OpenBSD

network stack internals

by Claudio Jeker

The OpenBSD network stack is under constant development mainly to implement features that are more and more used in todays core networks. Various changes were made over the last few years to allow features like flow tagging, route labels, multipath routing and multiple routing tables. New features like VRF (virtual routing and forwarding), MPLS and L2TP are worked on or planned. This paper tries to cover the design decisions and implementation of these features.

Introduction

The OpenBSD network stack is based on the original BSD4.4 design that is described in TCP/IP Illustrated, Volume 2[1]. Even though many changes were made since then [1] still gives the best in depth introduction to the BSD based network stacks. It is the number one reference for most part of the network stack. Additionally IPv6 Core Protocols Implementation[2] covers the KAME derived IPv6 network stack implementation and should be considered the reference for anything IPv6 related.

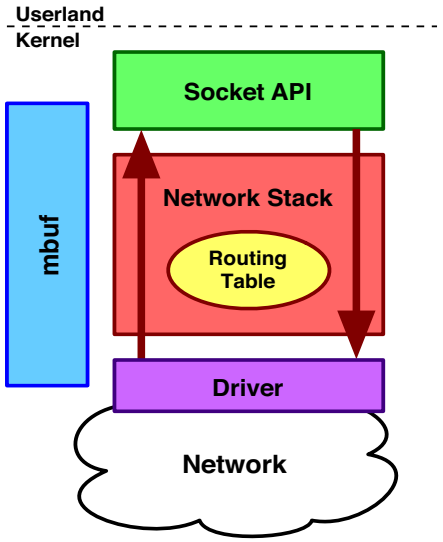

Figure 1: Building blocks of the network stack

The networking code can be split in four major building blocks: the network stack, the mbuf API, the routing table and the socket API. The various protocol support are hooked into the network stack on various defined borders. Protocols provide methods which are registered in a protosw structure. By looking up these structures instead of calling the functions directly a more dynamic and easily extensible stack can be built. There is an input and an output path that starts in the drivers and ends in the socket API and vice versa. Every packet passing through the network stack is stored in mbufs and mbuf clusters. It is the primary way to allocate packet memory in the network stack but mbufs are also used to store non packet data. Everything routing related is covered by the protocol independent routing table. The routing table covers not only the layer 3 forwarding information but is also used for layer 2 look ups -- e.g. arp or IPv6 network discovery. The calls to the routing table code are scattered all over the network stack. This entanglement of the routing code is probably one of the most problematic parts when making the network stack MP safe. Last but not least the socket API. It is the interface to the userland and a real success story. The BSD socket API is the most common way for applications to interact between remote systems. Almost any operating system implements this API for userland programs. I will not cover the socket API because non of the new features modified the socket API in detail.

mbufs

Packets are stored in mbufs. mbufs can be chained to build larger consecutive data via `m_next` or build a chain of independent packets by using the `m_nextpkt` header. `m_data` points to the first valid data byte in the mbuf which has the amount of `m_len` bytes stored. There are different mbuf types defined to indicate what the mbuf is used for. The `m_flags` are used in various ways. Some flags indicate the structure of the mbuf itself (`M_EXT`, `M_PKTHDR`, `M_CLUSTER`) and some indicate the way the packet was received (`M_BCAST`, `M_MCAST`, `M_ANYCAST6`). If `M_PKTHDR` is set an additional structure `m_pkthdr` is included in the mbuf. The first mbuf of a packet includes this `m_pkthdr` to store all important per packet meta data used by the network stack. The complete length of the mbuf chain and the interface a packet was received on are the most important ones.

Code Fragment 1: mbuf structures

```
struct m_hdr {
        struct  mbuf *mh_next;
        struct  mbuf *mh_nextpkt;
        caddr_t mh_data;
        u_int   mh_len;
        short   mh_type;
        u_short mh_flags;
};

struct  pkthdr {
        struct  ifnet *rcvif;
        SLIST_HEAD(packet_tags, m_tag) tags;
        int     len;
        int     csum_flags;
        struct  pkthdr_pf;
};

struct m_tag {
        SLIST_ENTRY(m_tag)      m_tag_link;
        u_int16_t               m_tag_id;
        u_int16_t               m_tag_len;
};
```

Mbuf tags are generic packet attributes that can be added to any packet. Mbuf tags are mostly used by the IPsec code and to prevent loops in the network stack when tunnelling interfaces are used. Up until OpenBSD 4.2 pf used the mbuf tags to store internal state information (`pkthdr_pf`). Every packet needs this state information if pf is enabled. Moving this structure from mbuf tags directly into the `m_pkthdr` almost doubled performance. The main reason of this speed up is that the allocation of mbuf tags is skipped. Mtag allocation is slow because `malloc(9)` needs to be used to allocate the dynamic length elements. Information that has to be added to every packet should probably be directly included in the packet header.

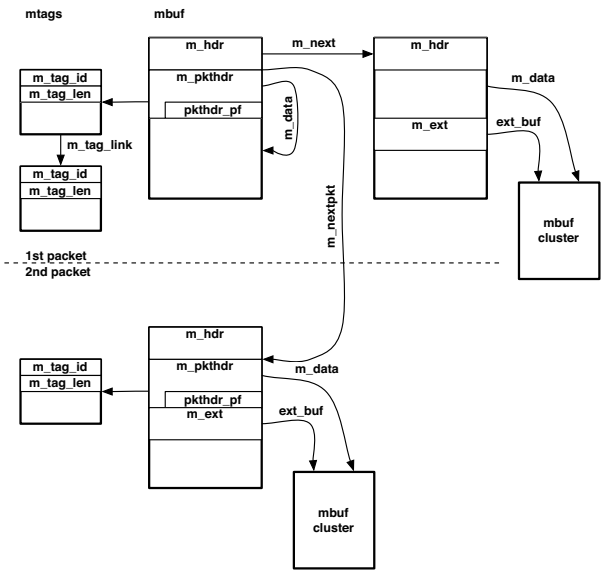

Figure 2: mbuf structures of two packets, 1st packet is built by an mbuf chain of two mbufs (first mbuf with internal data second with an external mbuf cluster).

Network Stack

Packets enter the network stack from userland through the socket API or by the network driver receive interrupt function. Packets received by the network card enter one of the layer 2 input functions -- `ether_input()` is the most commonly used one. This function decodes/pops off the layer 2 header and figures out the proper payload type of the data. In the Ethernet case the `ether_type` is inspected but first it is checked if it is a multicast or broadcast packet and the corresponding mbuf flags are set. Depending on the payload type an input queue is selected, the packet is enqueued and a softnet software interrupt is raised. This interrupt is delayed because `IPL_SOFTNET` has a lower precedence then `IPL_NET` used by the driver interrupt routine. So the driver can finish his work and when lowering the system priority level the softnet interrupt handler is called. The softnet handler checks `netisr` for any set bits and calls the corresponding protocol *interrupt* handler. Most of the time this is `ipintr()` or `ip6intr()` but the bridge, ppp and pppoe code use the softnet handler as well. So the `splnet()`/`splsoftnet()` dance has nothing to do with the layer 2/layer 3 border.

`ipintr()` dequeues all packets in the protocol input queue and passes them one after the other to `ipv4_input()`. `ipv4_input()` checks the IP header then calls `pf_test()` to do the input firewalling. Now the destination is checked and if the packet is not for this host it may be forwarded. Local packets are passed to the transport layer. Instead of hardcoding the corresponding handlers into `ipv4_input()` a more object oriented approach is used by calling the `pr_input()` function of a `protosw` structure. The `inetsw[]` array contains the various protosw structures indexed by protocol family.

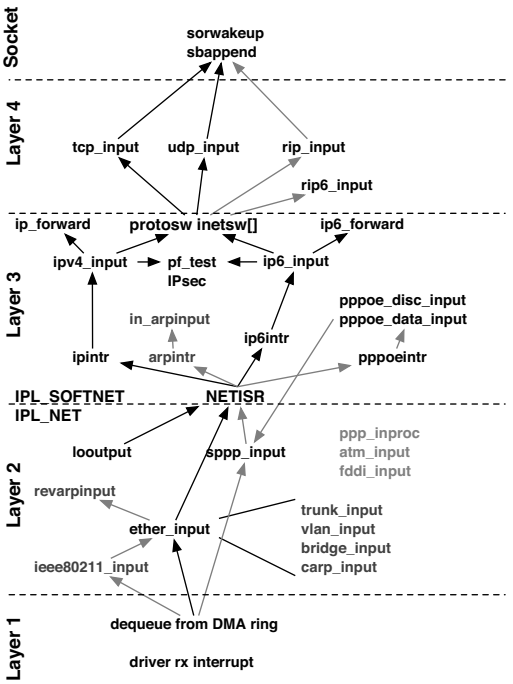

Figure 3: Network stack input flow

Common input functions are `tcp_input()`, `udp_input()` and `rip_input()` -- rip stands for raw IP and has nothing in common with the RIP routing protocol. These input functions check the payload again and do a pcb lookup. The pcb or protocol control block is the lower half of a socket. If all is fine the packet is appended to the socket receive buffer and any process waiting for data is woken up. Here the processing of a network interrupt ends. A process will later on call the `soreceive()` function to read out the data in the receive buffer.
In the forward path a route lookup happens and the packet is passed on to the output function.

Sending out data normally starts in userland by a `write()` call which ends up in `sosend()`. The socket code then uses the protosw `pr_usrreq()` function for every operation defined on a socket. In the `sosend()` case `pr_usrreq()` is called with `PRU_SEND` which will more or less directly call the output function e.g. `tcp_output()` or `udp_output()`. These functions encapsulate the data and pass them down to `ip_output()`. On the way down the output function is called directly (not like on the way up where between the layer 3 and 4 the protosw structure was used). In `ip_output()` the IP header is prepended and the route decision is done unless the upper layer passed a cached entry. Additionally the outbound firewalling is done by calling `pf_test()`. The layer 3 functions invoke the layer 2 output function via the `ifp->if_output()` function. For the Ethernet case, `ether_output()` will be called. `ether_output()` prepends the Ethernet header, raises the spl to `IPL_NET`, puts the packet on the interface

output queue, and then calls the `ifp->if_start()` function. The driver will then put the packet onto the transmit DMA ring where it is sent out to the network.

This is just a short fly through the network stack, in the real world the network stack is much more complex due to the complex nature of the protocols and added features. All the control logic of the network stack is left out even though it is probably the most obscure part of it.

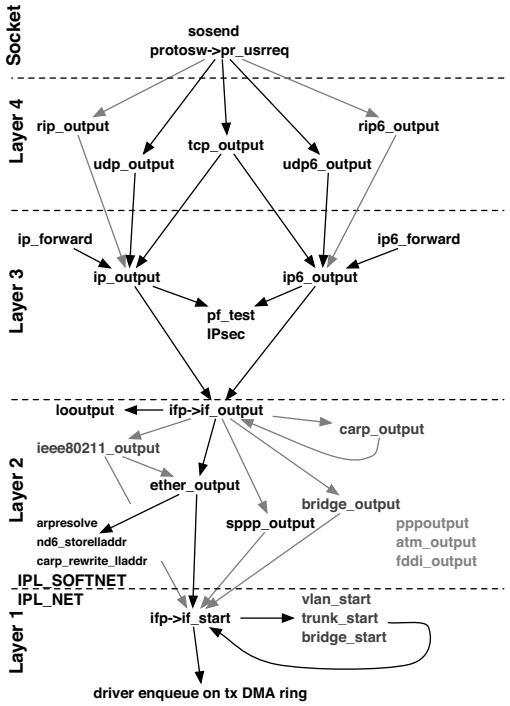

Figure 4: Network stack output flow

Routing Table

The routing table uses the same patricia trie as described in [1] even though minor extensions were done or are planned. The routing table stores not only layer 3 forwarding information but includes layer 2 information too. Additionally the patricia trie is also used by pf tables. There is a lot of magic in the routing table code and even minor changes may result in unexpected side-effects that often end up in panics. The interaction between layer 3 and layer 2 is a bit obscure and special cases like the arp-proxy are easily forgotten. The result is that routing changes take longer then expected and need to move slowly. Until now *only* routing labels, multiple routing tables, and multipath routing were implemented plus a major bump of the routing messages was done. The routing messages structure was changed to allow a clean integration of these features. Mainly the routing table ID had to be included into each routing header. With a few tricks it was even possible to have some minimal backward compatibility so that new kernels work with older binaries.

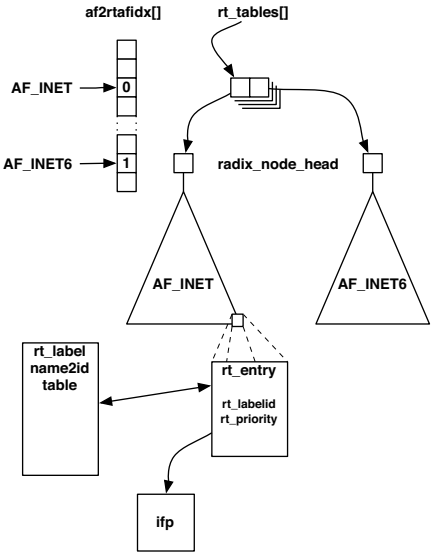

Figure 5: Overview of the routing tables

Routing Labels

Routing labels were the first OpenBSD specific extension. A routing label is passed to the kernel as an additional sockaddr structure in the routing message and so the impact was quite minimal. But instead of storing a string with the label on every node a per label unique ID is stored in the routing entry. The mapping is done by a name to ID lookup table. Labels can be set by userland and pf can make decisions based on these labels.

Multipath Routing

The implementation of multipath routing was initially from KAME but additional changes were necessary to make them usable. Especially the correct behaviour of the routing socket was a challenge. Some userland applications still have issues with correctly tracking multipath routes because the old fact of one route one nexthop is no longer true.

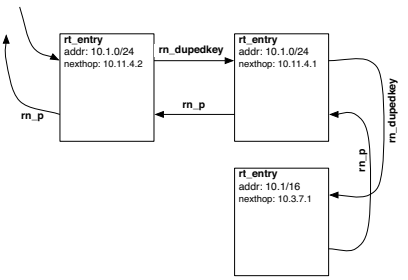

Figure 6: rn_dupedkey and multipath routes

Multipath routing is abusing the possibility of storing the same key multiple times in the routing table. This is allowed because of possible more specific routes with the same network address -- e.g. 10.0.0/24 and 10/8. Such identical keys

are stored in the `rn_dupedkey` list. This list is ordered by prefix length -- most specific first -- so all multipath routes are consecutive.

In the network stack some route look ups had to be exchanged with a multipath capable version. `rtalloc_mpath()` is the multipath aware route lookup which has an extra attribute -- the source address of the packet. The source and destination address are used to select one of the multiple paths. `rtalloc_mpath()` uses a hash-threshold mechanism[3] to select one of the equal routes and routes are inserted in the middle of the list of paths. This more complex mechanism to select and insert routes was chosen to keep the impact of route changes small.

Special sysctl buttons were added to enable and disable the multipath routing:

```
sysctl net.inet.ip.multipath=1
```

and/or:

```
sysctl net.inet6.ip6.multipath=1
```

Without these sysctl values set multipath routing is turned off even if multiple routes are available.

Multiple Routing Tables

Multiple routing tables are a prerequisite for VRF. The first step in supporting virtual routing and forwarding is to be able to select an alternate routing table in the forwarding path.

Every address family has a separate routing table and all routing table heads are stored in an array. With regard to VRF it was considered the best to always create a full set of routing tables instead of creating per address family specific routing tables. If a new routing table is created the `radix_node_heads` for all address families are created at once, see Figure 5. `pf(4)` is used to classify the traffic and to select the corresponding forwarding table. At the moment it is only possible to change the default routing table in the IP and IPv6 forwarding path. For link local addressing -- e.g. arp -- the default table is used.

VRF

The idea behind virtual routing and forwarding is the capability to divide a router into various domains that are independent. It is possible to use the same network in multiple domains without causing a conflict.

To support such a setup it is necessary to be able to bind interfaces to a specific routing table or actually building a routing domain out of a routing table and all interfaces which belong together. On packet reception the mbuf is marked with the ID of the receiving interface. Changes to the layer 2 code allow the use of alternate routing tables not only for IP forwarding but for arp look ups as well. With this, it is possible to have the same network address configured multiple times but completely independent of each other.

To create a system with virtualized routing many changes are needed. This starts with making the link local discovery protocols (arp, rarp, nd, ...) aware of the multiple domains. The ICMP code and all other code that replies or tunnels packets needs to ensure that the new packet is processed in the same domain. A special local tunnel interface is needed to pass traffic between domains and pf may need some modifications as well. Finally the socket layer needs a possibility to attach a socket to a specific routing domain. The easiest way to do this is via a `getsockopt()` call.

Unlike the vimage[4] approach for FreeBSD not a full virtualization is done. The main reason behind this different approach is in my opinion primarily the originator and his background. In FreeBSD, vimage was developed by network researchers the idea is to be able to simulate multiple full featured routers on a single box. Therefore vimage goes further then the OpenBSD implementation by having multiple routing sockets, protosw arrays and independent interface lists. In OpenBSD the development is pushed by networking people with an ISP background. The end result is similar but the userland hooks are different. In OpenBSD, userland will have access to all domains at once through one routing socket so that one routing daemon is able to modify multiple tables at once.

Routing Priorities

With the inclusion of bgpd, ospfd, and ripd the need for userland to easily manage various routing source in the kernel became more apparent. In case of conflicts it is necessary to have a deterministic mediator between the different daemons. E.g. prefixes learned via OSPF should have a higher preference than external BGP routes. Routing suites like xorp and quagga/zebra normally use a separate daemon to merge these various routing sources. This is a single point of failure and unnecessary because the kernel can do this just fine. Similar to commercial routers this can be done by the kernel by adding a priority to each route and giving each routing source one specific priority level. With the introduction of multipath routing it is possible to store a route multiple times in the kernel. Having a cost for each route and sorting the list of equal routes by the cost gets the desired behaviour. Only the routes with the lowest cost are used for routing -- in other words, this is now equal cost multipath routing.

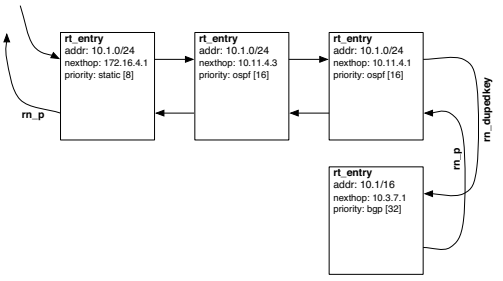

Figure 7: Using rn_dupedkey to implement priorities

Implementing this simple concept on the other hand released some evil dragons in the current routing code. Until now, the multipath code only had to handle inserts after the head element which is no longer true. The initial result were corrupted routing tables, crashes, a lot of head scratching, and even more debugging. The routing code and especially the radix tree implementation itself is complex, obscure, and a very sensitive part of the networking code.

References

[1] TCP/IP Illustrated, Volume 2
by Gary R. Wright, W. Richard Stevens

[2] IPv6 Core Protocols Implementation
by Qing Li, Tatuya Jinmei, Keiichi Shima

[3] Analysis of an Equal-Cost Multi-Path Algorithm, RFC 2992, November 2000

[4] The FreeBSD Network Stack Virtualization
http://www.tel.fer.hr/zec/vimage/
by Marko Zec