The Netlink protocol: Mysteries Uncovered

Jan Engelhardt

rev. 2010-October-30

Abstract

Netlink is a bitstream protocol for communication between the Linux kernel and userspace. It intends to replace ioctl calls, especially in the area of networking configuration, but is also being used for configuration other local services.

About the author

Jan is a consultant for systems and network administration with a strong focus on Linux-based environments. He uses Linux since fall 1999, and is involved in kernel development since 2003. His latest significant activities are in the field of firewalling with Netfilter and Xtables.

Thanks

Many thanks to Pablo Neira Ayuso who, with the release of libmnl, brought Netlink a lot closer to me.

Prerequisites

The reader should bring along an understanding of the C programming language, TCP/UDP networking and the POSIX socket API (which is not to be confused with the BSD socket API, even though they share similarities) as per `http://www.opengroup.org/onlinepubs/9699919799/functions/contents.html`.

This book is designed for, and made available in ISO A4 format, for the simple reason that most regular printed books have anything *but* a standard size.

Contents

1 Introduction

1.1 Overview

Netlink is a protocol used within the Linux operating system, for communication between kernel and userspace. It is described in RFC 3549[1] also as an intra-kernel protocol. In fact, it can also be "abused" as a userspace-to-userspace communications channel.

The structure of Netlink messages is basically a serialized packed stream of a hierarchial tree made of message headers and attributes. Nesting is possible; each attribute value is, without knowing the attribute more closely, just raw data, reminiscient of how IPv6-in-IPv6 is constructed, for example.

1.2 A bit of history

The driver was initially written by Alan Cox under the codename "Skiplink"[2] as a character device `/dev/netlink` for Linux kernel v1.3.31 dated 1995-Oct-04, where it was already included under the filename `netlink.c`, before it was superseded by Alexey Kuznetsov's socket-based variant `af_netlink.c` in v2.1.68 (1997-Dec-01), which also brought along rtnetlink that provided for network interface and routing configuration. (`struct nlmsghdr` exists since v2.1.15.)

1.3 RFC critique

The RFC has some academic bingo (cf. marketing bingo) terminology that at first may seem to have absolutely no relation to more well-known components, and only starting clearing up them in chapter 2. So if the RFC's Abstract put you off from reading the RFC, here's the deal:

- Control Plane (CP) – an execution environment: Userspace, usually.
- Control Plane Component (CPC): Daemon
- Forwarding Engine (FE): Kernel
- Forwarding Engine Component (FEC): Kernel subsystem

Of course one could also run a CPC/daemon in kernelspace, which is why CP is in fact not necessarily userspace.

While the choice of "forwarding engine" may still work out with configuring the Linux networking stack, this stops making sense when one talks about non-networking subsystems, such as DiskQuota[3] and Kobject Events[4].

1.4 Differences to IP

Users are most familiar with creating well-known IP-layer sockets, such as

```
int sk = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
sk = socket(AF_INET6, SOCK_STREAM, IPPROTO_SCTP);
```

[1]The first RFC I've encountered that documents a (currently) Linux-specific interface. Oh well, just wait until other operating systems adopt it.

[2]SKIP is "Simple Key Management for Internet Protocols", a 1994 Internet Draft.

[3]Delivers quota status messages via Netlink starting v2.6.24.

[4]Udev reads those, for example.

This would give them a socket that operates in stream mode. After setting up all the parameters and establishing a connection, one can write — in fact, stream — data to the socket, which will then take care of chopping stuff up or joining it accordingly. In stream mode, reads and writes generally have no correspondence to any sort of packet boundaries.

Second, there is datagram mode, reads and writes do correspond to packet boundaries. Even if a read cannot be completely satisfied, for example because the userspace buffer was too small, the packet will be consumed, in other words, you may lose bytes.

```
sk = socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDP);
```

Up to here, people may have also worked with local UNIX sockets that operate in very much the same fashion.

```
sk = socket(AF_LOCAL, SOCK_STREAM, 0);
sk = socket(AF_LOCAL, SOCK_DGRAM, 0);
```

The local sockets do not use any particular transport protocol — `AF_LOCAL` is the transport so to say, so the 3rd argument is just zero. Actually, the current Linux kernel allows using `socket(AF_LOCAL, *, AF_LOCAL)`. With local sockets also comes the introduction of a third mode, datagrams that are processed in sequential order. As it stands, SCTP also offers this[5].

```
sk = socket(AF_LOCAL, SOCK_SEQPACKET, 0);
sk = socket(AF_INET6, SOCK_SEQPACKET, IPPROTO_SCTP);
```

Of course there are many other possibilities. One of the newer additions is DCCP, which has no `SOCK_DGRAM` type even though DCCP is pretty much an (unreliable) datagram service.

```
sk = socket(AF_INET6, SOCK_DCCP, IPPROTO_DCCP);
```

However, let's leave it at that. What all of these modes have in common is that the operating system will cater for generating the headers that, together with the data, will form the packet. Similarly, on reception, the headers will be stripped again by the OS. The metainformation that is in these headers, like peer's address and/or port number are not lost though, but exposed through the socket API (5th argument to `recvfrom`(2); or 2nd arg to `accept`(2); or `getpeername`(2)).

1.5 When it gets raw

The next level of freedom (and more work) are so-called raw sockets, where the userspace program has to take care of crafting all the packets, including (layer-4) headers.

```
struct {
        struct udphdr u;
        char buf[64];
} pkt;
struct sockaddr_in6 dst = {
        .sin6_family = AF_INET6,
};
```

[5]This is indeed part of the SCTP specification, not a strange Linux enhancement.

```
int sk = socket(AF_INET6, SOCK_RAW, IPPROTO_UDP);
pkt.u.source = htons(1234);
pkt.u.dest   = htons(53);
pkt.u.len    = sizeof(pkt);
pkt.u.check  = 0;
inet_pton(AF_INET6, "::1", &dst.sin6_addr);
snprintf(pkt.buf, sizeof(pkt.buf), "Hello World");
sendto(sk, &pkt, sizeof(pkt), 0, &dst, sizeof(dst));
```

The IPv4 and IPv6 raw sockets also support remembering the address using `connect`(2) so that one can use `send`(2) instead of `sendto`(2). `bind`(2) is also supported in case a custom source address is desired. `dst.sin6_port` is ignored, as the ports are provided in `pkt` already. Specifying `IPPROTO_UDP` at socket creation time is a must, as that is what the IP header will get for the Next Header field.

One can also choose to build the layer-3 header by themselves, by using `IPPROTO_RAW`.

```
socket(AF_INET6, SOCK_RAW, IPPROTO_RAW);
socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
struct {
        struct ip6_hdr ih;
        struct udphdr u;
        char buf[64];
} pkt;
...
```

There is a subcase of IPv4 raw sockets:

```
static const int y = 1;
int sk = socket(AF_INET, SOCK_RAW, IPPROTO_UDP);
setsockopt(sk, SOL_IP, IP_HDRINCL, &y, sizeof(y));
```

IPv6 does not have `IP_HDRINCL`, as per RFC 3542 section 3. Trying to use it nevertheless with the `SOL_IPV6` socket level will actually lead to the `IPV6_2292HOPOPTS` option being set, which has the same value as `IP_HDRINCL`.

And finally, there is the ultimate socket interface where one has access to layer-2 too, on a cooked basis and raw basis, but which we won't go further into detail here:

```
sk = socket(AF_PACKET, SOCK_DGRAM, htons(ETH_P_ALL)); /* cooked */
sk = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL)); /* raw */
```

2 Netlink sockets

2.1 Services

The kernel provides a number of Netlink services, the list of which is below. These constants are used for the third argument to the socket(2) system call, in this context also referred to as Netlink families. Services use their own protocol that is derived from the Netlink base protocol. It is imperative that the proper protocol is spoken on a particular socket type, e. g. rtnetlink messages on NETLINK_ROUTE. Currently used services include:

- NETLINK_ROUTE – Routing as done with the ip(8) tool from the iproute2 packag.
- NETLINK_XFRM – Transformation database, i. e. IPsec and IPCOMP.
- NETLINK_FIREWALL – "Firewalling hook". This is used by ip_queue (obsoleted by libnetfilter_queue) to send packets to userspace, and receive reinjected packets or verdicts from userspace.
- NETLINK_INET_DIAG – Socket state monitoring using the ss(8) from the iproute2 package.
- NETLINK_NFLOG – Used by the ipt_ULOG and ebt_ulog targets. Probably obsoleted by libnetfilter_queue and/or libnetfilter_log.
- NETLINK_SELINUX Event notifications from SELinux.
- NETLINK_ISCSI Delivery of control PDUs from (Open)-iSCSI to userspace.

Significant initial confusion can ensue due to Netlink sockets always operating in raw mode. Malformed packets may be silently ignored by an operating system, and this is currently the case for Linux, though error reporting can be enabled if at least the nlmsghdr is correct.

##In contrast to what the RFC says, basic Netlink delivery is unicast, with the kernel side having the option to send broadcasts, such as gratitious network interface state change notifications, to attached listeners.##

2.2 Dissection

While Netlink sockets can be created using both SOCK_DGRAM and SOCK_RAW, they will currently both lead to a raw socket of the lowest level where no headers are ever autogenerated. This makes the initial understanding of Netlink harder than it needs to be, because we are so used to work ourselves from upper layers to lower ones rather than vice-versa.

AF_NETLINK, as well as AF_LOCAL, to include it in the picture for comparison, are both protocols for which it is hard to assign exact OSI layer numbers to them. Since they are limited to the local host, they are unlikely to lie anywhere below layer 4. Support for sequenced packets and/or streaming are also typical properties of layer 4. The following non-normative hint table should help relate.

	AF_INET6	AF_NETLINK	AF_LOCAL
Layer 3	ipv6hdr		
Layer 4	tcphdr	sockaddr_nl+nlmsghdr	sockaddr_un
4		(NETLINK_*subsys*)	
4		rtgenmsg / ifaddrmsg / etc.	
Layer 7	(data)	(attrs)	(data)

Netlink is a little bit strange in many regards when comparing to other socket types. Whereas `struct tcphdr` contains both source and destination locations, this information is split into `nlmsghdr.nlmsg_pid` and `sockaddr_nl.nl_pid`, respectively. Furthermore, Netlink messages carry no "Next Header" field akin to IPv6's NextHdr in `struct nlmsghdr`, so the actual protocol is implicitly given by the chosen Netlink family. Finally, `struct rtgenmsg` could be seen as taking the role of IPv6's Extension Headers, though it is mandatory rather than optional like an ExtHdr.

2.3 Simple user-to-user communication

The fact that the Linux kernel does not do any checks on Netlink packets when the destination of a datagram is not the kernel

allows for an interesting abuse of Netlink (which has originally been specified as a kernel-userspace communication channel). This also makes it possible to start understanding Netlink in smaller pieces. Let us thus begin with a code sample:

```
#include <sys/socket.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <linux/netlink.h>

int main(void)
{
        ssize_t ret;
        char buf[] = "Hello world";
        struct sockaddr_nl dst = {
                .nl_family = AF_NETLINK,
                .nl_pid = getpid(),
        };
        int sk = socket(AF_NETLINK, SOCK_RAW, 0);
        if (sendto(sk, buf, strlen(buf), 0,
            (void *)&dst, sizeof(dst)) < 0) {
                perror("sendto");
                abort();
        }
        ret = recv(sk, buf, sizeof(buf), 0);
        if (ret < 0) {
                perror("recv");
                abort();
        }
        printf("%.*s\n", (int)ret, buf);
        return EXIT_SUCCESS;
}
```

This creates a Netlink socket and starts sending some data (in this case, our "Hello World" string including trailing '\0' byte) to the specified destination, which, as chosen here, is the sending program itself. It is also possible to separate the sending and receiving side into two separate programs — and thus processes —, and to then specify the PID of the other process in `dst.nl_pid`. However, `nl_pid` does NOT actually specify a PID, but a Netlink address. More on that soon.

The careful reader may notice that `sendto`(2) may block and that `recv`(2) in fact may never be called. For brevity of code examples, some error checking is omitted, and it is assumed that the generic socket buffer will take care of this, so that there is room for queueing a few Netlink messages and not immediately block.

2.4 Socket addresses

Most socket types allow to use `bind`(2) to preselect the socket address — or part of it — that is to be used for further communication. It is possible to specify partial desires, such as `[::]:12345` and `[2a01::db8]:0`, in which case the zero fields will be filled in later at connect time. This is convenient if you want a specific source port, but do not want to deal with the outgoing address. If `bind`(2) is not explicitly called, it will be implicity invoked by the operating system when one is about to establish a connection and/or send a datagram. An appropriate address will be selected that the remote peer can reach too.

For Internet protocols, Linux will do a route lookup to the destination, and the "src" attribute of the resulting route is used. The routing tables can be viewed with '`ip route show table all`', and an actual route lookup can be performed on the command line using, for example, '`ip route get 2a00:1450:8004::67`'.

Netlink also supports explicit `bind`(2), and one might actually wonder whether that would not perhaps be a security risk if a process could impersonate another:

```
struct sockaddr_nl lo = {
        .nl_family = AF_NETLINK,
        .nl_pid = 1,
};
/* Look, I'm init! */
bind(sk, &lo, sizeof(lo));
/* Or not. */
```

However, `nl_pid` identifies a Netlink socket, not a process. `nl_pid` should better be thought of as "nl_port". Unlike Internet protocols, there are no privileged "ports", or actually: socket addresses, reachable in Netlink. An `nl_pid` of zero expresses a wildcard for `bind`(2), and is interpreted as "kernel" as a destination for `send`(2), so there is no way to get socket #0 in userspace. Even if there was (imagine fd passing from the kernel down through `AF_LOCAL`'s `SCM_RIGHTS` option), it would probably fail since the kernel already has that socket address in use. ## diff famililes?

Like IP sockets, Netlink sockets get automatically bound to a free socket address when `bind`(2) was not used, and `getsockname`(2) can be used to retrieve the address after it has been filled in. It can be observed that `sockaddr_nl.nl_pid` is usually set to the current process's PID, but given each socket's address must be, and is, unique, other numbers will be chosen for when the address is already in use. This is very easy to provoke: just create two sockets in the same process.

```
static const struct sockaddr_nl dst = {.nl_family = AF_NETLINK};
sk_1 = socket(AF_NETLINK, SOCK_RAW, 0);
sk_2 = socket(AF_NETLINK, SOCK_RAW, 0);
sendto(sk_1, NULL, 0, 0, (void *)&dst, sizeof(dst));
sendto(sk_2, NULL, 0, 0, (void *)&dst, sizeof(dst));
```

When there is a collision during autobinding, the Linux kernel will fallback to picking numbers decreasing monotonically from -4097 onwards. This selection however is an implementation detail that your C programs should not concern with.

To get an overview of the currently-active Netlink sockets in the system, use ‘`ss -f netlink`‘.

3 Message construction

In general, Netlink messages are rather simple, and it is only the amount of nesting one can do that makes them look a bit more complex on first sight. A basic message consists of a message header and its data. There may be padding after the header to have the data portion aligned, and there may be padding after the data, to have a subsequent message also aligned correctly.

```
struct nlmsghdr {
        uint32_t nlmsg_len;
        uint16_t nlmsg_type;
        uint16_t nlmsg_flags;
        uint32_t nlmsg_seq;
        uint32_t nlmsg_pid;
};
```

nlmsg_len The entire length of the message, including header, padding, data and trail padding.

4 Broadcasting

5 Using libmnl

6 Kernel side

7 Subsystems

NFNETLINK/GENETLINK

8 `asd`

There are more types defined, but there is currently no corresponding code in the Linux kernel to interpret them.
nlmsg_seq

Index