks described in RFC 1071 to speed up checksum calculation as much as possible.

Result

1 if the checksum is correct, 0 else.

Function

u16 *ipsum_calculate* (void * *frag*, uint *len*, ...) – compute an IP checksum

Arguments

void * *frag*
first packet fragment

uint *len*
length in bytes

... ...
variable arguments

Description

This function calculates a one's complement checksum of a given fragmented packet.
It uses all the clever tricks described in RFC 1071 to speed up checksum calculation as much as possible.

Function

u32 *u32_mkmask* (uint *n*) – create a bit mask

Arguments

uint *n*
number of bits

Description

u32_mkmask() returns an unsigned 32-bit integer which binary representation consists of *n* ones followed by zeroes.

Function

int *u32_masklen* (u32 *x*) – calculate length of a bit mask

Arguments

u32 *x*
bit mask

Description

This function checks whether the given integer *x* represents a valid bit mask (binary representation contains first ones, then zeroes) and returns the number of ones or -1 if the mask is invalid.

Function

u32 *u32_log2* (u32 *v*) – compute a binary logarithm.

Arguments

u32 *v*
number

Description

This function computes a integral part of binary logarithm of given integer *v* and returns it. The computed value is also an index of the most significant non-zero bit position.

Function

int *patmatch* (byte * *p*, byte * *s*) – match shell-like patterns

Arguments

byte * *p*
: pattern

byte * *s*
: string

Description

patmatch() returns whether given string *s* matches the given shell-like pattern *p*. The patterns consist of characters (which are matched literally), question marks which match any single character, asterisks which match any (possibly empty) string of characters and backslashes which are used to escape any special characters and force them to be treated literally.
The matching process is not optimized with respect to time, so please avoid using this function for complex patterns.

Function

int *bvsnprintf* (char * *buf*, int *size*, const char * *fmt*, va_list *args*) – BIRD's *vsnprintf()*

Arguments

char * *buf*
: destination buffer

int *size*
: size of the buffer

const char * *fmt*
: format string

va_list *args*
: a list of arguments to be formatted

Description

This functions acts like ordinary *sprintf()* except that it checks available space to avoid buffer overflows and it allows some more

format specifiers

`I` for formatting of IP addresses (any non-zero width is automatically replaced by standard IP address width which depends on whether we use IPv4 or IPv6; `%#I` gives hexadecimal format), `R` for Router / Network ID (u32 value printed as IPv4 address) `lR` for 64bit Router / Network ID (u64 value printed as eight :-separated octets) and `m` resp. `M` for error messages (uses *strerror()* to translate *errno* code to message text). On the other hand, it doesn't support floating point numbers.

Result

number of characters of the output string or `-1` if the buffer space was insufficient.

Function

int *bvsprintf* (char * *buf*, const char * *fmt*, va_list *args*) – BIRD's *vsprintf()*

Arguments

char * *buf*
: buffer

const char * *fmt*
: format string

va_list *args*
: a list of arguments to be formatted

Description
This function is equivalent to *bvsnprintf()* with an infinite buffer size. Please use carefully only when you are absolutely sure the buffer won't overflow.

Function
int *bsprintf* (char * *buf*, const char * *fmt*, ...) – BIRD's *sprintf()*

Arguments

char * *buf*
: buffer

const char * *fmt*
: format string

... ...
: variable arguments

Description
This function is equivalent to *bvsnprintf()* with an infinite buffer size and variable arguments instead of a `va_list`. Please use carefully only when you are absolutely sure the buffer won't overflow.

Function
int *bsnprintf* (char * *buf*, int *size*, const char * *fmt*, ...) – BIRD's *snprintf()*

Arguments

char * *buf*
: buffer

int *size*
: buffer size

const char * *fmt*
: format string

... ...
: variable arguments

Description
This function is equivalent to *bsnprintf()* with variable arguments instead of a `va_list`.

Function
void * *xmalloc* (uint *size*) – malloc with checking

Arguments

uint *size*
: block size

Description
This function is equivalent to *malloc()* except that in case of failure it calls *die()* to quit the program instead of returning a `NULL` pointer.
Wherever possible, please use the memory resources instead.

Function

void * *xrealloc* (void * *ptr*, uint *size*) – realloc with checking

Arguments

void * *ptr*
original memory block

uint *size*
block size

Description

This function is equivalent to *realloc()* except that in case of failure it calls *die()* to quit the program instead of returning a `NULL` pointer.
Wherever possible, please use the memory resources instead.

7.4 Message authentication codes

MAC algorithms are simple cryptographic tools for message authentication. They use shared a secret key a and message text to generate authentication code, which is then passed with the message to the other side, where the code is verified. There are multiple families of MAC algorithms based on different cryptographic primitives, BIRD implements two MAC families which use hash functions.

The first family is simply a cryptographic hash camouflaged as MAC algorithm. Originally supposed to be (m|k)-hash (message is concatenated with key, and that is hashed), but later it turned out that a raw hash is more practical. This is used for cryptographic authentication in OSPFv2, RIP and BFD.

The second family is the standard HMAC (RFC 2104), using inner and outer hash to process key and message. HMAC (with SHA) is used in advanced OSPF and RIP authentication (RFC 5709, RFC 4822).

Function

void *mac_init* (struct mac_context * *ctx*, uint *id*, const byte * *key*, uint *keylen*) – initialize MAC algorithm

Arguments

struct mac_context * *ctx*
context to initialize

uint *id*
MAC algorithm ID

const byte * *key*
MAC key

uint *keylen*
MAC key length

Description

Initialize MAC context *ctx* for algorithm *id* (e.g., `ALG_HMAC_SHA1`), with key *key* of length *keylen*. After that, message data could be added using *mac_update()* function.

Function

void *mac_update* (struct mac_context * *ctx*, const byte * *data*, uint *datalen*) – add more data to MAC algorithm

Arguments

struct mac_context * *ctx*
MAC context

const byte * *data*
data to add

uint *datalen*
length of data

Description
Push another *datalen* bytes of data pointed to by *data* into the MAC algorithm currently in *ctx*. Can be called multiple times for the same MAC context. It has the same effect as concatenating all the data together and passing them at once.

Function
byte * *mac_final* (struct mac_context * *ctx*) – finalize MAC algorithm

Arguments

struct mac_context * *ctx*
MAC context

Description
Finish MAC computation and return a pointer to the result. No more *mac_update*() calls could be done, but the context may be reinitialized later.
Note that the returned pointer points into data in the *ctx* context. If it ceases to exist, the pointer becomes invalid.

Function
void *mac_cleanup* (struct mac_context * *ctx*) – cleanup MAC context

Arguments

struct mac_context * *ctx*
MAC context

Description
Cleanup MAC context after computation (by filling with zeros). Not strictly necessary, just to erase sensitive data from stack. This also invalidates the pointer returned by *mac_final*().

Function
void *mac_fill* (uint *id*, const byte * *key*, uint *keylen*, const byte * *data*, uint *datalen*, byte * *mac*) – compute and fill MAC

Arguments

uint *id*
MAC algorithm ID

const byte * *key*
secret key

uint *keylen*
key length

const byte * *data*
message data

uint *datalen*
message length

byte * *mac*
place to fill MAC

Description
Compute MAC for specified key *key* and message *data* using algorithm *id* and copy it to buffer *mac*. *mac_fill()* is a shortcut function doing all usual steps for transmitted messages.

Function

int *mac_verify* (uint *id*, const byte * *key*, uint *keylen*, const byte * *data*, uint *datalen*, const byte * *mac*) – compute and verify MAC

Arguments

uint *id*
MAC algorithm ID

const byte * *key*
secret key

uint *keylen*
key length

const byte * *data*
message data

uint *datalen*
message length

const byte * *mac*
received MAC

Description
Compute MAC for specified key *key* and message *data* using algorithm *id* and compare it with received *mac*, return whether they are the same. *mac_verify()* is a shortcut function doing all usual steps for received messages.

Chapter 8: Resources

8.1 Introduction

Most large software projects implemented in classical procedural programming languages usually end up with lots of code taking care of resource allocation and deallocation. Bugs in such code are often very difficult to find, because they cause only 'resource leakage', that is keeping a lot of memory and other resources which nobody references to.

We've tried to solve this problem by employing a resource tracking system which keeps track of all the resources allocated by all the modules of BIRD, deallocates everything automatically when a module shuts down and it is able to print out the list of resources and the corresponding modules they are allocated by.

Each allocated resource (from now we'll speak about allocated resources only) is represented by a structure starting with a standard header (struct `resource`) consisting of a list node (resources are often linked to various lists) and a pointer to `resclass` – a resource class structure pointing to functions implementing generic resource operations (such as freeing of the resource) for the particular resource type.

There exist the following types of resources:

- *Resource pools* (`pool`)
- *Memory blocks*
- *Linear memory pools* (`linpool`)
- *Slabs* (`slab`)
- *Events* (`event`)
- *Timers* (`timer`)
- *Sockets* (`socket`)

8.2 Resource pools

Resource pools (`pool`) are just containers holding a list of other resources. Freeing a pool causes all the listed resources to be freed as well. Each existing `resource` is linked to some pool except for a root pool which isn't linked anywhere, so all the resources form a tree structure with internal nodes corresponding to pools and leaves being the other resources.

Example: Almost all modules of BIRD have their private pool which is freed upon shutdown of the module.

Function

pool * *rp_new* (pool * *p*, char * *name*) – create a resource pool

Arguments

pool * *p*
    parent pool

char * *name*
    pool name (to be included in debugging dumps)

Description

rp_new() creates a new resource pool inside the specified parent pool.

Function

void *rmove* (void * *res*, pool * *p*) – move a resource

Arguments

void * *res*
resource

pool * *p*
pool to move the resource to

Description
rmove() moves a resource from one pool to another.

Function
void *rfree* (void * *res*) – free a resource

Arguments

void * *res*
resource

Description
rfree() frees the given resource and all information associated with it. In case it's a resource pool, it also frees all the objects living inside the pool.
It works by calling a class-specific freeing function.

Function
void *rdump* (void * *res*) – dump a resource

Arguments

void * *res*
resource

Description
This function prints out all available information about the given resource to the debugging output.
It works by calling a class-specific dump function.

Function
void * *ralloc* (pool * *p*, struct resclass * *c*) – create a resource

Arguments

pool * *p*
pool to create the resource in

struct resclass * *c*
class of the new resource

Description
This function is called by the resource classes to create a new resource of the specified class and link it to the given pool. Allocated memory is zeroed. Size of the resource structure is taken from the *size* field of the `resclass`.

Function
void *rlookup* (unsigned long *a*) – look up a memory location

Arguments

unsigned long *a*
memory address

Description
This function examines all existing resources to see whether the address *a* is inside any resource. It's used for debugging purposes only.
It works by calling a class-specific lookup function for each resource.

Function

void *resource_init* (*void*) – initialize the resource manager

Description
This function is called during BIRD startup. It initializes all data structures of the resource manager and creates the root pool.

8.3 Memory blocks

Memory blocks are pieces of contiguous allocated memory. They are a bit non-standard since they are represented not by a pointer to `resource`, but by a void pointer to the start of data of the memory block. All memory block functions know how to locate the header given the data pointer.

Example: All "unique" data structures such as hash tables are allocated as memory blocks.

Function

void * *mb_alloc* (pool * *p*, unsigned *size*) – allocate a memory block

Arguments

pool * *p*
pool

unsigned *size*
size of the block

Description
mb_alloc() allocates memory of a given size and creates a memory block resource representing this memory chunk in the pool *p*.
Please note that *mb_alloc()* returns a pointer to the memory chunk, not to the resource, hence you have to free it using *mb_free()*, not *rfree()*.

Function

void * *mb_allocz* (pool * *p*, unsigned *size*) – allocate and clear a memory block

Arguments

pool * *p*
pool

unsigned *size*
size of the block

Description
mb_allocz() allocates memory of a given size, initializes it to zeroes and creates a memory block resource representing this memory chunk in the pool *p*.
Please note that *mb_allocz()* returns a pointer to the memory chunk, not to the resource, hence you have to free it using *mb_free()*, not *rfree()*.

Function

void * *mb_realloc* (void * *m*, unsigned *size*) – reallocate a memory block

Arguments

void * *m*
memory block

unsigned *size*
new size of the block

Description

mb_realloc() changes the size of the memory block *m* to a given size. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. Contrary to *realloc()* behavior, *m* must be non-NULL, because the resource pool is inherited from it.

Like *mb_alloc()*, *mb_realloc()* also returns a pointer to the memory chunk, not to the resource, hence you have to free it using *mb_free()*, not *rfree()*.

Function

void *mb_free* (void * *m*) – free a memory block

Arguments

void * *m*
memory block

Description

mb_free() frees all memory associated with the block *m*.

8.4 Linear memory pools

Linear memory pools are collections of memory blocks which support very fast allocation of new blocks, but are able to free only the whole collection at once.

Example: Each configuration is described by a complex system of structures, linked lists and function trees which are all allocated from a single linear pool, thus they can be freed at once when the configuration is no longer used.

Function

linpool * *lp_new* (pool * *p*, uint *blk*) – create a new linear memory pool

Arguments

pool * *p*
pool

uint *blk*
block size

Description

lp_new() creates a new linear memory pool resource inside the pool *p*. The linear pool consists of a list of memory chunks of size at least *blk*.

Function

void * *lp_alloc* (linpool * *m*, uint *size*) – allocate memory from a `linpool`

Arguments

linpool * *m*
linear memory pool

uint *size*
amount of memory

Description

lp_alloc() allocates *size* bytes of memory from a `linpool` *m* and it returns a pointer to the allocated memory. It works by trying to find free space in the last memory chunk associated with the `linpool` and creating a new chunk of the standard size (as specified during *lp_new()*) if the free space is too small to satisfy the allocation. If *size* is too large to fit in a standard size chunk, an ”overflow” chunk is created for it instead.

Function

void * *lp_allocu* (linpool * *m*, uint *size*) – allocate unaligned memory from a `linpool`

Arguments

linpool * *m*
  linear memory pool

uint *size*
  amount of memory

Description

lp_allocu() allocates *size* bytes of memory from a `linpool` *m* and it returns a pointer to the allocated memory. It doesn't attempt to align the memory block, giving a very efficient way how to allocate strings without any space overhead.

Function

void * *lp_allocz* (linpool * *m*, uint *size*) – allocate cleared memory from a `linpool`

Arguments

linpool * *m*
  linear memory pool

uint *size*
  amount of memory

Description

This function is identical to *lp_alloc()* except that it clears the allocated memory block.

Function

void *lp_flush* (linpool * *m*) – flush a linear memory pool

Arguments

linpool * *m*
  linear memory pool

Description

This function frees the whole contents of the given `linpool` *m*, but leaves the pool itself.

8.5 Slabs

Slabs are collections of memory blocks of a fixed size. They support very fast allocation and freeing of such blocks, prevent memory fragmentation and optimize L2 cache usage. Slabs have been invented by Jeff Bonwick and published in USENIX proceedings as 'The Slab Allocator: An Object-Caching Kernel Memory Allocator'. Our implementation follows this article except that we don't use constructors and destructors.

When the `DEBUGGING` switch is turned on, we automatically fill all newly allocated and freed blocks with a special pattern to make detection of use of uninitialized or already freed memory easier.

Example: Nodes of a FIB are allocated from a per-FIB Slab.

Function

slab * *sl_new* (pool * *p*, uint *size*) – create a new Slab

Arguments

pool * *p*
resource pool

uint *size*
block size

Description
This function creates a new Slab resource from which objects of size *size* can be allocated.

Function
void * *sl_alloc* (slab * *s*) – allocate an object from Slab

Arguments

slab * *s*
slab

Description
sl_alloc() allocates space for a single object from the Slab and returns a pointer to the object.

Function
void *sl_free* (slab * *s*, void * *oo*) – return a free object back to a Slab

Arguments

slab * *s*
slab

void * *oo*
object returned by *sl_alloc()*

Description
This function frees memory associated with the object *oo* and returns it back to the Slab *s*.

8.6 Events

Events are there to keep track of deferred execution. Since BIRD is single-threaded, it requires long lasting tasks to be split to smaller parts, so that no module can monopolize the CPU. To split such a task, just create an `event` resource, point it to the function you want to have called and call *ev_schedule()* to ask the core to run the event when nothing more important requires attention.

You can also define your own event lists (the `event_list` structure), enqueue your events in them and explicitly ask to run them.

Function
event * *ev_new* (pool * *p*) – create a new event

Arguments

pool * *p*
resource pool

Description
This function creates a new event resource. To use it, you need to fill the structure fields and call *ev_schedule()*.

Function

void *ev_run* (event * *e*) – run an event

Arguments

event * *e*
an event

Description

This function explicitly runs the event *e* (calls its hook function) and removes it from an event list if it's linked to any.
From the hook function, you can call *ev_enqueue()* or *ev_schedule()* to re-add the event.

Function

void *ev_enqueue* (event_list * *l*, event * *e*) – enqueue an event

Arguments

event_list * *l*
an event list

event * *e*
an event

Description

ev_enqueue() stores the event *e* to the specified event list *l* which can be run by calling *ev_run_list()*.

Function

void *ev_schedule* (event * *e*) – schedule an event

Arguments

event * *e*
an event

Description

This function schedules an event by enqueueing it to a system-wide event list which is run by the platform dependent code whenever appropriate.

Function

int *ev_run_list* (event_list * *l*) – run an event list

Arguments

event_list * *l*
an event list

Description

This function calls *ev_run()* for all events enqueued in the list *l*.

8.7 Timers

Timers are resources which represent a wish of a module to call a function at the specified time. The platform dependent code doesn't guarantee exact timing, only that a timer function won't be called before the requested time.

In BIRD, time is represented by values of the `bird_clock_t` type which are integral numbers interpreted as a relative number of seconds since some fixed time point in past. The current time can be read from variable *now* with reasonable accuracy and is monotonic. There is also a current 'absolute' time in variable *now_real* reported by OS.

Each timer is described by a `timer` structure containing a pointer to the handler function (*hook*), data private to this function (*data*), time the function should be called at (*expires*, 0 for inactive timers), for the other fields see `timer.h`.

Function

timer * *tm_new* (pool * *p*) – create a timer

Arguments

pool * *p*
    pool

Description

This function creates a new timer resource and returns a pointer to it. To use the timer, you need to fill in the structure fields and call *tm_start()* to start timing.

Function

void *tm_start* (timer * *t*, unsigned *after*) – start a timer

Arguments

timer * *t*
    timer

unsigned *after*
    number of seconds the timer should be run after

Description

This function schedules the hook function of the timer to be called after *after* seconds. If the timer has been already started, it's *expire* time is replaced by the new value.

You can have set the *randomize* field of *t*, the timeout will be increased by a random number of seconds chosen uniformly from range 0 .. *randomize*.

You can call *tm_start()* from the handler function of the timer to request another run of the timer. Also, you can set the *recurrent* field to have the timer re-added automatically with the same timeout.

Function

void *tm_stop* (timer * *t*) – stop a timer

Arguments

timer * *t*
    timer

Description

This function stops a timer. If the timer is already stopped, nothing happens.

Function

bird_clock_t *tm_parse_datetime* (char * *x*) – parse a date and time

Arguments

char * *x*
    datetime string

Description

tm_parse_datetime() takes a textual representation of a date and time (dd-mm-yyyy hh:mm:ss) and converts it to the corresponding value of type `bird_clock_t`.

Function

bird_clock_t *tm_parse_date* (char * *x*) – parse a date

Arguments

char * *x*
    date string

Description

tm_parse_date() takes a textual representation of a date (dd-mm-yyyy) and converts it to the corresponding value of type `bird_clock_t`.

Function

void *tm_format_datetime* (char * *x*, struct timeformat * *fmt_spec*, bird_clock_t *t*) – convert date and time to textual representation

Arguments

char * *x*
    destination buffer of size `TM_DATETIME_BUFFER_SIZE`

struct timeformat * *fmt_spec*
    specification of resulting textual representation of the time

bird_clock_t *t*
    time

Description

This function formats the given relative time value *t* to a textual date/time representation (dd-mm-yyyy hh:mm:ss) in real time.

8.8 Sockets

Socket resources represent network connections. Their data structure (`socket`) contains a lot of fields defining the exact type of the socket, the local and remote addresses and ports, pointers to socket buffers and finally pointers to hook functions to be called when new data have arrived to the receive buffer (*rx_hook*), when the contents of the transmit buffer have been transmitted (*tx_hook*) and when an error or connection close occurs (*err_hook*).

Freeing of sockets from inside socket hooks is perfectly safe.

Function

int *sk_setup_multicast* (sock * *s*) – enable multicast for given socket

Arguments

sock * *s*
    socket

Description

Prepare transmission of multicast packets for given datagram socket. The socket must have defined *iface*.

Result

0 for success, -1 for an error.

Function

int *sk_join_group* (sock * *s*, ip_addr *maddr*) – join multicast group for given socket

Arguments

sock * *s*
socket

ip_addr *maddr*
multicast address

Description

Join multicast group for given datagram socket and associated interface. The socket must have defined *iface*.

Result

0 for success, -1 for an error.

Function

int *sk_leave_group* (sock * *s*, ip_addr *maddr*) – leave multicast group for given socket

Arguments

sock * *s*
socket

ip_addr *maddr*
multicast address

Description

Leave multicast group for given datagram socket and associated interface. The socket must have defined *iface*.

Result

0 for success, -1 for an error.

Function

int *sk_setup_broadcast* (sock * *s*) – enable broadcast for given socket

Arguments

sock * *s*
socket

Description

Allow reception and transmission of broadcast packets for given datagram socket. The socket must have defined *iface*. For transmission, packets should be send to *brd* address of *iface*.

Result

0 for success, -1 for an error.

Function

int *sk_set_ttl* (sock * *s*, int *ttl*) – set transmit TTL for given socket

Arguments

sock * *s*
socket

int *ttl*
TTL value

Description

Set TTL for already opened connections when TTL was not set before. Useful for accepted connections when different ones should have different TTL.

Result

0 for success, -1 for an error.

Function

int *sk_set_min_ttl* (sock * *s*, int *ttl*) – set minimal accepted TTL for given socket

Arguments

sock * *s*
: socket

int *ttl*
: TTL value

Description

Set minimal accepted TTL for given socket. Can be used for TTL security. implementations.

Result

0 for success, -1 for an error.

Function

int *sk_set_md5_auth* (sock * *s*, ip_addr *local*, ip_addr *remote*, struct iface * *ifa*, char * *passwd*, int *setkey*) – add / remove MD5 security association for given socket

Arguments

sock * *s*
: socket

ip_addr *local*
: IP address of local side

ip_addr *remote*
: IP address of remote side

struct iface * *ifa*
: Interface for link-local IP address

char * *passwd*
: Password used for MD5 authentication

int *setkey*
: Update also system SA/SP database

Description

In TCP MD5 handling code in kernel, there is a set of security associations used for choosing password and other authentication parameters according to the local and remote address. This function is useful for listening socket, for active sockets it may be enough to set s->password field.

When called with passwd != NULL, the new pair is added, When called with passwd == NULL, the existing pair is removed.

Note that while in Linux, the MD5 SAs are specific to socket, in BSD they are stored in global SA/SP database (but the behavior also must be enabled on per-socket basis). In case of multiple sockets to the same neighbor, the socket-specific state must be configured for each socket while global state just once per src-dst pair. The *setkey* argument controls whether the global state (SA/SP database) is also updated.

Result

0 for success, -1 for an error.

Function

int *sk_set_ipv6_checksum* (sock * *s*, int *offset*) – specify IPv6 checksum offset for given socket

Arguments

sock * *s*
socket

int *offset*
offset

Description
Specify IPv6 checksum field offset for given raw IPv6 socket. After that, the kernel will automatically fill it for outgoing packets and check it for incoming packets. Should not be used on ICMPv6 sockets, where the position is known to the kernel.

Result
0 for success, -1 for an error.

Function
sock * *sock_new* (pool * *p*) – create a socket

Arguments

pool * *p*
pool

Description
This function creates a new socket resource. If you want to use it, you need to fill in all the required fields of the structure and call *sk_open()* to do the actual opening of the socket.
The real function name is *sock_new()*, *sk_new()* is a macro wrapper to avoid collision with OpenSSL.

Function
int *sk_open* (sock * *s*) – open a socket

Arguments

sock * *s*
socket

Description
This function takes a socket resource created by *sk_new()* and initialized by the user and binds a corresponding network connection to it.

Result
0 for success, -1 for an error.

Function
int *sk_send* (sock * *s*, unsigned *len*) – send data to a socket

Arguments

sock * *s*
socket

unsigned *len*
number of bytes to send

Description
This function sends *len* bytes of data prepared in the transmit buffer of the socket *s* to the network connection. If the packet can be sent immediately, it does so and returns 1, else it queues the packet for later processing, returns 0 and calls the *tx_hook* of the socket when the tranmission takes place.

Function

int *sk_send_to* (sock * *s*, unsigned *len*, ip_addr *addr*, unsigned *port*) – send data to a specific destination

Arguments

sock * *s*
socket

unsigned *len*
number of bytes to send

ip_addr *addr*
IP address to send the packet to

unsigned *port*
port to send the packet to

Description

This is a *sk_send()* replacement for connection-less packet sockets which allows destination of the packet to be chosen dynamically. Raw IP sockets should use 0 for *port*.

Function

void *io_log_event* (void * *hook*, void * *data*) – mark approaching event into event log

Arguments

void * *hook*
event hook address

void * *data*
event data address

Description

Store info (hook, data, timestamp) about the following internal event into a circular event log (*event_log*). When latency tracking is enabled, the log entry is kept open (in *event_open*) so the duration can be filled later.