Design and Implementation of *OpenOSPFD*

by Claudio Jeker <claudio@openbsd.org>
Internet Business Solutions AG

Abstract

OpenOSPFD is a free and secure implementation of the Open Shortest Path First protocol. It allows ordinary machines to be used as routers exchanging and calculating routes within an OSPF cloud.

OpenOSPFD is the next major step after OpenBGPD for full router capabilities in OpenBSD and other BSDs. Together with OpenBGPD it is possible to re-route traffic in case of link loss resulting in a higher-level of availability.

Overview

1.1 Routing Protocols

The Internet is split into regions called Autonomous Systems (AS). Each AS is under the control of a single administrative entity – for example a university or an ISP. The edge routers of these AS use an Exterior Gateway Protocol (EGP) to exchange routing information between AS. Currently BGP4, the Border Gateway Protocol is the only EGP in widespread use. Routers within an AS use an Interior Gateway Protocol to exchange routing information. There are different IGPs. OSPF, IS-IS, and RIP are the most commonly used. It is possible and common to have multiple IGPs running inside one AS.
The Routing Information Protocol (RIP) is a legacy protocol that is often found on appliances. It is not suitable for larger networks because the distance vector algorithm used by RIP converges slowly. Especially in the face of certain network failures (count to infinity). OSPF and IS-IS on the other hand are both link-state protocols. The Intermediate System to Intermediate System (IS-IS) protocol was developed for the OSI protocol suite under the lead of the ITU.

Why not use one protocol for everything, EGP and IGP? The requirements for an IGP differ from those of a an EGP. For an IGP it is important to recalculate the routing table quickly when the network changes. Another factor is automatic neighbor discovery. On the other hand the most important feature of an EGP is the ability to express routing policies. The resulting routing table is normally cost optimised.

1.2 Algorithms

There are two main concepts to exchange routing information. These algorithms are working in a totally different ways.

1.2.1 Distance Vector Algorithms

Distance vector algorithms got their name from the form of the routing updates: a vector of metrics.
In a distance vector algorithm every router exchanges its routing table with all his neighbors. The neighbors then walk through the list and compare if their current route entry is better or not. If not the route is replaced and redistributed again.
In case of RIP the list of routes and their metric is exchanged every 30 seconds. This results in a slow convergence because an update propagates only one hop every 30 seconds. On the other hand the protocol is simple and robust because every router cares only about his own neighbors. In other words the information about the network topology is distributed. This results in one of the biggest weaknesses of RIP – the count to infinity problem – resulting in slow convergence and routing loops if a network becomes unavailable. There are some countermeasures against this. The simplest is to pass the full routing path instead of only the metric. This path distance vector algorithm is used by BGP. It is easy to implement routing policies on distance vector algorithms.

1.2.2 Link-State Algorithms

In a link-state protocol every router or node sends out his current link-states. The link-state advertisements are distributed to all nodes in the network. The resulting replicated distributed database represents the entire network topology. Every node uses this connectivity map to calculate the shortest path to every other router. Link-state protocols have good convergence properties. The biggest weakness of link-state protocols is the replicated distributed database. If the database gets out of sync non optimal routes are used and in worst case routing loops are created. Link-state protocols are more complicated than distance vector protocols.

OSPF – the protocol

The OSPF routing protocol was developed within the IETF. The work started in 1987. The current version (OSPFv2) of the specification was published in 1998 as RFC 2328.

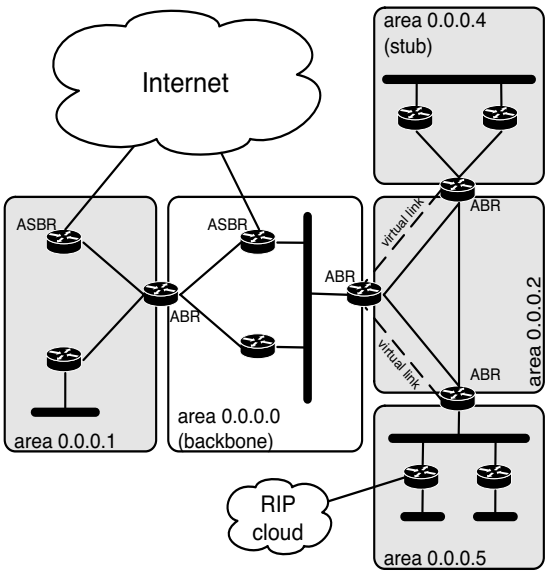

Figure 1: Sample OSPF network

2.1 Architecture

The Open Shortest Path First (OSPF) protocol is a link-state, hierarchical routing protocol. It is probably the most used IGP in the world. It is capable of doing neighbor discovery on different types of networks with minimal need for configuration. OSPF encapsulates its routing messages directly on top of IP as its own protocol type (89). TCP connections are not used because the link-state flooding algorithm already includes its own way for reliable communications – adding to OSPF's complexity. Most obvious the massive use of IP multicast in OSPF makes TCP infeasible.

2.1.1 Networks

An OSPF router discovers neighbors by periodically sending OSPF Hello packets out on all configured interfaces. Depending of the interface type different methods are used. The flooding algorithm depends on the interface type as well.
The simplest interface type is a point-to-point interface. Neighbor discovery is easy – there is only one neighbor on the other side of the link – and no special link-state flooding enhancement is required.
For ethernet and other broadcast networks OSPF uses multicast to find all neighbors on the segment. The link-state updates are flooded via multicast as well. To make the thing even more complicated a designated router (DR) was introduced. The DR has the duty to enforce the reliable flooding for all other routers connected to the same LAN. A backup designated router (BDR) was introduced to take over in case of a DR failure.

Additionally more flooding procedures where defined for other important network types like NBMA (non-broadcast multiple-access) or point-to-multipoint networks. Examples include X.25, Frame Relay, or ATM using full mesh or switched virtual circuits. OpenOSPFD does not support these exotic networks mostly because of lack of support by the OS and missing infrastructure.

2.1.2 Database synchronisation and reliable flooding

Database synchronisation in a link-state protocol is crucial. The routing calculation ensures a loop-free routing as long as the database remains perfectly synchronised. It is no wonder that this is the most fragile part of the specification. Especially with all the additional complexity added by multicasting of updates and the presence of DR and BDR routers. A reliable and robust flooding procedure is very important because a little inadvertence can result in a major network “melt down” where only a full reset of all routers cures the situation.

Database synchronisation takes two forms. First there is the initial database synchronisation. Following it the distributed copies of the database need to be kept in sync by reliably flooding updates to all routers in the network.
The initial database exchange is done when two routers build an adjacency. First a request list is built up through a TFTP like database exchange phase. In the exchange phase one of the two neighbors is elected as master of that session. This router sends a Database Description packet to the slave and waits for an answer. If none is received within some amount of time the packet is retransmitted. A sequence number identifies duplicates. At any given point in time only one packet can be outstanding. Afterwards Link-State Requests are sent between the two routers. The other side then sends the requested link-state announcement (LSA) back to the requesting router. A full adjacency has been set up when the request list is empty. Now reliable flooding needs to ensure that the databases remain perfectly synchronised. Every time a link changes state or after a 30 minute timeout a LSA needs to be reflooded. A LS update received on one interface needs to be sent out on all other interfaces. This simple rule is unfortunately not sufficient because the flooding would never stop. So the router checks his database to see if the update was already received on a different path. In that case the update does not need to get reflooded. It is also necessary to acknowledge the updates because an non reliable transport layer was chosen. Additionally implicit acknowledgements and timeouts, throttling the generated LS updates, help to make the flooding more robust and the implementation more complex, yet again.

2.1.3 Areas

One problem of a link-state protocol is the computation cost bourn by every router, particularly in large networks. Many routers have an underpowered CPU and so OSPF areas where invented to divide a large network into smaller pieces. Every area is connected to a special backbone area. In most cases inter-area routing goes via the backbone. Routers that are connected to multiple areas are area border routers (ABR) and are always connected to the backbone area. If no direct connection to the backbone is possible, a virtual-link has to be established to at least one backbone router. Areas where no transit traffic is exchanged can be converted into stub areas, reducing the routing table to a bare minimum. Stub areas are useful to connect routers with minimal memory configurations to large OSPF clouds.

LSAs are flooded only inside an area. The ABR has the duty to reflood the other areas with special summary-LSAs to inform them of available prefixes inside the originating area.

2.1.4 Border routers

Besides ABRs another kind of boarder router exists. A router is automatically an AS border router (ASBR) if it imports routes from external sources into the link-state database. External sources are other routing protocols or manually configured static routes. These routers are on the boarder of the OSPF cloud but are not necessary on the real AS border. The external routes redistributed by a ASBR are special as they are flooded through the full OSPF cloud instead of per area as all other LSAs. Only stub areas are left out to avoid overloading those poor little routers in them.

2.2 Packets

There are five different packet types defined. Every packet starts with a common 24 byte OSPF header. This header includes all necessary information for the recipient to determine if it should be accepted and processed or ignored and dropped.

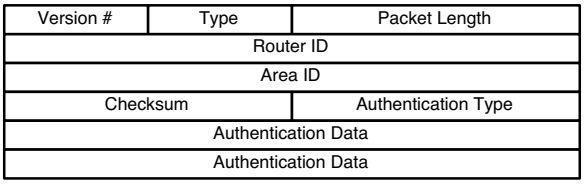

Figure 2: Common OSPF header

The standard IP CRC checksum is used to validate packet integrity. Multiple authentication procedures are defined but only one can be considered useful. Only the cryptographic authentication is enough strong to protect OSPF traffic. Only cryptographic authentication can prevent spoofing and replay attacks. After the verification the payload of the packet is examined.
The following packet types are defined:

Table 1: OSPF packet types

1	Hello
2	Database Description
3	Link-State Request
4	Link-State Update
5	Link-State Acknowledgement

2.2.1 Hello

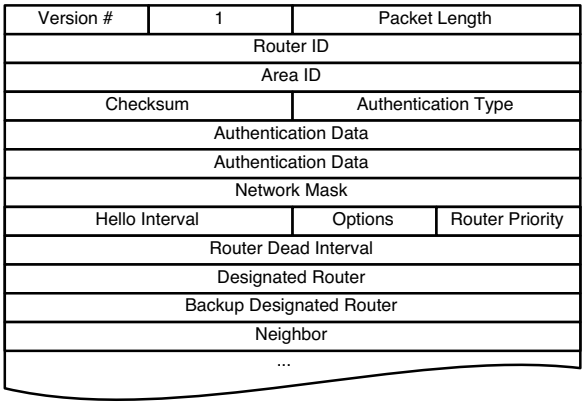

Figure 3: Hello Header

Hello packets are sent periodically in order to establish and maintain neighbor relationships. Hello packets are sent to a multicast group to enable dynamic discovery of neighboring routers. All routers to a common network must agree on certain parameters. The most important part of the hello packet is the neighbor list at the end. The router ID of each router from which a valid Hello packet has recently been received is added to that list. Only after the own router ID is seen in a neighbors Hello packet an adjacency can be formed.

2.2.2 Database Description

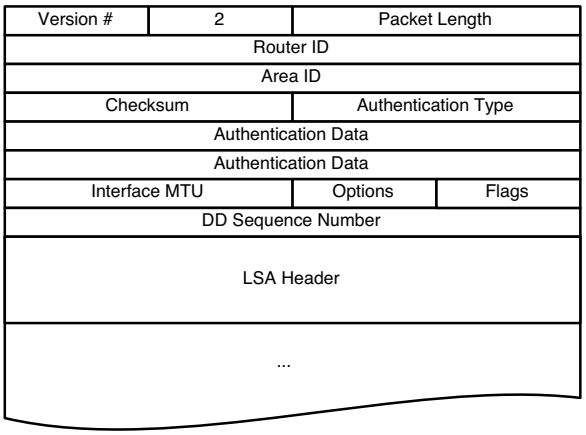

Figure 4: Database Description Header

These packets are exchanged when an adjacency is initialised. They describe the contents of the link-state database. The initial database exchange is done similar to the TFTP protocol. For that reason a sequence number is included in the header.

Additionally the MTU of the outgoing interface is included to detect possible forwarding issues with large packets. The rest of the packet consists of a list of LSA headers. A LSA header contains all information to uniquely identify a LSA.

2.2.3 Link-State Request

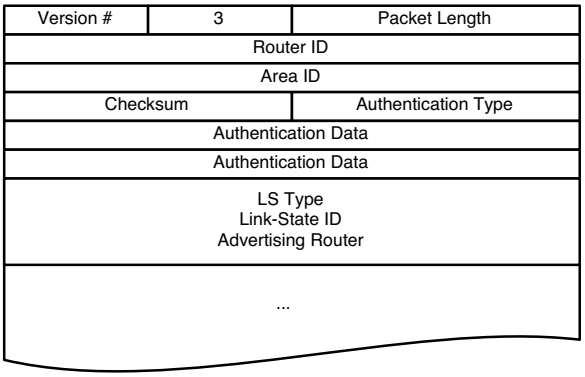

Figure 5: Link-State Request Header

After exchanging Database Description packets with the neighboring router, Link-State Request packets request pieces of the neighbors LS database that are more up-to-date. Each LSA requested is specified by its LS type, Link-State ID, and Advertising Router. This uniquely identifies the LSA, but not its instance. Link-State Request packets are understood to be requests for the most recent instance. It is possible to request multiple LSA with one LS request packet.

2.2.4 Link-State Update

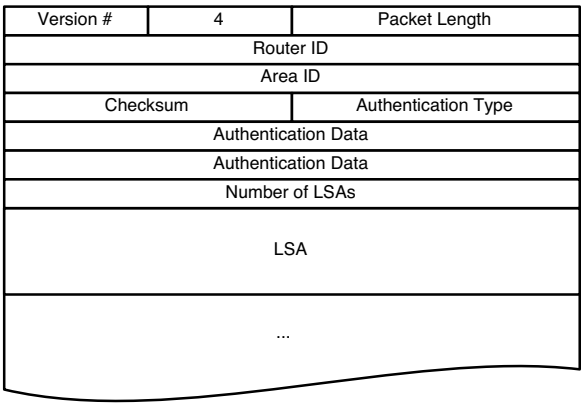

Figure 6: Link-State Update Header

These packets implement the flooding of LSAs. Each Link-State Update packet carries a collection of LSAs one hop further from their origin. Several LSAs may be included in a single packet. The body of the Link-State Update packet consists of a list of LSAs.

2.2.5 Link-State Acknowledgement

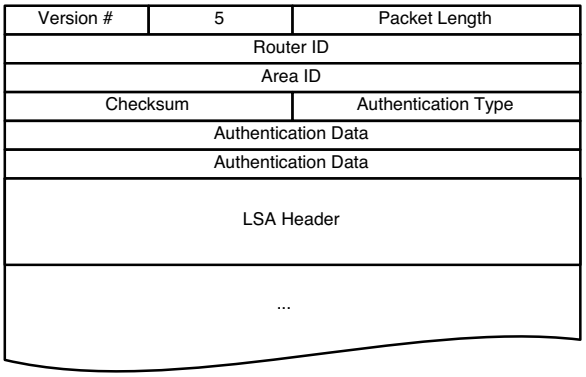

Figure 7: Link-State Acknowledgement Header

In order to make the flooding procedure reliable, flooded LSAs are acknowledged in Link-State Acknowledgement packets. Multiple LSAs can be acknowledged in a single Link-State Acknowledgement packet. The format of this packet is similar to that of the Data Description packet. The body of both packets is simply a list of LSA headers.

2.2.6 Link-State Advertisements Header

Each LSA begins with a common 20 byte header. This header is enough to uniquely identify a LSA. So it is enough to use the LSA header in LS acknowledgements and Database Description packets. LSAs are identified by the LS type, Link-State ID, and Advertising Router triple. Additionally a LS sequence number and LS age are included to determine which instance is more recent. The LS checksum protects the integrity of LSAs. Instead of the known CRC algorithm specified in many IP protocols a ISO checksum algorithm – also known as Fletcher Checksum – is employed.

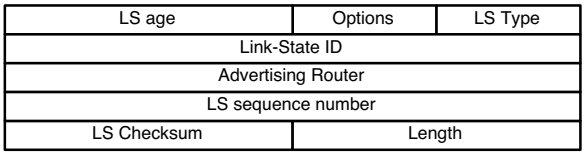

Figure 8: Link-State Advertisements Header

Each LSA type has a separate advertisement format. The LS types defined in the OSPF standard are as follows:

Table 2: LS types

1	Hello
2	Database Description
3	Link-State Request
4	Link-State Update
5	Link-State Acknowledgement

Router- and Network-LSA describe the network inside an area. Summary-LSA are injected by area border routers (ABRs) and describe inter-area destinations. AS-external-LSAs are originated by ASBRs to describe destinations external to the OSPF routing domain.

Design

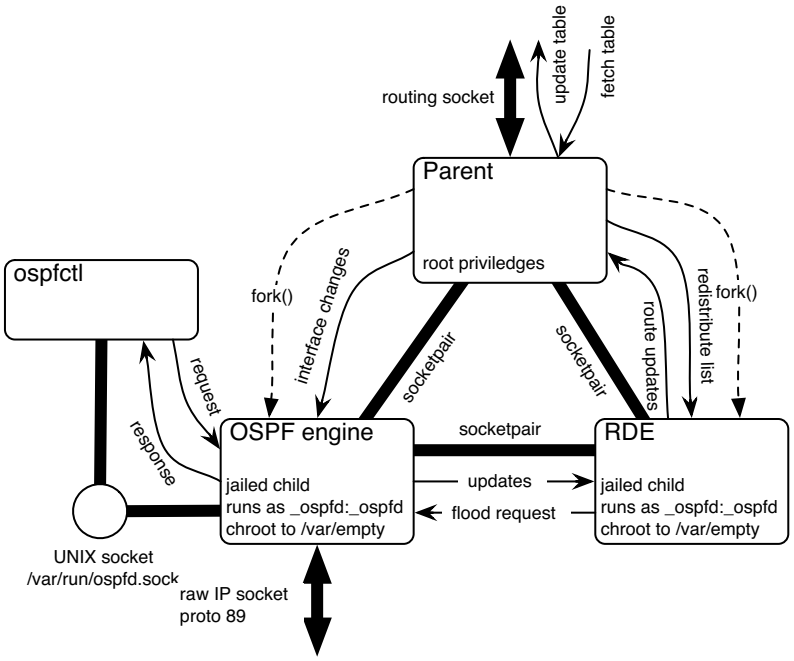

Figure 9: Design of OpenOSPFD

The design of OpenOSPFD is based on the one in OpenBGPD. The routing daemon is split into three processes. The privileged parent process handles the kernel routing table updates. The OSPF engine handles all incoming packets and the state machines with all the necessary periodic events and timeouts. Finally the route decision engine stores the LS database, calculates the SPF tree and the resulting routing table. This separation into three processes does not only enhance the security but also the stability. Even a large database recomputation in the RDE will not hold up the keep alive packets sent out by the OSPF engine. The Inter-Process Communication (IPC) system is almost the same as in OpenBGPD. The only major difference is the use of libevent for timers and file descriptor polling instead of poll(2). The basic imsg framework is still the same. OpenOSPFD switched to libevent mostly because of the OSPF engine. The engine is mostly event driven with many concurrent timers running. OpenOSPFD can be controlled and monitored via ospfctl. It works very similar to bgpctl for OpenBGPD.

3.1 Processes

3.1.1 ospfd parent

The ospfd parent process is the only one running with root privileges. This is necessary to update the kernel routing table. This process listens on a routing socket for changes and updates and distributes that information to the OSPF engine or the RDE. At a later time config-file reloads will be handled by the parent process too.

3.1.2 OSPF engine

The OSPF engine listens to the network and processes the OSPF packets. Both the interface and the neighbor finite state machine are implemented in the OSPF engine. This includes the DR/BDR election process. Additionally the reliable flooding of LS updates with retransmission and acknowledgement is done by the engine.

3.1.3 Route Decision Engine

The RDE stores the LS database, calculates the SPF tree, and informs the parent process about changes in the resulting routing table. Premature LSA aging is done by the RDE as well. Additionally redistribution of networks is handled by the process. The RDE synchronises multiple areas if the router is acting as ABR and refloods summary-LSA into the different areas if necessary.

3.1.4 ospfctl

ospfctl is the tool to control and monitor OpenOSPFD. It uses a UNIX local socket to communicate with ospfd. Over this socket imsgs are passed which encapsulate the information. There is no command line interface to OpenOSPFD because it doesn't make sense to write a clumsy CLI on a UNIX system shipping with very powerful shells and many tools to manipulate the status output. ospfctl is mostly an adapted bgpctl.

Implementation

OpenOSPFD currently consist of around 12'000 lines of C code. For comparison OpenBGPD is currently a bit under 20'000 lines. Zebra/Quagga ospfd has almost 40'000 lines of code. And that is just the ospfd directory, not including the 35'000 lines in lib and the 15'000 lines for the zebra daemon.

Lets start with a short overview of the source files.

Table 3: Overview of source files

area.c	Area handling which is actually very simple.
auth.c	Implementing all OSPF authentication extensions. Nobody wants to run a OSPF network without using cryptographic authentication.
buffer.c	buffer handling mostly for the imsg framework but also used to generate outgoing packets.
control.c	ospfctl session management and message verification.
database.c	Code for the initial database exchange. This is not related the LS database that is managed by the RDE.
hello.c	Generating and parsing of Hello packets is done here.
imsg.c	imsg framework mostly copied from OpenBGPD.
in_cksum.c	Implementation of the CRC16 checksum of the TCP/IP standards.
interface.c	Interface finite state machine, event handling and interface specific functions.
iso_cksum.c	ISO checksum also known as Fletcher checksum for LSAs.
kroute.c	Kernel routing socket handling including the FIB table.
log.c	Various logging functions mostly adapted from OpenBGPD.
lsack.c	Link-State Acknowledgement construction and parsing.
lsreq.c	Link-State Request construction and parsing, including the request list functions.
lsupdate.c	Link-State Updates construction and parsing, including the flooding function and retransmission lists.
neighbor.c	Neighbor finite state machine and event handling.

Table 3: Overview of source files

ospfd.c	Parent process, home of main().
ospfe.c	OSPF engine main event loop plus functions for self originated LSAs.
packet.c	Packet reception and sending.
parse.y	Configuration parser.
printconf.c	Configuration dumping used by the -n switch.
rde.c	RDE main event loop plus other RDE specific functions.
rde_lsdb.c	LS database code.
rde_spf.c	SPF algorithm and RIB calculation.

4.1 Important datastructures

There are four main datastructures in OpenOSPFD. It is important to know what such a structure represents to understand the code. Most of the time when the term interface is used, the actual `struct iface` of that interface is meant. Ditto for neighbor or area.

4.1.1 ospfd_conf

This is the main config of the router. It holds the parameters like the router ID, `spf_delay` or `redistribute_flags`. The `lsa_tree` and `cand_list` are used in the RDE by the LS database and SPF algorithm. The `area_list` holds all configured areas. Finally there is one event handler used for polling the raw socket or implementing the SPF timer depending on the process it is used in.

Code snip 1: struct ospfd_conf

```
struct ospfd_conf {
        struct event            ev;
        struct in_addr          rtr_id;
        struct lsa_tree         lsa_tree;
        LIST_HEAD(, area)       area_list;
        LIST_HEAD(, vertex)     cand_list;
        u_int32_t               opts;
        u_int32_t               spf_delay;
        u_int32_t               spf_hold_time;
        int                     spf_state;
        int                     ospf_socket;
        int                     flags;
        int                     redistribute_flags;
        int                     options; /* OSPF options */
        u_int8_t                rfc1583compat;
        u_int8_t                border;
};
```

4.1.2 area

Area specific configurations are stored in the area descriptor. There are many parameters that are mostly used by the OSPF engine. Exclusively for the RDE are `lsa_tree` and the `nbr_list`. The first stores the per area LS database. The second is a list of all active neighbors from the RDE perspective. The OSPF engine tells the RDE when neighbors are created, deleted, or when their state changes. On the other hand `active` is only used by

the OSPF engine. `active` tracks the number of neighbors which are in state *FULL*. If the number is zero the area is considered inactive. This counter is used to determine if a router is an area border router.

Code snip 2: struct area

```
struct area {
        LIST_ENTRY(area)        entry;
        struct in_addr          id;
        struct lsa_tree         lsa_tree;
        LIST_HEAD(, iface)      iface_list;
        LIST_HEAD(, rde_nbr)    nbr_list;
        u_int32_t               stub_default_cost;
        u_int32_t               num_spf_calc;
        u_int32_t               dead_interval;
        int                     active;
        u_int16_t               transmit_delay;
        u_int16_t               hello_interval;
        u_int16_t               rxmt_interval;
        u_int16_t               metric;
        u_int8_t                priority;
        u_int8_t                transit;
        u_int8_t                stub;
};
```

4.1.3 interface

Every configured interface is represented by a `struct iface`. It stores values like the `link_state`, `baudrate`, MTU, and interface type. There are some additional OSPF specific parameters like the `auth_type`, list of keys used for cryptographic authentication (`auth_md_list`), interface `metric` and interface `state`. Lets have a look at the neighbor list and the three neighbor pointers `dr`, `bdr`, and `self`. `dr` and `bdr` are pointers to the active DR or BDR neighbor or `NULL` if there is none. `self` is used for a dummy neighbor structure that represents the router himself. Using this dummy neighbor simplifies many cases but additional care needs to be taken to not remove it by accident or doing some other stupid action with it. A back pointer to the parent area this interface is part of is also included. An interface can have up to three concurrent timers running and therefore three different event structures are needed.

Code snip 3: struct iface

```
struct iface {
        LIST_ENTRY(iface)       entry;
        struct event            hello_timer;
        struct event            wait_timer;
        struct event            lsack_tx_timer;

        LIST_HEAD(, nbr)        nbr_list;
        TAILQ_HEAD(, auth_md)   auth_md_list;
        struct lsa_head         ls_ack_list;

        char                    name[IF_NAMESIZE];
        struct in_addr          addr;
        struct in_addr          dst;
        struct in_addr          mask;
        struct in_addr          abr_id;
        char                   *auth_key;
        struct nbr             *dr;
        struct nbr             *bdr;
        struct nbr             *self;
        struct area            *area;

        u_int32_t               baudrate;
        u_int32_t               dead_interval;
        u_int32_t               ls_ack_cnt;
        u_int32_t               crypt_seq_num;
        unsigned int            ifindex;
        int                     fd;
        int                     state;
        int                     mtu;
        u_int16_t               flags;
        u_int16_t               transmit_delay;
        u_int16_t               hello_interval;
        u_int16_t               rxmt_interval;
        u_int16_t               metric;
        enum iface_type         type;
        enum auth_type          auth_type;
        u_int8_t                auth_keyid;
        u_int8_t                linkstate;
        u_int8_t                priority;
        u_int8_t                passive;
};
```

4.1.4 neighbor

`Struct neighbor` represents the neighbor relationship from the local point of view. To maintain a session successfully a LS retransmission and request list is required plus a list for the database snapshot. Then a few values – `dd_seq_num`, `dd_pending`, `last_rx_options`, `last_rx_bits`, and `master` – are only used in the *EXCHANGE* phase when Database Description packets are transmitted. `peerid` is a unique ID used in all three processes. The `peerid` is used in imsgs to tell the recipient of the message which neighbor is guilty for the just received message. The interface, over which this neighbor is reached, is stored in `iface`. The neighbor structure is per interface so if two routers are connected via two different networks two different neighbor structures will be created for the same router but the structures are added to different interfaces.

Code snip 4: struct nbr

```
struct nbr {
        LIST_ENTRY(nbr)  entry, hash;
        struct event     inactivity_timer;
        struct event     db_tx_timer;
        struct event     lsreq_tx_timer;
        struct event     ls_retrans_timer;
        struct event     adj_timer;

        struct nbr_stats stats;
        struct lsa_head  ls_retrans_list;
        struct lsa_head  db_sum_list;
        struct lsa_head  ls_req_list;

        struct in_addr   addr;
        struct in_addr   id;
        struct in_addr   dr;    /* designated router */
        struct in_addr   bdr;   /* backup DR */

        struct iface    *iface;
        struct lsa_entry*ls_req;
        struct lsa_entry*dd_end;

        u_int32_t        dd_seq_num;
        u_int32_t        dd_pending;
        u_int32_t        peerid;/* unique ID in DB */
        u_int32_t        ls_req_cnt;
        u_int32_t        crypt_seq_num;

        int              state;
        u_int8_t         priority;
        u_int8_t         options;
        u_int8_t         last_rx_options;
        u_int8_t         last_rx_bits;
        u_int8_t         master;
};
```

4.2 Parent Process

4.2.1 Start-up

On start-up ospfd first initialises the log subsystem and fetches the list of available interfaces. This list is required for the next step, the configuration file parsing. The yacc parser used by ospfd is based on bgpds parser which in turn has his origin in the pf parser. Explaining

the parser goes beyond the scope of this paper. Important to know is that the configuration is parsed into a hierarchy of structures.
The configuration consists of a list of areas and every area holds a list of interfaces that are part of this area. Last but not least every interface has a list of neighbors that is dynamically created as soon as a valid Hello packet is received from an other OSPF router on that interface.
After the file got parsed ospfd daemonises and starts the child processes. Beforehand a set of socketpairs – a special sort of pipes – are created. Finally the event handlers are set up, rest of the kroute structures is initialised and the parent reports ready for service.
Meanwhile both children have started. First of all both *chroot(2)* to */var/empty* and drop privileges by switching to the special user *_ospfd*. Before doing that the OSPF engine creates a UNIX local socket for ospfctl and opens the raw IP socket to receive and send packets to the network. After dropping privileges the OSPF engine initialises the different subsystems, sets the event handlers and starts the actual work by kicking the interface finite state machine. The RDE start-up is even simpler as it just has to initialise the internal structures and event handlers.

4.2.2 Routing socket and FIB

The main purpose of the parent process is to maintain the Forward Information Base (FIB) and keep the information in sync with the kernel routing table. This synchronisation is to be done in both directions. Additionally link-state changes and arrival or departure of interfaces are handled via the routing socket as well.
The kroute code maintains two primary data structures. A prefix tree (`kroute`) and an interface tree (`kif`). These two trees are kept in sync with the kernel through the routing socket. On start-up `fetchtable()` loads the kroute tree and `fetchifs()` does the same for the kif tree. Routing changes are tracked by `dispatch_rtmsg()` which handles kroute changes directly but off-loads interface specific messages to `if_change()` and `if_announce()`. To modify the kernel routing table `send_rtmsg()` is used. `send_rtmsg()` translates a `struct kroute` into a `rt_msg` structure expected by the routing socket. The parent process uses `kr_change()` to add or modify routes and `kr_delete()` to remove routes. These changes are propagated to the kernel routing table if needed.

Both the kroute and kif tree are implemented as red-black trees – a balanced binary tree. An API to find, insert and remove nodes is specified to simplify the tree manipulation.
Everytime a route is added or removed to the kroute tree `kr_redistribute()` is called. This function transmits possible candidates for redistribution to the RDE. In the RDE `kif_validate()` verifies that the nexthop is actually reachable. This is a work a round that should be fixed later as it is currently not possible to track and handle newly arriving network interfaces at runtime. Last but not least `kr_show_route()` and `kr_ifinfo()` pass information about kroutes or interfaces to ospfctl.

4.3 OSPF Engine

The finite state machines implemented in ospfd are simple table driven state machines. Any state transition may result in an specific action to be run. The resulting next state can either be a result of the action or is fixed and pre-determined.

4.3.1 Interface state machine

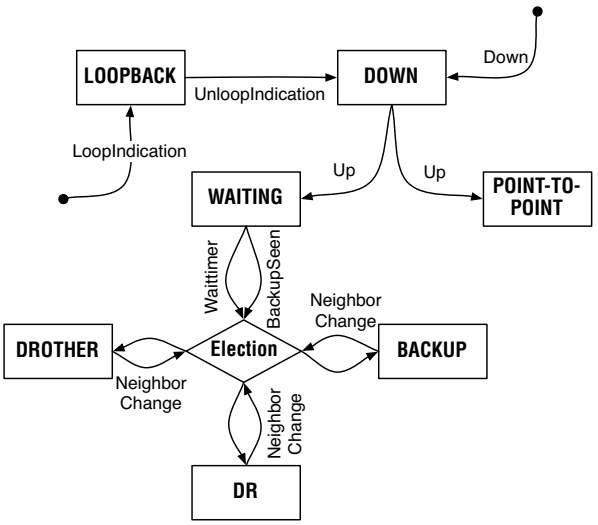

Figure 10: Interface FSM

DOWN

In this state, the lower-level protocols have indicated that the interface is unusable. No protocol traffic at all will be sent or received on such an interface.

LOOPBACK

In this state, the router's interface to the network is looped back. Loopback interfaces are advertised in router-LSAs as single host routes, whose destination is the interface IP address.

POINT-TO-POINT

Point-to-point networks or virtual links enter this state as soon as the interface is operational.

WAITING

Broadcast or NBMA interfaces enter this state when the interface gets operational. While in this state no DR/BDR election is allowed. Receiving and sending of Hello packets is allowed and is used to try to determine the identity of the DR/BDR routers.

DROTHER

The router is neither DR nor BDR on the connected network. In this state the router will only form adjacencies to both the DR and the BDR. All other neighbors will stay in neighbor state *2-WAY*.

BACKUP

The router is the BDR on the connected network segment. If the DR fails it will promote itself to be the new DR. The router forms adjacencies to all neighbors in the network segment.

DR

The router is the DR on the connected network segment. Adjacencies are established to all neighbors in the network segment. Additional duties are origination of a network-LSA for the network node and flooding of LS updates on behalf of all other neighbors.

Only a few events are needed. The events *UP*, *DOWN*, *LOOP*, *UNLOOP* are obvious. The other events *WAITTIMER*, *BACKUPSEEN* and *NEIGHBORCHANGE* are restricted to broadcast and NBMA networks. *WAITTIMER* and *BACKUPSEEN* are used to move out of state *WAITING* by running the election process. The *NEIGHBORCHANGE* event is issued when there is a change in the set of the bidirectional neighbors. This event will force a re-election of the DR and BDR.
The most important actions are `if_act_start()` and `if_act_elect()`. `if_act_start()` sets the correct next state (*POINT-TO-POINT* or *WAITING*), initialises the interface and starts the hello timer to begin with the neighbor discovery process. `if_act_elect()` elects a DR and BDR for a network. This function caused major problems because of subtle bugs and a sloppy written RFC.
First a backup designated router has to be elected.

Code snip 5: BDR election

```
/* elect backup designated router */
LIST_FOREACH(nbr, &iface->nbr_list, entry) {
        if (nbr->priority == 0 ||   /* not electable */
            nbr->state & NBR_STA_PRELIM ||
                                    /* not available */
            nbr->dr.s_addr == nbr->addr.s_addr ||
            nbr == dr)              /* don't elect DR */
                continue;
        if (bdr != NULL) {
                /*
                 * routers announcing themselves as BDR
                 * have higher precedence over those
                 * routers announcing a different BDR.
                 */
                if (nbr->bdr.s_addr == nbr->addr.s_addr) {
                        if (bdr->bdr.s_addr ==
                            bdr->addr.s_addr)
                                bdr = if_elect(bdr, nbr);
                        else
                                bdr = nbr;
                } else if (bdr->bdr.s_addr !=
                    bdr->addr.s_addr)
                        bdr = if_elect(bdr, nbr);
        } else
                bdr = nbr;
}
```

Every neighbor is evaluated, neighbors with a priority of 0 are skipped. Additionally all neighbors that are not in state *2-WAY* or higher plus possible DRs are skipped. From the remaining set a BDR is selected. Routers announcing themselves as BDR have higher precedence so the code checks if the current neighbor is announcing himself BDR. The same thing is done with the current candidate. If both are announcing themselves as BDR or both are not announcing themselves as BDR `if_elect()` elects a new candidate. The helper function `if_elect()` compares two neighbors and returns the preferred one. In the other two cases no additional comparison needs to be done as the next candidate is known.

Code snip 6: DR election

```
/* elect designated router */
LIST_FOREACH(nbr, &iface->nbr_list, entry) {
        if (nbr->priority == 0 ||
            nbr->state & NBR_STA_PRELIM ||
            (nbr != dr &&
            nbr->dr.s_addr != nbr->addr.s_addr))
            /* only DR may be elected check priority too */
                continue;
        if (dr == NULL)
                dr = nbr;
        else
                dr = if_elect(dr, nbr);
}

if (dr == NULL) {
        /* no designate router found use backup DR */
        dr = bdr;
        bdr = NULL;
}
```

Almost the same process is done for electing a DR. Neighbors that are neither in state *2-WAY* or higher or have a priority of 0 are skipped again. Additionally all neighbors that don't announce themselves as DR are skipped as well, with the only exception of the current DR itself. This is done because the election process can be restarted with the current candidates. If no DR was elected the current BDR is promoted DR. If the router is involved in the election it has to redo the election.

Code snip 7: final step of election

```
/*
 * if we are involved in the election (e.g. new DR or no
 * longer BDR) redo the election
 */
if (round == 0 &&
    ((iface->self == dr && iface->self != iface->dr) ||
    (iface->self != dr && iface->self == iface->dr) ||
    (iface->self == bdr && iface->self != iface->bdr) ||
    (iface->self != bdr && iface->self == iface->bdr))) {
        /*
         * Reset announced DR/BDR to calculated one, so
         * that we may get elected in the second round.
         * This is needed to drop from a DR to a BDR.
         */
        iface->self->dr.s_addr = dr->addr.s_addr;
        if (bdr)
                iface->self->bdr.s_addr = bdr->addr.s_addr;
        round = 1;
        goto start;
}
```

Before doing that we set the current candidates in our own structure so that the second round will actually modify the behaviour. It is well possible that some checks are unnecessary or to complex but this current implementation seems to behave correctly and so we keep it as is.

After the election process a bit of housekeeping has to be performed. If the DR or BDR changed, all neighbors have to be checked if the adjacency is still OK. Additionally it may be necessary to join or leave the `AllDRouters` multicast group. In case the router was or is now the DR an updated network-LSA needs to be reflooded.
Getting the DR/BDR election right was one of the most difficult parts of the development. Often unexpected behaviours where found because of small mistakes here and in `recv_hello()`. It took multiple retries and many debugging sessions to get that code where it is now. The poorly written RFC doesn't help much in clarifying the issues.

4.3.2 Neighbor state machine

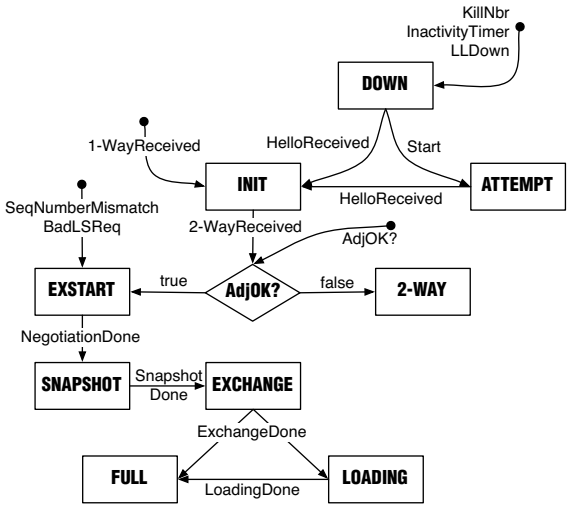

Figure 11: Neighbor FSM

DOWN

A neighbor is considered down if no hello has been received for more than router-dead-time seconds. This is also the initial state of a neighbor.

ATTEMPT

This state is only valid for neighbors attached to NBMA networks. Therefore it is currently unused.

INIT

In this state, a Hello packet has recently been seen from the neighbor. However, bidirectional communication has not yet been established.

2-WAY

The communication between the neighbor and the router is bidirectional. Neighbors will remain in this state if both the router itself and the neighbor are neither DR nor BDR.

EXSTART

This is the first step in creating an adjacency between the two routers. In this state the initial DD sequence number and the master is selected for the upcoming database exchange phase.

SNAPSHOT

This state is actually an extension of the state machine defined by the RFC. Because the LS database is stored in the RDE, a current snapshot of all LSA headers have to be requested by the OSPF engine. The database exchange will start after the snapshot is done.

EXCHANGE

This is the database exchange phase. Additionally all neighbors in state *EXCHANGE* or higher (*LOADING*, *FULL*) participate in the flooding procedure. Starting from this state all packet types can be received inclusive flooded LS updates.

LOADING

The state is only entered if the Link-State Request list is not empty. In that case Link-State Request packets are sent out to fetch the more recent LSAs from the neighbors LS database.

FULL

The two routers are now fully adjacent. The connection will now appear in router-LSAs and network-LSAs. Only in this state real traffic will be routed between the two routers.

4.3.3 Packet reception

The OSPF engine uses the `recv_packet()` libevent handler to receive packets from the raw IP socket. The packet is validated via `ip_hdr_sanity_check()` and `ospf_hdr_sanity_check()`. Some additional length checks are done to ensure that no access outside of the packet is done. It is currently not possible in OpenBSD 3.8 to get the incoming interface via *recvfrom(2)* so we need to find the interface the hard way. `find_iface()` does this job by walking through all configured interfaces and comparing the source address of the incoming packet with the interface address. This is not optimal and will be changed soon. The next step is looking up the neighbor and afterwards the OSPF authentication is run. `nbr_find_id()` takes the unique router ID to get the neighbor structure with all information needed. This is done before `auth_validate()` because the cryptographic authentication method uses a per neighbor specific sequence number to immunize against replay attacks. If necessary `auth_validate()` does the CRC

checksumming of the packet. Finally the packet is passed on according to its packet type to one of the following functions.

recv_hello()

Every hello-interval seconds a Hello packet is sent to all neighbors. On broadcast networks this is done with one multicast packet. The Hello packet is used for neighbor discovery and to maintain neighbor relationships. As first step all the common options need to be compared. If one of hello-interval, router-dead-time, or the stub area flag differs the packet is not accepted. So all routers on a common network must have the same configuration for these values.

Code snip 8: neighbor look up

```
switch (iface->type) {
case IF_TYPE_POINTOPOINT:
case IF_TYPE_VIRTUALLINK:
        /* match router-id */
        LIST_FOREACH(nbr, &iface->nbr_list, entry) {
                if (nbr == iface->self)
                        continue;
                if (nbr->id.s_addr == rtr_id)
                        break;
        }
        break;
case IF_TYPE_BROADCAST:
case IF_TYPE_NBMA:
case IF_TYPE_POINTOMULTIPOINT:
        /* match src IP */
        LIST_FOREACH(nbr, &iface->nbr_list, entry) {
                if (nbr == iface->self)
                        continue;
                if (nbr->addr.s_addr == src.s_addr)
                        break;
        }
        break;
default:
        fatalx("recv_hello: unknown interface type");
}

if (!nbr) {
        nbr = nbr_new(rtr_id, iface, 0);
        /* set neighbor parameters */
        nbr->dr.s_addr = hello.d_rtr;
        nbr->bdr.s_addr = hello.bd_rtr;
        nbr->priority = hello.rtr_priority;
        nbr_change = 1;
}
```

The packet is now accepted and the neighbor is looked up. Depending on the interface type either by router ID or by interface address. If no neighbor could be found a new one is created. A new neighbor is considered a *NEIGHBORCHANGE* and the `nbr_change` flag is set that an interface neighbor change event can be issued later.

Code snip 9: bidirectional or not

```
nbr_fsm(nbr, NBR_EVT_HELLO_RCVD);

while (len >= sizeof(nbr_id)) {
        memcpy(&nbr_id, buf, sizeof(nbr_id));
        if (nbr_id == ospfe_router_id()) {
                /* seen myself */
                if (nbr->state & NBR_STA_PRELIM)
                        nbr_fsm(nbr, NBR_EVT_2_WAY_RCVD);
                break;
        }
        buf += sizeof(nbr_id);
        len -= sizeof(nbr_id);
}

if (len == 0) {
        nbr_fsm(nbr, NBR_EVT_1_WAY_RCVD);
        /* set neighbor parameters */
        nbr->dr.s_addr = hello.d_rtr;
        nbr->bdr.s_addr = hello.bd_rtr;
        nbr->priority = hello.rtr_priority;
        return;
}
```

Multiple neighbor events have to be generated. First of all is the hello received event. Next it is checked if there is already bidirectional communication between the routers. This is done by walking through the list of neighbors in the hello packet and compared it with the own router ID. If no match was found a *1-WAY* received event gets issued. If the match is done the first time – the neighbor is in an embryonic state like *INIT* – a *2-WAY* received event is generated.

Now the scariest part of OpenOSPFD is coming. Handling fast start-ups and the famous interface event *BACKUPSEEN*. This part of the Hello protocol was rewritten multiple times and the result was always some other obscure problem in the election process. In the end OpenOSPFD had to violate the RFC a bit. The RFC is not very clear about how to handle the event *BACKUPSEEN* correctly.

From the RFC:

- *If the neighbor is both declaring itself to be Designated Router (Hello Packet's Designated Router field = Neighbor IP address) and the Backup Designated Router field in the packet is equal to 0.0.0.0 and the receiving interface is in state Waiting, the receiving interface's state machine is scheduled with the event BACKUPSEEN. …*
- *If the neighbor is declaring itself to be Backup Designated Router (Hello Packet's Backup Designated Router field = Neighbor IP address) and the receiving interface is in state Waiting, the receiving interface's state machine is scheduled with the event BACKUPSEEN. …*

Now this sounds simple but it isn't. The first case is not problematic but the second one is. Why? Because it is not known in which order hello packets are received. What does happen if we start an election process and the actual DR neighbor is still in state *1-WAY*? A major confusion is the result. The election process evaluates the BDR as DR and himself as BDR or something like this and the result is a network with too many DR / BDR routers.

Code snip 10: scary fast start-ups

```
if (iface->state & IF_STA_WAITING &&
    hello.d_rtr == nbr->addr.s_addr && hello.bd_rtr == 0)
        if_fsm(iface, IF_EVT_BACKUP_SEEN);

if (iface->state & IF_STA_WAITING &&
    hello.bd_rtr == nbr->addr.s_addr) {
        /*
         * In case we see the BDR make sure that the DR is
         * around with a bidirectional connection
         */
        LIST_FOREACH(dr, &iface->nbr_list, entry)
                if (hello.d_rtr == dr->addr.s_addr &&
                    dr->state & NBR_STA_BIDIR)
                        if_fsm(iface, IF_EVT_BACKUP_SEEN);
}
```

To clear up the situation OpenOSPFD does an additional check. It verifies that the DR has a bidirectional connection to the router and only if that is true a backup seen event is issued. The result is that it may take a bit longer to establish an adjacency and that some initial Database Description packet are dropped. But the confusion of too many DR/BDRs is avoided. The rest of `recv_hello()` is simply here to issue the possible neighbor change events that were detected earlier.

recv_db_description()

While the `send_db_description()` function ended up pretty simple `recv_db_description()` turned out to be more problematic. Usual sanity checking is done first. Afterwards additional checks are performed to verify the MTU and detect possible duplicates because of retransmissions. The MTU check is required by the RFC, the problem is that some OSPF implementations are lying about their MTU and so only bigger MTUs are considered a problem.
The code path is dependent on the neighbor state. Packets received from neighbors in unexpected states are just ignored. This includes state *SNAPSHOT* because during the time the LSA snapshot is done we cannot respond to a received packet. Funnily it is allowed to get Database Description packets in state *INIT*. In that case some kind of super fast start-up needs to be done. It looks like it was simpler to fix the RFC than to fix someone's OSPF implementation. So both the interface and neighbor FSM are kicked and afterwards the new neighbor state has to be checked again. If it is now in state *EXSTART* a fall-through into the next case can be done.
In case *EXSTART* there are two possible scenarios. The first is the reception of a Christmas packet – one with all flags turned on. This is the initial packet and OpenOSPFD has to evaluate if it is master or slave of the database exchange phase. The slave will issue a negotiation done event and sends back a packet with just the *M* bit set.

Code snip 11: EXSTART scenario 1

```
/*
 * check bits: either I,M,MS or only M
 */
if (dd_hdr.bits == (OSPF_DBD_I | OSPF_DBD_M |
    OSPF_DBD_MS)) {
        /* if nbr Router ID is larger than own -> slave */
        if ((ntohl(nbr->id.s_addr)) >
            ntohl(ospfe_router_id())) {
                /* slave */
                nbr->master = 0;
                nbr->dd_seq_num = ntohl(dd_hdr.dd_seq_num);

                /* event negotiation done */
                nbr_fsm(nbr, NBR_EVT_NEG_DONE);
        }
```

The second scenario – a packet with just the *M* bit set, is received. The *M* bit stands for “more” as in more data. The master will finally issue the negotiation done event. So the slave is actually sending valid data ahead of the master. This is a bit strange but we are used to it.

Code snip 12: EXSTART scenario 2

```
} else if (!(dd_hdr.bits & (OSPF_DBD_I | OSPF_DBD_MS))) {
        /* M only case: we are master */
        if (ntohl(dd_hdr.dd_seq_num) != nbr->dd_seq_num) {
                log_warnx("recv_db_description: invalid "
                    "seq num, mine %x his %x",
                    nbr->dd_seq_num,
                    ntohl(dd_hdr.dd_seq_num));
                nbr_fsm(nbr, NBR_EVT_SEQ_NUM_MIS);
                return;
        }
        nbr->dd_seq_num++;

        /* packet may already have data so pass it on */
        if (len > 0) {
                nbr->dd_pending++;
                ospfe_imsg_compose_rde(IMSG_DD,
                    nbr->peerid, 0, buf, len);
        }

        /* event negotiation done */
        nbr_fsm(nbr, NBR_EVT_NEG_DONE);

}
```

Afterwards the actual transfer starts or continues. First of all, packets with invalid flags and options result in a reset of the session (sequence number mismatch event). If the slave receives a duplicate packet it has to resend the last packet. The master does not care about duplicate packets. Actually the master should never see a duplicate – the slave will never send a packet by its own. If the neighbor state is either *LOADING* or *FULL* the only packets received should be duplicates. Anything else is considered an error and the session is reset. Side effect of this is that sending a packet with the Initialise (*I*) bit set can be used to reset a neighbor relationship. Now the sequence number is checked. Only the master is increasing the number so the slave receives packets with the current sequence number plus one. In case of the master the sequence numbers are equal on receive and afterwards the sequence number is increased. Our first implementation was a bit buggy and it took some debugging to find all the small issues like forgetting to bump the sequence number in a specific case.

Code snip 13: synchronising part 1

```
/* forward to RDE and let it decide which LSAs to request
 */
if (len > 0) {
        nbr->dd_pending++;
        ospfe_imsg_compose_rde(IMSG_DD, nbr->peerid, 0,
            buf, len);
}
```

The received LSA headers have to be sent to the RDE where they are compared with the LS database. This resulted in an interesting issue: if the RDE was busy the OSPF engine could move forward and suddenly think that no LSAs have to be requested and move the neighbor directly into state *FULL*. Afterwards the RDE would send some LSAs to request to the OSPF engine but it was too late. To solve this race condition the `dd_pending` counter was added. It gets increased for each sent database description packet.

Code snip 14: synchronising part 2 ospfe_dispatch_rde()

```
nbr->dd_pending--;
if (nbr->dd_pending == 0 && nbr->state & NBR_STA_LOAD) {
        if (ls_req_list_empty(nbr))
                nbr_fsm(nbr, NBR_EVT_LOAD_DONE);
        else
                start_ls_req_tx_timer(nbr);
}
```

When an `IMSG_DD_END` message arrives from the RDE the counter gets decremented. If the counter drops to zero no DD packets are pending. In case that the neighbor state is now *LOADING* we actually hit the race condition and so we have to either move to state *FULL* if the request list is empty or start sending out LS requests. Sometimes running a single daemon as three processes needs some additional work to synchronise the processes. This is a nice example. Finally the next packet is prepared for being sent by `send_db_description()`. If there is nothing left to send and the received packet has no *M* bit set then the exchange phase is mostly done. The slave is finished but the master has to ensure that at least one packet without the *M* bit has been sent and acknowledged. The result is that the slave will always change state before the master. Why should the end of the exchange be less strange than the beginning?

recv_ls_req()

Link-State Requests are simply passed to the RDE but only if the neighbor state is *EXCHANGE* or higher. In all other states Link-State Request packets are ignored.

recv_ls_update()

Link-State Updates are simply dropped if the neighbor is not in state *EXCHANGE* or higher. Otherwise all LSAs are extracted from the packet and sent to the RDE one after the other. While doing that additional length checks are done to guard against buffer overflows.

recv_ls_ack()

Link-State Acknowledgements are only accepted in neighbor state *EXCHANGE* or higher. Otherwise the packet is dropped. Every LSA header included in the packet needs to be roughly validated with `lsa_hdr_check()` and then possibly deleted from the retransmission list. In case the interface is in state *DROTHER* `ls_retrans_list_del()` will be called twice. First it deletes LSAs from the global retransmission list of updates sent to the `AllDRouters` multicast address. Second the per-neighbor queue is purged in case the interface state changed lately.

4.3.4 Packet delivery

send_hello()

`send_hello()` is called by the `if_hello_timer()` function that is run every hello-interval seconds if an interface is not in state *DOWN*. Sending hellos is pretty simple so it is a good example how the buffer framework is used in OpenOSPFD.

Code snip 15: Allocate dynamic buffer

```
/* XXX READ_BUF_SIZE */
if ((buf = buf_dynamic(PKG_DEF_SIZE,
    READ_BUF_SIZE)) == NULL)
        fatal("send_hello");
```

First a dynamic buffer is allocated. Currently a fixed size of `PKG_DEF_SIZE` bytes is used but the buffer is allowed to grow till `READ_BUF_SIZE`. This is not optimal as packets should not be fragmented by OSPF. For Hello packets this is not a big issue because the embedded data is often very small. Other send functions use a different approach by limiting the resulting packet size to the MTU of the corresponding interface.

Code snip 16: Set correct destination

```
dst.sin_family = AF_INET;
dst.sin_len = sizeof(struct sockaddr_in);

switch (iface->type) {
case IF_TYPE_POINTOPOINT:
case IF_TYPE_BROADCAST:
        inet_aton(AllSPFRouters, &dst.sin_addr);
        break;
case IF_TYPE_NBMA:
case IF_TYPE_POINTOMULTIPOINT:
        /* XXX not supported */
        break;
case IF_TYPE_VIRTUALLINK:
        dst.sin_addr = iface->dst;
        break;
default:
        fatalx("send_hello: unknown interface type");
}
```

The outgoing address needs to be determined. For broadcast and point-to-point networks this is the multicast address `AllSPFRouters`. Virtual links are sent as unicast. NBMA and point-to-multipoint are special and currently not supported. For NBMA and point-to-multipoint the packet has to be sent to all neighbors directly and `send_packet()` would be called for every neighbor once.

Code snip 17: create Hello packet

```
/* OSPF header */
if (gen_ospf_hdr(buf, iface, PACKET_TYPE_HELLO))
        goto fail;

/* hello header */
hello.mask = iface->mask.s_addr;
hello.hello_interval = htons(iface->hello_interval);
hello.opts = oeconf->options;
hello.rtr_priority = iface->priority;
hello.rtr_dead_interval = htonl(iface->dead_interval);

if (iface->dr) {
        hello.d_rtr = iface->dr->addr.s_addr;
        iface->self->dr.s_addr = iface->dr->addr.s_addr;
} else
        hello.d_rtr = 0;
```

```
if (iface->bdr) {
        hello.bd_rtr = iface->bdr->addr.s_addr;
        iface->self->bdr.s_addr = iface->bdr->addr.s_addr;
} else
        hello.bd_rtr = 0;

if (buf_add(buf, &hello, sizeof(hello)))
        goto fail;
```

Finally the packet is constructed. First of all the common OSPF header is added. This is done for every packet type and so a helper function `gen_ospf_hdr()` is used. The Hello specific contents are filled in afterwards and added with `buf_add()`.

Code snip 18: Add active neighbors

```
/* active neighbor(s) */
LIST_FOREACH(nbr, &iface->nbr_list, entry) {
        if ((nbr->state >= NBR_STA_INIT) &&
            (nbr != iface->self))
                if (buf_add(buf, &nbr->id,
                    sizeof(nbr->id)))
                        goto fail;
}
```

The Hello packets include a list of all bidirectional neighbors (state *2-WAY* or higher). Again the neighbor IDs are added directly with `buf_add()`. The neighbor ID is stored in network byte order or `htonl()` is used to correctly switch byte order.

Code snip 19: Final step

```
        /* update authentication and calculate checksum */
        if (auth_gen(buf, iface))
                goto fail;

        ret = send_packet(iface, buf->buf, buf->wpos,
            &dst);
        buf_free(buf);
        return (ret);
fail:
        log_warn("send_hello");
        buf_free(buf);
        return (-1);
```

Last is updating authentication and checksum of the outgoing packet. The interface pointer is passed to `auth_gen()` to get the necessary keys and sequence number for the simple and cryptographic authentication. The packet gets sent out via `send_packet()`. Before sending the packet it is necessary to set the outgoing interface for multicast traffic. This is done by `if_set_mcast()` inside of `send_packet()`. Finally the no longer needed buffer is freed.

send_db_description()

`send_db_description()` implements the sending part of the database exchange. It sends out the initial Database Description packet when moving the neighbor state to *EXSTART*.

Code snip 20: Allocate fixed buffer

```
if ((buf = buf_open(nbr->iface->mtu - sizeof(struct ip)))
    == NULL)
        fatal("send_db_description");

/* OSPF header */
if (gen_ospf_hdr(buf, nbr->iface, PACKET_TYPE_DD))
        goto fail;

/* reserve space for database description header */
if (buf_reserve(buf, sizeof(dd_hdr)) == NULL)
        goto fail;
```

Obvious differences to `send_hello()` are the use of `buf_open()` instead of `buf_dynamic()`. `Buf_open()` allocates a fixed size buffer of size `nbr->iface->mtu - sizeof(struct ip)` – which is the maximum packet size that does not get fragmented. Later `buf_reserve()` is used on that buffer to reserve `sizeof(dd_hdr)` bytes. The rest of the packet can be added and later `buf_seek()` can be used to write into the reserved space like this:

Code snip 21: Usage of buf_seek()

```
memcpy(buf_seek(buf, sizeof(struct ospf_hdr),
    sizeof(dd_hdr)), &dd_hdr, sizeof(dd_hdr));
```

The remainder of the function sets up the Database Description header with its bit fields and sequence number. If in state *EXCHANGE*, as many LSA headers as possible are appended. While appending LSA headers one must keep in mind that the cryptographic authentication will append `MD5_DIGEST_LENGTH` bytes to the end of the packet.

send_ls_req()

`send_ls_req()` uses like `send_db_description()` `buf_open()` to get a buffer that doesn't get fragmented. While filling in the requested LSA headers some additional space gets reserved for the possible MD5 sum.

Code snip 22: Filling packet with requests

```
/* LSA header(s), keep space for a possible md5 sum */
for (le = TAILQ_FIRST(&nbr->ls_req_list); le != NULL &&
    buf->wpos + sizeof(struct ls_req_hdr) < buf->max -
    MD5_DIGEST_LENGTH; le = nle) {
        nbr->ls_req = nle = TAILQ_NEXT(le, entry);
        ls_req_hdr.type = htonl(le->le_lsa->type);
        ls_req_hdr.ls_id = le->le_lsa->ls_id;
        ls_req_hdr.adv_rtr = le->le_lsa->adv_rtr;
        if (buf_add(buf, &ls_req_hdr, sizeof(ls_req_hdr)))
                goto fail;
}
```

The rest is straight forward and mostly the same as in `send_hello()`.

send_ls_ack()

Actually we have to start in `ls_ack_tx_timer()` because `send_ls_ack()` is just the last step to send out an ack. `send_ls_ack()` will add the common OSPF header and add the data passed to the function to the packet. The list of acknowledgements is created by `ls_ack_tx_timer()` in a not so nice way and therefore it should not be used as example for other code. Especially as it will be rewritten soon.

send_ls_update()

Sending out LS updates is easy but the retransmission list and flooding procedure are a bit tricky. `send_ls_update()` will just add a LSA to a buffer together with a common OSPF header and send the

results out. But there is one thing that must to be done with the LSA first. It has to be aged with the value of transmit-delay.

Code snip 23: LSA aging

```
pos = buf->wpos;
if (buf_add(buf, data, len))
        goto fail;

/* age LSA before sending it out */
memcpy(&age, data, sizeof(age));
age = ntohs(age);
if ((age += iface->transmit_delay) >= MAX_AGE)
        age = MAX_AGE;
age = htons(age);
memcpy(buf_seek(buf, pos, sizeof(age)), &age, sizeof(age));
```

First the current write position is stored and the LSA is added to the buffer. The LS Age is stored in the first two bytes of the LSA. The `memcpy()` extracts the age because a direct memory access could end on unaligned memory. Then the LSA is aged and written into the buffer with the help of `buf_seek()` and the previously stored position.

4.3.5 Control handling

The handling of control sessions is actually a small UNIX local socket server. There is a listener event (`control_listen()`) that accepts (`control_accept()`) connections and creates a per control connection structure. `control_dispatch_imsg()` reads the request from ospfctl. First the per connection structure are retrieved and then the imsg's sent are extracted. They get either forwarded to the parent, the RDE, or directly answered. Messages forwarded to the other processes will often require a response that needs to be relayed to ospfctl because neither the RDE nor the parent process have access to the socket. Relaying is done by `control_imsg_relay()`. It has to be called for those imsgs that need to get forwarded. This is done in the imsg dispatch functions `ospfe_dispatch_main()` and `ospfe_dispatch_rde()`.

4.4 Route Decision Engine

4.4.1 LS Database

The LS database is implemented as a red-black tree – actually multiple trees exist – one per area and a global one for AS-external-LSAs. The key is the LS-type LS-ID advertising router triple. The LSA is part of a `vertex` that builds a node of the network connectivity graph.

Code snip 24: struct vertex

```
struct vertex {
        RB_ENTRY(vertex) entry;
        TAILQ_ENTRY(vertex) cand;
        struct event     ev;
        struct in_addr   nexthop;
        struct vertex   *prev;
        struct rde_nbr  *nbr;
        struct lsa      *lsa;
        time_t           changed;
        time_t           stamp;
        u_int32_t        cost;
        u_int32_t        ls_id;
        u_int32_t        adv_rtr;
        u_int8_t         type;
        u_int8_t         flooded;
};
```

The vertex contains all necessary information not only for the LS Database but for the SPF calculation too. `entry` and `cand` are used to put the vertex into the red-black tree or into the candidate list respectively. The event `ev` is for a per-LSA entry timeout for aging. Additionally `stamp` is used for aging as well. `changed` is set to the time the last modification was done to the LSA. `ls_id`, `adv_rtr` and `type` are shorthands for the actual values that are stored inside of `lsa`. These are used by the tree search routine. The `flooded` flag should indicate that a LSA was received as part of a flooding. Flooded LSA are locked for `MIN_LS_ARRIVAL` seconds whereas requested LSA are not. `nbr` represents the neighbor from which the LSA was received. `nbr` has nothing to do with the actual originator of the LSA. This is only done to correctly flood out LSAs and sending an acknowledgement back to the neighbor. `prev` is the parent vertex in the SPF tree. It is possible to construct the actual path through the network by following all `prev` pointers. This is used to calculate the `nexthop`. The `nexthop` is the address for forwarding packets to that destination. It is normally the address of the last router-LSA before the root node.

4.4.2 LSA aging

Before using a LSA that is in the DB it normally needs to be aged. This is done by `lsa_age()` with help of the vertex time stamp.

Code snip 25: LSA aging

```
now = time(NULL);
d = now - v->stamp;
/* set stamp so that at least new calls work */
v->stamp = now;

if (d < 0) {
        log_warnx("lsa_age: time went backwards");
        return;
}

age = ntohs(v->lsa->hdr.age);
if (age + d > MAX_AGE)
        age = MAX_AGE;
else
        age += d;

v->lsa->hdr.age = htons(age);
```

Normally it is enough to just add the difference of the current time and stamp. Nonetheless some additional care is needed. First of all `time()` returns the system time and this can be modified by the user. I remember a complete network outage at an ISP because the UNIX time got changed on a Zebra/Quagga router. Afterwards Zebra/Quagga was no longer working until a reboot on the changed machines was performed. So by checking whether the difference is positive it is at least possible to fail in a save way. The other case that needs to be considered is that a LSA may never get older than `MAX_AGE` (1 hour).

4.4.3 Comparing LSA

There are two functions to compare LSA. `lsa_equal()` is similar to a `memcmp()` but compares a bit more. One thing is important to note: LSA with age `MAX_AGE` are never considered equal. This comes from the fact that `lsa_equal()` is mostly used to determine if a recalculation of the SPF tree is required or for similar situations. In that context LSAs with an age of `MAX_AGE` are always special and it is OK to force an update.
The other compare function is `lsa_newer()` and implements the RFC specification of newer, equal and older LSA. It works similar to other compare functions by returning -1 if the first LSA is older, 1 if newer and 0 if equal to the second LSA passed. The function compares the sequence number, the LS checksum, and the LS age. Once again a bit care needs to be taken when comparing ages.

Code snip 26: Comparing ages

```
a16 = ntohs(a->age);
b16 = ntohs(b->age);

if (a16 >= MAX_AGE && b16 >= MAX_AGE)
        return (0);
if (b16 >= MAX_AGE)
        return (-1);
if (a16 >= MAX_AGE)
        return (1);

i = b16 - a16;
if (abs(i) > MAX_AGE_DIFF)
        return (i > 0 ? 1 : -1);

return (0);
```

If both LSA are at age `MAX_AGE` they are considered equal. If only one has age `MAX_AGE` that one is newer and last but not least the LS ages need to be at least `MAX_AGE_DIFF` (15 minutes) apart to be not considered equal.

4.4.4 LSA refresh

All `LS_REFRESH_TIME` seconds a LSA needs to be refreshed by its originator. The age is reset to the initial value and the sequence number is bumped. After modifying the LSA the checksum has to be recalculated. The LSA is flooded and a new timeout event is registered. Non self originated LSA have the same timer running but with `MAX_AGE` instead of `LS_REFRESH_TIME`. If the timer fires the LSA will be deleted from the LS database by flooding it out with age `MAX_AGE`. How to delete LSA will be explained later as it is fairly complex.

4.4.5 LSA merging

If a self originated LSA changes, for example because a neighbor relationship is established or lost, an updated LSA needs to be reflooded. `lsa_merge()` takes care of replacing the LSA in the database with the new one and sets the LS sequence number of the new LSA to the current used number.

Code snip 27: First set sequence number

```
if (v == NULL) {
        lsa_add(nbr, lsa);
        rde_imsg_compose_ospfe(IMSG_LS_FLOOD, nbr->peerid,
            0, lsa, ntohs(lsa->hdr.len));
        return;
}

/*
 * set the seq_num to the current one.
 * lsa_refresh() will do the ++
 */
lsa->hdr.seq_num = v->lsa->hdr.seq_num;
/* recalculate checksum */
len = ntohs(lsa->hdr.len);
lsa->hdr.ls_chksum = 0;
lsa->hdr.ls_chksum = htons(iso_cksum(lsa, len,
    LS_CKSUM_OFFSET));
```

Sure if there was no LSA in the database in the first place there is no need to merge. It is enough to just add and flood the LSA. When changing the sequence number the checksum has to be recalculated. The sequence number is only set to the current value because there is no need to increase it already. Especially if `lsa_merge()` is used to remove a self originated LSA from the database there is no need to rise the sequence number, it is sufficient to set the age to `MAX_AGE`.

Code snip 28: Then overwrite and reflood if necessary

```
/*
 * compare LSA; most header fields are equal
 * so don't check them
 */
if (lsa_equal(lsa, v->lsa)) {
        free(lsa);
        return;
}

/* overwrite the lsa all other fields are unaffected */
free(v->lsa);
v->lsa = lsa;
start_spf_timer();

/* set correct timeout for reflooding the LSA */
now = time(NULL);
timerclear(&tv);
if (v->changed + MIN_LS_INTERVAL >= now)
        tv.tv_sec = MIN_LS_INTERVAL;
evtimer_add(&v->ev, &tv);
```

Now `lsa_equal()` is used to determine whether to actually reflood the LSA. If the LSA did not change there is nothing to modify and we're done. Otherwise the LSAs are exchanged and a SPF recalculation is issued. Finally the reflooding is prepared. This is done via a timer because it is not allowed to send out updates faster than `MIN_LS_INTERVAL` (5) seconds.

4.4.6 lsa_self()

Identifying self originated LSA is an important task. This comes from the fact that if a router leaves the network the other routers will not remove the LSAs of this router until the LS age hits `MAX_AGE`. If the router joins the network again – after a reboot for example – the old LSAs are still floating around. So it is the routers duty to detect those old self originated LSAs and renew them or remove them from the database. This task is done by `lsa_self()`.

Code snip 29: Detect self originated LSA

```
if (nbr->self)
        return (0);

if (rde_router_id() == new->hdr.adv_rtr)
        goto self;

if (new->hdr.type == LSA_TYPE_NETWORK)
        LIST_FOREACH(iface, &nbr->area->iface_list, entry)
            if (iface->addr.s_addr == new->hdr.ls_id)
                    goto self;
return (0);
```

First of all the newly received LSA (new) gets classified. If the router ID is the same or if an interface address matches the LS ID of a network-LSA the LSA is considered self originated.

Code snip 30: Remove or update

```
self:
if (v == NULL) {
        /*
         * LSA is no longer announced, remove by premature
         * aging. The problem is that new may not be
         * altered so a copy needs to be added to the LSA
         * DB first.
         */
        if ((dummy = malloc(ntohs(new->hdr.len))) == NULL)
                fatal("lsa_self");
        memcpy(dummy, new, ntohs(new->hdr.len));
        dummy->hdr.age = htons(MAX_AGE);
        /*
         * The clue is that by using the remote nbr as
         * originator the dummy LSA will be reflooded via
         * the default timeout handler.
         */
        lsa_add(rde_nbr_self(nbr->area), dummy);
        return (1);
}

/*
 * LSA is still originated, just reflood it. But we need to
 * create a new instance by setting the LSA sequence number
 * equal to the one of new and calling lsa_refresh().
 * Flooding will be done by the caller.
 */
v->lsa->hdr.seq_num = new->hdr.seq_num;
lsa_refresh(v);
return (1);
```

In case of a self originated LSA there are two cases. The first one is that the LSA is no longer announced. In that case the LSA gets added to the Database with a LS age of MAX_AGE. The database code will then reflood the LSA as soon as possible and by doing that removing it from the database. There is no other way in doing this because removing LSAs is a complex task that only works if the LSA is in the database. The other case is much simpler because there is already a self originated LSA in the local database but the sequence number is lower then the new one. In this case the sequence number is bumped like in the lsa_merge() case and lsa_refresh() is called to flood the LSA.

4.4.7 LSA check

Before even accepting a LS update the embedded LSA has to be verified. Once again lengths are compared and especially the ISO checksum is verified. Additionally the LS age and sequence number are checked to be in a valid range. Per LS type checks follow the generic ones. It is verified that the packet has the right size for this type and that values like the metric – which is a 24bit value stored as 32bit integer is in the correct range. AS-external-LSAs that are sent to stub areas get silently discarded. At the end the LS age is checked and if it is MAX_AGE some special care needs to be taken.

Code snip 31: MAX_AGE handling

```
if (lsa->hdr.age == htons(MAX_AGE) &&
    !nbr->self && lsa_find(area, lsa->hdr.type,
    lsa->hdr.ls_id, lsa->hdr.adv_rtr) == NULL &&
    !rde_nbr_loading(area)) {
        /*
         * if no neighbor in state Exchange or Loading
         * ack LSA but don't add it. Needs to be a direct
         * ack.
         */
        rde_imsg_compose_ospfe(IMSG_LS_ACK, nbr->peerid, 0,
            &lsa->hdr, sizeof(struct lsa_hdr));
        return (0);
}
```

If the LS age is MAX_AGE and the LSA is not in the database there is actually no need to add the LSA to the database. However this is a fallacy, there are some additional checks required. The RFC mentions that if a neighbor is currently establishing an adjacency – state *EXCHANGE* or *LOADING* – no short-cuts are allowed. Additionally self originated LSA generated by the OSPF engine have to be passed. Therefore nbr->self is tested. If all conditions are met the LSA will not be added. Instead only a direct acknowledgement is sent back.

4.4.8 Deleting LSA

Deleting something from a replicated distributed database is not a trivial task. Especially if there is no LS remove packet type. Removing is done via the LS age. LSA with LS age MAX_AGE are ready to be removed from the database. Especially for OpenOSPFD removing LSAs is even more complicated. To remove a LSA it first has to be reflooded and all neighbors have to acknowledge the reception before removing it from the database. In OpenOSPFD the database and the retransmission logic are in two different processes so additional IPC is needed. If the RDE tries to delete the LSA either because it exceeds the MAX_AGE age or because of premature aging – used to clean the database from no longer valid LSAs – it simply sets the age to MAX_AGE and sends a flood request to the OSPF engine. The OSPF engine will then start the flooding procedure. The LSA is added to the LSA cache and the different retransmission lists refer to the cached LSA. If the last reference to the cached object drops the following happens:

Code snip 32: lsa_cache_put()

```
void
lsa_cache_put(struct lsa_ref *ref, struct nbr *nbr)
{
        if (--ref->refcnt > 0)
                return;

        if (ntohs(ref->hdr.age) >= MAX_AGE)
                ospfe_imsg_compose_rde(IMSG_LS_MAXAGE,
                    nbr->peerid, 0, ref->data,
                    sizeof(struct lsa_hdr));

        free(ref->data);
        LIST_REMOVE(ref, entry);
        free(ref);
}
```

The LS age is compared with MAX_AGE and if true a IMSG_LS_MAXAGE is sent back to the RDE. In the RDE the message is received and verified. If something is incorrect the RDE bombs out.

Code snip 33: IMSG_LS_MAXAGE handling

```
case IMSG_LS_MAXAGE:
        nbr = rde_nbr_find(imsg.hdr.peerid);
        if (nbr == NULL)
                fatalx("rde_dispatch_imsg: "
                    "neighbor does not exist");

        if (imsg.hdr.len != IMSG_HEADER_SIZE +
            sizeof(struct lsa_hdr))
                fatalx("invalid size of OE request");
        memcpy(&lsa_hdr, imsg.data, sizeof(lsa_hdr));

        if (rde_nbr_loading(nbr->area))
                break;

        v = lsa_find(nbr->area, lsa_hdr.type,
            lsa_hdr.ls_id, lsa_hdr.adv_rtr);
        if (v == NULL)
                db_hdr = NULL;
        else
                db_hdr = &v->lsa->hdr;

        /*
         * only delete LSA if the one in the db isn't newer
         */
        if (lsa_newer(db_hdr, &lsa_hdr) <= 0)
                lsa_del(nbr, &lsa_hdr);
        break;
```

If there is still a neighbor in state *EXCHANGE* or *LOADING* the LSA may not be removed. It is possible that the neighbor may request that LSA just a bit later. Now the LSA is searched in the database and the entry of the database is compared with the LSA that should be removed. If the database entry is newer the entry will not be removed else it would get finally removed from the database and freed.

4.4.9 SPF and RIB calculation

The SPF calculation is still a large construction area. The code should be split up as some steps are not necessary in all cases. Especially on ABRs this is not optimal and creates a lot of superfluous load. Worth knowing: RIB and FIB are terms from BGP and got inherited into OpenOSPFD. RIB is the Routing Information Base and FIB is the Forwarding Information Base. The FIB is mostly the kernel routing table and is stripped from unneeded ballast whereas the RIB contains all additional protocol specific informations.
To calculate the routing table three calculations are performed. First the SPF tree gets built. Then the local LSAs are added to the RIB and finally the AS-external-LSAs are inserted.

Code snip 34: SPF calculation

```
/* calculate SPF tree */
do {
        /* loop links */
        for (i = 0; i < lsa_num_links(v); i++) {
                switch (v->type) {
                case LSA_TYPE_ROUTER:
                        rtr_link = get_rtr_link(v, i);
                        switch (rtr_link->type) {
                        case LINK_TYPE_STUB_NET:
                                /* skip */
                                continue;
                        case LINK_TYPE_POINTTOPOINT:
                        case LINK_TYPE_VIRTUAL:
                                /* find router LSA */
                                w = lsa_find(area,
                                    LSA_TYPE_ROUTER,
                                    rtr_link->id,
                                    rtr_link->id);
                                break;
                        case LINK_TYPE_TRANSIT_NET:
                                /* find network LSA */
                                w = lsa_find_net(area,
                                    rtr_link->id);
                                break;
                        default:
                                fatalx("spf_calc: "
                                    "invalid link type");
                        }
                        break;
                case LSA_TYPE_NETWORK:
                        net_link = get_net_link(v, i);
                        /* find router LSA */
                        w = lsa_find(area, LSA_TYPE_ROUTER,
                            net_link->att_rtr,
                            net_link->att_rtr);
                        break;
                default:
                        fatalx("spf_calc: "
                            "invalid LSA type");
                }

                ...
                cand_list_add(w);
        }
        /* get next vertex */
        v = cand_list_pop();
        w = NULL;
} while (v != NULL);
```

The loops starts at the root vertex and moves through one vertex after another. After a vertex is selected all next vertices that are connected to this vertex are extracted and added to the candidate list. After all vertices are added the one with the lowest cost is popped from the list and the loops starts over with this vertex. Before a vertex is added to the candidate list it is verified that the connection is still valid.

Code snip 35: the three dots in the previous snippet

```
if (w == NULL)
        continue;

if (w->lsa->hdr.age == MAX_AGE)
        continue;

if (!linked(w, v))
        continue;

if (v->type == LSA_TYPE_ROUTER)
        d = v->cost + ntohs(rtr_link->metric);
else
        d = v->cost;

if (cand_list_present(w)) {
        if (d > w->cost)
                continue;

        if (d < w->cost) {
                w->cost = d;
                w->prev = v;
                calc_next_hop(w, v);
                /*
                 * need to readd to candidate list
                 * because the list is sorted
                 */
                TAILQ_REMOVE(&cand_list, w, cand);
        }
} else if (w->cost == LS_INFINITY && d < LS_INFINITY) {
        w->cost = d;
        w->prev = v;
        calc_next_hop(w, v);
}
```

On leaf nodes – w is NULL – there is nothing to do. If the next vertex has an age of MAX_AGE it is no longer considered valid and dropped. The connection between the two vertices has to be bidirectional and this is checked by

linked(). The next steps calculate the cost to the new vertex w. There is one important thing to note: only links into a network have a cost but links from the network to the router have no cost. The result is that modifying the cost of an interface will often not change incoming traffic flow only outgoing traffic may be rerouted due to the change. Before adding a vertex to the candidate list it is necessary to check if the vertex is already on the list. If it is, then the calculated cost is compared with the current one. The new path must be shorter than the current selected one. In that case the cost and the prev pointer are modified and the nexthop is recalculated. The vertex is also removed from the candidate list and later added back to keep the list sorted. If the vertex is not on the candidate list then cost and prev pointer are initialised and the nexthop is calculated. Finally the new candidate is added to the list of candidates.
Now the RIB needs to be built. To start the area specific routes are added. First of all, all LSAs with LS age MAX_AGE, a cost of LS_INFINITY, or a zero nexthop address are skipped. They are invalid. All valid network-LSAs are added to the RIB and all router-LSAs for ABRs and ASBRs are added as well. Summary-LSAs are put into the RIB. On ABRs only for area 0. On non ABRs there is no limitation. A summary-LSA is only valid if the ABR was previously added to the RIB. The last step is adding of the AS-external routes to the RIB. This is done only once and not for every area. Similarly to summary-LSAs AS-external-LSAs will do a look-up of the ASBR router and if the router is not found the route is considered invalid. When updating the RIB with rt_update() some order is retained. Intra-area routes (router and network-LSAs) have highest priority, inter-area routers (summary-LSAs) follow and Type1 and Type2 AS-external routes have the lowest priority. So if a network is added multiple times that order will favour intra-area traffic over inter-area or external routes.

4.5 Workflow

4.5.1 Flooding

The flooding and retransmission of LS updates is entirely done in the OSPF engine. The RDE sends a IMSG_LS_FLOOD imsg with the peer ID of the neighbor from which the update was initially received. The OSPF engine uses that information to flood out the LS update to all affected networks.

Code snip 36: flooding part 1

```
ref = lsa_cache_add(imsg.data, l);

if (lsa_hdr.type == LSA_TYPE_EXTERNAL) {
        /*
         * flood on all areas but stub areas and
         * virtual links
         */
        LIST_FOREACH(area, &oeconf->area_list, entry) {
            if (area->stub)
                    continue;
            LIST_FOREACH(iface, &area->iface_list,
                entry) {
                    noack += lsa_flood(iface, nbr,
                        &lsa_hdr, imsg.data, l);
            }
        }
} else {
        /*
         * flood on all area interfaces on
         * area 0.0.0.0 include also virtual links.
         */
        area = nbr->iface->area;
        LIST_FOREACH(iface, &area->iface_list, entry) {
                noack += lsa_flood(iface, nbr,
                    &lsa_hdr, imsg.data, l);
        }
}
```

Before starting the flooding decision process the LS update is added to the LSA cache. Later, if the LSA is added to different retransmission queues, only a reference to the LSA cache is retained. Depending on the LS type it must be flooded to all areas (AS-external-LSA) or only to the current area (all other LSAs). lsa_flood() is doing the per interface specific part of the flooding. More about that a bit later.

Code snip 37: flooding part2

```
/* remove from ls_req_list */
le = ls_req_list_get(nbr, &lsa_hdr);
if (!(nbr->state & NBR_STA_FULL) && le != NULL) {
        ls_req_list_free(nbr, le);
        /*
         * XXX no need to ack requested lsa
         * the problem is that the RFC is very
         * unclear about this.
         */

        noack = 1;
}

if (!noack && nbr->iface != NULL &&
    nbr->iface->self != nbr) {
        if (!(nbr->iface->state & IF_STA_BACKUP) ||
            nbr->iface->dr == nbr) {
                /* delayed ack */
                lhp = lsa_hdr_new();
                memcpy(lhp, &lsa_hdr, sizeof(*lhp));
                ls_ack_list_add(nbr->iface, lhp);
        }
}

lsa_cache_put(ref, nbr);
break;
```

After flooding the LSA out on all affected interfaces an acknowledgement has to be sent back to the initial sender of the LS update. In some cases there is no requirement to send a LS acknowledge back. One of those cases are requested LSAs – sending back a LSA ack to an explicitly requested LSA does not make much sense. However the RFC is not very clear about this fact. So let's be prepared for some broken implementations out there. The last step adds the LSA to the LS acknowledge list so that a, possibly delayed, acknowledge can be sent back. This is only done if an ack is required, the neighbor where the ack is sent to is not ourselves and additionally no acks were sent from the BDR to the DR. Finally the acquired reference of the LSA gets passed back. Reference counting makes careful programming a necessity to avoid missing a reference change somewhere.

lsa_flood()

As mentioned earlier `lsa_flood()` is used for flooding on a per interface scope. In particular it loops over all neighbors and decides if it has to send the update to this neighbor or not.

Code snip 38: neighbor loop part 1

```
LIST_FOREACH(nbr, &iface->nbr_list, entry) {
        if (nbr == iface->self)
                continue;
        if (!(nbr->state & NBR_STA_FLOOD))
                continue;
```

First of all `self` is skipped. Then all neighbors which are not available for flooding – their state is neither *FULL* nor *LOADING* nor *EXCHANGE* – are skipped as well.

Code snip 39: neighbor loop part 2

```
if (iface->state & IF_STA_DROTHER && !queued)
        if ((le = ls_retrans_list_get(iface->self,
            lsa_hdr)))
                ls_retrans_list_free(iface->self, le);

if ((le = ls_retrans_list_get(nbr, lsa_hdr)))
        ls_retrans_list_free(nbr, le);
```

Afterwards the retransmission lists are searched for an older LS update for the same LSA. If an older LSA is found it is removed and replaced later with the new one. A special queue is used for interfaces with state *DROTHER* as explained later on. Because only one queue is used, redoing this check after the LSA got queued once results in unexpected behaviour. So this case is protected by the `!queued` check.

Code snip 40: neighbor loop part 3

```
if (!(nbr->state & NBR_STA_FULL) &&
    (le = ls_req_list_get(nbr, lsa_hdr)) != NULL) {
        r = lsa_newer(lsa_hdr, le->le_lsa);
        if (r > 0) {
                /* to flood LSA is newer than requested */
                ls_req_list_free(nbr, le);
                /* new needs to be flooded */
        } else if (r < 0) {
                /* to flood LSA is older than requested */
                continue;
        } else {
                /* LSA are equal */
                ls_req_list_free(nbr, le);
                continue;
        }
}
```

If the adjacency is not yet full, the LS request list is examined. If a LSA is found we know the exact LSA the neighbor has in his database. So if the LSA in the request list is older than the new one, the requested one is removed and the new one will be flooded. Otherwise if the LSA is older than the requested one, there is no need to flood it to the neighbor and the request list is left alone so that the newer LSA of that neighbor is requested later. In case both LSAs are equal there is no need to request the LSA anymore. There is also no need to flood the LSA to that neighbor.

Code snip 41: neighbor loop part 4

```
if (nbr == originator) {
        dont_ack++;
        continue;
}

/* non DR or BDR router keep all lsa in one retrans list */
if (iface->state & IF_STA_DROTHER) {
        if (!queued)
                ls_retrans_list_add(iface->self, data);
        queued = 1;
} else {
        ls_retrans_list_add(nbr, data);
        queued = 1;
}
```

If the current neighbor is the initial sender of this LS update there is high chances that no ack has to be sent back. This decision is done later. At least there is also no need to flood the LS update back to this router.
Finally the LS update or actually a reference to the LS update is added to the retransmission queue. Depending on the interface state, different queues are chosen. If the interface is not in state *DROTHER* it will be added to the neighbor retransmission list. In case of *DROTHER* only one global queue is used because all updates go to the `AllDRouters` address. For this special case `iface->self` is “abused”. Because only one queue is used it is important to protect the queue from multiple adds. Currently there is a known feature in the queuing behaviour of OpenOSPFD that needs to be solved. In case of the router being BDR it will queue the update to all neighbors on that interface including the DR. The DR therefore is required to send an acknowledge to the BDR. This will not happen and so one retransmission is done from the BDR to the DR and the DR will then answer with a direct acknowledge. This is unnecessary and no updates to the DR should be queued unless they are self originated or from a different interface.

Code snip 42: sending LS update

```
if (!queued)
        return (0);

if (iface == originator->iface &&
    iface->self != originator) {
        if (iface->dr == originator ||
            iface->bdr == originator)
                return (0);
        if (iface->state & IF_STA_BACKUP)
                return (0);
        dont_ack++;
}

/* flood LSA but first set correct destination */
switch (iface->type) {
case IF_TYPE_POINTOPOINT:
        inet_aton(AllSPFRouters, &addr);
        send_ls_update(iface, addr, data, len);
        break;
case IF_TYPE_BROADCAST:
        if (iface->state & IF_STA_DRORBDR)
                inet_aton(AllSPFRouters, &addr);
        else
                inet_aton(AllDRouters, &addr);
        send_ls_update(iface, addr, data, len);
        break;
...
}

return (dont_ack == 2);
```

After inspecting every neighbor and adding LSA references to the retransmission lists an initial flooding gets sent out. If nothing got queued there is no reason to send the LSA, do a return. In the other cases we send the update to the correct address. For point-to-point links it is always `AllSPFRouters`. For broadcast networks it is either `AllSPFRouters` or `AllDRouters` to multicast the update to the correct group. All other interface types use unicast to send the updates. Before sending out the LS update a special check is done mostly for broadcast and NBMA networks. In case the originator of the initial LS update is on the now outgoing interface more checks have to be done. First of all if the originator is DR or BDR there is no need to send an update. The actual flooding was already done by the DR respectively BDR. Additionally if the router itself is BDR there is no need to flood the network. This will be done by the DR. If none of these two tests where true it is now clear that no acknowledgement needs to be sent back. Therefore `dont_ack` is bumped a second time and so `lsa_flood()` will return true.

4.5.2 Retransmission Lists and LSA Cache

Now lets have a look at the retransmission lists. All other lists – acknowledge, request, and database descriptor list – are implemented in a similar way. The retransmission list is a bit more complex because of the LSA cache. To add a LS update to the request list `ls_retrans_list_add()` is used.

Code snip 43: ls_retrans_list_add()

```
if ((ref = lsa_cache_get(lsa)) == NULL)
        fatalx("King Bula sez: somebody forgot to
lsa_cache_add");

if ((le = calloc(1, sizeof(*le))) == NULL)
        fatal("ls_retrans_list_add");

le->le_ref = ref;
TAILQ_INSERT_TAIL(&nbr->ls_retrans_list, le, entry);

if (!evtimer_pending(&nbr->ls_retrans_timer, NULL)) {
        timerclear(&tv);
        tv.tv_sec = nbr->iface->rxmt_interval;

        if (evtimer_add(&nbr->ls_retrans_timer, &tv) == -1)
                log_warn("ls_retrans_list_add: evtimer_add
failed");
}
```

First of all a LSA cache reference is acquired via `lsa_cache_get()`. If this call fails we have an internal program error and the OSPF engine has no way to recover from that. The reference is added to a list element that in turn is added to the retransmission list. And if there is no timer pending a new retransmission timer is started.
Removing works in a similar way. First the correct entry is searched with the help of `ls_retrans_list_get()` and afterwards it gets freed if the LSA was the same. `ls_retrans_list_get()` uses the known LSA triple to identify a LSA.

Code snip 44: ls_retrans_list_free()

```
void
ls_retrans_list_free(struct nbr *nbr, struct lsa_entry *le)
{
        TAILQ_REMOVE(&nbr->ls_retrans_list, le, entry);

        lsa_cache_put(le->le_ref, nbr);
        free(le);
}
```

`ls_retrans_list_free()` will not only unlink the LSA from the request list but hands the LSA cache reference back by calling `lsa_cache_put()`. Again it is important to take care of those references.

How does this LSA cache work?
The LSA cache is nothing more than a hash list. A simple hash is built over the LSA header and used to find the correct hash bucket. In the LSA cache a LSA is identified not only by LS type, LS ID, and advertising router. The sequence number and LS checksum is compared as well. To find a LSA in the cache the internal `lsa_cache_look()` function is used.
`lsa_cache_get()` returns a new reference to an existing LSA.

Code snip 45: lsa_cache_get()

```
struct lsa_ref *
lsa_cache_get(struct lsa_hdr *lsa_hdr)
{
        struct lsa_ref  *ref;

        ref = lsa_cache_look(lsa_hdr);
        if (ref)
                ref->refcnt++;

        return (ref);
}
```

This function is very simple and the only important step is not to forget to bump the reference count.
`lsa_cache_add()` works very similar to `lsa_cache_get()`. Again `lsa_cache_look()` is used to find already added LSAs. In that case a bump of the reference count is enough. Else a new reference object gets allocated and filled in. There is a timestamp included to age the LSA when it is sent out. The initial reference count is set to one because a reference is immediately returned to the caller.

Code snip 46: lsa_cache_add()

```
struct lsa_ref *
lsa_cache_add(void *data, u_int16_t len)
{
        struct lsa_cache_head*head;
        struct lsa_ref  *ref, *old;

        if ((ref = calloc(1, sizeof(*ref))) == NULL)
                fatal("lsa_cache_add");
        memcpy(&ref->hdr, data, sizeof(ref->hdr));

        if ((old = lsa_cache_look(&ref->hdr))) {
                free(ref);
                old->refcnt++;
                return (old);
        }

        if ((ref->data = malloc(len)) == NULL)
                fatal("lsa_cache_add");
        memcpy(ref->data, data, len);
        ref->stamp = time(NULL);
        ref->len = len;
        ref->refcnt = 1;
```

```
        head = lsa_cache_hash(&ref->hdr);
        LIST_INSERT_HEAD(head, ref, entry);
        return (ref);
}
```

lsa_cache_put() was only roughly explained in the MAX_AGE handling. First the reference count is decreased and if it hits zero the cache is no longer referenced and can be freed. Now the known MAX_AGE dance comes. Sending back an IMSG_LS_MAXAGE if the LSA has an age of MAX_AGE to make it possible to remove the LSA from the LS DB. Afterwards the cache object is cleaned and removed.

Code snip 47: lsa_cache_put()

```
void
lsa_cache_put(struct lsa_ref *ref, struct nbr *nbr)
{
        if (--ref->refcnt > 0)
                return;

        if (ntohs(ref->hdr.age) >= MAX_AGE)
                ospfe_imsg_compose_rde(IMSG_LS_MAXAGE,
                    nbr->peerid, 0, ref->data,
                    sizeof(struct lsa_hdr));

        free(ref->data);
        LIST_REMOVE(ref, entry);
        free(ref);
}
```

4.5.3 Self originated LSA

There are three kinds of self originated LSAs. First router and network-LSAs – those are generated in the OSPF engine. Then AS-external-LSAs which are generated in the RDE with the help of the parent process. Finally on ABRs summary-LSAs are generated – this happens in the RDE as well.
To create a self originated LSA in the OSPF engine and commit it to the LS DB in the RDE is a bit tricky. Let's have a look at orig_net_lsa() because it is a lot simpler than orig_rtr_lsa().

Code snip 48: originate network-LSA

```
if ((buf = buf_dynamic(sizeof(lsa_hdr), READ_BUF_SIZE)) ==
    NULL)
        fatal("orig_net_lsa");

/* reserve space for LSA header and LSA Router header */
if (buf_reserve(buf, sizeof(lsa_hdr)) == NULL)
        fatal("orig_net_lsa: buf_reserve failed");

/* LSA net mask and then all fully adjacent routers */
if (buf_add(buf, &iface->mask, sizeof(iface->mask)))
        fatal("orig_net_lsa: buf_add failed");

/* fully adjacent neighbors + self */
LIST_FOREACH(nbr, &iface->nbr_list, entry)
        if (nbr->state & NBR_STA_FULL) {
                if (buf_add(buf, &nbr->id,
                    sizeof(nbr->id)))
                        fatal("orig_net_lsa: "
                            "buf_add failed");
                num_rtr++;
        }

if (num_rtr == 1) {
        /*
         * non transit net therefor no need to generate
         * a net lsa
         */
        buf_free(buf);
        return;
}

/* LSA header */
if (iface->state & IF_STA_DR)
        lsa_hdr.age = htons(DEFAULT_AGE);
else
        lsa_hdr.age = htons(MAX_AGE);

lsa_hdr.opts = oeconf->options;/* XXX */
lsa_hdr.type = LSA_TYPE_NETWORK;
lsa_hdr.ls_id = iface->addr.s_addr;
lsa_hdr.adv_rtr = oeconf->rtr_id.s_addr;
lsa_hdr.seq_num = htonl(INIT_SEQ_NUM);
lsa_hdr.len = htons(buf->wpos);
lsa_hdr.ls_chksum = 0;/* updated later */
memcpy(buf_seek(buf, 0, sizeof(lsa_hdr)), &lsa_hdr,
    sizeof(lsa_hdr));

chksum = htons(iso_cksum(buf->buf, buf->wpos,
    LS_CKSUM_OFFSET));
memcpy(buf_seek(buf, LS_CKSUM_OFFSET, sizeof(chksum)),
    &chksum, sizeof(chksum));

imsg_compose(ibuf_rde, IMSG_LS_UPD, iface->self->peerid, 0,
    -1, buf->buf, buf->wpos);

buf_free(buf);
```

Once again the buf API is used. First space for the header is reserved then the network mask is added and finally a list of all fully adjacent routers is added. The router itself needs to be added as well but this is no problem because of the special self neighbor. If there is no other OSPF router on the network it is not necessary to create a network-LSA. A stub network entry in the router-LSA will do the job. In that case the buffer gets freed and the function returns. Otherwise the LSA header has to be built. First the correct age is set. To remove a network-LSA the age is set to MAX_AGE else the initial DEFAULT_AGE is used. Other important fields are LS type, LS ID and advertising router. Also the sequence number has to be set but the correct instance number is only known by the RDE. The RDE uses lsa_merge() later on to merge this LSA into the database and lsa_merge() will take care of the sequence number – so here we set it just to the initial value. Copy the header into the buffer, calculate the checksum and finally send this self originated LSA with the peerid of the special neighbor self to the RDE.
Originating a router-LSA is done in a similar way. It is just more complex because many additional informations are added in the router-LSA. One tricky part is setting the correct router flags.

Code snip 49: originate router-LSA

```
/* LSA router header */
lsa_rtr.flags = 0;
/*
 * Set the E bit as soon as an as-ext lsa may be
 * redistributed, only setting it in case we redistribute
 * something is not worth the fuss.
 */
if (oeconf->redistribute_flags &&
    (oeconf->options & OSPF_OPTION_E))
        lsa_rtr.flags |= OSPF_RTR_E;

border = area_border_router(oeconf);

if (border != oeconf->border) {
        oeconf->border = border;
        orig_rtr_lsa_all(area);
}

if (oeconf->border)
        lsa_rtr.flags |= OSPF_RTR_B;
if (virtual)
        lsa_rtr.flags |= OSPF_RTR_V;
```

There are three bits that have to be set. The *E* bit indicates that the router is an AS border router and will announce AS-external routes. The *E* bit is used in the SPF calculation and for summary-LSAs. In the SPF calculation routers with *E* bit set are added to the RIB. Without setting the *E* bit all AS-external routes using this router as advertising router are considered invalid because the router is not present in the RIB. Similar happens for summary-LSAs. On ABRs router summary-LSAs will be generated for every router with *E* bit set. OpenOSPFD tricks a bit with the *E* bit by setting the bit as soon as it is possible that a AS-external route is redistributed and not when the router actually redistributes a route. Other implementations have the same sloppy behaviour. Even more complex is setting the *B* bit, which is used to mark ABRs. As soon as a router is part of two active areas the *B* bit has to be set on all router-LSA. `area_border_router()` returns true if there are two or more active areas. If the state of the ABR changes all self originated router-LSAs in all areas have to be updated. This is done via `orig_rtr_lsa_all()` which in turn calls `orig_rtr_lsa()` for all areas but the current one. Afterwards setting the *B* bit is no longer a problem. The last bit that can be set is the *V* bit. It is used to mark interfaces where a virtual link is terminated. Areas where one router has a *V* bit set are transit areas. Transit areas need some special handling in the SPF calculation as example it is not allowed to send aggregated summary routing information into a transit area.

4.5.4 ABR and summary-LSA

The code handling ABRs and summary-LSAs is still in some flux. There are to many work a rounds and some stuff is still missing. Lets have a look at it anyway. It actually starts in the SPF calculation. The code that recalculates the RIB looks currently like this:

Code snip 50: SPF timer

```
rt_invalidate();

LIST_FOREACH(area, &conf->area_list, entry)
        spf_calc(area);

RB_FOREACH(r, rt_tree, &rt) {
        LIST_FOREACH(area, &conf->area_list, entry)
                rde_summary_update(r, area);

        if (r->d_type != DT_NET)
                continue;

        if (r->invalid)
                rde_send_delete_kroute(r);
        else
                rde_send_change_kroute(r);
}

LIST_FOREACH(area, &conf->area_list, entry)
        lsa_remove_invalid_sums(area);

start_spf_holdtimer(conf);
```

First the RIB is invalidated by flagging routes as invalid. While doing that old invalid routes are removed from the tree. Afterwards the SPF calculation is run for every area. This is one of the things that should be changed. There is no need to recalculate an area if there was no changes in that area. In the next step a walk over the RIB is done. By calling `rde_summary_update()` for every area and any route all required summary informations are generated. Afterwards the kernel routing table is updated by sending change or delete messages to the parent process. This is only done for routes that describe networks. After that old invalid summary-LSAs get removed from all areas. Finally the hold timer is started. This is specified in the RFC so that the SPF calculation does not kill the underpowered routers.
`rde_summary_update()` does the decision if it necessary to create a summary-LSA.

Code snip 51: Is summary-LSA needed?

```
/* first check if we actually need to announce this route
*/
if (!(rte->d_type == DT_NET || rte->flags & OSPF_RTR_E))
        return;
/* never create summaries for as-ext LSA */
if (rte->p_type == PT_TYPE1_EXT || rte->p_type ==
PT_TYPE2_EXT)
        return;
/* no need for summary LSA in the originating area */
if (rte->area.s_addr == area->id.s_addr)
        return;
/* TODO nexthop check, nexthop part of area -> no summary
*/
if (rte->cost >= LS_INFINITY)
        return;
/* TODO AS border router specific checks */
/* TODO inter-area network route stuff */
/* TODO intra-area stuff -- condense LSA ??? */
```

First of all only network routes or router routes where the *E* bit is set are summarised into other areas. The *E* bit is the same as the one in router-LSAs specifying that the router is an ASBR. An ASBR has to be added to other areas so that they can validate the AS-external-LSAs. As AS-external routes are flooded through all areas there is no need to create summaries for those networks. The originating area and all invalid routes are skipped. Finally there are some other minor but very complicated things left out for now.

Code snip 52: update summary-LSA

```
/* update lsa but only if it was changed */
if (rte->d_type == DT_NET) {
        type = LSA_TYPE_SUM_NETWORK;
        v = lsa_find(area, type, rte->prefix.s_addr,
            rde_router_id());
} else if (rte->d_type == DT_RTR) {
        type = LSA_TYPE_SUM_ROUTER;
        v = lsa_find(area, type, rte->adv_rtr.s_addr,
            rde_router_id());
} else
        fatalx("orig_sum_lsa: unknown route type");

lsa = orig_sum_lsa(rte, type);
lsa_merge(rde_nbr_self(area), lsa, v);

if (v == NULL) {
        if (rte->d_type == DT_NET)
                v = lsa_find(area, type,
                    rte->prefix.s_addr, rde_router_id());
        else
                v = lsa_find(area, type,
                    rte->adv_rtr.s_addr, rde_router_id());
}
v->cost = rte->cost;
```

To update the LS DB `lsa_merge()` is used. Before it is possible to call `lsa_merge()` two things have to be done. First the current database version of the LSA has to be found. Secondly a new LSA is generated by `orig_sum_lsa()`. After merging the LSA it is necessary

to update the cost of the vertex so that a later call to `lsa_remove_invalid_sums()` sees that this vertex is still in use. In case the LSA was newly added the previous `lsa_find()` returned `NULL` so the search has to be repeated to get a valid vertex.
`lsa_remove_invalid_sums()` does nothing more than a tree walk looking for summary-LSAs with a cost of `LS_INFINITY` and removes those by setting their age to `MAX_AGE` and calling `lsa_timeout()` to flood them out.

4.5.5 Originating AS-external-LSA

To redistribute AS-external-LSA the parent process sends a list of candidates to the RDE. The RDE uses `rde_asext_get()` to convert the kroute into a LSA and with the help of `lsa_find()` and `lsa_merge()` the LSA is added to the database. Similarly on remove `rde_asext_put()` is used to get the no longer needed LSA and again `lsa_find()` and `lsa_merge()` do the actual job.
`rde_asext_put()` has a more or less simple job. Find the kroute, remove it from the list and create a LSA with LS age `MAX_AGE` if the LSA was used.

Code snip 53: rde_asext_put()

```
LIST_FOREACH(ae, &rde_asext_list, entry)
        if (kr->prefix.s_addr == ae->kr.prefix.s_addr &&
            kr->prefixlen == ae->kr.prefixlen) {
                LIST_REMOVE(ae, entry);
                used = ae->used;
                free(ae);
                if (used)
                        return (orig_asext_lsa(kr,
                            MAX_AGE));
                break;
        }
return (NULL);
```

On the other hand `rde_asext_get()` has a bit more to do. It first looks if the route was added already before. In that case the route needs to be updated, else a new one is created.

Code snip 54: rde_asext_get() part 1

```
LIST_FOREACH(ae, &rde_asext_list, entry)
        if (kr->prefix.s_addr == ae->kr.prefix.s_addr &&
            kr->prefixlen == ae->kr.prefixlen)
                break;

if (ae == NULL) {
        if ((ae = calloc(1, sizeof(*ae))) == NULL)
                fatal("rde_asext_get");
        LIST_INSERT_HEAD(&rde_asext_list, ae, entry);
}

memcpy(&ae->kr, kr, sizeof(ae->kr));

wasused = ae->used;
ae->used = rde_redistribute(kr);
```

Next task is to find out if the route should be redistributed. The actual logic is in `rde_redistribute()` and so lets have a look at that.

Code snip 55: rde_redistribute()

```
int
rde_redistribute(struct kroute *kr)
{
        struct area*area;
        struct iface*iface;
        int              rv = 0;

        if (!(kr->flags & F_KERNEL))
                return (0);

        if ((rdeconf->options & OSPF_OPTION_E) == 0)
                return (0);

        if ((rdeconf->redistribute_flags &
            REDISTRIBUTE_DEFAULT) &&
            (kr->prefix.s_addr == INADDR_ANY &&
            kr->prefixlen == 0))
                return (1);

        /* only allow 0.0.0.0/0 if REDISTRIBUTE_DEFAULT */
        if (kr->prefix.s_addr == INADDR_ANY &&
            kr->prefixlen == 0)
                return (0);

        if ((rdeconf->redistribute_flags &
            REDISTRIBUTE_STATIC) &&
            (kr->flags & F_STATIC))
                rv = 1;
        if ((rdeconf->redistribute_flags &
            REDISTRIBUTE_CONNECTED) &&
            (kr->flags & F_CONNECTED))
                rv = 1;

        /*
         * interface is not up and running so don't
         * announce
         */
        if (kif_validate(kr->ifindex) == 0)
                return (0);

        LIST_FOREACH(area, &rdeconf->area_list, entry)
                LIST_FOREACH(iface, &area->iface_list,
                    entry) {
                        if ((iface->addr.s_addr &
                            iface->mask.s_addr) ==
                            kr->prefix.s_addr &&
                            iface->mask.s_addr ==
                            prefixlen2mask(kr->prefixlen))
                                /* already announced
                                 * as net LSA */
                                rv = 0;
                }

        return (rv);
}
```

First it is checked if we have to redistribute anything. Afterwards the default route gets handled. The default route is only redistributed if explicitly enforced via "*redistribute default*". Dependent on the flags it is now decided if routes gets redistributed. The interface state is checked and finally all configured interfaces are inspected to see if the route is not already part of a network-LSA or is announced as a stub network.
After the `rde_redistribute()` call it is now clear what remains to be done.

Code snip 56: rde_asext_get() part 2

```
if (ae->used)
        /* update of seqnum is done by lsa_merge */
        return (orig_asext_lsa(kr, DEFAULT_AGE));
else if (wasused)
        /*
         * lsa_merge will take care of removing the
         * lsa from the db
         */
        return (orig_asext_lsa(kr, MAX_AGE));
else
        /* not in lsdb, superseded by a net lsa */
        return (NULL);
```

If the route has to be redistributed a LSA with the initial LS age is generated and returned. If it is no longer used a LSA with LS age `MAX_AGE` is generated and returned.

Otherwise the work is completed and function returns.
In case an interface state changes, `rde_update_redistribute()` is called and all routes that depend on this interface are recalculated very similar to the presented code here. Again going through `rde_redistribute()`, `orig_asext_lsa()`, `lsa_find()`, and `lsa_merge()`.

4.6 Issues and other stuff

There are still some problems in OpenOSPFD that have to be solved. Some features are incomplete and so there is still a lot of work to be done. Lets look back at the solved problems. The first problem encountered was probably the privilege separation because a clever splitting had to be done. This is still sometimes an issue – for example the current redistribute code is partially done in the wrong place. The result is massive overhead if the router does "*redistribute static*" with a full view in the routing table. All ~170'000 routes are passed to the RDE and evaluated there. It works but is inefficient. Other problems with privsep were solved like the `MAX_AGE` or the database exchange problems explained earlier. A good example of a work a round is the multicast handling. A real fix for this problem is in progress but some kernel patches are required to make it fly. At least many issues and bugs were identified and fixed in the flooding and database exchange phase – the most important part of the protocol.
Things that remain to be fixed include the redistribute code or the missing support for interface aliases. The ABR code is still not optimal and is not as good tested as the normal case. Virtual links still need a lot of work to get them flying – a lot of code is around but some important bits are missing. Interface handling should be improved, like supporting aliases and dynamic interfaces. Last but not least there are all those supercool new features planned but that's a different paper. :)

Bibliography

[1] Moy, J. *OSPF version 2*. RFC 2328, April 1998.

[2] Moy, J. *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley, September 1998

[3] OpenBSD, *http://www.openbsd.org/*

[4] OpenBGPD, *http://www.openbgpd.org/*

[5] OpenOSPFD source code, *http://www.openbsd.org/cgi-bin/cvsweb/src/usr.sbin/ospfd/*