Manipulating the Networking Environment Using RTNETLINK

Asanga Udugama

Abstract

How to use RTNETLINK to develop applications that control networking.

NETLINK is a facility in the Linux operating system for user-space applications to communicate with the kernel. NETLINK is an extension of the standard socket implementation. Using NETLINK, an application can send/receive information to/from different kernel features, such as networking, to check current status and control them.

In this article, I describe how a programmer can use the networking environment manipulation capability of NETLINK known as RTNETLINK. I discuss some areas of use of RTNETLINK, the relevant socket operations, the functionality, how RTNETLINK messages are formed and finally, provide a set of sample code that uses RTNETLINK. RTNETLINK for the IP version 4 environment is referred to as NETLINK_ROUTE, and for the IP version 6 environment, it is referred to as NETLINK_ROUTE6. The explanations given here are applicable for both IP versions 4 and 6.

Developers of network layer protocol handlers can use RTNETLINK to modify and monitor different components of networking, such as the routing table and network interfaces. There are many existing and upcoming protocol standards at the Internet Engineering Task Force (IETF) that can be implemented in user space. These implementations will require manipulating the routing and knowing what is being modified by other processes. Some of these protocol categories are as follows:

- Dynamic routing protocols: protocols of this category, including the Routing Information Protocol (RIP), Open Shortest Path First (OSPF) and Exterior Gateway Protocol (EGP) actively manage the routing environment of a host while communicating with other equally capable hosts or routers in the network or Internet.

- Mobility protocols: hosts that are mobile and connect to different networks at different times use protocols such as Mobile IP (MIP), Session Initiation Protocol (SIP) and Network Mobility (NEMO) to manage routing to maintain connectivity and continuity of communications.

- Ad hoc networking protocols: hosts that are mobile and located in places where there is no networking infrastructure, such as routers and WLAN access points, require peer-to-peer communications with differently configured hosts. Mobile computers of rescue workers in an earthquake-struck area or other such emergencies can use ad hoc networking protocols. These protocols, such as the Ad hoc On-demand Distance Vector (AODV) and Optimized Link State Routing (OLSR), require managing the routing to find and communicate with other hosts using neighboring hosts as routers and gateways.

It helps reduce the complexity of the kernel code if you implement these protocols in user space. Further, it simplifies the development and testing of these protocols because of the availability of many user-space development tools. Problems, such as kernel crashes, that are likely with kernel-based code when testing or when used by end users will not occur in a user-space protocol handler.

Socket Operations

The socket implementation of Linux allows two end points to communicate. The socket API provides a standard set of functions and data structures. With RTNETLINK, the two end points in communication are user space and kernel space. The following sequence of socket calls have to be made when manipulating the networking environment through RTNETLINK:

1. Open socket.
2. Bind socket to local address (using process ID).
3. Send message to the other end point.
4. Receive message from the other end point.
5. Close socket.

The socket() function opens an unattached end point to communicate with the kernel. The function prototype of this call is as follows:

```
int socket(int domain, int type, int protocol);
```

The domain refers to what type of socket is being used. For RTNETLINK, we use AF_NETLINK (PF_NETLINK). type refers to the type of protocol used when communicating. This can be raw (SOCK_RAW) or datagram (SOCK_DGRAM). This is not relevant for RTNETLINK sockets and either can be used. protocol refers to the exact NETLINK capability that we use; in our case, it is NETLINK_ROUTE. This function returns an integer with a positive number called the socket descriptor, if the socket opening was successful. This descriptor will be used in all the future RTNETLINK calls until the socket is closed. If there was a failure, a negative value is returned, and the system error variable errno included in errno.h is set to the appropriate error code.

The following is an example of a call to open an RTNETLINK socket:

```
int fd;
...
fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
```

Once the socket is opened, it has to be bound to a local address. The user application can use a unique 32-bit ID to identify the local address. The function prototype of bind is as follows:

```
int bind(int fd, struct sockaddr *my_addr,
                              socklen_t addrlen);
```

To bind, the caller must provide a local address using the sockaddr_nl structure. This structure in the linux/netlink.h #include file has the following format:

```
struct sockaddr_nl
{
  sa_family_t     nl_family; // AF_NETLINK
  unsigned short  nl_pad;    // zero
```

```
  __u32           nl_pid;    // process pid
  __u32           nl_groups; // multicast grps mask
};
```

The nl_pid must contain a unique ID, which can be created using the return of the getpid() function. This function returns the process ID of the current user process that opened the RTNETLINK socket. But, if our process consists of multiple threads with each thread opening different RTNETLINK sockets, a modified process ID can be used.

Once this structure is filled, the binding can be done. The bind function returns zero if the operation succeeded. A negative number is returned in the case of failure, and the system error variable is set. The following is an example of calling bind:

```
struct sockaddr_nl la;
...
bzero(&la, sizeof(la));
la.nl_family = AF_NETLINK;
la.nl_pad = 0;
la.nl_pid = getpid();
la.nl_groups = 0;
rtn = bind(fd, (struct sockaddr*) &la, sizeof(la));
```

If the operation you require is multicast-based, you must set nl_groups to join the multicast group associated with the required RTNETLINK operation. For example, if you want to be notified of the changes to the routing table by other processes, you must OR (|) the RTMGRP_IPV4_ROUTE and RTMGRP_NOTIFY.

Sending routing RTNETLINK messages to the kernel is done through the use of the standard sendmsg() function of the socket interface. The following is the prototype of this function:

```
ssize_t sendmsg(int fd, const struct msghdr *msg,
                                      int flags);
```

msg is a pointer to a msghdr structure. The following is the format of this structure:

```
struct msghdr
{
  void *msg_name;        //Address to send to
  socklen_t msg_namelen; //Length of address data

  struct iovec *msg_iov; //Vector of data to send
  size_t msg_iovlen;     //Number of iovec entries

  void *msg_control;     //Ancillary data
  size_t msg_controllen; //Ancillary data buf len

  int msg_flags;         //Flags on received msg
};
```

The msg_name is a pointer to a variable of the type struct sockaddr_nl. This is the destination address of the sendmsg() function. Because this message is directed to the kernel, all variables of sockaddr_nl will be initialized to zero, except the nl_family member variable. The field msg_namelen should contain the size of a struct sockaddr_nl.

msg_iov should contain a pointer to a struct iovec, which is filled with the RTNETLINK message relevant to the request being made. The caller is allowed to place multiple RTNETLINK requests, if required. msg_iovlen points to the number of struct iovec structures that were placed in msg_iov. The rest of the variables are initialized to zero.

To receive RTNETLINK messages, the recv() function is used. Here is the prototype of this function:

```
ssize_t recv(int fd, void *buf, size_t len,
                                    int flags);
```

The second and third variables are a pointer to a buffer to place the bytes read and the length of this buffer, respectively. For RTNETLINK, the buffer will contain a set of RTNETLINK messages that have to be read one after the other using a set of macros provided in the netlink.h and rtnetlink.h #include files. flags is a set of flags to indicate how the receive should be performed. For RTNETLINK, this simply can be initialized to zero.

Once the socket communications are complete, the socket has to be closed using the close() function. Here's the prototype of this function:

```
int close(int fd);
```

RTNETLINK Functionality

A programmer who develops applications that use RTNETLINK must include the following #include files at a minimum:

```
#include <bits/sockaddr.h>
#include <asm/types.h>
#include <linux/rtnetlink.h>
#include <sys/socket.h>
```

These files contain the different definitions, such as data types and structures, required to make RTNETLINK calls. Here is a short explanation of what is defined in these files relevant to RTNETLINK:

- bits/sockaddr.h: provides the definitions for the addresses used by socket functions.
- asm/types.h: provides the definitions of the data types used in the header files related to NETLINK and RTNETLINK.
- linux/rtnetlink.h: provides the macros and data structures that are used in RTNETLINK. Because RTNETLINK is based on NETLINK, this also includes the linux/netlink.h. netlink.h defines the general macros and structures that are used in all the NETLINK-based capabilities.
- sys/socket.h: provides the function prototypes and the different data structures related to the socket implementation.

The operations that can be invoked using RTNETLINK are defined in the rtnetlink.h file. Each of the operations provides three possibilities of manipulation: add/update, delete or query. These three possibilities are identified by NEW, DEL and GET. Following are the manipulation operations allowed by RTNETLINK.

General networking environment manipulation services:

- Link layer interface settings: identified by RTM_NEWLINK, RTM_DELLINK and RTM_GETLINK.
- Network layer (IP) interface settings: RTM_NEWADDR, RTM_DELADDR and RTM_GETADDR.
- Network layer routing tables: RTM_NEWROUTE, RTM_DELROUTE and RTM_GETROUTE.
- Neighbor cache that associates network layer and link layer addressing: RTM_NEWNEIGH, RTM_DELNEIGH and RTM_GETNEIGH.

Traffic shaping (management) services:

- Routing rules to direct network layer packets: RTM_NEWRULE, RTM_DELRULE and RTM_GETRULE.
- Queuing discipline settings associated with network interfaces: RTM_NEWQDISC, RTM_DELQDISC and RTM_GETQDISC.
- Traffic classes used together with queues: RTM_NEWTCLASS, RTM_DELTCLASS and RTM_GETTCLASS.
- Traffic filters associated with a queuing: RTM_NEWTFILTER, RTM_DELTFILTER and RTM_GETTFILTER.

Forming and Reading RTNETLINK Messages

RTNETLINK employs a request-response mechanism to send and receive information to manipulate the networking environment. A request or a response of RTNETLINK consists of a stream of message structures. These structures are filled by the caller, in the case of a request, or filled by the kernel, in the case of a response. To place information into these structures or to retrieve information, RTNETLINK provides a set of macros (using #define statements). Every request must contain the following structure at the beginning:

```
struct nlmsghdr
{
  __u32  nlmsg_len;   //Length of msg incl. hdr
  __u16  nlmsg_type;  //Message content
  __u16  nlmsg_flags; //Additional flags
  __u32  nlmsg_seq;   //Sequence number
  __u32  nlmsg_pid;   //Sending process PID
}
```

This structure provides information about what type of RTNETLINK message is specified in the rest of the request. It is also called the NETLINK header. Here is a brief explanation of these fields:

- nlmsg_len: should contain the length of the whole RTNETLINK message, including the length of the nlmsghdr structure. This field can be filled using the macro NLMSG_ALIGN(len), where len is the length of the message that follows this structure.
- nlmsg_type: a 16-bit flag to indicate what is contained in the message, such as RTM_NEWROUTE.
- nlmsg_flags: another 16-bit flag that further clarifies the operation specified in nlmsg_type, such as NLM_F_REQUEST.

- nlmsg_seq and nlmsg_pid: these two fields are used to identify an RTNETLINK request uniquely. The caller can place the process ID and a sequence number in these fields.

Following the nlmsghdr structure are the structures relevant to the operation being requested. Depending on the type of RTNETLINK operation, the caller must include one or more of the following structures. These are called the RTNETLINK operation headers:

- struct rtmsg: retrieving or modifying entries of the routing table requires the use of this structure.
- struct rtnexthop: a next hop in a routing entry is the next host to consider on the way to the destination. A single routing entry can have multiple next hops. Each next hop entry has many types of attributes, such as the network interface in addition to the next hop IP address.
- struct rta_cacheinfo: each route entry consists of status information that the kernel updates regularly, mainly usage information. Using this structure, a user can retrieve this information.
- struct ifaddrmsg: retrieving or modifying network layer attributes associated with a network interface requires the use of this structure.
- struct ifa_cacheinfo: similar to a route entry, a network interface also consists of information about its status, which is updated by the kernel. This structure is used to retrieve this information.
- struct ndmsg: retrieving or modifying the association information between link layer addressing and network layer addressing of neighbors, referred to as neighbor discovery, is specified through this structure.
- struct nda_cacheinfo: holds the kernel updated information related to each neighbor discovery entry.
- struct ifinfomsg: retrieving or modifying the link layer attributes related to a network interface requires the use of this structure.
- struct tcmsg: retrieving or modifying traffic shaping attributes is supplied using this structure.

Following the RTNETLINK operation header are the attributes related to the operation, such as an interface number and IP address. These attributes are specified using the struct rtattr. There is one structure for each attribute. This structure has the following format:

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

Immediately following this structure is the value of the attribute. An attribute such as an IP version 4 address will occupy a 4-byte area. The variable rta_len should contain the size of this structure plus the size of the attribute. rta_type should contain the value identifying the attribute, which are given in the enumerations defined in rtnetlink.h. enum rtattr_type_t and other enumerations provide the attribute identifiers, such as IFA_ADDRESS and NDA_DST, to be used in this field. The maximum number of attributes that you can attach is up to only the macro RTATTR_MAX. An example of attaching an attribute is as follows:

```
rtap->rta_type = RTA_DST;
rtap->rta_len = sizeof(struct rtattr) + 4;
inet_pton(AF_INET, dsts,
```

```
    ((char *)rtap) + sizeof(struct rtattr));
```

Information that is received from an RTNETLINK socket is again a stream of structures. A programmer has to identify and extract information by moving a pointer along this byte stream. To simplify this process, RTNETLINK provides a set of macros to make the buffer positioning easier:

- NLMSG_NEXT(nlh, len): returns the pointer to the next NETLINK header. nlh is the header returned previously, and len is the total length of the message. This will be called in a loop to read every message.
- NLMSG_DATA(nlh): returns the pointer to the RTNETLINK header related to the requested operation given the NETLINK header. If a route entry is being manipulated, this will return a pointer to a struct rtmsg.
- RTM_RTA(r), IFA_RTA(r), NDA_RTA(r), IFLA_RTA(r) and TCA_RTA(r): return a pointer to the start of the attributes of the respective RTNETLINK operation given the header of the RTNETLINK message (r).
- RTM_PAYLOAD(n), IFA_PAYLOAD(n), NDA_PAYLOAD(n), IFLA_PAYLOAD(n) and TCA_PAYLOAD(n): return the total length of the attributes that follow the RTNETLINK operation header given the pointer to the NETLINK header (n).
- RTA_NEXT(rta, attrlen): returns a pointer to the start of the next attribute given the last returned attribute (rta) and the remaining size (attrlen) of the attributes.

Considering a simple example where an RTNETLINK request to retrieve the routing table was sent, the reply is processed in the following manner:

```
char *buf; // ptr to RTNETLINK data
int nll; // byte length of all data
struct nlmsghdr *nlp;
struct rtmsg *rtp;
int rtl;
struct rtattr *rtap;

nlp = (struct nlmsghdr *) buf;
for(;NLMSG_OK(nlp, nll); nlp=NLMSG_NEXT(nlp, nll))
{

  // get RTNETLINK message header
  rtp = (struct rtmsg *) NLMSG_DATA(nlp);

  // get start of attributes
  rtap = (struct rtattr *) RTM_RTA(rtp);

  // get length of attributes
  rtl = RTM_PAYLOAD(nlp);

  // loop & get every attribute
  for(;RTA_OK(rtap, rtl); rtap=RTA_NEXT(rtap, rtl))
  {

    // check and process every attribute

  }
}
```

RTNETLINK Sample Walk-Through

The sample code presented here focuses on three of the operations that can be performed on the routing table:

- get_routing_table: reads the main routing table in the system.
- set_routing_table: inserts a new routing entry to the table.
- mon_routing_table: monitors the routing table changes.

All three samples use a similar main() function that calls a set of subfunctions to form RTNETLINK messages and send, receive and process the received messages. To simplify the explanation, no error handling is considered. These samples perform on the IP version 4 environment of the system (AF_INET). Here is the main() function:

```
int main(int argc, char *argv[])
{

  // open socket
  fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);

  // setup local address & bind using
  // this address
  bzero(&la, sizeof(la));
  la.nl_family = AF_NETLINK;
  la.nl_pid = getpid();
  bind(fd, (struct sockaddr*) &la, sizeof(la));


  // sub functions to create RTNETLINK message,
  // send over socket, receive reply & process
  // message
  form_request();
  send_request();
  recv_reply();
  read_reply();

  // close socket
  close(fd);
}
```

Similar to the above function, the two functions that perform the socket communications are almost common to all the samples. These two functions simply send a formed message to the kernel and receive messages sent by the kernel. Exceptions here are the set_routing_table and mon_routing_table samples. In set_routing_table, a receive phase is not considered. In the mon_routing_table, a send phase is not present as it attempts to monitor only the state of the routing environment to see what is being changed. This information is mulitcast by the kernel to all the RTNETLINK sockets that are in the appropriate receiving state.

First, here's the code for send_request():

```
void send_request()
{
  // create the remote address
  // to communicate
  bzero(&pa, sizeof(pa));
  pa.nl_family = AF_NETLINK;
```

```
  // initialize & create the struct msghdr supplied
  // to the sendmsg() function
  bzero(&msg, sizeof(msg));
  msg.msg_name = (void *) &pa;
  msg.msg_namelen = sizeof(pa);

  // place the pointer & size of the RTNETLINK
  // message in the struct msghdr
  iov.iov_base = (void *) &req.nl;
  iov.iov_len = req.nl.nlmsg_len;
  msg.msg_iov = &iov;
  msg.msg_iovlen = 1;

  // send the RTNETLINK message to kernel
  rtn = sendmsg(fd, &msg, 0);
}
```

And, here's the recv_reply():

```
void recv_reply()
{
  char *p;

  // initialize the socket read buffer
  bzero(buf, sizeof(buf));

  p = buf;
  nll = 0;

  // read from the socket until the NLMSG_DONE is
  // returned in the type of the RTNETLINK message
  // or if it was a monitoring socket
  while(1) {
    rtn = recv(fd, p, sizeof(buf) - nll, 0);

    nlp = (struct nlmsghdr *) p;

    if(nlp->nlmsg_type == NLMSG_DONE)
      break;

    // increment the buffer pointer to place
    // next message
    p += rtn;

    // increment the total size by the size of
    // the last received message
    nll += rtn;

    if((la.nl_groups & RTMGRP_IPV4_ROUTE)
                     == RTMGRP_IPV4_ROUTE)
      break;
  }
}
```

The above functions and the following ones use a set of globally defined variables. These are used for all the socket operations as well as for forming and processing RTNETLINK messages:

```
// buffer to hold the RTNETLINK request
struct {
  struct nlmsghdr nl;
  struct rtmsg    rt;
  char            buf[8192];
```

```
} req;

// variables used for
// socket communications
int fd;
struct sockaddr_nl la;
struct sockaddr_nl pa;
struct msghdr msg;
struct iovec iov;
int rtn;

// buffer to hold the RTNETLINK reply(ies)
char buf[8192];

// RTNETLINK message pointers & lengths
// used when processing messages
struct nlmsghdr *nlp;
int nll;
struct rtmsg *rtp;
int rtl;
struct rtattr *rtap;
```

The get_routing_table sample retrieves the main routing table of the IPv4 environment. The form_request() function is as follows:

```
void form_request()
{
  // initialize the request buffer
  bzero(&req, sizeof(req));

  // set the NETLINK header
  req.nl.nlmsg_len
             = NLMSG_LENGTH(sizeof(struct rtmsg));
  req.nl.nlmsg_flags = NLM_F_REQUEST | NLM_F_DUMP;
  req.nl.nlmsg_type = RTM_GETROUTE;

  // set the routing message header
  req.rt.rtm_family = AF_INET;
  req.rt.rtm_table = RT_TABLE_MAIN;
}
```

The received message for the RTNETLINK request in the buf variable to retrieve the routing table is processed by the read_reply() function. Here is the code of this function:

```
void read_reply()
{
  // string to hold content of the route
  // table (i.e. one entry)
  char dsts[24], gws[24], ifs[16], ms[24];

  // outer loop: loops thru all the NETLINK
  // headers that also include the route entry
  // header
  nlp = (struct nlmsghdr *) buf;
  for(;NLMSG_OK(nlp, nll);nlp=NLMSG_NEXT(nlp, nll))
  {

    // get route entry header
    rtp = (struct rtmsg *) NLMSG_DATA(nlp);

    // we are only concerned about the
    // main route table
    if(rtp->rtm_table != RT_TABLE_MAIN)
```

```
      continue;

    // init all the strings
    bzero(dsts, sizeof(dsts));
    bzero(gws, sizeof(gws));
    bzero(ifs, sizeof(ifs));
    bzero(ms, sizeof(ms));

    // inner loop: loop thru all the attributes of
    // one route entry
    rtap = (struct rtattr *) RTM_RTA(rtp);
    rtl = RTM_PAYLOAD(nlp);
    for(;RTA_OK(rtap, rtl);rtap=RTA_NEXT(rtap,rtl))
    {
      switch(rtap->rta_type)
      {
        // destination IPv4 address
        case RTA_DST:
          inet_ntop(AF_INET, RTA_DATA(rtap),
                                    dsts, 24);
          break;

        // next hop IPv4 address
        case RTA_GATEWAY:
          inet_ntop(AF_INET, RTA_DATA(rtap),
                                    gws, 24);
          break;

        // unique ID associated with the network
        // interface
        case RTA_OIF:
          sprintf(ifs, "%d",
                    *((int *) RTA_DATA(rtap)));
        default:
          break;
      }
    }
    sprintf(ms, "%d", rtp->rtm_dst_len);

    printf("dst %s/%s gw %s if %s\n",
                          dsts, ms, gws, ifs);
  }
}
```

The set_routing_table sample sends an RTNETLINK request to insert an entry to the routing table. The route entry that is inserted is a host route (32-bit network prefix) to a private IP address (192.168.0.100) through interface number 2. These values are defined in the variables dsts (destination IP address), ifcn (interface number) and pn (prefix length). You can run the get_routing_table sample to get an idea about the interface numbers and the IP network in your system. Here's the form_request():

```
void form_request()
{
  // attributes of the route entry
  char dsts[24] = "192.168.0.100";
  int ifcn = 2, pn = 32;

  // initialize RTNETLINK request buffer
  bzero(&req, sizeof(req));

  // compute the initial length of the
  // service request
  rtl = sizeof(struct rtmsg);

  // add first attrib:
```

```
  // set destination IP addr and increment the
  // RTNETLINK buffer size
  rtap = (struct rtattr *) req.buf;
  rtap->rta_type = RTA_DST;
  rtap->rta_len = sizeof(struct rtattr) + 4;
  inet_pton(AF_INET, dsts,
     ((char *)rtap) + sizeof(struct rtattr));
  rtl += rtap->rta_len;

  // add second attrib:
  // set ifc index and increment the size
  rtap = (struct rtattr *) (((char *)rtap)
            + rtap->rta_len);
  rtap->rta_type = RTA_OIF;
  rtap->rta_len = sizeof(struct rtattr) + 4;
  memcpy(((char *)rtap) + sizeof(struct rtattr),
           &ifcn, 4);
  rtl += rtap->rta_len;

  // setup the NETLINK header
  req.nl.nlmsg_len = NLMSG_LENGTH(rtl);
  req.nl.nlmsg_flags = NLM_F_REQUEST | NLM_F_CREATE;
  req.nl.nlmsg_type = RTM_NEWROUTE;

  // setup the service header (struct rtmsg)
  req.rt.rtm_family = AF_INET;
  req.rt.rtm_table = RT_TABLE_MAIN;
  req.rt.rtm_protocol = RTPROT_STATIC;
  req.rt.rtm_scope = RT_SCOPE_UNIVERSE;
  req.rt.rtm_type = RTN_UNICAST;
  // set the network prefix size
  req.rt.rtm_dst_len = pn;
}
```

The mon_routing_table sample reads the RTNETLINK messages received when other processes change the system's main routing table. This function will use the same read_reply() function to process the messages. The main() function requires a slight change. Because this operation involves listening to multicast messages of the kernel, the local address to which we bind, it also must include the two flags RTMGRP_IPV4_ROUTE and RTMGRP_NOTIFY. Here is the required change:

```
la.nl_groups = RTMGRP_IPV4_ROUTE | RTMGRP_NOTIFY;
```

Once mon_routing_table is executed, run a **`route add`** or a **`route del`** command from another shell prompt to see the results.

Conclusion

RTNETLINK is a simple, yet versatile way of manipulating the networking environment of a Linux host. User-space network protocol handlers are ideal candidates for using RTNETLINK. The advanced IP routing command suite, referred to as IPROUTE2, is based on RTNETLINK. More information about the different operations and flags of RTNETLINK can be found at NETLINK(7) and RTNETLINK(7).

The sample code for this article is available at ftp://ftp.ssc.com/pub/lj/listings/issue145/8498.tgz.

Acknowledgement

I sincerely thank Professor Carmelita Goerg.