Communicating between the kernel and user-space in Linux using Netlink Sockets: Source code reference

Pablo Neira Ayuso

This document is the continuation of *Communication between the kernel and user-space in Linux using Netlink Sockets* published in *Software Practise and Experience*. This part covers the programming aspects of Netlink and GeNetlink with explained source code examples.

1 PROGRAMMING NETLINK SOCKETS

Adding Netlink support from scratch for some Linux kernel subsystem requires some coding in user and kernel-space. There are a lot of common tasks in parsing, validating, constructing of both the Netlink header and TLVs that are repetitive and easy to get wrong. Instead of replicating code, Linux provides a lot of helper functions.

Moreover, many existing subsystems in the Linux kernel already support Netlink sockets, thus, the kernel-space coding could be skipped. However, for the purpose of this tutorial we cover both scenarios: programming Netlink from user and kernel-space [1].

1.1 Netlink sockets from kernel-space

We have coded a simple Linux kernel module that implements Netlink support to print the "Hello world" message [2] when it receives a message from user-space. In Figure. **??**, we have represented the sequence diagram of our example.

We use the Listing. **??** as reference to start explaining the steps that you have to follow to add a new Netlink-based interface, that are:

1. Reserve a Netlink bus for your new kernel subsystem: you have to edit *include/linux/netlink.h* to add your new Netlink bus [3]. You may also

[1] Adding new Netlink busses is almost guaranteed to be reject for mainline inclusion unless that you have a good reason to do so. The current policy is to use GeNetlink that we cover in this work in Section **??**. It is a good idea to contact Linux networking developers via the *linux-netdev* development mailing-list before going ahead with your new Netlink subsystem.

[2] The message is stored in the kernel logging ring buffer which is usually stored in the log files by syslogd or, alternatively, it can be displayed by means of *dmesg*.

[3] We have selected NETLINK_EXAMPLE which is not used in the Linux kernel 2.6.30

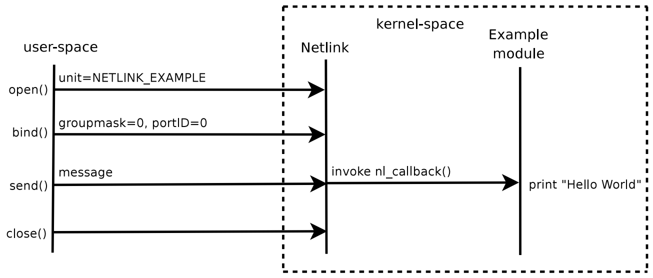

Figure 1: Hello world example via Netlink

include the bus definition in your code to avoid modifying the Linux kernel source code (lines 7-9). However, you have to make sure that you use an available Netlink bus between the existing slots. Netlink busses are unique so you have to avoid possible overlaps with other kernel subsystems.

2. Include the appropriate headers (lines 1-5), in particular it is important to include the headers that define the prototypes of the functions required to register a new Netlink subsystem (lines 3-5) that are the generic BSD socket infrastructure (line 3), the generic network socket buffer (line 4) and the Netlink definitions and prototypes (line 5).

3. Call *netlink_kernel_create()* in the module initialization path to create the Netlink kernel socket: you have to pass a callback that is invoked when your new Netlink bus receives a message from user-space (see lines 23-28). The function *netlink_kernel_create()* returns a pointer to the *sock* structure, which is used to store the kernel part of a generic BSD socket.

4. Call *netlink_kernel_release()* in the module exit path, which is executed when the Linux kernel module is removed to unregister the given Netlink socket bus (lines 38).

```
1  #include <linux/kernel.h>
2  #include <linux/module.h>
3  #include <net/sock.h>
4  #include <linux/skbuff.h>
5  #include <linux/netlink.h>
6
7  #ifndef NETLINK_EXAMPLE 21
8  #define NETLINK_EXAMPLE 21
9  #endif
10
11 #define NLEX_GRP_MAX 0
12
13 static struct sock *nlsk;
14
15 static void
16 nl_callback(struct sk_buff *skb)
17 {
```

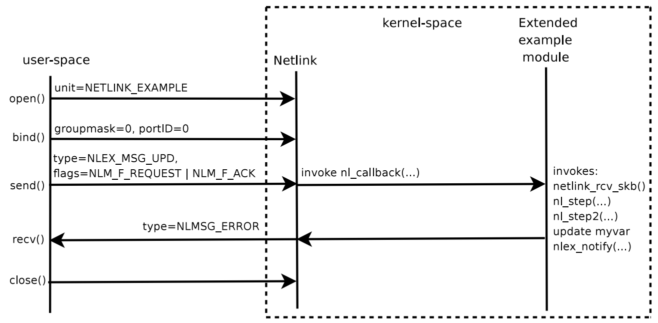

Figure 2: Update of the variable *myvar* via Netlink

```
18      printk("Hello world\n");
19  }
20
21  static int __init nlexample_init(void)
22  {
23      nlsk = netlink_kernel_create(&init_net,
24                                   NETLINK_EXAMPLE,
25                                   NLEX_GRP_MAX,
26                                   nl_callback,
27                                   NULL,
28                                   THIS_MODULE);
29      if (nlsk == NULL) {
30          printk(KERN_ERR "Can't create netlink\n");
31          return -ENOMEM;
32      }
33      return 0;
34  }
35
36  void __exit nlexample_exit(void)
37  {
38      netlink_kernel_release(nlsk);
39  }
40
41  module_init(nlexample_init);
42  module_exit(nlexample_exit);
```

Listing 1: Simple Netlink kernel module

Note that our example module that we have represented in Listing. **??** registers no multicast groups (see constant NLEX_GRP_MAX which has been set to zero in line 11) that has been passed as parameter to *netlink_kernel_create()* in lines 23-28).

Using Netlink to print some output when a message is received is a good starting point, but it is not very useful. For that reason, we present a new example that exposes the variable *myvar* to user-space by means of Netlink in Listing. **??** and Listing. **??**. This example supports two new operations: *a)* update the value of *myvar* and *b)* get the current value of *myvar*.

In Figure. **??**, we have represented the sequence diagram of an update of the variable *myvar*. With regards to coding, we initially have to declare these two new actions in the header file of the Linux kernel module in Listing. **??**.

```
 1  #ifndef _NLEXAMPLE_H_
 2  #define _NLEXAMPLE_H_
 3
 4  #ifndef NETLINK_EXAMPLE
 5  #define NETLINK_EXAMPLE 21
 6  #endif
 7
 8  enum nlexample_msg_types {
 9      NLEX_MSG_BASE = NLMSG_MIN_TYPE,
10      NLEX_MSG_UPD = NLEX_MSG_BASE,
11      NLEX_MSG_GET,
12      NLEX_MSG_MAX
13  };
14
15  enum nlexample_attr {
16      NLE_UNSPEC,
17      NLE_MYVAR,
18      /* add your new attributes here */
19      __NLE_MAX,
20  };
21  #define NLE_MAX (__NLE_MAX - 1)
22
23  #define NLEX_GRP_MYVAR (1 << 0)
24  #endif
```

Listing 2: Example header file

This listing contains the following sections that are:

1. The Netlink bus that we are using to register our new Netlink subsystem (lines 4-6), as we previously did in our "Hello World" example.

2. The definition of supported actions (lines 8-13): the actions are mapped to Netlink message types starting by 16 to skip the reserved control message types from 0 to 15. We have defined *NLEX_MSG_UPD* to update the value of *myvar*, and *NLEX_MSG_GET* to obtain the current value.

3. The definition of the attribute types (lines 15-20): we have defined the attribute *NLE_MYVAR* which can be used to store the new value of *myvar* in the TLV Netlink payload. This new attribute can be used in messages going from user to kernel-space to set the new value of *myvar*, but it can also be used to encapsulate the value of *myvar* for messages going from kernel to user-space as a result of a request to obtain the current value.

4. The definition of the only multicast group supported by this example (NLEX_GRP_MYVAR in line 23) that allows you to subscribe to event notifications of changes in *myvar*.

In Listing. **??**, we have replaced the *nl_callback()* function that we have previously shown in Listing. **??** (line 16) to support the new actions defined in Listing. **??**.

```c
1  static const
2  struct nla_policy nlex_policy[NLE_MAX+1] = {
3    [NLE_MYVAR] = { .type = NLA_U32 },
4  };
5
6  static int
7  nl_step(struct sk_buff *skb,
8          struct nlmsghdr *nlh)
9  {
10    int err;
11    struct nlattr *cda[NLE_MAX+1];
12    struct nlattr *attr = NLMSG_DATA(nlh);
13    int attrlen = nlh->nlmsg_len - NLMSG_SPACE(0);
14
15    if (security_netlink_recv(skb, CAP_NET_ADMIN))
16       return -EPERM;
17
18    if (nlh->nlmsg_len < NLMSG_SPACE(0))
19       return -EINVAL;
20
21    err = nla_parse(cda, NLE_MAX,
22                    attr, attrlen, nlex_policy);
23    if (err < 0)
24       return err;
25
26    return nl_step2(cda);
27 }
28
29 static void
30 nl_callback(struct sk_buff *skb)
31 {
32    netlink_rcv_skb(skb, &nl_step);
33 }
```

Listing 3: Extending the nl_callback function

In this new version of *nl_callback()* (lines 29-33), we call *netlink_rcv_skb()* (line 32) which is a generic function that is used to perform initial sanity checkings on the message received. This function also appropriately handles Netlink message batches coming from user-space. The sanity checkings consists of the following:

1. Check if the message contains enough room for the Netlink header.

2. Check if the *NLM_F_REQUEST* flag has been set, which is used to tell that this message is going from user to kernel-space.

3. Skip control messages if they are sent from user-space: Netlink replies with an acknowledgment message in this case. User-space applications do not usually send control messages to user-space.

4. Check if the *NLM_F_ACK* flag has been set, in that case, the user-space process has explicitly requested an acknowledgment from kernel-space to make sure that the request has been performed successfully.

Once the sanity checkings have been passed, the *netlink_rcv_skb()* function invokes the callback function passed as argument, which we have called *nl_step()* (lines 6-27 in Listing. **??**). The *nl_step()* function initially performs further non-generic sanity checkings, one of them to ensure that only user-space processes with *CAP_NET_ADMIN* capabilities (that is, the process can perform network administration tasks. Generally speaking, processes that run as superuser have this capability) (lines 15-16) can send messages to kernel-space. Then, we proceed to parse the payload of the Netlink message that contains the attributes in TLV format. In order to perform the payload parsing, we use *nla_parse()* which takes an array of pointers to Netlink attributes (see lines 21-22), whose size is the maximum number of attributes plus one (line 11), and updates the array to allow easy access to the attributes. The parsing includes the data validation to ensure that the data type stored in the attribute payload is correct (line 22, fifth parameter). This is done by means of a structure that defines the correspondence between the type stored in the attribute payload and the attribute type (lines 1-4). If there are problems in the parsing, an error is reported to user-space.

Once the parsing has been done, we call the function *nl_step2()* (line 26) which is implemented in Listing. **??**.

```
1  static int myvar;
2
3  static int
4  nl_step2(struct nlattr *cda[],
5           struct nlmsghdr *nlh)
6  {
7     int echo = nlh->nlmsg_flags & NLM_F_ECHO;
8     int pid = nlh->nlmsg_pid;
9
10    switch(nlh->nlmsg_type) {
11       case NLEX_MSG_UPD:
12          if (!cda[NLE_MYVAR])
13             return -EINVAL;
14
15          myvar = nla_get_u32(cda[NLE_MYVAR]);
16          nlex_notify(echo, pid);
17          break;
18       case NLEX_MSG_GET:
19          nlex_unicast(pid);
20          break;
21    };
22    return 0;
23 }
```

Listing 4: nl_step2() function

This function takes the array of Netlink attributes and it handles the message depending on its type according to the two possible actions that we have already defined in Listing. **??**. We handle the message depending on its type, that can be:

- NLEX_MSG_UPD: we check if the *NLE_MYVAR* attribute is set (lines 12-13), in that case, it takes the 32-bits value and it updates *myvar* (line 15).

Otherwise, it returns *invalid argument* (EINVAL) to report a malformed message since a mandatory attribute is missing.

- NLEX_MSG_GET: it creates and delivers a Netlink message to user-space containing the current value of *myvar* via unicast. We have implemented the unicast deliver by means of the function *nlex_unicast()*, which is implemented in Listing. **??**.

The NLEX_MSG_UPD command also notifies to user-space listeners that the value of *myvar* has changed (line 16 in Listing. **??**). This event notification is implemented by means of the *nlex_notify()* function in Listing. **??**.

```
 1  static int
 2  nlex_notify(int rep, int pid)
 3  {
 4      struct sk_buff *skb;
 5
 6      skb = nlmsg_new(NLMSG_DEFAULT_SIZE,GFP_KERNEL);
 7      if (skb == NULL)
 8          return -ENOMEM;
 9
10      nlmsg_put(skb, pid, rep, NLEX_MSG_UPD, 0, 0);
11      NLA_PUT_U32(skb, NLE_MYVAR, myvar);
12
13      nlmsg_notify(nlsk, skb, pid,
14                   NLEX_GRP_MYVAR,
15                   rep, GFP_KERNEL);
16      return 0;
17
18  nla_put_failure:
19      return -1;
20  }
```

Listing 5: Broadcast notification of changes in myvar

This function allocates memory for the Netlink message that will be send to the user-space listeners that are subscribed to the NLEX_GRP_MYVAR group. The message is stored in a network buffer of one memory page size (NLMSG_DEFAULT_SIZE is the size of one memory page minus the header of a Netlink message) for performance reasons (line 6).

We use the function *nlmsg_put()* to add the netlink header (line 10) and the *NLA_PUT_U32()* macro to add the value of *myvar* in TLV format as the payload of the Netlink message (line 11). To deliver the Netlink message, we use *nlmsg_notify()* which sends notifications to all user-space listeners that are subscribed to the Netlink multicast group that alerts about changes in *myvar*. This function includes the Port-ID in the message (third parameter, line 13) to report other listeners that this change was triggered by another user-space process with that Port-ID. The multicast group is also passed (line 14) to tell Netlink to what multicast group it has to deliver the change report.

Note that the *nlmsg_notify()* function also supports the NLM_F_ECHO flag that, if it is set, results in an unicast delivery to the origin of the change. This is useful in case that the origin is also subscribed to changes via multicast. Thus, it allows the origin to identify that the change report that it has received was generated by itself.

In Listing. 6, we have implemented *nlex_unicast* which performs the allocation (lines 6-8), building (lines 10-11) and delivery (line 13) of a Netlink message via unicast as response to a request to get the current value of *myvar* (NLEX_MSG_GET message).

```
1  static int
2  nlex_unicast(int pid)
3  {
4     struct sk_buff *skb;
5
6     skb = nlmsg_new(NLMSG_DEFAULT_SIZE,GFP_KERNEL);
7     if (skb == NULL)
8        return -ENOMEM;
9
10    nlmsg_put(skb, pid, 0, NLEX_MSG_UPD, 0, 0);
11    NLA_PUT_U32(skb, NLE_MYVAR, myvar);
12
13    nlmsg_unicast(nlsk, skb, pid);
14    return 0;
15
16 nla_put_failure:
17    return -1;
18 }
```

Listing 6: Unicast delivery to obtain the value of *myvar*

1.2 Netlink sockets in user-space

Netlink sockets are implemented on top of the generic BSD sockets interface. Thus, programming Netlink sockets in user-space is similar to programming common TCP/IP sockets. However, we have to take into consideration the aspects that make Netlink sockets different from other socket families, more relevantly:

1. Netlink sockets do not hide protocol details to user-space as other protocols do. In fact, Netlink passes the whole message, including the Netlink header and attributes in TLV format as well as multi-part messages, to user-space. This makes the data handling different than common TCP/IP sockets since the user-space program have to appropriately parse and build Netlink messages according to its format. However, there are no standard facilities to perform these tasks so you would need to implement your own functions or use some existing library to assist your development.

2. Errors that comes from Netlink and kernel subsystems are not returned by *recvmsg()* as an integer. Instead, errors are encapsulated in the Netlink error message. There is one exception to this rule that is the *No buffer space available* (ENOBUFS) error, which is not encapsulated since its purpose is to report that we cannot enqueue a new Netlink message. Standard

generic socket errors, like *Resource temporarily unavailable* (EAGAIN), which are commonly used together with polling primitives, like *poll()* and *select()*, are still returned as an integer by *recvmsg()*.

In order to simplify the work with Netlink sockets, we propose in this work the use of *libmnl* [?], written in C, which is a minimalistic user-space Netlink library oriented to Netlink developers. This library allows you to re-use code and it avoids re-inventing the wheel. The main features of this library are:

1. Small: the shared library requires around 20KB in a x86-based computer.
2. Simple: this library avoids complexity and elaborated abstractions that tend to hide Netlink details.
3. Easy to use: the library simplifies the work for Netlink-wise developers. It provides functions to make socket handling, message building, parsing and sequence tracking, easier.
4. Easy to re-use: you can use the library to build your own abstraction layer on top.
5. Decoupling: the interdependency of the main bricks that compose the library are reduced, eg. the library provides helper facilities like a callback-oriented API to handle Netlink messages but the programmer is not forced to use it.

There are other existing user-space Netlink libraries like:

- *libnetlink* which was the primer library, written in C. It was developed by A.Kuznetsov and distributed within the Linux's advanced routing tools *iproute2* [?]. This library is intended for internal use of the *iproute2* command line tools.
- *libnl* [?] is a complete library, also written in C, that aims to simplify the work with Netlink sockets. The development was started 2003 by Thomas Graf, it is currently the recommended library in the Linux kernel documentation [?]. It provides an object-oriented abstraction layer to forget about Netlink sockets details. Thus, obtaining information from Netlink kernel subsystems is fairly easy and it allows very rapid application development for non-Netlink developers. This library hides many details of Netlink, for the purpose of this work, we have preferred to focus on a library that adds very few abstractions.

1.3 Netlink sockets in user-space using libmnl

We have written two user-space programs using the *libmnl* library for our example Linux kernel module that we have exposed in Section **??**, they are:

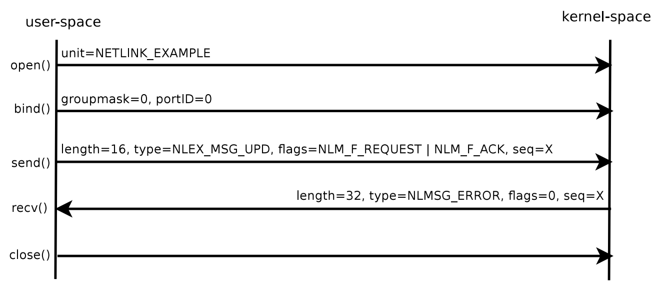
user-space
kernel-space
open()
unit=NETLINK_EXAMPLE
bind()
groupmask=0, portID=0
send()
length=16, type=NLEX_MSG_UPD, flags=NLM_F_REQUEST | NLM_F_ACK, seq=X
recv()
length=32, type=NLMSG_ERROR, flags=0, seq=X
close()

Figure 3: Update of the *myvar* variable via Netlink

- *change.c* that is used to change the current value of *myvar*, the sequence diagram of this program is represented in Figure. **??** and the implementation in Listing 7.
- *event.c* which allows to listen to asynchronous event notification that report changes in the *myvar* value. This program is available in Listing 8. In Figure. **??**, we have represented the sequence diagram of the asynchronous event notification.

In Listing 7, we initially reserve room (line 17) for the Netlink header in a large enough buffer that we have previously allocated in the stack (line 11). Then, we fill the Netlink header fields with the message type, flags and sequence number (lines 18-20). The message type (line 18) is NLEX_MSG_UPD in this case to tell the kernel module that this message contains an update for *myvar*. With regards to the flags, we have set NLM_F_REQUEST which is mandatory for requests that go from user to kernel-space and NLM_F_ACK to ask for an explicit Netlink error message from kernel-space containing the result of the operation (0 in case of success, otherwise a standard errno value). Finally, to generate the sequence numbering (line 20), we have selected the function *time()* which returns the seconds since 1970. The kernel uses the same sequence number in the acknowledgment and data replies, thus, the sequence number provides a way to identify that a given message comes as reply of certain request. Therefore, there is no need to use incremental sequence numbers.

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <unistd.h>
 4
 5  #include <libmnl/libmnl.h>
 6
 7  #include "nlexample.h"
 8
 9  int main(void)
10  {
11    char buf[getpagesize()];
12    struct nlmsghdr *nlh;
13    struct mnl_socket *nl;
14     int ret, numbytes;
15     unsigned int seq, oper;
16 
17     nlh = mnl_nlmsg_put_header(buf);
18     nlh->nlmsg_type = NLEX_MSG_UPD;
19     nlh->nlmsg_flags = NLM_F_REQUEST|NLM_F_ACK;
20     nlh->nlmsg_seq = seq = time(NULL);
21 
22     mnl_attr_put_u32(nlh,NLE_MYVAR,10);
23 
24     numbytes = mnl_nlmsg_get_len(nlh);
25 
26     nl = mnl_socket_open(NETLINK_EXAMPLE);
27     if (nl == NULL) {
28       perror("mnl_socket_open");
29       exit(EXIT_FAILURE);
30     }
31 
32     ret = mnl_socket_bind(nl,0,0);
33     if (ret == -1) {
34       perror("mnl_socket_bind");
35       exit(EXIT_FAILURE);
36     }
37 
38     ret = mnl_socket_sendto(nl,nlh,numbytes);
39     if (ret == -1) {
40       perror("mnl_socket_send");
41       exit(EXIT_FAILURE);
42     }
43 
44     ret = mnl_socket_recvfrom(nl,buf,sizeof(buf));
45     if (ret == -1) {
46       perror("recvfrom");
47       exit(EXIT_FAILURE);
48     }
49 
50     ret = mnl_cb_run(buf,ret,seq,NULL,NULL);
51     if (ret == -1) {
52       perror("callback");
53       exit(EXIT_FAILURE);
54     }
55     mnl_socket_close(nl);
56   }
```

Listing 7: *change.c* program

Once we have filled the Netlink header, we proceed with the TLV-payload building. In our case, the payload is simple since it is only composed of one attribute, which is NLE_MYVAR (line 22). This Netlink attribute contains the new value that we want to assign to the kernel-space variable *myvar*.

Now, it is time to create an user-space Netlink socket to send the message that we have built to kernel-space. Basically, we open the Netlink socket (line 26), then bind it (line 32) to no multicast groups (second parameter, line 32) and use the automatic Port-ID selection facility by using zero (third parameter, line 32).

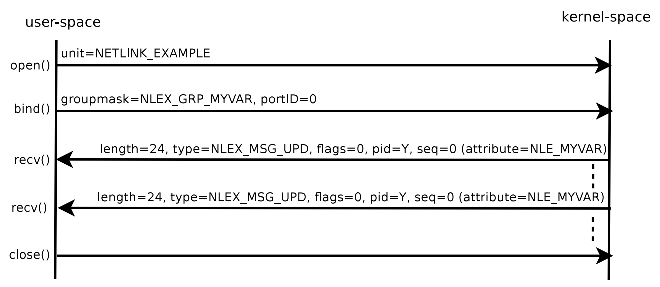
user-space
kernel-space
open()
unit=NETLINK_EXAMPLE
bind()
groupmask=NLEX_GRP_MYVAR, portID=0
recv()
length=24, type=NLEX_MSG_UPD, flags=0, pid=Y, seq=0 (attribute=NLE_MYVAR)
recv()
length=24, type=NLEX_MSG_UPD, flags=0, pid=Y, seq=0 (attribute=NLE_MYVAR)
close()

Figure 4: Subscribe to asynchronous event reports of *myvar* changes via Netlink

The Netlink message is sent to kernel-space (line 38). Since we have explicitly requested an acknowledgment, we wait for it (line 44). The Netlink message received is passed to the callback runqueue which returns -1 (line 50) and it sets errno appropriately in case of error. Then, the user-space Netlink socket is closed (line 55).

We can subscribe to changes of the variable *myvar* by means of the asynchronous notification facility that Netlink provides. In Figure. **??**, we have represented the sequence diagram of the subscription to events that report about changes in the variable *myvar*.

In Listing 8, we have implemented the event subscription. Basically, we open an user-space Netlink socket (line 30) and bind it (line 36) to the multicast group NLEX_GRP_MYVAR (second parameter, line 36). Then, we wait to receive Netlink messages that contain updates of the variable (lines 41-48).

Once we receive a Netlink message, we pass it to the callback runqueue which invokes the *data_cb()* function (line 9-21). This function parses the TLV-based payload of the Netlink message (line 15) and it initializes an array of pointers that contain the address of the attributes (variable *tb* in line 15, second parameter). Thus, we can easily access the attributes to retrieve the value of the attribute and print it (lines 17-18).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  #include <libmnl/libmnl.h>
6
7  #include "nlexample.h"
8
9  static int
10 data_cb(const struct nlmsghdr *nlh, void *data)
11 {
12   struct nlattr *tb[NLE_MAX+1];
13   struct nlattr *attr;
14
15   mnl_attr_parse(nlh, tb, NLE_MAX);
16   if (tb[NLE_MYVAR])
17     printf("myvar=%u\n",
18            mnl_attr_get_u32(tb[NLE_MYVAR]));
```

```

    return MNL_CB_OK;
}

int main()
{
    struct mnl_socket *nl;
    char buf[getpagesize()];
    struct nlmsghdr *nlh = (struct nlmsghdr *) buf;
    int ret;

    nl = mnl_socket_open(NETLINK_EXAMPLE);
    if (nl == NULL) {
        perror("mnl_socket_open");
        exit(EXIT_FAILURE);
    }

    if (mnl_socket_bind(nl,NLEX_GRP_MYVAR,0) < 0){
        perror("mnl_socket_bind");
        exit(EXIT_FAILURE);
    }

    ret = mnl_socket_recvfrom(nl,buf,sizeof(buf));
    while (ret > 0) {
        ret = mnl_cb_run(buf,ret,0,data_cb,NULL);
        if (ret <= 0)
            break;
        ret = mnl_socket_recvfrom(nl,buf,
                                  sizeof(buf));
    }
    if (ret == -1) {
        perror("error");
        exit(EXIT_FAILURE);
    }

    mnl_socket_close(nl);

    return 0;
}
```

Listing 8: *event.c* program

2 PROGRAMMING GENETLINK

This section provides an overview on the GeNetlink programming from both user and kernel-space.

2.1 Programming GeNetlink from kernel-space

We have ported the previous example Linux kernel module that uses Netlink to export the variable *myvar* to use GeNetlink. To do so, we have defined two commands in Listing. **??**:

- NLEX_CMD_UPD, that allows you to update the current value of *myvar*.

- NLEX_CMD_GET, that is used to retrieve the current value of *myvar* via unicast.

This command values are used in the command field in the GeNetlink header. With regards to the attribute definitions (lines 10-15), we use the same declarations described in our previous example in Section **??**.

```
1  #ifndef _NLEXAMPLE_H_
2  #define _NLEXAMPLE_H_
3
4  enum nlexample_msg_types {
5      NLEX_CMD_UPD = 0,
6      NLEX_CMD_GET,
7      NLEX_CMD_MAX
8  };
9
10 enum nlexample_attr {
11     NLE_UNSPEC,
12     NLE_MYVAR,
13     __NLE_MAX,
14 };
15 #define NLE_MAX (__NLE_MAX - 1)
16
17 #define NLEX_GRP_MYVAR 1
18
19 #endif
```

Listing 9: GeNetlink example header file

In Listing. **??**, we show the structures that are required to register the new GeNetlink family, they are:

1. The GeNetlink family structure (lines 11-17) that includes the type of ID number (line 12) which is automatically set by GeNetlink since GENL_ID_GENERATE is used. Then, the unique string name that is *nlex*, and the maximum number of TLV attributes that can contain the GeNetlink message (line 16). The version field (line 15) allows to declare different versions of the same family, this can be used in the future to introduce changes in the GeNetlink message format, and operations without breaking backward compatibility.

2. The family operations (lines 23-32), that include the two commands supported. This structure links the command type (lines 25 and 29) with a callback function, which is the *doit* field in the structure (lines 26 and 30). Thus, if a GeNetlink message for this family is received, it is passed to the appropriate callback.

3. The family multicast group (lines 19-21), that is only one group in this case that is identified by the string name *example*. This group allows user-space listeners to subscribe to asynchronous reports on changes in *myvar*.

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/skbuff.h>
#include <linux/genetlink.h>
#include <net/genetlink.h>

#include "genlexample.h"

static int myvar;

static struct genl_family genl_ex_family = {
  .id = GENL_ID_GENERATE,
  .name = "nlex",
  .hdrsize = 0,
  .version = 1,
  .maxattr = NLE_MAX,
};

static struct genl_multicast_group genl_ex_mc = {
  .name     = "example",
};

static struct genl_ops genl_ex_ops[] = {
  {
    .cmd = NLEX_CMD_GET,
    .doit = genl_get_myvar,
  },
  {
    .cmd = NLEX_CMD_UPD,
    .doit = genl_upd_myvar,
  },
};
```

Listing 10: GeNetlink example declarations

Once the appropriate structures have been declared, we proceed to register them into GeNetlink. In our example, the registration is done in the initialization and exit path of the Linux kernel module. The code snippet in Listing. **??** contains the registration (lines 1-27) and the unregistration (lines 29-32) routines.

```c
1  static int __init nlexample_init(void)
2  {
3    int i, ret = -EINVAL;
4
5    ret = genl_register_family(&genl_ex_family);
6    if (ret < 0)
7      goto err;
8
9    for (i = 0; i < ARRAY_SIZE(genl_ex_ops); i++) {
10     ret = genl_register_ops(&genl_ex_family,
11                             &genl_ex_ops[i]);
12     if (ret < 0)
13       goto err_unregister;
14   }
15
16    ret = genl_register_mc_group(&genl_ex_family,
17                                 &genl_ex_mc);
18    if (ret < 0)
19      goto err_unregister;
20
21    return ret;
22
23  err_unregister:
24    genl_unregister_family(&genl_ex_family);
25  err:
26    return ret;
27  }
28
29  void __exit nlexample_exit(void)
30  {
31    genl_unregister_family(&genl_ex_family);
32  }
33
34  module_init(nlexample_init);
35  module_exit(nlexample_exit);
```

Listing 11: GeNetlink family registration and unregistration routines

The initialization consists of the registration of the new GeNetlink family (lines 5-7), the registration of the operations (lines 9-14) and the registration of the multicast group (lines 16-19).

The operation to retrieve the current value of *myvar* is implemented in the Listing. **??**. This function consists of the allocation of the Netlink message (lines 7-9), the initialization of the GeNetlink header (lines 11-12) and the addition of the TLV-based payload that contains the current value of the *myvar* (line 14). Then, the GeNetlink message is delivered via unicast to the user-space process (line 16).

```
1   static int
2   genl_get_myvar(struct sk_buff *skb,
3                  struct genl_info *info)
4   {
5     struct sk_buff *msg;
6
7     msg = nlmsg_new(NLMSG_DEFAULT_SIZE,GFP_KERNEL);
8     if (msg == NULL)
9       return -ENOMEM;
10
11    genlmsg_put(msg, info->snd_pid, info->snd_seq,
12                &genl_ex_family, 0, NLEX_CMD_UPD);
13
14    NLA_PUT_U32(msg, NLE_MYVAR, myvar);
15
16    genlmsg_unicast(msg, info->snd_pid);
17
18    return 0;
19
20  nla_put_failure:
21    return -ENOBUFS;
22  }
```

Listing 12: Get *myvar* routine

The update of the value of *myvar* via the example GeNetlink family is implemented in the Listing. **??**.

```
 1  static int
 2  genl_upd_myvar(struct sk_buff *skb,
 3                 struct genl_info *info)
 4  {
 5    struct sk_buff *msg;
 6
 7    if (!info->attrs[NLE_MYVAR])
 8      return -EINVAL;
 9
10    myvar = nla_get_u32(info->attrs[NLE_MYVAR]);
11
12    msg = nlmsg_new(NLMSG_DEFAULT_SIZE, GFP_KERNEL);
13    if (msg == NULL)
14      return -ENOMEM;
15
16    genlmsg_put(msg, info->snd_pid, info->snd_seq,
17                &genl_ex_family, 0, NLEX_CMD_UPD);
18
19    NLA_PUT_U32(msg, NLE_MYVAR, myvar);
20
21    genlmsg_multicast(msg, 0, genl_ex_mc.id,
22                      GFP_KERNEL);
23    return 0;
24
25  nla_put_failure:
26    return -ENOBUFS;
27  }
```

Listing 13: Update *myvar* routine

Initially, this routine verifies that the message coming from user-space contains the NLE_MYVAR attribute with the new *myvar* value that user-space wants to set (lines 7-8). If the attribute is missing, the routine returns an error to report that the message is invalid since the NLE_MYVAR attribute is mandatory. If the message contains the attribute, the routine updates the value of the variable *myvar* (line 10). Since user-space process can subscribe to the *example* multicast to track changes in the *myvar* variable, this routine has to report the change in the variable. The reporting consists of allocating memory for the Netlink message (lines 12-14), filling the GeNetlink header (lines 16-17) with the Port-ID of the process that has changed the variable (second parameter), the same sequence number used in the message received from user-space (third parameter), the GeNetlink family (line 17, fourth parameter) and the Netlink message flags (fifth parameter) which are none in this case, and the command type of the message (sixth parameter). To conclude the message building, the routine adds the new value of *myvar* in TLV format as payload of the GeNetlink message. Then, the message is delivered to the user-space listeners that are subscribed to the *example* group (lines 21-22).

2.2 Programming GeNetlink from user-space

Since the family and multicast IDs are assigned in run-time, we initially have to look up for the IDs to send requests and to subscribe to GeNetlink multicast groups from user-space. For that task, we use a program that takes the GeNetlink family name as first parameter, it sends a request to the GeNetlink control family *nlctrl* which is the only family with a fixed family ID, and it displays the family ID and the list of available multicast groups and their corresponding IDs. The body of the main() function of this resolver has been represented in Listing. 14.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <libmnl/libmnl.h>
#include <linux/genetlink.h>

int main(int argc, char *argv[])
{
  struct mnl_socket *nl;
  char buf[getpagesize()];
  struct nlmsghdr *nlh;
  struct genlmsghdr *genl;
  int ret, numbytes, hdrsiz;
  unsigned int seq;

  if (argc != 2) {
    printf("%s [family_name]\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  nlh=mnl_nlmsg_put_header(buf);
  nlh->nlmsg_type =GENL_ID_CTRL;
  nlh->nlmsg_flags=NLM_F_REQUEST | NLM_F_ACK;
  nlh->nlmsg_seq=seq=time(NULL);

  hdrsiz=sizeof(struct genlmsghdr));
  genl=mnl_nlmsg_put_extra_header(nlh, hdrsiz);
  genl->cmd=CTRL_CMD_GETFAMILY;
  genl->version=1;

  mnl_attr_put_str_null(nlh,
                        CTRL_ATTR_FAMILY_NAME,
                        argv[1]);

  numbytes=mnl_nlmsg_get_len(nlh));

  nl=mnl_socket_open(NETLINK_GENERIC);
  if (nl == NULL) {
    perror("mnl_socket_open");
    exit(EXIT_FAILURE);
  }

  if (mnl_socket_bind(nl, 0, 0) < 0) {
    perror("mnl_socket_bind");
```

```
46        exit(EXIT_FAILURE);
47     }
48
49     if (mnl_socket_sendto(nl, nlh, numbytes) < 0){
50        perror("mnl_socket_send");
51        exit(EXIT_FAILURE);
52     }
53
54     ret=mnl_socket_recvfrom(nl,buf,sizeof(buf));
55     while (ret > 0) {
56        ret=mnl_cb_run(buf,ret,seq,data_cb, NULL);
57        if (ret <= 0)
58           break;
59        ret=mnl_socket_recvfrom(nl,buf,sizeof(buf));
60     }
61     if (ret == -1) {
62        perror("error");
63        exit(EXIT_FAILURE);
64     }
65     mnl_socket_close(nl);
66
67     return 0;
68  }
```

Listing 14: GeNetlink family and multicast resolver

The resolver initially checks for the required family name as argument (lines 17-20). Then, it starts by building the Netlink header (lines 22-25) in which the message type is GENL_ID_CTRL that is the fixed ID of the GeNetlink control family (GENL_ID_CTRL equals 16 which is the first available message type for Netlink headers, lower values are reserved for Netlink control messages). It follows the GeNetlink header building (lines 27-30) in which the command used to retrieve the GeNetlink family information is CTRL_CMD_GETFAMILY (line 29). The payload of the message is composed of one Netlink attribute that contains the GeNetlink family name that we want to retrieve information.

Then, we open the Netlink socket (lines 38-42) using the NETLINK_GENERIC bus which corresponds GeNetlink. We bind it to no multicast groups (lines 44-47) and we finally send the message (lines 49-52). Then, we wait for the reply (line 54 and 59) and we handle the data received by means of the callback run-queue (line 56) which calls the function *data_cb()* to perform the interpretation of the reply. If no GeNetlink family was found, we receive a Netlink message that contains the ENOENT error (No such entry exists).

We have represented the function *data_cb()* in Listing. 15.

```
1   static int
2   data_cb(const struct nlmsghdr *nlh, void *data)
3   {
4      struct nlattr *tb[CTRL_ATTR_MAX+1];
5      struct genlmsghdr *genl =
6            mnl_nlmsg_get_data(nlh);
7
8      mnl_attr_parse_at_offset(nlh, sizeof(*genl),
9                               tb, CTRL_ATTR_MAX);
10     if (tb[CTRL_ATTR_FAMILY_ID]) {
```

```
11        printf("family-id: %d\n",
12          mnl_attr_get_u16(tb[CTRL_ATTR_FAMILY_ID]));
13      }
14      if (tb[CTRL_ATTR_MCAST_GROUPS]) {
15        printf("multicast -> id\n");
16        parse_mc_grps(tb[CTRL_ATTR_MCAST_GROUPS]);
17      }
18
19      return MNL_CB_OK;
20    }
```

Listing 15: Digesting GeNetlink control messages

This function parses the payload of the message (lines 8-9) and it sets the array of pointers to access the attributes. Then, we check if the family ID attribute is available in the message (line 10). If so, the routine displays the family ID (lines 11-12) which is required to fill the Netlink type to send a message for that family. Moreover, the routine parses the set of multicast groups that are available in the GeNetlink family (line 16).

The set of multicast groups available in a given family is stored in a three-level nesting. The first level uses the attribute type CTRL_ATTR_MCAST_GROUPS, then the second level contains a set of attribute types starting by 1 to n, where n is the number of multicast groups available. Finally, the third level contains the attributes CTRL_ATTR_MCAST_GRP_NAME, which contains the multicast group name, and CTRL_ATTR_MCAST_GRP_ID, that is the ID of that given multicast group. We have represented the multicast group parsing in Listing. 16.

```
static void
parse_one_mc_group(struct nlattr *pos)
{
 struct nlattr *tb[CTRL_ATTR_MCAST_GRP_MAX+1];

 mnl_attr_parse_nested(pos, tb,
                       CTRL_ATTR_MCAST_GRP_MAX);
 if (tb[CTRL_ATTR_MCAST_GRP_NAME] &&
     tb[CTRL_ATTR_MCAST_GRP_ID]) {
  printf("\"%s\" -> %d\n",
   mnl_attr_get(tb[CTRL_ATTR_MCAST_GRP_NAME]),
   mnl_attr_get_u32(tb[CTRL_ATTR_MCAST_GRP_ID]));
 }
}

static void
parse_mc_grps(struct nlattr *nested)
{
 struct nlattr *pos;
 int len;
 const char *mcast_grp_name;
 int mcast_grp_id;

 mnl_attr_for_each_nested(pos, nested, len)
  parse_one_mc_group(pos);
}
```

Listing 16: Parsing GeNetlink multicast groups

Basically, the function *parse_mc_groups()* (lines 16-26) iterates over the second level of nested attributes (lines 24-25) and parses it by means of *parse_one_mc_groups()* (line 25). This function parses the attributes (lines 6-7) and it displays the multicast group name and the ID (lines 10-12).

Now that we have a program to obtain the GeNetlink family ID and the multicast group ID, we can implement the user-space support for our example GeNetlink Linux kernel module that we have previously represented in Listings. 9, 10, 11, 12 and 13. We have implemented the GeNetlink version of two programs that we have previously discussed in Section **??**, they are:

- *change.c* that can be used to update the *myvar* variable. This program takes as first argument the GeNetlink family ID that we have obtained from the resolver.
- *event.c* that allows to subscribe to events of our example GeNetlink family.

In Listing. 17, we show the implementation of the *change.c* program. To update the *myvar* variable, we initially build the Netlink header (lines 24-27), whose message type has been obtained by means of the *resolve* program and it has been passed as argument (line 25). It follows the GeNetlink header (lines 29-31) and the payload that contains the attribute NLE_MYVAR that encapsulates the new value of myvar that has been passed as second argument (line 33). Then, we open the Netlink socket (line 37) and bind it to no multicast groups (line 42) and send it (line 48). We wait for the reply from kernel-space (line 54) and pass the data received to the runqueue callback (line 60) to handle the acknowledgment received in return to inform about the operation success or failure. The routines ends by closing the Netlink socket (line 65).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  #include <libmnl/libmnl.h>
6  #include <linux/genetlink.h>
7
8  #include "genlexample.h"
9
10 int main(int argc, char *argv[])
11 {
12   char buf[getpagesize()];
13   struct nlmsghdr *nlh;
14   struct genlmsghdr *genl;
15   struct mnl_socket *nl;
16   int ret, hdrsiz, numbytes;
17   unsigned int seq, oper;
18
19   if (argc != 3) {
20     printf("%s␣[Family␣ID]␣[myvar]\n",argv[0]);
```

```
    exit(EXIT_FAILURE);
  }

  nlh = mnl_nlmsg_put_header(buf);
  nlh->nlmsg_type = atoi(argv[1]);
  nlh->nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK;
  nlh->nlmsg_seq = seq = time(NULL);

  hdrsiz = sizeof(struct genlmsghdr);
  genl = mnl_nlmsg_put_extra_header(nlh, hdrsiz);
  genl->cmd = NLEX_CMD_UPD;

  mnl_attr_put_u32(nlh,NLE_MYVAR, atoi(argv[2]));

  numbytes = mnl_nlmsg_get_len(nlh);

  nl = mnl_socket_open(NETLINK_GENERIC);
  if (nl == NULL) {
    perror("mnl_socket_open");
    exit(EXIT_FAILURE);
  }
  ret = mnl_socket_bind(nl, 0, 0);
  if (ret == -1) {
    perror("mnl_socket_bind");
    exit(EXIT_FAILURE);
  }

  ret = mnl_socket_sendto(nl,nlh,numbytes);
  if (ret == -1) {
    perror("mnl_socket_send");
    exit(EXIT_FAILURE);
  }

  ret = mnl_socket_recvfrom(nl,buf,sizeof(buf));
  while (ret == -1) {
    perror("recvfrom");
    exit(EXIT_FAILURE);
  }

  ret = mnl_cb_run(buf, ret, seq, NULL, NULL);
  if (ret == -1) {
    perror("callback");
    exit(EXIT_FAILURE);
  }
  mnl_socket_close(nl);
}
```

Listing 17: change.c for GeNetlink

We have represented the event subscription program in Listing. 18. This routine initially opens a Netlink socket (line 38) and it binds to no multicast groups (line 44). Since the common binding operation only allows to subscribe by means of a group mask (Thus, this allows to subscribe up to the limited number 32 groups.), we use the socket option NETLINK_ADD_MEMBERSHIP (lines 50-53) which allows to subscribe to a group by an integer. Thus, we can subscribe to any of the existing 2^{32} multicast groups. This multicast ID is passed as first argument to the program and it has been previously obtained by means of the *resolve* program.

Once we are subscribed to the event reporting, we wait for event messages (lines 55 and 60) and we handle the messages received by means of the callback runqueue (line 57) that invokes the *data_cb()* function. This function parses the payload of the GeNetlink messages received (lines 17-18) and it displays the new value (lines 20-21).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  #include <libmnl/libmnl.h>
6  #include <linux/genetlink.h>
7
8  #include "genlexample.h"
9
10 static int
11 data_cb(const struct nlmsghdr *nlh, void *data)
12 {
13   struct nlattr *tb[NLE_MAX+1];
14   struct nlattr *attr;
15   int hdrsiz = sizeof(struct genlmsghdr);
16
17   mnl_attr_parse_at_offset(nlh, hdrsiz,
18                            tb, NLE_MAX);
19   if (tb[NLE_MYVAR])
20     printf("myvar=%u\n",
21            mnl_attr_get_u32(tb[NLE_MYVAR]));
22
23   return MNL_CB_OK;
24 }
25
26 int main(int argc, char *argv[])
27 {
28   struct mnl_socket *nl;
29   char buf[getpagesize()];
30   struct nlmsghdr *nlh = (struct nlmsghdr *) buf;
31   int ret, grp;
32
33   if (argc != 2) {
34     printf("%s␣[GeNL␣multicast␣ID]\n",argv[0]);
35     exit(EXIT_FAILURE);
36   }
37
38   nl = mnl_socket_open(NETLINK_GENERIC);
39   if (nl == NULL) {
40     perror("mnl_socket_open");
41     exit(EXIT_FAILURE);
42   }
43
44   ret = mnl_socket_bind(nl, 0, 0);
45   if (ret == -1) {
46     perror("mnl_socket_bind");
47     exit(EXIT_FAILURE);
48   }
```

```

    grp = atoi(argv[1]);
    mnl_socket_setsockopt(nl,
                          NETLINK_ADD_MEMBERSHIP,
                          &grp, sizeof(grp));

    ret = mnl_socket_recvfrom(nl,buf,sizeof(buf));
    while (ret > 0) {
      ret = mnl_cb_run(buf,ret,0,data_cb,NULL);
      if (ret <= 0)
        break;
      ret=mnl_socket_recvfrom(nl,buf,sizeof(buf));
    }
    if (ret == -1) {
      perror("error");
      exit(EXIT_FAILURE);
    }

    mnl_socket_close(nl);

    return 0;
}
```

Listing 18: event.c for GeNetlink