Understanding And Programming With Netlink Sockets

Neil Horman

Version 0.3

December 6, 2004

Contents

1 Introduction 3

2 The Netlink Address Family 4
- 2.1 Socket Creation 4
- 2.2 Sending and Receiving Datagrams 4
- 2.3 The Netlink Socket Address Structure 5

3 The Netlink Message Format 7
- 3.1 The Netlink Header 7
- 3.2 Netlink Utility Macros 9
- 3.3 A Visual Overview 10

4 The NETLINK_FIREWALL protocol 11
- 4.1 Creation and Use 11
- 4.2 Message Types 12
  - 4.2.1 IPQM_MODE 12
  - 4.2.2 IPQM_PACKET 13
  - 4.2.3 IPQM_VERDICT 14
- 4.3 Example use of the NETLINK_FILTER protocol 16

5 The NETLINK_ROUTE Protocol 18
- 5.1 Creation and Use 18
- 5.2 the NETLINK_ROUTE message macros 18
- 5.3 Another Visual Overview 19
- 5.4 Message Types 19
  - 5.4.1 The LINK messages 20
  - 5.4.2 The ADDR messages 21
  - 5.4.3 The ROUTE messages 23
  - 5.4.4 The NEIGH messages 24
  - 5.4.5 The RULE messages 24
  - 5.4.6 The QDISC messages 24
  - 5.4.7 The CLASS messages 24
  - 5.4.8 The FILTER messages 24

6 Adding Netlink Protocols in the Kernel 25

List of Figures

1 Netlink message layout and netlink macro interaction 10
2 Netlink route protocol message layout and macro interaction . 20

1 Introduction

Network configuration in Linux, and in various implementations of Unix in general, has always been something of an afterthought from the programmers point of view. Adding routes, neighbor table entries, or interface configuration options required a somewhat haphazard combination of raw sockets, ioctl calls and specialized pseudo-network protocols. In the 2.4 Linux kernel, developers began an effort to create a more standard interface for the configuration of the network *control plane*[1]. This configuration interface and protocol, known as *netlink sockets* aims to create a communication framework suitable for setting all aspects of the network control plane. While the build out of the netlink system‘ is not complete, it is clearly the new method for network configuration, and the infrastructure is reasonably solidified and operational. This paper aims to document the format and use of both the netlink socket family and its currently implemented protocols.

This whitepaper is intended to be used as a programming guide and reference. It assumes that the reader has a prior familiarity both with C programming, and with socket programming.

[1] The logical subsection of network communication responsible for controlling the devices which forward network data

2 The Netlink Address Family

2.1 Socket Creation

The netlink address family uses the standard BSD socket API as its interface between user-space programs and various kernel components. The creation of a netlink socket is preformed in the exact same way as any other network socket,via the **socket** library call:

socket_fd=socket(AF_NETLINK,SOCK_RAW,protocol);

The address family is always given as AF_NETLINK, and its type is always given as SOCK_RAW. The only variation in the creation of a netlink socket is the protocol provided. The list of available protocols is and will continue to change as configuration aspects of the Linux network stack add in their configurability. As of this writing, the list of available protocols is as follows:

- NETLINK_ARPD
- NETLINK_FIREWALL
- NETLINK_IP6_FW
- NETLINK_NFLOG
- NETLINK_ROUTE
- NETLINK_ROUTE6
- NETLINK_TAPBASE
- NETLINK_TCPDIAG
- NETLINK_XFRM

Each of these protocols is individually described in its own section.

2.2 Sending and Receiving Datagrams

Netlink sockets are connectionless, and operate in much the same way UDP[2] sockets do. Messages are sent to recipients on an open netlink socket via the **sendto** and **sendmsg** library calls. Messages are received by the **recvfrom** and **recvmsg** library calls. Note that messages are not exchanged with the

[2] User Datagram Protocol

send and **recv** library calls. This is because netlink sockets are connectionless. Much like UDP sockets, netlink messages are transferred in datagrams. As such there is no guarantee of delivery between socket endpoints, although there are mechanisms in the netlink message header which are designed to help the programmer add a level of reliability to the protocol for those applications which require it.

2.3 The Netlink Socket Address Structure

The netlink socket address address structure, named **struct sockaddr_nl** is passed to all calls which send or receive netlink sockets. This structure both informs the kernel networking stack of a datagrams destination, and informs a user-space program of a received frames source. The strucutre is defined as follows:

```
struct sockaddr_nl
{
   sa_family_t nl_family;
   unsigned short nl_pad;
   __u32 nl_pid;
   __u32 nl_groups;
}
```

- **nl_family** - This field defines the address family of the message being sent over the socket. This should always be set to AF_NETLINK.
- **nl_pad** - This field is unused and should always be set to zero.
- **nl_pid** - This field is PID[3] of the process that should receive the frame being sent, or the process which sent the frame being received. Set this value to the PID of the process you wish to recieve the frame, or to zero for a multicast message or to have the kernel process the message.
- **nl_groups** - This field is used for sending multicast messages over netlink. Each netlink protocol family has 32 multicast groups which processes can send and receive messages on. To subscribe to a particular group, the **bind** library call is used, and the nl_groups field is set to the appropriate bitmask. Sending multicast frames works in a simmilar fashion, by setting the nl_groups field to an appropriate set of values when calling **sendto** or **sendmsg**. Each protocol uses the multicast groups differently, if at all, and their use is defined by convention.

[3] Process Identifier

The **sockaddr_nl** structure is cast to a standard **sockaddr** structure and passed in as the appropriate parameter to the **send** and **recv** familes of library calls.

3 The Netlink Message Format

In the same way that every IP[4] message has an standard IP header, netlink messages all have an identical header on each message sent and received. Unlike other protocols however, the programmer is required to build this header for each frame. This header is used to store metadata about each netlink message and forms the base infrastructure of every netlink protocol.

3.1 The Netlink Header

The netlink header is defined as follows:

```
struct nlmsghdr {
   __u32 nlmsg_len;
   __u16 nlmsg_type;
   __u16 nlmsg_flags;
   __u32 nlmsg_seq;
   __u32 nlmsg_pid;
}
```

- **nlmsg_len** - Each netlink message header is followed by zero or more bytes of ancilliary data. This 4 byte field records the total amount of data in the message, including the header itself.

- **nlmsg_type** - This 2 byte field defines the format of the data which follows the netlink message header

- **nlmsg_flags** - This 2 byte field or logically OR'ed bits defines various control flags which determine who each message is processed and interpreted:

  - **NLM_F_REQUEST** - This flag identifies a request message. It should be set on almost all application initiated messages
  - **NLM_F_ACK** - This flag identifies a response to a previous request packet. The sequence and pid values can be used to correlate the request to the response.
  - **NLM_F_ECHO** - This flag causes the associated packet to be eched back to the sending process on the same socket.

[4] Internet Protocol

  - **NLM_F_MULTI** - This flag indicates the message is part of a multipart message. The next successive message in the chain can be obtained with the NLMSG_NEXT macro, which is detailed in section 3.2.
  - **NLM_F_ROOT** - Used With various data retrieval (GET) operations for various netlink protocols, request messages with this flag set indicate that an entire table of data should be returned rather than a single entry. Setting this flag usually results in a response message with the NLM_F_MULTI flag set. Note that while this flag is set in the netlink header, the get request is protocol specific, and the specific get request is specified in the **nlmsg_type** field.
  - **NLM_F_MATCH** - This flag indicates only a subset of data should be returned on a protocol specific GET request, as specified by a protocol specific filter. Not yet implemented.
  - **NLM_F_ATOMIC** - Indicates that any data returned by a protocol specific GET requests should be gathered atomically, which prevents the contents of the table from changing accross multiple GET requests.
  - **NLM_F_DUMP** - Not yet defined/implemented
  - **NLM_F_REPLACE** - Replace an entry in a table with a protocol specific SET request. Some netlink protocols allow the population of various tables data tables, and setting this flag allows the message to override an entry in that table.
  - **NLM_F_EXCL** - Used in conjunction with the CREATE or APPEND flags, causes a protocol specific SET message, causes the request message to fail if the entry key already exists.
  - **NLM_F_CREATE** - Indicates that the entry associated with this SET request message should be created in the table specified by the protocol data.
  - **NLM_F_APPEND** - Indicates that the entry specified in this protocol SET request should be created, specifically at the end of the table

- **nlmsg_seq** - This 4 byte field is an arbitrary number, and is used by processes that create netlink request messages to correlate those requests with thier responses.

- **nlmsg_pid** - This 4 byte field is used in a simmilar fashion to **nlmsg_seq**. It can be used to correlate request messages to response messages, and is primarily usefull in netlink sockets that use the multicast groups feature (implying that there may be several senders and receivers of data. Any message which is originated by a user process should set this field to the value returned by the **getpid** library call. Also note that it is imperative that any program receiving netlink socket messages from the kernel verify that this field is set to zero, or it is possible to expose the software to unexpected influences from other non-privlidged user space programs.

3.2 Netlink Utility Macros

The netlink header fields and the anciliary protocol data have associated utiilty macros to facilitate their computation. The following utility macros are used to build and manipulate netlink messages

- **int NLMSG_ALIGN(size_t len)** This macro accepts the length of a netlink message and rounds it up to the nearest NLMSG_ALIGNTO[5] boundary. It returns the rounded length. This macro is not normally used by applications, but rather is used by the other utility macros internally.

- **int NLMSG_LENGTH(size_t len)** Given the size of the ancilliary data to be sent with a netlink message, this macro returns the size of the payload, plus the netlink header size, rounded up to the nearest NLMSG_ALIGNTO bytes. This macro is used to set the nlmsg_len field of a netlink message header.

- **int NLMSG_SPACE(size_t len)** Returns the aligned size of the passed length. Effectively this is the same as NLMSG_LENGTH without including the size of the netlink message header

- **void *NLMSG_DATA(struct nlmsghdr *nlh)** Given a netlink header structure, this macro returns a pointer to the ancilliary data which it contains.

- **struct nlmsghdr *NLMSG_NEXT(struct nlmsghdr *nlh)** Many netlink protocols have request messages that result in multiple response messages. In these cases, if the buffer passed to the **recv** library call is large enough, multiple responses will be copied into the buffer, and the netlink header in each response will have the NLM_F_MULTI bit set in the flags field. In this case this macro can be used to walk the

[5] Usually defined as 4 bytes

chain of responses. Returns NULL in the event the message is the last in the chain for the given buffer.

- **int NLMSG_OK(struct nlmsghdr *nlh, int len)** In the event that multiple messages have been returned in a received buffer, this macro is used to ensure that the passed netlink header has not had it or its anciliary data truncated. It also guarantees that it is safe to parse using the other macros.

- **int NLMSG_PAYLOAD(nlmsghdr *nlh, int len)** This macro accepts a netlink message header, and returns the length of the ancilliary data associated with that header.

3.3 A Visual Overview

The following figure illustrates how a netlink message is laid out in memory

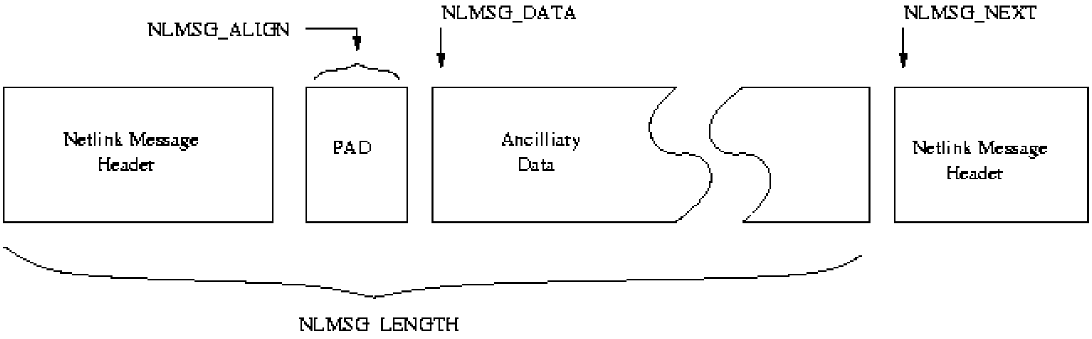

Figure 1: Netlink message layout and netlink macro interaction

4 The NETLINK_FIREWALL protocol

The NETLINK_FIREWALL protocol is an extreemely usefull development protocol. Primarily it is intended to be used a development tool to facilitate the construction and debugging of iptables modules in user space, but in conjunction with a combination of packet sockets and/or raw sockets, it can be used to drive a wide variety of network traffic patterns, for any number of applications.

This protocol is used in conjunction with several iptables modules. The ip_queue module (included in the linux kernel source) registers this protocol within the kernel. Any attempt to create a socket on this protocol before this module is added to the kernel (via insmod or modprobe) will result in failure. Also the iptables_filter module is needed, as only packets which are selected by the filter will be sent to the NETLINK_FIREWALL protocol socket. Use the iptables command to configure the filter to select those packets you are interested in examining with your user space application. For instance the following command will select all frames originating from the local host with a protocol of tcp and a destination port of 7551 to be sent to the user space program:

```
iptables -I OUTPUT -j QUEUE -p tcp –destination-port
7551
```

Note the use of the QUEUE target. This option tells iptables that any frame that matches the specified filter that it should be sent to the QUEUE target, which is implmented by the ip_queue module, which in turn sends it to the user space program via the netlink socket.

4.1 Creation and Use

NETLINK_FIREWALL packets are created with the standard socket library call, specifying NETLINK_FIREWALL as the protocol. The protocol makes no use of the netlink multicast groups, so any messages sent on a socket bound to this protocol should set the groups field of the sockaddr_nl structure to 0. Sockets on this protocol also need not make use of the **bind** library call, as packets are sent only between the kernel and a the originating process. This implies that any packets sent from a user process should set the pid field of the sockaddr_nl structure to 0, although other user processes may be designated as recipients by setting the appropriate value in the sockaddr's pid field.

4.2 Message Types

The NETLINK_FIREWALL protocol has 3 messages which can be assigned in the nlmsg_type field of the netlink message header (see section 3.1):

- **IPQM_MODE**
- **IPQM_PACKET**
- **IPQM_VERDICT**

Each message type, along with its requisite anciliary netlink data structures are detailed in their own sections below.

4.2.1 IPQM_MODE

After establishing a connection to the ip_queue module in the kernel by opening the NETLINK_FIREWALL socket, an operational mode must be established. The mode of the queue protocol defines what information about each enqueued packet to send to user space. The ancilliary data which is appended to the netlink message header has the following format:

```
typedef struct ipq_mode_msg {
   unsigned char value;
   size_t range;
};
```

- **value** - This field has one of three values:
  - **IPQ_COPY_NONE** - This is the initial mode, and causes no filter reporting to be done. All queued packets are dropped.
  - **IPQ_COPY_META** - This mode sends only metadata about the frame to userspace (the data contained in the IPQM_PACKET message)
  - **IPQ_COPY_PACKET** - This mode copies metadata and a portion of the packet to the application. Note that setting the protocol mode to IPQ_COPY_PACKET and the range to 0 will cause the entire packet to be copied in the payload for each IPQM_PACKET message.
- **range** - This optional setting is only valid when value is set to IPQ_COPY_PACKET. The range specifies how many bytes of data should be copied into the application on the receipt of a IPQM_PACKET message.

4.2.2 IPQM_PACKET

Once a mode other than IPQ_COPY_NONE is set on the socket, the socket will start receiving messages when a packet is selected by the ip_queue module in the kernel[6]. When a packet is intercepted the queue module, The application will read a mesage from the socket with the netlink message header type set to IPQM_PACKET. The ancilliary data appended to the message will have the following format:

```
typedef struct ipq_packet_msg {
   unsigned long packet_id;
   unsigned long mark;
   long timestamp_sec;
   long timestamp_usec;
   unsigned int hook;
   char indev_name[IFNAMSIZ];
   char outdev_name[IFNAMSIZ];
   unsigned short hw_protocol;
   unsigned short hw_type;
   unsigned char hw_addrlen;
   unsigned char hw_addr[8];
   size_t data_len;
   unsigned char payload[0];
};
```

- **packet_id** - This a unique number generated in the kernel and use to correlate incomming packet messages with their disposition sent back in the IPQM_VERDICT messages, described in section 4.2.3.

- **mark** - Netfilter mark value. Used in conjunction with the netfilter mangle table.

- **timestap_sec** - Arrival time of the packet.

- **timestap_usec** - Arrival time of the packet.

- **hook** - The netfilter hook number at which the packet was redirected to the queue target.

- **indev_name** - The device name the packet arrived on.

- **outdev_name** - The device name the packet was destined to (if any).

[6] Based on the iptables configuration you have set

- **hw_protocol** - The protocol specified in the MAC[7] header. Usually set to ETH_P_IP.

- **hw_type** - The media type of the interface that the packet arrived on. Usually set to ARPHDR_ETHER.

- **hw_addrlen** - The length of the MAC address of the packet.

- **hw_addr** - The source hardware address of the incomming packet.

- **data_len** - The length of the data pointed to by the payload pointer.

- **payload** - The payload of the packet. Maximum length defined by the range field of the mode message as discussed in section 4.2.1.

4.2.3 IPQM_VERDICT

The NETLINK_FIREWALL protocol is synchronous. In an effort to keep packets properly ordered, the impelmentation of the protocol requires that the user space application send an IPQM_VERDICT message after every IPQM_PACKET message is received. If the application attempts to receive two IPQM_PACKET frames without an sending an intervening IPQM_VERDICT message, the second *recvfrom* or *recvmsg* call will block. The IPQM_VERDICT message is used to release packets from the kernel ip_queue module. The packets destination (if any) any is determined by the ancilliary data in the verdict message. The ancilliary data appended to the message has the following format:

```
typedef struct ipq_verdict_msg {
   unsigned int value;
   unsigned long id;
   size_t data_len;
   unsigned char payload;
};
```

- **value** - This is the verdict to provide to the kernel queue module, which determines the action the kernel will take for a given packet. Its value is one of the following:
  - **NF_DROP** - This verdict causes the packet to be dropped immediately.

[7] Media Access Control

  - **NF_ACCEPT** - This verdict causes the packet to be accepted. The packet is sent on through the network stack without any further iptables tests being preformed.
  - **NF_STOLEN** - Take ownership of the packet but don't release netfilter resources. This is best described as "drop but allow later re-injection"
  - **NF_QUEUE** - Queue the packet to the ip_queue module for user space examination. Not often used by user space NETLINK_FIREWALL enabled applications.
  - **NF_REPEAT** - Send the packet on. Allow and subsequent check in the iptables chain to be preformed.

- **id** - This value should always be set to the **packet_id** value specified in a received IPQM_PACKET message. It is used by the kernel to correlate a particular verdict to a particular packet message.

- **data_len** - The length of the data pointed to by the payload value. This may be set to zero if there is no payload to pass back to the kernel

- textbfpayload - When sending a verdict back to the kernel, a segment of the packet may be rewritten prior to acceptance. This modification, beginning at the MAC header, is pointed to by the payload pointer. This is only usefull when the verdict returned on a packet is NF_ACCEPT, NF_QUEUE or NF_REPEAT.

4.3 Example use of the NETLINK_FILTER protocol

```
...

int netlink_socket;
int seq=0;
struct sockaddr_nl addr;
struct nlmsghdr *nl_header = NULL;
struct ipq_mode_msg *mode_data = NULL;
struct ipq_packet_msg *pkt_data = NULL;
struct ipq_verdict_msg *ver_data = NULL;
unsigned char buf1[128];
unsigned char buf2[128];

/*create the socket*/
netlink_socket = socket(AF_NETLINK,SOCK_RAW,NETLINK_FIREWALL);

...

/*set up the socket address structure*/
memset(&addr,0,sizeof(struct sockaddr_nl);
addr.nl_family=AF_NETLINK;
addr.nl_pid=0;/*packets are destined for the kernel*/
addr.nl_groups=0;/*we don't need any multicast groups*/

/*
 *we need to send a mode message first, so fill
 *out the nlmsghdr structure as such
 */
nl_header=(struct nlsmghdr *)buf1;
nl_header->nlmsg_type=IPQM_MODE;
nl_header->nlmsg_len=NLMSG_LENGTH(sizeof(struct ipq_mode_msg));
nl_header->nlmsg_flags=(NLM_F_REQUEST);/*this is a request, don't ask for an ans
nl_header->nlmsg_pid=getpid();
nl_header->nlmsg_seq=seq++;/*arbitrary unique value to allow response correlatio
mode_data=NLMSG_DATA(nl_header);
mode_data->value=IPQ_COPY_META;
mode_data->range=0;/*when mode is PACKET, 0 here means copy whole packet*/
if(sendto(netlink_socket,(void *)nl_header,nl_header->nlmsg_len,0,
         (struct sockaddr *)&addr,sizeof(struct sockaddr_nl)) < 0) {
   perror("unable to send mode message");
   exit(0);
}

/*
 *we’re ready to fileter packets
 */
for(;;) {
   if(recvfrom(netlink_socket,buf1,NLMSG_LENGTH(sizeof(struct ipq_packet_msg)),
0,&addr,sizeof(struct sockaddr_nl)) < 0) {
      perror("Unable to receive packet message");
      exit(0);
   }
   /*
    *once we have the packet message, lets extract the header and ancilliary dat
    */
   nl_header=(struct nlmsghdr *)buf1;
   pkt_data=NLMSG_DATA(nl_header);

   /*for the example just forward all packets*/
   nl_header=buf2;
   nl_header->nlmsg_type=IPQM_VERDICT;
   nl_header->nlmsg_len=NLMSG_LENGTH(sizeof(struct ipq_verdict_msg));
   nl_header->nlmsg_flags=(NLM_F_REQUEST);/*this is a request, don’t ask for an
   nl_header->nlmsg_pid=getpid();
   nl_header->nlmsg_seq=seq++;/*arbitrary unique value to allow response correla
   ver_data=(struct ipq_verdict_msg *)NLMSG_DATA(nl_header);
   ver_data->value=NF_ACCEPT;
   ver_data->id=pkt_data->packet_id;
   if(sendto(netlink_socket,(void *)nl_header,nl_header->nlmsg_len,0,(struct soc
      perror("unable to send mode message");
      exit(0);
   }
}
```

5 The NETLINK_ROUTE Protocol

5.1 Creation and Use

The NETLINK_ROUTE protocol has the largest scope and is the most mature of all the netlink family protocols. It contains its own subset of message manipulation macros that mirror the behavior and function of the netlink address family macros described in section 3.2. These macros allow the NETLINK_ROUTE protocol to implement additional ancilliary data segments that can be customized to each particular message type.

The messages that make up the NETLINK_ROUTE protocol can be divided into families, each of which control a specific aspect of the Linux kernels network routing system. The message families are:

- **LINKS**
- **ADDRESSES**
- **ROUTES**
- **NEIGHBORS**
- **RULES**
- **DISCIPLINES**
- **CLASSES**
- **FILTERS**

Each of these families are implemeted in the same message type namespace, and each family has a common structure of ancilliary data. Each ancilliary strucutre may be followed by one or more message family specific attributes, as described in section 5.2.

Each family consists of three methods; a NEW method, a DEL method, and a GET method. As each family controls a set of data objects that can be represented as a table, these methods allow a user to create table entries, delete table entries, and retrieve one or more of the objects in the table.

5.2 the NETLINK_ROUTE message macros

The NETLINK_ROUTE protocol allows for each message to follow up its defined ancilliary data with a set of subsequent segments of varying length. Each segment of data contains a header which defines its data type, and its length, as follows:

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

A user can parse the set of subsequent data segments using the following macros, which operate identically to the netlink macros as described in section 3.2:

- **int RTA_OK(struct rtattr *rta, int rtabuflen);** - Verify the data integrity of the data which succedes this rtattr header.
- **void * RTA_DATA(struct rtattr *rta);** - Return a pointer to the ancilliary data associated with this rtattr header.
- **struct rtattr *RTA_NEXT(struct rtattr *rta);** - Return a pointer to the next rtattr header in the chain.
- **unsigned int RTA_PAYLOAD(struct rtattr *rta);** - Return the length of the ancilliary data associated with the passed rtattr header.
- **unsigned int RTA_LENGTH(unsigned int length);** - Return the aligned length for the passed payload length. This value is assigned to the rta_len field of the rtattr header
- **unsigned int RTA_SPACE(unsigned int length);** - Return the length of the ancilliary data, when aligned.

5.3 Another Visual Overview

The following figure is an extention of the figure shown in section 3.3. It illustrates the memory layout of a sample netlink message which uses data structures from the NETLINK_ROUTE protocol

5.4 Message Types

As mentioned in section 5.1, The message namespace for the NETLINK_ROUTE protocol can be subdivided into families, each of which interfaces to a particular aspect of the networking subsystem in the kernel. Each group is detailed in its own subsequent section. Note that all families in this protocols operate on tables of objects. Each family has a GET member which, when used in conjunction with the NLM_F_ROOT flag (described in section **??**), can dump the entire contents of the table.

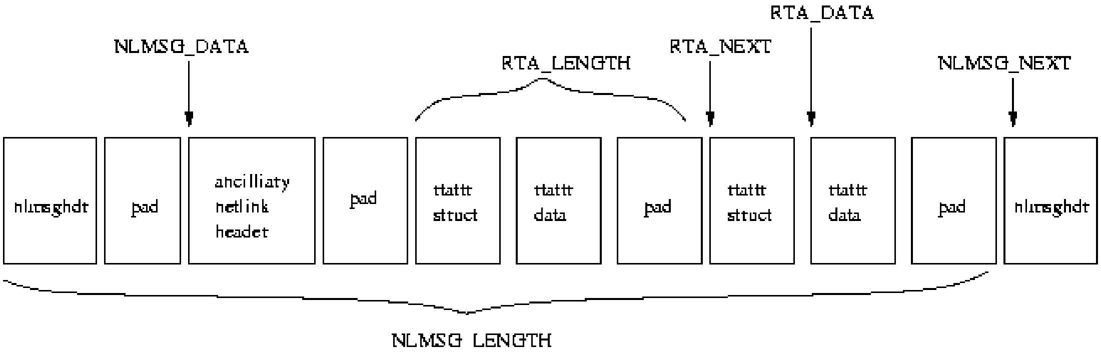

Figure 2: Netlink route protocol message layout and macro interaction

5.4.1 The LINK messages

The LINK family of messages allow a user of the NETLINK_ROUTE protocol to set and retrieve information about network interfaces on the system. It consists of three message types:

- **RTM_NEWLINK** - Create a new network interface
- **RTM_DELLINK** - Destroy a network interface
- **RTM_GETLINK** - Retrieve information about a network interface

Each of these messages contains as its ancilliary data a infomsg structure which is defined as:

```
struct ifinfomsg {
   unsigned char ifi_family;
   unsigned short ifi_type;
   int ifi_index;
   unsigned int ifi_flags;
   unsigned int ifi_change;
};
```

- **ifi_family** - The address family that this interface belongs to. For interfaces with ipv6 addresses associated this field is AF_INET6, otherwise it is set to AF_UNSPEC
- **ifi_type** - The media type of the interface. Usually set to ARPHRD_ETHER
- **ifi_index** - The unique interface number associated with this interface. Note that this number is simply a unique identifier number associated with a interface, and is in no way related to the interface name

- **ifi_flags** - The interface flags
- **ifi_change** - Reserved field. Always set this is 0xffffffff

The ifinfo data structure may be followed by zero or more attributes, each lead by an rtattr structure. The possible attributes are:

rtattr_type	**description**	**data length**	**data type**
IFLA_ADDRESS	Hardware MAC address	Varies based on ifi_type	char array
IFLA_BROADCAST	Hardware MAC broadcast address	Varies based on ifi_type	char array
IFLA_IFNAME	Interface Name	0 ¡ length ¡ IFNAMSIZ	char array
IFLA_MTU	Max transmission unit	4 bytes	unsigned int
IFLA_LINK	Link Type	4 bytes	int
IFLA_QDISC	Queue discipline	varies	char array
IFLA_STATS	interface statistics	sizeof(struct net_device_stats	struct net_device_stats

5.4.2 The ADDR messages

The ADDR family of NETLINK_ROUTE protocol messages are used to manipulate network addresses on network interfaces. This family contains three messages:

- **RTM_NEWADDR**
- **RTM_DELADDR**
- **RTM_GETADDR**

Each of these messages carries as ancilliary data a struct ifaddrmsg:

```
struct ifaddrmsg {
   unsigned char ifa_family;
   unsigned char ifa_prefixlen;
   unsigned char ifa_flags;
   unsigned char ifa_scope;
   int ifa_index;
}
```

- **ifa_family** - The address family to which this address belongs. Most often set to AF_INET
- **ifa_prefixlen** - The length of the address mask of the family, if defined
- **ifa_flags** - The flags associated with this address. Specifies (among other settings) if this address is the primary address for the interface, or if it is a secondary one. Defined as the IFA_F_* flags in rtnetlink.h
- **ifa_scope** - The scope of the address. This is kind of a loose definition of a distance to the address. Also defined in rtnetlink.hi, and always set to RT_SCOPE_HOST or RT_SCOPE_LINK.
- **ifa_ifindex** - The interface index number of the interface this address is associated to. Defined identically to ifi_ifindex from section 5.4.1.

Following the primary ancilliary data, a series of route attributes (struct rtattrs with ancilliary data) may follow. The table below details all of the route attributes that may be returned in conjunction with this family of messages.

rtattr_type	**description**	**data length**	**data type**
IFA_ADDRESS	Protocol Address	varies depending on protocol	varies by protocol
IFA_LOCAL	Protocol Address	varies depending on protocol	varies by protocol
IFA_LABEL	Interface Name	0 ¡ length ¡ IFNAMSIZ	char array
IFA_BROADCAST	Broadcast Protocol Address	varies depending on protocol	varies by protocol
IFA_ANYCAST	Anycast Protocol Address	varies by protocol	varies by protocol
IFA_CACHEINFO	Provides additional information about this address	sizeof(struct ifa_cacheinfo	struct ifa_cacheinfo

5.4.3 The ROUTE messages

The ROUTE family of messages are responsible for managing the IPV4 route table. This family consists of three messages, following with the same convention as the preceding families:

- **RTM_NEWROUTE**
- **RTM_DELROUTE**
- **RTM_GETROUTE**

Each of the ROUTE family of messages contains the following strucutres as ancilliary data[8]:

```
struct rtmsg {
   unsigned char rtm_family;
   unsigned char rtm_dst_len;
   unsigned char rtm_src_len;
   unsigned char rtm_tos;
   unsigned char rtm_table;
   unsigned char rtm_protocol;
   unsigned char rtm_scope;
   unsigned char rtm_type;
   unsigned int rtm_flags;
}
```

- **rtm_family** - The address family of the route. most commonly AF_INET.
- **rtm_dst_len** - The length of the destination address of the route entry.
- **rtm_src_len** - The length of the source address of the route entry.
- **rtm_tos** - The type of service indicator for the route.
- **rtm_table** - Specifies the routing table to operate on. See the table below for an enumeration of the values which this field can contain.
- **rtm_protocol** - Specifies the routing protocol to operate on. This value differentiates static vs. dynamic vs. other types of routes. See the table below for an enumeration of the values this field can contain.

[8] For the RTM_GETROUTE message all fields use 0 as a wildcard match except for rtm_table and rtm_protocol, which must be exactly specified

- **rtm_scope** - The network horizon to which this route applies. See the table below for an enumeration as to the value that this field can contain.

- **rtm_type** - The type of route this route is (unreachable, broadcast, blackhole, etc). See the table below for an enumeration of the values that this field can contain.

- **rtm_flags** - Miscelaneous operational flags applied to the route. See the table below for a list of value that this field can contain.

rtm_type value	**Meaning**
RT_TABLE_UNSPEC	All routing tables
RT_TABLE_DEFAULT	The default routing table
RT_TABLE_MAIN	The main routing table
RT_TABLE_LOCAL	The local routing table

rtm_protocol value	**Route Origin**
RTPROT_UNSPEC	All routing protocols
RTPROT_REDIRECT	Redirect via ICMP message
RTPROT_KERNEL	Route managed by the kernel
RTPROT_BOOT	Route added during boot
RTPROT_STATIC	Route added by admin

rtm_scope value	**Horizon of route**
RT_SCOPE_UNIVERSE	Global route
RT_SCOPE_SITE	Route within interior system (ipv6 only)

5.4.4 The NEIGH messages

5.4.5 The RULE messages

5.4.6 The QDISC messages

5.4.7 The CLASS messages

5.4.8 The FILTER messages

6 Adding Netlink Protocols in the Kernel

Just as user space programs can use the Netlink Socket family and manipulation macros to communicate with various kernel subsystems, there is an api within the kernel which allows developers to create modules which allow the netlink protocol family to be extended to provide further user space access to other kernel functionality.