Netlink Sockets - Overview

Gowri Dhandapani, Anupama Sundaresan
Information and Telecommunications Technology Center
Department of Electrical Engineering & Computer Science
The University of Kansas
Lawrence, KS 66045-2228

September 13, 1999

Contents

1 Introduction

Linux has support for a lot of advanced networking features, these include firewalls, QoS support in the form of queues, classes and filters, traffic conditioning, netlink sockets etc.This document discusses briefly about netlink sockets from the implementation and usage point of view. Section 2 gives an overview of the creation and usage of netlink sockets and how in general protocol families are registered in the kernel, Section 3 explains packaging a netlink packet from user space with an example and how this packet is handled in the kernel. Also a complete example code listing is provided in the appendix.

2 Netlink sockets

2.1 An Overview of netlink sockets

Netlink is used to transfer information between kernel modules and user space processes, it provides kernel/user space bidirectional communication links. It consists of a standard sockets based interface for user processes and an internal kernel API for kernel modules.

A netlink socket in the user space can be created by

```
sock_fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
```

The domain is AF_NETLINK, the type of socket is SOCK_RAW, however netlink is a datagram oriented service. Both SOCK_RAW and SOCK_DGRAM are valid values for socket_type but the netlink protocol does not distinguish between datagram and raw sockets.

Netlink_family selects the kernel module or netlink group to communicate with. The currently assigned netlink families are:

- NETLINK_ROUTE : Receives routing updates and may be used to modify the IPv4 routing table, network routes, ip addresses, link parameters, neighbour setups, queueing disciplines, traffic classes and packet classifiers may all be controlled through NETLINK_ROUTE sockets

- NETLINK_FIREWALL : Receives packets sent by the IPv4 firewall code.

- NETLINK_ARPD : For managing the arp table from user space.
- NETLINK_ROUTE6 : Receives and sends IPv6 routing table updates.
- NETLINK_IP6_FW : To receive packets that failed the IPv6 firewall checks (currently not implemented).
- NETLINK_TAPBASE...NETLINK_TAPBASE+15 : Are the instances of the ethertap device. Ethertap is a pseudo network tunnel device that allows an ethernet driver to be simulated from user space.
- NETLINK_SKIP : Reserved for ENskip.

2.2 Protocol Registration

Before actually seeing how netlink sockets are created and used, let us just take a look at how the various socket types are registered and supported by the kernel.

When the linux system comes up and memory and process management modules start working, its time to get some real work done. There is a function __init do_basic_setup() in init/main.c which essentially does all the initialisation including the sock_init(). This function in net/socket.c does the initialisation of the firewalls, protocols etc, we are interested in the protocol initialisation, there is a call to the function proto_init() which kicks off all the configured protocols. The list of protocols is maintained as a table in net/protocols.c, the section of the code is given below:

```
struct net_proto protocols[] = {
#ifdef  CONFIG_NETLINK
  { "NETLINK",  netlink_proto_init  },
#endif
#ifdef  CONFIG_INET
  { "INET", inet_proto_init },          /* TCP/IP */
#endif
```

```
struct net_proto
{
    const char *name;       /* Protocol name */
    void (*init_func)(struct net_proto *);  /* Bootstrap */
};
```

The table contains net_proto structures, every entry in the table contains the protocol name and the init function corresponding to that protocol. So the proto_init() contains the following code for the initialisation of the protocols, it picks up every entry in the table and calls the init of every protocol.

```
void __init proto_init(void)
{
extern struct net_proto protocols[];    /* Network protocols */
struct net_proto *pro;

    pro = protocols;
    while (pro->name != NULL)
    {
        (*pro->init_func)(pro);
        pro++;
    }
}
```

Now lets see what happens in the netlink_proto_init function.

There is yet another structure of interest,

```
struct net_proto_family
{
    int family;
    int (*create)(struct socket *sock, int protocol);
    /* These are counters for the number of different methods of
       each we support */
    short   authentication;
    short   encryption;
    short   encrypt_net;
};
```

Now for the netlink protocol, the binding between the family and the create function for the socket is given by

```
struct net_proto_family netlink_family_ops = {
    PF_NETLINK,
    netlink_create
};
```

In netlink_proto_init(), look at the following code,

```
void netlink_proto_init(struct net_proto *pro)
{
    .
    .
    sock_register(&netlink_family_ops);
    .
}
```

The sock_register() function is called by a protocol handler when it wants to advertise its address family, and have it linked to the socket module. It creates an entry for this protocol in the net_families table. The net_families contains the protocol list and all the protocols are registered here.

```
int sock_register(struct net_proto_family *ops)
{
    .
    .
    net_families[ops->family]=ops;
    .
}
```

Now that we know the basic structures, let us see what happens when a netlink socket is created in the user space. The socket() is a system call which is then resolved in the kernel, this brings us to a point where we can understand a little about how system calls work, refer to the Appendix.

2.3 Netlink Socket creation

All socket related calls are handled by the sys_socketcall() in net/socket.c, depending on the type of operation requested say SYS_SOCKET, SYS_BIND, SYS_CONNECT etc, the appropriate function is invoked.

For socket creation, take a look at the code of sys_socketcall()

```
asmlinkage int sys_socketcall(int call, unsigned long *args)
{
    .
  case SYS_SOCKET:
            err = sys_socket(a0,a1,a[2]);
            break;
    .
}
```

In sys_socket in net/socket.c the socket is created and a socket descriptor assigned for future reference. The section of the code is

```
asmlinkage int sys_socket(int family, int type, int protocol)
{
    .
    retval = sock_create(family, type, protocol, &sock);
    .
    retval = get_fd(sock->inode);
    .
}
```

The sections of code relevant in sock_create are

```
int sock_create(int family, int type, int protocol, struct socket **res)
{
    .
    .
    sock = sock_alloc();
    i = net_families[family]->create(sock, protocol);
    .
    .
}
```

For netlink sockets, as described earlier, netlink_create is called. This function associates the operations of the protocol with the socket.

```
static int netlink_create(struct socket *sock, int protocol)
{
    .
    .
    sock->ops = &netlink_ops;
    .
    .
}
```

netlink_ops gives the list of function pointers for the various operation associated with the netlink sockets.

```
struct proto_ops netlink_ops = {
    PF_NETLINK,

    sock_no_dup,
    netlink_release,
    netlink_bind,
    netlink_connect,
    sock_no_socketpair,
    sock_no_accept,
    netlink_getname,
    datagram_poll,
    sock_no_ioctl,
    sock_no_listen,
    sock_no_shutdown,
    sock_no_setsockopt,
    sock_no_getsockopt,
    sock_no_fcntl,
    netlink_sendmsg,
    netlink_recvmsg
};
```

After the netlink socket is created, the next step is to bind the socket, when a bind is issued from the user level, the sys_bind finction is called in the kernel, this in turn calls the bind function corresponding to the socket created, in our case it will be the netlink_bind that is called.

In netlink_bind, netlink_insert() is called which creates an entry for this netlink socket in the nl_table which is a list of sock structures.

```
static void netlink_insert(struct sock *sk)
{
    sk->next = nl_table[sk->protocol];
    nl_table[sk->protocol] = sk;
}
```

So the user code for the creation and binding of the netlink socket can be summarised as

```
struct sockaddr_nl address;
sock_fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
bind(sock_fd, (struct sockaddr*)&address, sizeof(address));
```

where sockaddr_nl is defined in include/linux/netlink.h

```
struct sockaddr_nl
{
    sa_family_t nl_family;  /* AF_NETLINK   */
    unsigned short  nl_pad;     /* zero     */
    __u32       nl_pid;     /* process pid  */
    __u32       nl_groups;  /* multicast groups mask */
};
```

the family is AF_NETLINK, nl_groups is used for multicast options and the nl_pid is used to represent the process id, if this given as zero, the kernel gets the current->pid from the task structure and fills it. In the kernel the check for the pid is done and if is zero, netlink_autobind() is called which does the following,

```
static int netlink_autobind(struct socket *sock)
{
    struct sock *sk = sock->sk;
    struct sock *osk;
    sk->protinfo.af_netlink.groups = 0;
    sk->protinfo.af_netlink.pid = current->pid;
```

```
    .
    .
    netlink_insert(sk);
}
```

2.4 Sending and Receiving messages

After the socket is created and bound, we can read and write using recvmsg() and sendmsg() functions.

```
sendmsg(sock_fd, &msg, 0);
```

where msg is of struct msghdr defined in /include/linux/socket.h

```
struct msghdr {
    void    *   msg_name;   /* Socket name          */
    int     msg_namelen;    /* Length of name       */
    struct iovec *  msg_iov;    /* Data blocks          */
    __kernel_size_t msg_iovlen; /* Number of blocks     */
    void    *   msg_control;    /* Per protocol magic (eg BSD file descriptor
   passing) */
    __kernel_size_t msg_controllen; /* Length of cmsg list */
    unsigned    msg_flags;
};
```

This msghdr is filled as follows,

```
  struct msghdr msg = {
        (void*)&nladdr, sizeof(nladdr),
        &iov,   1,
        NULL,   0,
        0
    };

    memset(&nladdr, 0, sizeof(nladdr));
    nladdr.nl_family = AF_NETLINK;
    nladdr.nl_pid = 0;
    nladdr.nl_groups = 0;
```

where iov is of type struct iovec.

```
struct iovec
{
    void *iov_base;     /* BSD uses caddr_t (1003.1g requires void *) */
    __kernel_size_t iov_len; /* Must be size_t (1003.1g) */
};
```

This iovec structure is filled with the data pointer and the length of the data to be passed to the kernel. In netlink_sendmsg the data sent through the iovec is copied into the kernel space as follows,

```
static int netlink_sendmsg(struct socket *sock, struct msghdr *msg, int len,
               struct scm_cookie *scm)
{
    .
    memcpy_fromiovec(skb_put(skb,len), msg->msg_iov, len);
    .
}
```

So, the data to be sent to the kernel is filled up, the iovec structure initialised and the packet sent to the kernel. The data from the iovec is copied and processed in the kernel, how the data needs to be filled depends on the kind of operation that we need to perform. Note that the nladdr's pid is set to zero, the significance of which will be explained later.

2.5 NETLINK_ROUTE Family

Now let us take a look at how the kernel netlink socket is created. For now let us focus on how the socket for the NETLINK_ROUTE family is created.

In net/core/rtnetlink.c, there is an rtnetlink_init which is of interest to us.

```
__initfunc(void rtnetlink_init(void))
{
#ifdef RTNL_DEBUG
    printk("Initializing RT netlink socket\n");
#endif
    rtnl = netlink_kernel_create(NETLINK_ROUTE, rtnetlink_rcv);
    if (rtnl == NULL)
        panic("rtnetlink_init: cannot initialize rtnetlink\n");
```

```
    register_netdevice_notifier(&rtnetlink_dev_notifier);
    rtnetlink_links[PF_UNSPEC] = link_rtnetlink_table;
    rtnetlink_links[PF_PACKET] = link_rtnetlink_table;
}
```

This function is called as part of the sock_init function in net/socket.c The function creates a netlink socket in the kernel which handles the user requests. The code of the netlink_kernel_create is

```
struct sock *
netlink_kernel_create(int unit, void (*input)(struct sock *sk, int len))
{
    .
    .
    if (netlink_create(sock, unit) < 0) {
        sock_release(sock);
        return NULL;
    }
    sk = sock->sk;
    if (input)
        sk->data_ready = input;

    netlink_insert(sk);
    .
    .
}
```

The function creates a netlink socket and then makes an entry in the nl_table, infact since this socket is created when the system comes up, it will be the first entry in that table. This netlink socket which is created will have a pid = 0, which is the reason that all user netlink sockets which want to perfrom NETLINK_ROUTE related functions have to contact this socket by setting the pid to be 0. Also note that the function is called with a function pointer rtnetlink_rcv and the data_ready pointer is set to this value. This function is significant in the sense that this is the entry point into the kernel.

The link_rtnetlink_table is a table of structures

```
struct rtnetlink_link
{
    int (*doit)(struct sk_buff *, struct nlmsghdr*, void *attr);
    int (*dumpit)(struct sk_buff *, struct netlink_callback *cb);
};
```

which consists of the doit and dumpit function pointers. The table can be indexed by the action to be performed say RTM_NEWQDISC, RTM_DELQDISC etc and the corresponding function called.

This table is furthur filled up in sched/sch_api.c as

```
link_p[RTM_NEWQDISC-RTM_BASE].doit = tc_modify_qdisc;
link_p[RTM_DELQDISC-RTM_BASE].doit = tc_get_qdisc;
link_p[RTM_GETQDISC-RTM_BASE].doit = tc_get_qdisc;
link_p[RTM_GETQDISC-RTM_BASE].dumpit = tc_dump_qdisc;
link_p[RTM_NEWTCLASS-RTM_BASE].doit = tc_ctl_tclass;
link_p[RTM_DELTCLASS-RTM_BASE].doit = tc_ctl_tclass;
link_p[RTM_GETTCLASS-RTM_BASE].doit = tc_ctl_tclass;
link_p[RTM_GETTCLASS-RTM_BASE].dumpit = tc_dump_tclass;
```

and the route related function pointers are stored in /net/ipv4/devinet.c

```
static struct rtnetlink_link inet_rtnetlink_table[RTM_MAX-RTM_BASE+1] =
{
    .
    .
    { inet_rtm_newroute,    NULL,           },
    { inet_rtm_delroute,    NULL,           },
    { inet_rtm_getroute,    inet_dump_fib,  },
    .
    .
}
rtnetlink_links[PF_INET] = inet_rtnetlink_table;
```

Now let us trace how the netlink packet from the user space finds its way in the kernel. The send_msg is mapped to sys_sendmsg which inturn calls the netlink_sendmsg() in our case, this function calls the netlink_unicast() or netlink_broadcast() as the case may be. This function identifies to which netlink socket this message has to be passed by comparing the pids of all the netlink sockets in the nl_table and calls the data_ready function of that socket which is the rtnetlink_rcv() for NETLINK_ROUTE case. The relevant section of the code is

```
int netlink_unicast(struct sock *ssk, struct sk_buff *skb, u32 pid, int
nonblock)
{
    .
    .
    for (sk = nl_table[protocol]; sk; sk = sk->next) {
        if (sk->protinfo.af_netlink.pid != pid)
            continue;
    .
    sk->data_ready(sk, len);
}
```

The flow of code from rtnetlink_rcv() is that the skb is dequeued and then passed on to rtnetlink_rcv_skb() which inturn calls the rtnetlink_rcv_msg(), this function actually extracts the operation to be performed from the netlink packet and calls the corresponding doit function by indexing into the rtnetlink_links array depending on the family, eg. for queue and class related stuff, the family is AF_UNSPEC and the indexing is done into the link_rtnetlink_table, whereas for route modifications, the indexing is done into the inet_rtnetlink_table because the family is AF_INET. Thus the appropriate function is reached and the necessary action taken and the success/failure reported to the user.

3 Packaging a netlink packet

Packaging a netlink packet involves close association with the kernel, we have to take a look at the kernel data structures and the corresponding code to understand how they are interpreted, in other words there is a format in which parameters have to be filled in the netlink packet so that the appropriate module in the kernel can perform the necessary action required. In this section let us take a simple example to illustrate how the netlink packet is formed in the user space to add an entry in the kernel routing table.

Now there are two important structures, to be noted.

```
struct nlmsghdr
{
    __u32       nlmsg_len;  /* Length of message including header */
    __u16       nlmsg_type; /* Message content */
    __u16       nlmsg_flags;    /* Additional flags */
    __u32       nlmsg_seq;  /* Sequence number */
    __u32       nlmsg_pid;  /* Sending process PID */
};

struct rtmsg
{
    unsigned char       rtm_family;
    unsigned char       rtm_dst_len;
    unsigned char       rtm_src_len;
    unsigned char       rtm_tos;

    unsigned char       rtm_table;  /* Routing table id */
    unsigned char       rtm_protocol;   /* Routing protocol; see below  */
    unsigned char       rtm_scope;  /* See below */
    unsigned char       rtm_type;   /* See below    */

    unsigned        rtm_flags;
};
```

A netlink packet for interacting with the routing table should essentially consist of the following data structure, the members of the structure include the netlink message header, the *rtmsg* and the parameters to be passed in the buffer. For setting up queues, classes and filters the rtmsg structure is replaced with the traffic control structure *tcmsg*.

Specific details are covered in the following subsection.

```
struct
{
    struct nlmsghdr     netlink_header;
    struct rtmsg        rt_message;
    char            buffer[1024];
} request;
```

This is the general structure used for encapsulating route information. The character buffer in the above structure is filled with rtattributes, each consisting of a type/len value followed by the actual value. So, an rtattribute is specified as a tuple <length, type, value>, the length value includes the size of the structure *rta* and the value for the data.

```
struct rtattr
{
    unsigned short  rta_len;
    unsigned short  rta_type;
};
```

3.1 Illustration of the usage of netlink sockets with an example

The example shown below is taken from zebra, a GNU licensed package used to configure a linux box as a router. The *zebra* and *iproute2* packages are excellent sources to understand how netlink sockets work and how they interact with the kernel. This section just provides a starting point to understand the basic data structures involved, for furthur details, code walkthrough of zebra and iproute2 is recommended.

3.1.1 Macros for handling rtattributes in the kernel

Before we actually see the parameters, it is necessary for us to understand a few macros to handle the rtattributes. These are defined in include/linux/rtnetlink.h.

The RTA_ALIGN macro is used to round off length to the nearest nibble boundary. For eg.

```
#define RTA_ALIGNTO 4
#define RTA_ALIGN(len) ( ((len)+RTA_ALIGNTO-1) & ~(RTA_ALIGNTO-1) )

RTA_ALIGN(3) translates to (7 & ~3)  returns 4
RTA_ALIGN(9) translates to (12 & ~3) returns 12

#define RTA_OK(rta,len) ((len)>0 && (rta)->rta_len>=sizeof(struct rtattr)&& \
                        (rta)->rta_len <= (len))
#define RTA_LENGTH(len) (RTA_ALIGN(sizeof(struct rtattr)) + (len))
```

The RTA_OK macro checks to see if the given len is greater than 0, if the length of the attribute is atleast the size of the struct *rta* and if the length of the attribute is lesser than the argument len passed to it.

The RTA_LENGTH macro adds the length of the type/len fields to the length of the value of the parameter, i.e RTA_LENGTH(4) translates to RTA_ALIGN(4+4) which after rounding off to the nearest nibble gives 8 as the output.

The RTA_DATA returns a pointer to the data portion of the attribute, i.e it offsets the pointer beyond the type and length and takes you to the beginning of the parameter value. RTA_LENGTH(0) returns the size of the structure *rta*, so the pointer is offset to the beginning of the data just after the type and length. RTA_PAYLOAD returns the length of the parameter value i.e the actual payload.

```
#define RTA_DATA(rta)   ((void*)(((char*)(rta)) + RTA_LENGTH(0)))
#define RTA_PAYLOAD(rta) ((int)((rta)->rta_len) - RTA_LENGTH(0))
```

Also take a look at the other macros defined in include/linux/netlink.h

```
#define NLMSG_ALIGNTO   4
#define NLMSG_LENGTH(len) ((len)+NLMSG_ALIGN(sizeof(struct nlmsghdr)))
#define NLMSG_ALIGN(len) ( ((len)+NLMSG_ALIGNTO-1) & ~(NLMSG_ALIGNTO-1) )
```

NLMSG_ALIGN is similar to the RTA_ALIGN macro, just aligns length to the nearest nibble boundary. NLMSG_LENGTH(len) adds the length given by len to the size of structure *nlmsghdr*.

3.1.2 Utility function for adding the rtattribute

This is the standard utility function used to add an rtattribute in the netlink packet, this function is taken from iproute2, thanks to Alexey Kuznetsov.

```
int addattr_l(struct nlmsghdr *n, int maxlen, int type, void *data, int alen)
{
    int len = RTA_LENGTH(alen);
    struct rtattr *rta;
    if (NLMSG_ALIGN(n->nlmsg_len) + len > maxlen)
        return -1;
    rta = (struct rtattr*)(((char*)n) + NLMSG_ALIGN(n->nlmsg_len));
    rta->rta_type = type;
    rta->rta_len = len;
    memcpy(RTA_DATA(rta), data, alen);
    n->nlmsg_len = NLMSG_ALIGN(n->nlmsg_len) + len;
    return 0;
}
```

len = RTA_LENGTH(alen) assigns the length of the data given by alen plus the length of the structure *rta* as explained by the macro. n->nlmsg_len always has the length of the packet.What the function basically does is, it offsets the pointer by getting past the length of the packet so that a new parameter can be added at that point. The pointer is typecast to struct *rta* and then the type and length of the parameter is filled followed by the copying of the actual data. As mentioned earlier, RTA_DATA takes the pointer beyond the struct *rta* and the data is then copied. The n->nlmsg_len is incremented by the length of the new parameter so that it now reflects the length of the packet after adding the new parameter.

All parameters are stuffed into the netlink packet by calling this function.

3.1.3 Adding a new route

Now as an example, we can see what parameters are required for adding a new route to the kernel routing table.

The initialisations include

```
memset(&request, 0, sizeof(request));
request.netlink_header.nlmsg_len = NLMSG_LENGTH(sizeof(struct rtmsg));
request.netlink_header.nlmsg_flags = NLM_F_REQUEST|NLM_F_CREATE;
request.netlink_header.nlmsg_type = RTM_NEWROUTE;
request.rt_message.rtm_family = AF_INET;
request.rt_message.rtm_table = RT_TABLE_MAIN;

if (cmd != RTM_DELROUTE)
{
    request.rt_message.rtm_protocol = RTPROT_BOOT;
    request.rt_message.rtm_scope = RT_SCOPE_UNIVERSE;
    request.rt_message.rtm_type = RTN_UNICAST;
}
```

Initially the netlink_header->nlmsg_len contains the length of the structures *nlmsghdr* and *rtmsg* before adding any parameter, so for the first parameter we can go to the beginning of the character buffer in the packet. The family is set as AF_INET so that we could index the inet_rtnetlink_table in the kernel by the command RTM_NEWROUTE to get to the function inet_rtm_newroute, this function in the kernel adds the new route to the kernel routing table. Also note the RT_TABLE_MAIN whose significance will be explained later.

For ipv4, The ipaddress of both the destination and the gateway needs to filled in as an unsigned integer array of 4 bytes, then the packaging into the netlink packet is done by calling the function addattr_l()

```
addattr_l(&request.netlink_header, sizeof(request), RTA_GATEWAY, &gw,4);
addattr_l(&request.netlink_header, sizeof(request), RTA_DST, &dst,4);
addattr32 (&request.netlink_header, sizeof(request), RTA_OIF, index);
```

The outgoing interface is denoted by index. Note that the parameters and the names to be filled such as RTA_GATEWAY, RTA_DST etc are defined in include/linux/rtnetlink.h, for each type of operation intended, different parameters have to be filled.

For adding a route, the gateway, destination address and the interface will suffice, now the netlink packet is all set to go to the kernel.

To send the packet to the kernel, look at the following code.

```
int status;
    struct sockaddr_nl nladdr;
    struct iovec iov = { (void*)netlink_header, netlink_header->nlmsg_len };
    char   buf[8192];
    struct msghdr msg = {
        (void*)&nladdr, sizeof(nladdr),
        &iov,   1,
        NULL,   0,
        0
    };

    memset(&nladdr, 0, sizeof(nladdr));
    nladdr.nl_family = AF_NETLINK;
    nladdr.nl_pid = 0;
    nladdr.nl_groups = 0;

    n->nlmsg_seq = ++rtnl->seq;
    if (answer == NULL)
        n->nlmsg_flags |= NLM_F_ACK;

    status = sendmsg(rtnl->fd, &msg, 0);
```

The family is set as AF_NETLINK. The iovector is formed and the message formed and sendmsg is called. The success/failure is returned as the status.

3.1.4 What happens in the kernel

When the netlink packet comes to the kernel, as explained earlier, the doit function in the inet_rtnetlink_table indexed by RTM_NEWROUTE is called and control reaches inet_rtm_newroute() in net/ipv4/fib_frontend.c

```
int inet_rtm_newroute(struct sk_buff *skb, struct nlmsghdr* nlh, void *arg)
{
    struct fib_table * tb;
    struct rtattr **rta = arg;
    struct rtmsg *r = NLMSG_DATA(nlh);
```

```
    if (inet_check_attr(r, rta))
        return -EINVAL;

    tb = fib_new_table(r->rtm_table);
    if (tb)
        return tb->tb_insert(tb, r,(struct kern_rta*)rta,nlh,&NETLINK_CB(skb));
    return -ENOBUFS;
}
```

NLMSG_DATA macro takes you to the start of the rtmessage in tha netlink packet.

inet_check_attr essentially loops through the parameter list and creates an array of parameters consisting of only the data, this is important because this is later typecasted to struct kern_rta in include/net/fib.h

```
struct kern_rta
{
    void        *rta_dst;
    void        *rta_src;
    int     *rta_iif;
    int     *rta_oif;
    void        *rta_gw;
    u32     *rta_priority;
    void        *rta_prefsrc;
    struct rtattr   *rta_mx;
    struct rtattr   *rta_mp;
    unsigned char   *rta_protoinfo;
    unsigned char   *rta_flow;
    struct rta_cacheinfo *rta_ci;
};
```

Note that this structure has a one to one correspondance with the routing table attributes so that the typecasting makes sense.

```
enum rtattr_type_t
{
    RTA_UNSPEC,
    RTA_DST,
```

```
    RTA_SRC,
    RTA_IIF,
    RTA_OIF,
    RTA_GATEWAY,
    RTA_PRIORITY,
    RTA_PREFSRC,
    RTA_METRICS,
    RTA_MULTIPATH,
    RTA_PROTOINFO,
    RTA_FLOW,
    RTA_CACHEINFO
};

static int inet_check_attr(struct rtmsg *r, struct rtattr **rta)
{
    int i;

    for (i=1; i<=RTA_MAX; i++) {
        struct rtattr *attr = rta[i-1];
        .
        .
        if (i != RTA_MULTIPATH && i != RTA_METRICS)
                rta[i-1] = (struct rtattr*)RTA_DATA(attr);
        }
    }
    return 0;
}
```

A new fib_table is created, for this the RT_TABLE_MAIN parameter is used and the insert is a function pointer which takes us to fn_hash_insert() in net/ipv4/fib_hash.c, here the individual parameters are extracted and the entry added to the forwarding information base in the kernel.

So, as said earlier, to pack a netlink packet, the corresponding code in the kernel has to be understood for the right interpretation of the packets. To perform a specific function, the kernel expects the netlink packet to be packaged in a particular format, so from the user space the parameters have to be filled in that order for them to make sense in the kernel.

A Appendix - System calls

A system call works on the basis of a defined transition from User Mode to System Mode. In Linux, this is done by calling the interrupt 0x80, together with the actual register values and the number of the system call. The kernel (in system mode) calls a kernel function out of the _sys_call_table table in arch/i386/kernel/entry.S. The conversion from a function used by a program to the system call is carried out in the C library. The actual work of the system calls is taken care of by the interrupt routine, this starts at the entry address _system_call() held in arch/i386/kernel/entry.S. When the interrupt service routine returns, the return value is read from the appropriate transfer register and the library function terminates.

As an example, there is a single system call sys_socket_call() that allows the entire programming of the sockets. The corresponding sections of the code are:

This is the entry in the sys_call_table for resolving the system call.

```
ENTRY(sys_call_table)

    .long SYMBOL_NAME(sys_socketcall)
```

The number and name of the system call is defined in include/asm-i386/unistd.h as

```
#define __NR_socketcall     102
```

B Appendix - Code Listing routeadd.c

The file included below summarises the code for adding a route to the kernel routing table, the purpose of the code is to show the flow of control, however it may not complete in all respects.

```
#include <stdio.h>
#include <asm/types.h>
#include <linux/netlink.h>
#include <linux/rtnetlink.h>

struct rtnl_handle
{
        int             fd;
        struct sockaddr_nl   local;
        struct sockaddr_nl   peer;
        __u32           seq;
        __u32           dump;
};


// This code may not be complete in all respects, it just shows the flow of
// control, for more detailed information refer to the code of iproute2 and
// zebra packages.



// This function is to open the netlink socket as the name suggests.
int netlink_open(struct rtnl_handle* rth)
{
        int addr_len;

        memset(rth, 0, sizeof(rth));

        // Creating the netlink socket of family NETLINK_ROUTE

        rth->fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
        if (rth->fd < 0) {
                perror("cannot open netlink socket");
                return -1;
        }

        memset(&rth->local, 0, sizeof(rth->local));
        rth->local.nl_family = AF_NETLINK;
```

```
        rth->local.nl_groups = 0;

        // Binding the netlink socket
        if (bind(rth->fd, (struct sockaddr*)&rth->local, sizeof(rth->local)) < 0)
        {
                perror("cannot bind netlink socket");
                return -1;
        }
        addr_len = sizeof(rth->local);

        if (getsockname(rth->fd, (struct sockaddr*)&rth->local, &addr_len) < 0)
        {
                perror("cannot getsockname");
                return -1;
        }

        if (addr_len != sizeof(rth->local)) {
                fprintf(stderr, "wrong address lenght %d\n", addr_len);
                return -1;
        }

        if (rth->local.nl_family != AF_NETLINK) {
                fprintf(stderr, "wrong address family %d\n", rth->local.nl_family);
                return -1;
        }
        rth->seq = time(NULL);
        return 0;
}

// This function does the actual reading and writing to the netlink socket
int rtnl_talk(struct rtnl_handle *rtnl, struct nlmsghdr *n, pid_t peer,
                unsigned groups, struct nlmsghdr *answer)
{
        int status;
        struct nlmsghdr *h;
        struct sockaddr_nl nladdr;

        // Forming the iovector with the netlink packet.
        struct iovec iov = { (void*)n, n->nlmsg_len };
        char    buf[8192];

        // Forming the message to be sent.
        struct msghdr msg = {
                (void*)&nladdr, sizeof(nladdr),
                &iov,   1,
```

```
                NULL,   0,
                0
        };

        // Filling up the details of the netlink socket to be contacted in the
        // kernel.
        memset(&nladdr, 0, sizeof(nladdr));
        nladdr.nl_family = AF_NETLINK;
        nladdr.nl_pid = peer;
        nladdr.nl_groups = groups;

        n->nlmsg_seq = ++rtnl->seq;
        if (answer == NULL)
                n->nlmsg_flags |= NLM_F_ACK;

        // Actual sending of the message, status contains success/failure
        status = sendmsg(rtnl->fd, &msg, 0);

        if (status < 0)
                return -1;
}

// This function forms the netlink packet to add a route to the kernel routing
// table
route_add(__u32* destination, __u32* gateway)
{
        struct rtnl_handle rth;

        // structure of the netlink packet.
        struct {
                struct nlmsghdr     n;
                struct rtmsg        r;
                char                buf[1024];
        } req;

        char   mxbuf[256];
        struct rtattr * mxrta = (void*)mxbuf;
        unsigned mxlock = 0;

        memset(&req, 0, sizeof(req));

        // Initialisation of a few parameters
        req.n.nlmsg_len = NLMSG_LENGTH(sizeof(struct rtmsg));
        req.n.nlmsg_flags = NLM_F_REQUEST|NLM_F_CREATE;
        req.n.nlmsg_type = RTM_NEWROUTE;
```

```
        req.r.rtm_family = AF_INET;
        req.r.rtm_table = RT_TABLE_MAIN;

        req.r.rtm_protocol = RTPROT_BOOT;
        req.r.rtm_scope = RT_SCOPE_UNIVERSE;
        req.r.rtm_type = RTN_UNICAST;

        mxrta->rta_type = RTA_METRICS;
        mxrta->rta_len = RTA_LENGTH(0);

        // RTA_DST and RTA_GW are the two esential parameters for adding a route,
        // there are other parameters too which are not discussed here. For ipv4,
        // the length of the address is 4 bytes.
        addattr_l(&req.n, sizeof(req), RTA_DST, destination, 4);
        addattr_l(&req.n, sizeof(req), RTA_GATEWAY, gateway, 4);

        // opening the netlink socket to communicate with the kernel
        if (rtnl_open(&rth) < 0)
        {
                fprintf(stderr, "cannot open rtnetlink\n");
                exit(1);
        }

        // sending the packet to the kernel.
        if (rtnl_talk(&rth, &req.n, 0, 0, NULL) < 0)
                exit(2);

        return 0;
}

// This is the utility function for adding the parameters to the packet.
int addattr_l(struct nlmsghdr *n, int maxlen, int type, void *data, int alen)
{
        int len = RTA_LENGTH(alen);
        struct rtattr *rta;

        if (NLMSG_ALIGN(n->nlmsg_len) + len > maxlen)
                return -1;
        rta = (struct rtattr*)(((char*)n) + NLMSG_ALIGN(n->nlmsg_len));
        rta->rta_type = type;
        rta->rta_len = len;
        memcpy(RTA_DATA(rta), data, alen);
        n->nlmsg_len = NLMSG_ALIGN(n->nlmsg_len) + len;
        return 0;
}
```