IPv6 & IoT:
Implementation of a Kernel Space Module and User Space Utility for retrieving L2 Network Interface Information

Andrii Konotop

April 2025

Contents

1 Introduction

The motivation for this project is to establish a flexible and robust inter-process communication (IPC) between user space and kernel space using Generic Netlink. The subject of the communication is the retrieval of Layer 2 (L2) network interface information. This information is obtained in kernel space using an appropriate kernel API. The project includes a user-space utility that queries and presents the retrieved data in two formats. The first format displays a comprehensive list of L2 network interfaces, including basic details such as interface name, Maximum Transmission Unit (MTU), MAC address, broadcast address, and transmit (TX) queue length. The second format enables users to query interface-specific statistics, including the number of good bytes received and transmitted, the total number of received errors, and the number of transmit problems.

2 Background

Netlink, introduced in Linux 2.2 (1999), originally provided a fixed set of protocol families for different kernel subsystems (e.g., routing, firewall, audit). As Netlink's popularity grew, the limited range of family numbers became a concern. To address this, the Linux kernel introduced the Generic Netlink family in version 2.6.15 (released in 2006), allowing dynamic allocation of logical families under a single protocol. When user space requests a family ID, the kernel's Generic Netlink controller matches the requested name to a registered family and replies with the dynamically assigned numeric family ID. This ID is then used for all further communication with that Generic Netlink family. Moreover, Generic Netlink enforces TLV attribute payloads, and validates messages using explicit policies, making the API more straightforward and secure.

3 Implementation Details

3.1 Kernel Module

The kernel module implements a Generic Netlink family to facilitate communication between kernel space and user space for retrieving Layer 2 (L2) network interface information. Before registering the family, its name, attributes, nested sub-attributes, and commands must be defined. These definitions are shared between the kernel module and the user-space utility to ensure consistent communication.

The Type-Length-Value (TLV) attribute format supports nested sub-attributes, allowing related attributes to be grouped under a top-level attribute. This structure simplifies attribute type validation through the `nla_policy` structure, which defines policies for both parent and nested sub-attributes. In this implementation, a single top-level attribute, `NL_UTIL_A_NETDEV`, encapsulates several nested sub-attributes, including `NL_UTIL_NESTED_A_IFINDEX`, `NL_UTIL_NESTED_A_IFNAME`, `NL_UTIL_NESTED_A_IFMTU`, and others, corresponding to L2 interface information being transmitted.

```c
enum NL_UTIL_ATTRS {
    NL_UTIL_A_UNSPEC,      /* Unspecified attribute (placeholder) */
    NL_UTIL_A_NETDEV,      /* Network interface-related attribute */
    __NL_UTIL_A_MAX        /* Internal max value for validation */
};
enum NL_UTIL_NESTED_ATTRS {
    NL_UTIL_NESTED_A_UNSPEC,     /* Unspecified nested sub-attribute (placeholder) */
    NL_UTIL_NESTED_A_IFINDEX,    /* Interface index (e.g., 2 for eth0) */
    NL_UTIL_NESTED_A_IFNAME,     /* Interface name (e.g., "eth0") */
    NL_UTIL_NESTED_A_IFMTU,      /* Interface MTU (maximum transmission unit) */
    __NL_UTIL_NESTED_A_MAX       /* Internal max value for validation */
};
```

Listing 1: Top-level and nested attribute definitions

The Generic Netlink family relies on commands and their corresponding callback functions, registered via the `genl_ops` structure during initialization. Two commands are defined: `NL_UTIL_C_L2_LIST` and `NL_UTIL_C_L2_IID`.

```c
enum NL_UTIL_CMDS {
    NL_UTIL_C_UNSPEC,               /* Unspecified command (placeholder) */
    NL_UTIL_C_L2_LIST,              /* List all Layer 2 network interfaces */
    NL_UTIL_C_L2_IID,               /* Query a Layer 2 interface by index */
    __NL_UTIL_C_MAX                 /* Internal max value for validation */
};
```

Listing 2: Commands

The `NL_UTIL_C_L2_LIST` command is associated with the `l2_list_doit` callback function. And the `NL_UTIL_C_L2_IID` command is associated with the `l2_iid_doit` callback function.

```c
int l2_list_doit(struct sk_buff *sender_buff, struct genl_info *info);
int l2_iid_doit(struct sk_buff *sender_buff, struct genl_info *info);
```

Listing 3: Callback function signatures

```c
struct genl_ops nl_util_gnl_ops[NL_UTIL_C_MAX] = {
    {
        .cmd = NL_UTIL_C_L2_LIST,       /* Command to list L2 interfaces */
        .doit = l2_list_doit,           /* Callback handler for listing interfaces */
    },
    {
        .cmd = NL_UTIL_C_L2_IID,        /* Command to get interface by index */
        .doit = l2_iid_doit,            /* Callback handler for interface lookup */
    },
};
```

Listing 4: Operations

The `NL_UTIL_C_L2_IID` command expects an additional interface index as a part of payload. The `l2_iid_doit` callback parses this index from the nested sub-attribute structure (`NL_UTIL_A_NETDEV` → `NL_UTIL_NESTED_A_IFINDEX` → `(u32) ifindex`). A nested validation policy ensures the interface index is correctly typed as `u32`.

```c
struct nla_policy nl_util_nested_policy[NL_UTIL_NESTED_A_MAX + 1] = {
    [NL_UTIL_NESTED_A_IFINDEX] = { .type = NLA_U32 }, /* Interface index */
};
```

Listing 5: Nested validation policy

The type of `NL_UTIL_A_NETDEV` is specified to `NLA_NESTED` using `NLA_POLICY_NESTED` macro, which ensures that the attribute is structured as a nested container. This allows applying the `nl_util_nested_policy` sub-policy to the nested content of the top-level attribute.

```c
struct nla_policy nl_util_genl_policy[NL_UTIL_A_MAX + 1] = {
    [NL_UTIL_A_NETDEV] = NLA_POLICY_NESTED(
        nl_util_nested_policy),
};
```

Listing 6: Top-level validation policy

Finally, after defining attributes, their policies, and commands, all of that is encapsulated in `genl_family` structure.

```
static struct genl_family nl_util_genl_family = {
    .id = 0, /* Auto-assigned ID */
    .hdrsize = 0, /* No custom header */
    .name = FAMILY_NAME, /* Family name for user-space identification */
    .version = 1, /* Family version number */
    .ops = nl_util_gnl_ops, /* Array of operations */
    .n_ops = NL_UTIL_C_MAX, /* Number of operations */
    .policy = nl_util_genl_policy, /* Attribute validation policy */
    .maxattr = NL_UTIL_A_MAX, /* Maximum number of attributes */
    .module = THIS_MODULE, /* Owning kernel module */
};
```

Listing 7: Generic Netlink family definition

The Generic Netlink family, defined by the `nl_util_genl_family` structure, is registered during the kernel module's initialization. The registration is performed in the module initialization function, by calling `genl_register_family`.

```
static int __init netlink_mod_init(void)
{
    return genl_register_family(&nl_util_genl_family);
}
```

Listing 8: Generic Netlink Family Registration

To ensure proper resource management, the module cleanup function, `netlink_mod_exit`, deregisters the Generic Netlink family using `genl_unregister_family` function. This step releases allocated resources and prevents memory leaks when the module is unloaded.

```
static void __exit netlink_mod_exit(void)
{
    genl_unregister_family(&nl_util_genl_family);
}
```

Listing 9: Generic Netlink Family Deregistration

The primary functionality of this kernel module is implemented within the two callback functions. The `l2_list_doit` does not expect a payload; once a valid request with the corresponding `NL_UTIL_C_L2_LIST` command is sent, the function is invoked. It begins by allocating an `sk_buff` reply buffer with a size of `NLMSG_GOODSIZE` and with a flag `GFP_KERNEL`, which is suitable for kernel-internal allocations.

```
struct sk_buff *reply_buff = genlmsg_new(NLMSG_GOODSIZE, GFP_KERNEL);
```

Listing 10: Reply message buffer allocation

After allocating memory for the response, Netlink and Generic Netlink headers are constructed and added to the response buffer by calling `genlmsg_put`.

```
void *msg_hdr = genlmsg_put(reply_buff, info->snd_portid, info->snd_seq + 1,
                  &nl_util_genl_family, 0, NL_UTIL_C_L2_LIST);
```

Listing 11: Message headers initialization

The next step in the callback function is the RCU-protected iteration over all network interfaces in the `init_net` namespace, filtered by the `ARPHRD_ETHER` type, which corresponds to Ethernet (Layer 2) network interfaces. On each iteration, a new level of nested sub-attributes is started, filled with the desired network interface information, and ended. In case of an error, nested sub-attribute creation is stopped, the `reply_buffer` is freed, and the `-ENOMEN` code is returned.

```
rcu_read_lock();
for_each_netdev_rcu(&init_net , netdev)
    if (netdev ->type == ARPHRD_ETHER) {
        struct nlattr *start = nla_nest_start_noflag(reply_buff , NL_UTIL_A_NETDEV);
        nla_nest_end(reply_buff , start);
    }
rcu_read_unlock();
```

Listing 12: Iteration over network interfaces

Between calling `nla_nest_start` and `nla_nest_end`, nested sub-attributes are added to the `reply buffer` using the `put_nested_basic` helper function. Which internally uses `nla_put` or type-specific wrapper functions like `nla_put_u32` or `nla_put_string` for known types (e.g., `u32` or a `string`).

```
int put_nested_basic(...) {
    return nla_put_u32(...) || nla_put_string(...) || ... ? -1 : 0;
}
```

Listing 13: Nested sub-attributes placement

Finally, the constructed Generic Netlink message is finalized with `genlmsg_end` and sent back to userspace using `genlmsg_unicast`.

```
genlmsg_end(reply_buff , msg_hdr);
genlmsg_unicast(genl_info_net(info), reply_buff , info ->snd_portid);
```

Listing 14: Finalization and sending

The `l2_iid_doit` callback function operates similarly to the `l2_list_doit` callback function in terms of reply buffer allocation, nested sub-attributes placement, and message sending. The key difference is that it expects the `NL_UTIL_A_NETDEV` top-level attribute containing the `NL_UTIL_NESTED_A_IFINDEX` nested sub-attribute in the request message. The expected payload from userspace is in the following format: `struct *nlmsghdr` → `struct *genlmsghdr` → `struct *nlattr` → `NL_UTIL_A_NETDEV` + `(struct *nlattr` → `NL_UTIL_NESTED_A_IFINDEX + (u32) ifindex)`.

Firstly, the top-level `NL_UTIL_A_NETDEV` is accessed from the request.

```
struct nlattr *na, *na_nested;
na = info ->attrs[NL_UTIL_A_NETDEV];
```

Listing 15: Top-level attribute access

Secondly, the interface index is retrieved from the nested sub-attribute.

```
na_nested = nla_data(na);
int ifindex = nla_get_u32(na_nested);
```

Listing 16: Nested sub-attribute access

Afterwards, the network interface is retrieved by the parsed interface index in an RCU-protected manner.

The RCU read lock is held to ensure the interface information remains valid during the operation and is released after message finalization or in case of an error.

```c
rcu_read_lock();
netdev = dev_get_by_index_rcu(&init_net, ifindex);
if (!netdev) {
        rcu_read_unlock();
        nlmsg_free(reply_buff);
        return -ENODEV;
    }
```

Listing 17: Network interface information retrieval

l2_iid_doit builds upon the existing put_nested_basic helper function by adding information from the rtnl_link_stats64 network interface statistics container, retrieved using dev_get_stats.

```c
int put_nested_detailed(struct sk_buff *b, struct net_device *dev)
{
    struct rtnl_link_stats64 stats;
    if (put_nested_basic(b, dev)) return -1;
    dev_get_stats(dev, &stats);
    return nla_put(b, NL_UTIL_NESTED_A_STATS, sizeof(stats), &stats) ? -1 : 0;
}
```

Listing 18: Network interface statistics retrieval

3.2 User-Space Utility

The user-space program begins by allocating memory for context variables, including outgoing and incoming messages, the socket file descriptor, the dynamically retrieved family id, and the Netlink address. Initially, the socket file descriptor and the family id are set to -1, as the socket is not yet bound and the family id has not yet been obtained from the kernel.

```c
struct nl_context {
    int fd;
    int fam_id;
    struct sockaddr_nl nl_address;
    struct nl_msg *req;
    struct nl_msg *res;
};
```

Listing 19: Netlink context

```c
struct nl_msg {
    struct nlmsghdr n;
    struct genlmsghdr g;
    char buf[NL_MSG_BUF_SIZE];
};
```

Listing 20: Netlink message structure

A socket is created with the address family set to AF_NETLINK, the socket type set to SOCK_RAW, and the protocol number set to NETLINK_GENERIC.

```c
ctx->fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_GENERIC);
```

Listing 21: Socket creation

Netlink uses sockaddr_nl with nl_pid as the address identifier. If the file descriptor is non-negative, the socket is bound to the Netlink address with a PID matching the calling process's PID using the bind function.

```
set_nl_addr(ctx, getpid());
bind(ctx->fd, (struct sockaddr *)&ctx->nl_address, sizeof(struct sockaddr_nl))
```

Listing 22: Socket binding

The next step involves resolving the family id using the FAMILY_NAME, which is defined as an arbitrary string in the nl_common.h header file. The name of the family is defined as 'nl_util'. Family ID resolution is achieved by sending a request to the Generic Netlink controller by setting the nlmsghdr->nlmsg_type field to GENL_ID_CTRL. Additionally, CTRL_CMD_GETFAMILY is used as the command in the Generic Netlink header.

```
memset(ctx->req, 0, sizeof(struct nl_msg));
ctx->req->n.nlmsg_len = NLMSG_LENGTH(GENL_HDRLEN);
ctx->req->n.nlmsg_type = GENL_ID_CTRL;
ctx->req->n.nlmsg_flags = NLM_F_REQUEST;
ctx->req->n.nlmsg_seq = 0;
ctx->req->n.nlmsg_pid = getpid();
ctx->req->g.cmd = CTRL_CMD_GETFAMILY;
ctx->req->g.version = 0x1;
```

Listing 23: Request message initialization

The payload of the request includes the Netlink attribute CTRL_ATTR_FAMILY_NAME, which contains the name of the family terminated by a null byte.

```
struct nlattr *na;
na = (struct nlattr *)GENLMSG_DATA(req);
na->nla_len = strlen(FAMILY_NAME) + 1 + NLA_HDRLEN;
na->nla_type = CTRL_ATTR_FAMILY_NAME;
strcpy(NLA_DATA(na), FAMILY_NAME);
ctx->req->n.nlmsg_len += NLMSG_ALIGN(na->nla_len);
```

Listing 24: Request message payload

The prepared request is then sent to the kernel, which has PID 0.

```
set_nl_addr(ctx, 0);
rxtx_len = sendto(ctx->fd, (char *)req, req->n.nlmsg_len, 0,
                  (struct sockaddr *)&ctx->nl_address,
                  sizeof(struct sockaddr_nl));
```

Listing 25: Sending the request

In case the request was sent successfully, the recv is called.

```
memset(ctx->res, 0, sizeof(struct nl_msg));
rxtx_len = recv(ctx->fd, (char *)res, sizeof(*res), 0);
```

Listing 26: Receiving a response

After successfully receiving a response from the kernel, the family ID is extracted from the response. It is assumed to be the second top-level attribute, following the CTRL_ATTR_FAMILY_NAME attribute.

```
na = NLA_NEXT((struct nlattr *)GENLMSG_DATA(res);
if (na->nla_type == CTRL_ATTR_FAMILY_ID) {
    ctx->fam_id = *(__u16 *)NLA_DATA(na);
}
```

Listing 27: Family ID parsing

Depending on whether the user has passed an additional `ifindex` parameter, the appropriate command handler is invoked. There are two handlers defined as follows:

```
int handle_l2_list(struct nl_context *ctx);
int handle_l2_by_ifindex(struct nl_context *ctx, const int ifindex);
```

Listing 28: Core functionality functions

The `handle_l2_list` handler does not send any payload. The `nlmsghdr’s nlmsg_type` field of the request message is set to the obtained family id, the `genlmsghdr’s cmd` field is set to `NL_UTIL_C_L2_LIST`. The request is prepared and sent to the kernel.

Upon receiving a successful response, the list of L2 network interfaces are parsed and printed to the `stdout`. The parsing process, which is common for both handlers, is described later.

In contrast, the `handle_l2_by_ifindex` handler is required to send the interface index as the part of the nested sub-attribute `NL_UTIL_NESTED_A_IFINDEX`. The request headers preparation is the same as for the previous handler, but in this case involves setting the `genlmsghdr’s cmd` field to `NL_UTIL_C_L2_IID`. Specifically, the top-level attribute `NL_UTIL_A_NETDEV` must be marked with `NLA_F_NESTED` to indicate that it contains nested sub-attributes; otherwise, the kernel will reject the request.

```
na = (struct nlattr *)GENLMSG_DATA(ctx->req);
na->nla_type = NLA_F_NESTED | NL_UTIL_A_NETDEV;
na->nla_len = NLA_HDRLEN;
na_nested = (struct nlattr *)NLA_DATA(na);
na_nested->nla_type = NL_UTIL_NESTED_A_IFINDEX;
na_nested->nla_len = NLA_HDRLEN + sizeof(int);
*(int *)NLA_DATA(na_nested) = ifindex;
na->nla_len += NLA_ALIGN(na_nested->nla_len);
ctx->req->n.nlmsg_len += NLMSG_ALIGN(na->nla_len);
```

Listing 29: Request payload with the interface index

Upon receiving a successfull response, the details of a specific L2 network interface are obtained, including `rtnl_link_stats64` statistics.

The set of nested sub-attributes encapsulated by the `NL_UTIL_A_NETDEV` top-level attributes, which are received from the kernel, is parsed into the corresponding dynamic array of custom-defined `netdev` structures.

```
 struct netdev {
    unsigned int ifindex;
    char ifname[IFNAMSIZ];
    unsigned int flags;
    unsigned int mtu;
    unsigned int operstate;
    unsigned int qlen;
    uint8_t ifmac[ETH_ALEN];
    uint8_t ifbrd[MAX_ADDR_LEN];
    struct rtnl_link_stats64 stats;
    unsigned int initialized_fields;
};
```

Listing 30: User space netdev structure

The function responsible for doing that is:

```
int parse_into_netdev(struct netdev *dev, struct nlattr *nl_na, size_t rem);
```

Listing 31: Attribute parsing function signature

This function takes as arguments an address of the dynamically allocated array, address of the beginning of the message payload and the size of the payload. The maximum size of the array is equal to `MAX_NETDEV_COUNT` multiplied by the padded size of the `netdev` structure.

The function iterates over each top-level attribute and its nested sub-attributes. The top-level attribute is matched by its type (i.e., `NL_UTIL_A_NETDEV`) in an if statement. The nested sub-attributes are matched in the switch statement, that lists all possible nested sub-attributes.

```
int parse_into_netdev(struct netdev *dev, struct nlattr *nl_na, size_t rem)
{
    int dev_count = 0;
    while (rem >= sizeof(*nl_na) && dev_count < MAX_NETDEV_COUNT) {
        if (nl_na->nla_type == NL_UTIL_A_NETDEV) {
            dev_count++;
            struct nlattr *pos      = NLA_DATA(nl_na);
            int nest_rem            = NLMSG_ALIGN(nl_na->nla_len) - NLA_HDRLEN;
            while (nest_rem >= sizeof(*pos)) {
                void *data = NLA_DATA(pos);
                switch (pos->nla_type) {
                    /* Parse nested sub-attributes... */
                }
                nest_rem -= NLA_ALIGN(pos->nla_len);
                pos       = NLA_NEXT(pos);
            }
            dev++;
        }
        rem   -= NLA_ALIGN(nl_na->nla_len);
        nl_na  = NLA_NEXT(nl_na);
    }
    return dev_count;
}
```

Listing 32: Simplified `parse_into_netdev` implementation

As the `l2_iid_doit` callback function is designed to send extra fields for the (TX) queue length and statistics, the issue of the uninitialized fields `qlen` and `rtnl_link_stats64` is resolved by tracking their bit positions using this defenitions.

```
#define NETDEV_QLEN_SET (1 << 0)
#define NETDEV_STATS_SET (1 << 1)
```

Listing 33: Bit positions

Therefore, those special cases are handled in the switch statement:

```
case NL_UTIL_NESTED_A_QLEN:
    dev->qlen = *(uint32_t *)data;
    dev->initialized_fields |= NETDEV_QLEN_SET;
    break;
case NL_UTIL_NESTED_A_STATS:
    memcpy(&dev->stats, data, sizeof(struct rtnl_link_stats64));
    dev->initialized_fields |= NETDEV_STATS_SET;
    break;
```

Listing 34: Special cases in the switch statement

After obtaining the number of parsed interfaces, the initialized array of L2 network interfaces is iterated over. Information corresponding to each network interface in the array is printed, replicating the output format of the `ip link show` command.

```
for (int i = 0; i < n; i++) {
    struct netdev *d = &netdev[i];
    printf("\n    link/ether %02x:%02x:%02x:%02x:%02x:%02x brd %02x:%02x:%02x:%02x:%02x
    :%02x\n",
           d->ifmac[0], d->ifmac[1], d->ifmac[2], d->ifmac[3],
           d->ifmac[4], d->ifmac[5], d->ifbrd[0], d->ifbrd[1],
           d->ifbrd[2], d->ifbrd[3], d->ifbrd[4], d->ifbrd[5]);
}
```

Listing 35: Example of printing a MAC address

4 Usage Instructions

The user-space utility and the kernel module can be compiled by running `make` in the terminal. The Makefile inserts the kernel module if it isn't already loaded and removes it if it is. Additionally, it clears the kernel's ring buffer that stores previous kernel logs.

```
make
```

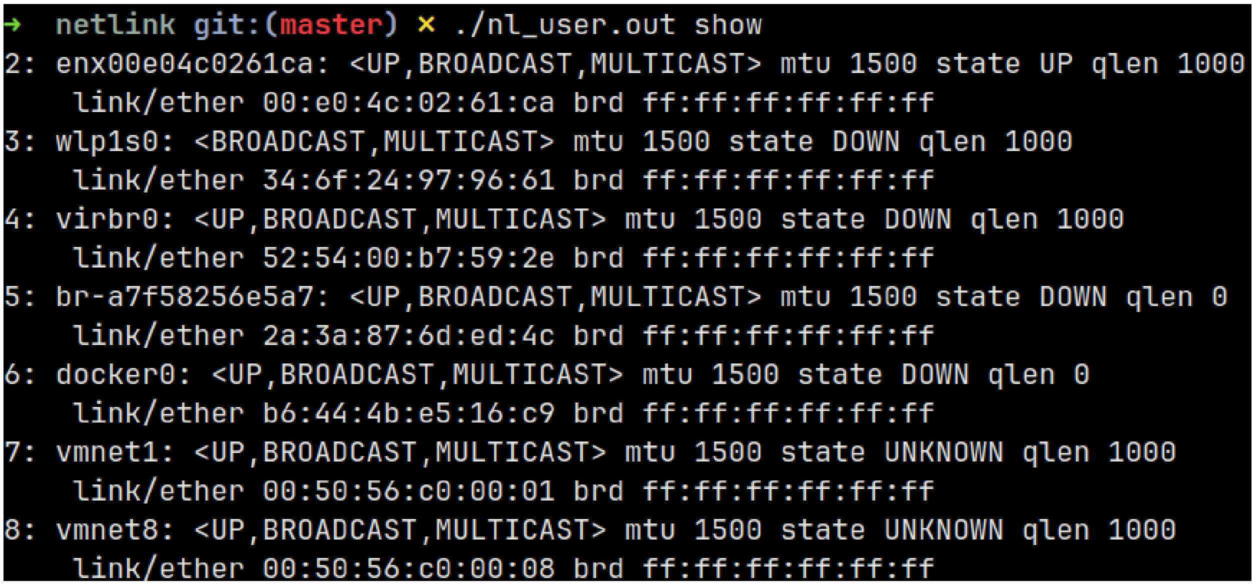
Listing 36: Compiling the kernel module and the user-space utility

To display a list of available L2 network interfaces, the following command is used:

```
./nl_user.out show
```

Listing 37: Listing L2 network interfaces

Example output from this command:



```
→  netlink git:(master) ✗ ./nl_user.out show
2: enx00e04c0261ca: <UP,BROADCAST,MULTICAST> mtu 1500 state UP qlen 1000
    link/ether 00:e0:4c:02:61:ca brd ff:ff:ff:ff:ff:ff
3: wlp1s0: <BROADCAST,MULTICAST> mtu 1500 state DOWN qlen 1000
    link/ether 34:6f:24:97:96:61 brd ff:ff:ff:ff:ff:ff
4: virbr0: <UP,BROADCAST,MULTICAST> mtu 1500 state DOWN qlen 1000
    link/ether 52:54:00:b7:59:2e brd ff:ff:ff:ff:ff:ff
5: br-a7f58256e5a7: <UP,BROADCAST,MULTICAST> mtu 1500 state DOWN qlen 0
    link/ether 2a:3a:87:6d:ed:4c brd ff:ff:ff:ff:ff:ff
6: docker0: <UP,BROADCAST,MULTICAST> mtu 1500 state DOWN qlen 0
    link/ether b6:44:4b:e5:16:c9 brd ff:ff:ff:ff:ff:ff
7: vmnet1: <UP,BROADCAST,MULTICAST> mtu 1500 state UNKNOWN qlen 1000
    link/ether 00:50:56:c0:00:01 brd ff:ff:ff:ff:ff:ff
8: vmnet8: <UP,BROADCAST,MULTICAST> mtu 1500 state UNKNOWN qlen 1000
    link/ether 00:50:56:c0:00:08 brd ff:ff:ff:ff:ff:ff
```

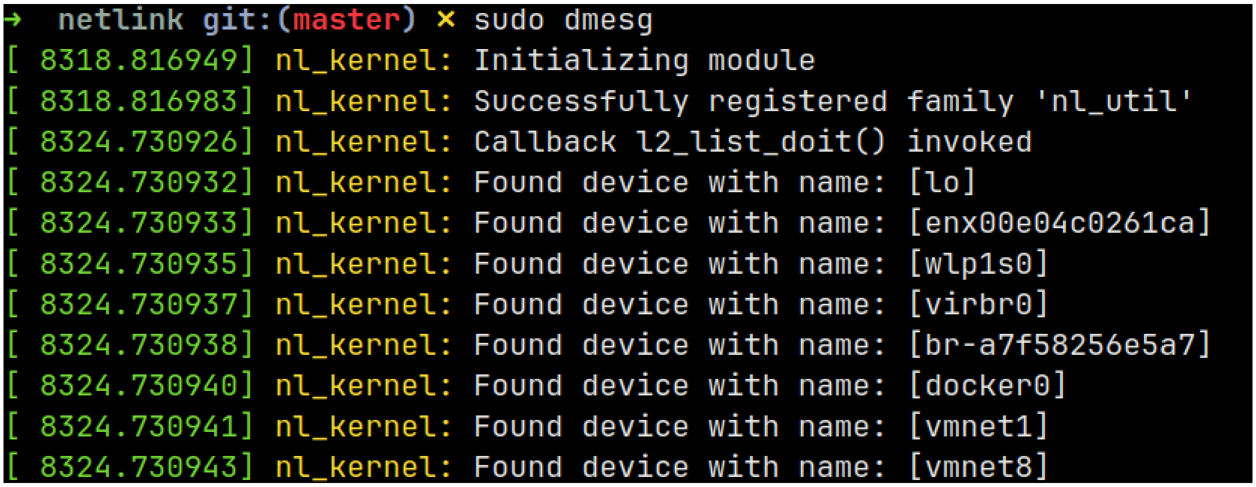
Figure 1: Example output of the list of L2 network interfaces

Kernel logs after running the command can be inspected by running `dmesg` with root privileges:

```
sudo dmesg
```

Listing 38: Showing last kernel logs

Example output of the kernel logs after running the `./nl_user.out show` command:



```
→  netlink git:(master) ✗ sudo dmesg
[ 8318.816949] nl_kernel: Initializing module
[ 8318.816983] nl_kernel: Successfully registered family 'nl_util'
[ 8324.730926] nl_kernel: Callback l2_list_doit() invoked
[ 8324.730932] nl_kernel: Found device with name: [lo]
[ 8324.730933] nl_kernel: Found device with name: [enx00e04c0261ca]
[ 8324.730935] nl_kernel: Found device with name: [wlp1s0]
[ 8324.730937] nl_kernel: Found device with name: [virbr0]
[ 8324.730938] nl_kernel: Found device with name: [br-a7f58256e5a7]
[ 8324.730940] nl_kernel: Found device with name: [docker0]
[ 8324.730941] nl_kernel: Found device with name: [vmnet1]
[ 8324.730943] nl_kernel: Found device with name: [vmnet8]
```

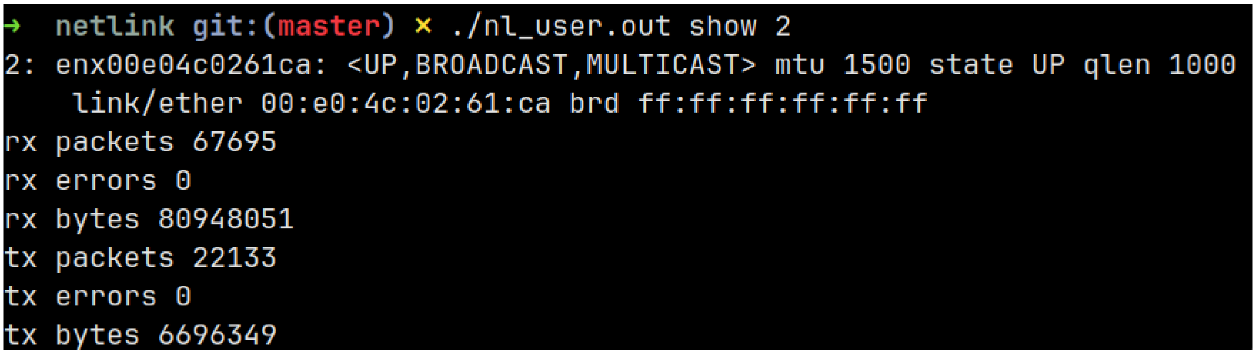
Figure 2: Example output of kernel logs after running the `show` command

To display details for a specific interface, the `./nl_user.out show <ifindex>` command is used, where `ifindex` is the numeric index of the interface.

```
./nl_user.out show 2
```

Listing 39: Displaying interface details

Example output from this command:

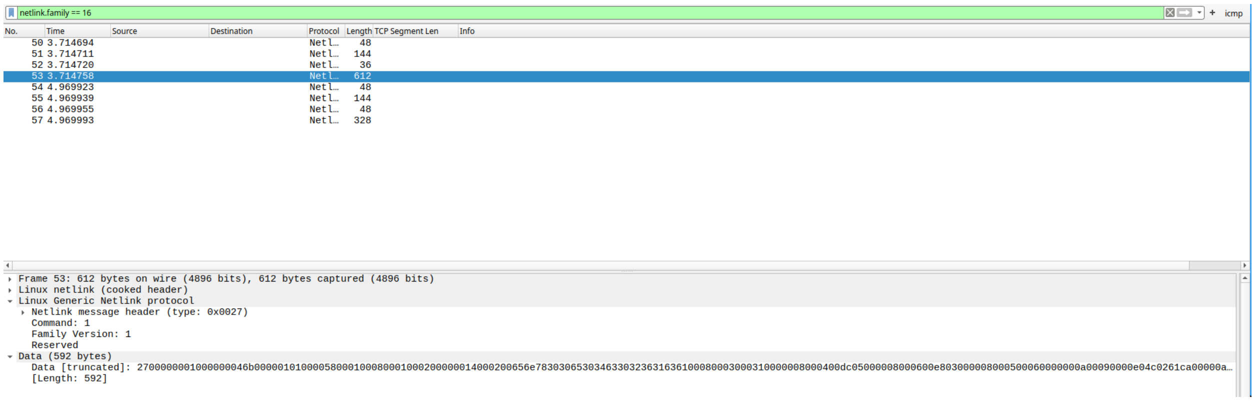


```
→  netlink git:(master) ✗ ./nl_user.out show 2
2: enx00e04c0261ca: <UP,BROADCAST,MULTICAST> mtu 1500 state UP qlen 1000
    link/ether 00:e0:4c:02:61:ca brd ff:ff:ff:ff:ff:ff
rx packets 67695
rx errors 0
rx bytes 80948051
tx packets 22133
tx errors 0
tx bytes 6696349
```

Figure 3: Example output of detailed information for a specific L2 interface

5 Traffic Capture Analysis

Wireshark has built-in dissectors for well-known Generic Netlink families like `nl80211`, which parse commands and attributes based on their definitions (e.g., from `<linux/nl80211.h>`). However, the `'nl_util'` family does not have one, so its payload (commands and attributes) appears as raw hex data.

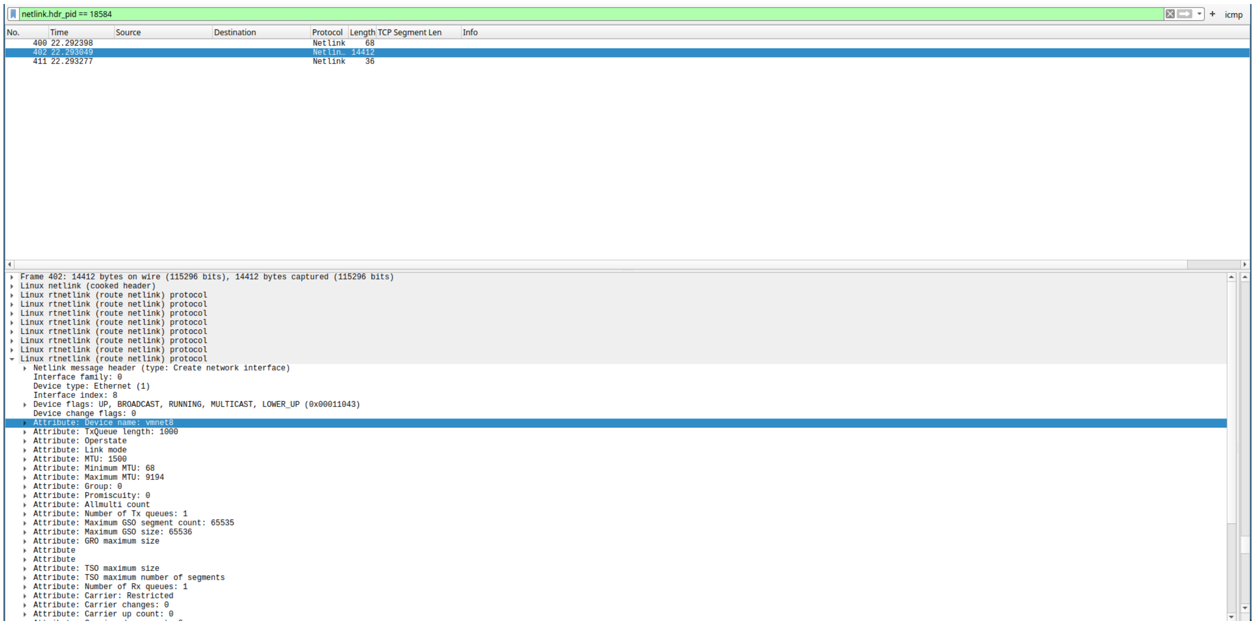


```
netlink.family == 16

No.  Time      Source  Destination  Protocol  Length  TCP Segment Len  Info
  50 3.714694                       Netl…       48
  51 3.714711                       Netl…      144
  52 3.714720                       Netl…       36
  53 3.714758                       Netl…      612
  54 4.969923                       Netl…       48
  55 4.969939                       Netl…      144
  56 4.969955                       Netl…       48
  57 4.969993                       Netl…      328

▸ Frame 53: 612 bytes on wire (4896 bits), 612 bytes captured (4896 bits)
▸ Linux netlink (cooked header)
▾ Linux Generic Netlink protocol
  ▸ Netlink message header (type: 0x0027)
    Command: 1
    Family Version: 1
    Reserved
▾ Data (592 bytes)
    Data [truncated]: 2700000001000000046b000001010000580001000800010002000000014000200656e78303065303463303236316361000800030003100000008000400dc050000008000600e8030000008000500060000000a00090000e04c0261ca00000a…
    [Length: 592]
```

Figure 4: Netlink capture of the `./nl_user.out show` command

To identify netlink messages related to the execution of `ip link show` command, the underlying system calls are traced using the `strace` tool. This allows retrieval of the process ID of the user-space process responsible for sending requests to the kernel and receiving Layer 2 interface information. The captured Netlink traffic can then be filtered in Wireshark based on this PID value.

The protocol number used by the `ip` tool is `NETLINK_ROUTE`, which is responsible for communication between user space and the kernel regarding network configuration. When `ip link show` is executed, the kernel responds by dumping information about all network interfaces. Each interface's information is encapsulated within an `rtnetlink` message header.



```
netlink.hdr_pid == 18584

No.  Time       Source  Destination  Protocol  Length  TCP Segment Len  Info
 400 22.292398                       Netlink      68
 402 22.293049                       Netlin…   14412
 411 22.293277                       Netlink      36

▸ Frame 402: 14412 bytes on wire (115296 bits), 14412 bytes captured (115296 bits)
▸ Linux netlink (cooked header)
▸ Linux rtnetlink (route netlink) protocol
▸ Linux rtnetlink (route netlink) protocol
▸ Linux rtnetlink (route netlink) protocol
▸ Linux rtnetlink (route netlink) protocol
▸ Linux rtnetlink (route netlink) protocol
▸ Linux rtnetlink (route netlink) protocol
▸ Linux rtnetlink (route netlink) protocol
▾ Linux rtnetlink (route netlink) protocol
  ▸ Netlink message header (type: Create network interface)
    Interface family: 0
    Device type: Ethernet (1)
    Interface index: 8
  ▸ Device flags: UP, BROADCAST, RUNNING, MULTICAST, LOWER_UP (0x00011043)
    Device change flags: 0
  ▸ Attribute: Device name: vmnet8
  ▸ Attribute: TxQueue length: 1000
  ▸ Attribute: Operstate
  ▸ Attribute: Link mode
  ▸ Attribute: MTU: 1500
  ▸ Attribute: Minimum MTU: 68
  ▸ Attribute: Maximum MTU: 9194
  ▸ Attribute: Group: 0
  ▸ Attribute: Promiscuity: 0
  ▸ Attribute: Allmulti count
  ▸ Attribute: Number of Tx queues: 1
  ▸ Attribute: Maximum GSO segment count: 65535
  ▸ Attribute: Maximum GSO size: 65536
  ▸ Attribute: GRO maximum size
  ▸ Attribute
  ▸ Attribute
  ▸ Attribute: TSO maximum size
  ▸ Attribute: TSO maximum number of segments
  ▸ Attribute: Number of Rx queues: 1
  ▸ Attribute: Carrier: Restricted
  ▸ Attribute: Carrier changes: 0
  ▸ Attribute: Carrier up count: 0
```

Figure 5: Netlink traffic capture of the `ip link show` command

6 Conclusion

The implemented Generic Netlink family `'nl_util'` supports future extensibility, allowing additional nested sub-attributes to be added without modifying the top-level structure. As a part of improvements to be made, the `l2_list_doit` callback function should ideally perform the same task as the underlying callback function of the `ip link show` command does - dumping the information instead of sending it all at once. However, it is retained as a doit handler for simplicity. Also, it would be beneficial to implement a dissector for the implemented family.

7 References

- RFC 3549: Linux Netlink as an IP Services Protocol, `https://datatracker.ietf.org/doc/html/rfc3549`
- Introduction to Generic Netlink, or How to Talk with the Linux Kernel, `https://www.yaroslavps.com/weblog/genl-intro/`
- Generic Netlink HOW-TO based on Jamal's original doc `https://lwn.net/Articles/208755/`
- Introduction to Netlink - Linux Kernel documentation `https://docs.kernel.org/userspace-api/netlink/intro.html`
- Linux Networking and Network Devices APIs `https://docs.kernel.org/networking/kapi.html`

Appendix

The source code is available on GitHub: `https://github.com/sappChak/netdev-genl`