Netlink and Netlink for Wifi

Comunicações Móveis

2021/2022

DETI – UA

Daniel Corujo

Outline

- Sockets
- Netlink library (libnl)
- nl80211 library

Netlink Library (libnl) in a slide

- Allows the use of Netlink sockets Communication
  - Connecting/Disconnecting
  - Sending/Receiving Data
  - Message construction/parsing
  - Message reception state machines
  - Data structures
- Uses the Netlink Protocol
  - Socket-based inter-process Communications mechanism
    - User-space processes <-> kernel
    - User-space process <-> User-space process
  - Is a datagram-oriented service

(What is a socket?)

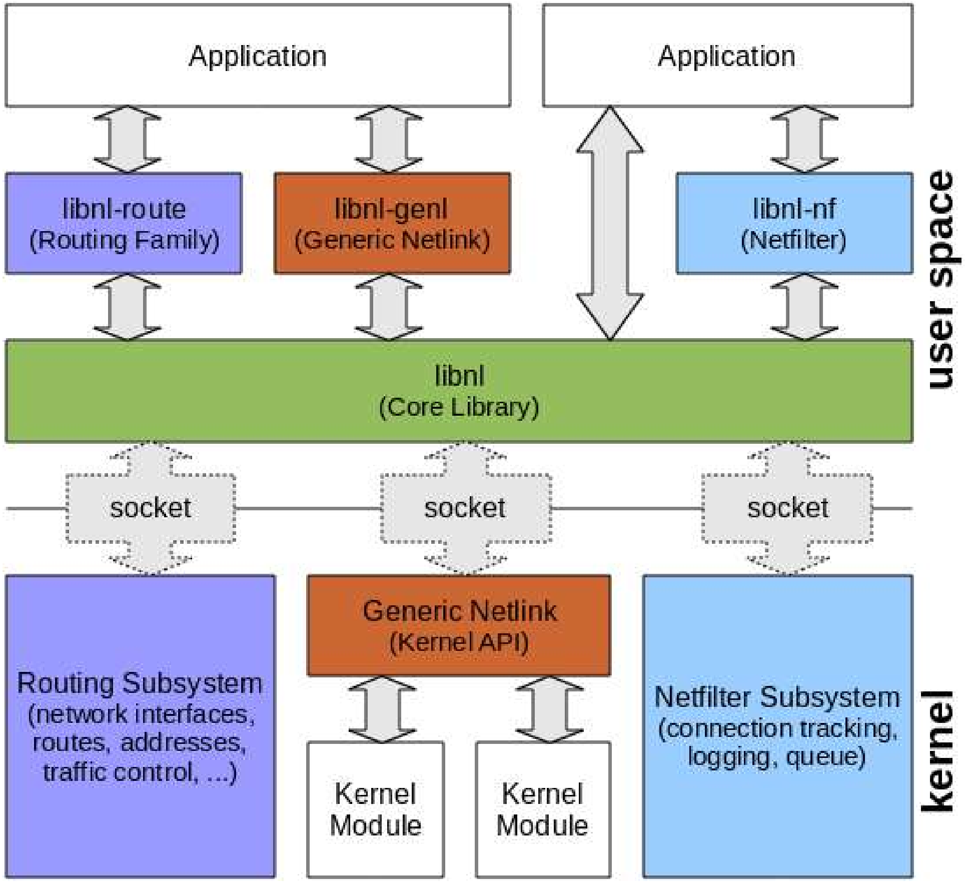
- Abstraction of an end-point for exchanging data between processes
  - Pipeline
    - `history | grep gcc`
  - Network Sockets
    - Between different hosts
  - Berkeley Sockets
    - Unix Domain socket / IPC socket
    - Internet Socket
  - Netlink Sockets
    - Host-only
    - Based on BSD sockets

(What is a socket)

- Berkeley socket uses a File Descriptor
  - Identifies an object for system resources
  - File descriptors cannot be directly accessed by user-space processes.
  - User-space processes does a system call to the kernel, providing it with the file descriptor reference, and the kernel accesses the resource (i.e., input/ouput) on its behalf
- Netlink socket used to use PID
  - Not anymore (multi-threading!)
  - Now uses a 32-bit port number

Netlink library (libnl)

- Libnl
  - Netlink Library
    - Socket handling
    - Sending and receiving
    - Message construction and parsing
- Libnl-genl
  - Generic Netlink Library
    - Controller API
    - Family and command registration



Netlink socket familiies

- AF_NETLINK
  - Supports different subsects, each targeting different kernel components and messages (i.e., address families)

- Socket types
  - SOCK_RAW
  - SOCK_DGRAM

- Netlink family
  - Selects the kernel module or netlink group to communicate with

Netlink Families

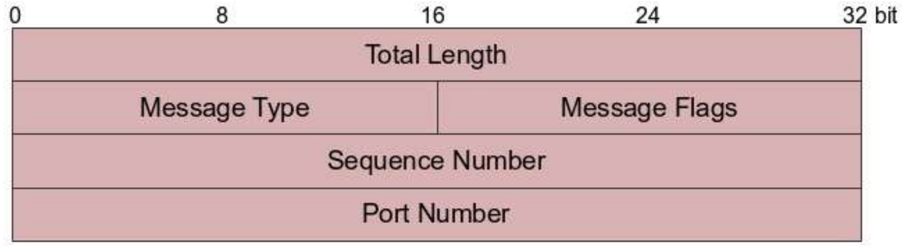
- NETLINK_ROUTE
- NETLINK_W1
  - 1-wire
- NETLINK_USERSOCK
- NETLINK_FIREWALL
- NETLINK_SOCK_DIAG
- NETLINK_NFLOG
  - Netfilter/iptables log
- NETLINK_XFRM
- NETLINK_SELINUX
- NETLINK_ISCSI
- NETLINK_AUDIT
- NETLINK_FIB_LOOKUP
- NETLINK_CONNECTOR
- NETLINK_NETFILTER
- NETLINK_GENERIC
- …

Netlink Protocol

Netlink Protocol

- Socket-based inter-process Communications mechanism
  - User-space processes <-> kernel
  - User-space process <-> User-space process
- Is a datagram-oriented service

Datagram Message Format



0 8 16 24 32 bit

Total Length	
Message Type	Message Flags
Sequence Number	
Port Number	

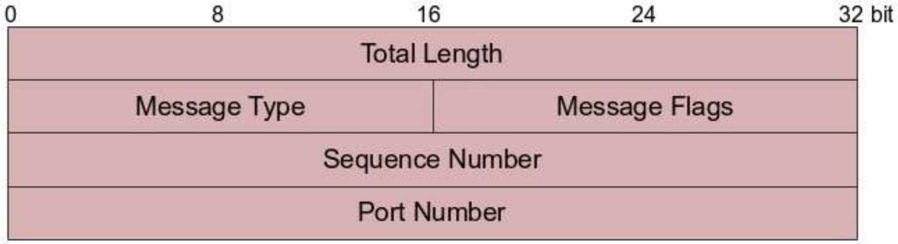
- Message Type
  - Type of payload
- Message Flags
  - Modify the behavior of the message type
    - Request
    - Multicast
    - Acknowledgement
    - Echo

Types of Netlink Messages

- Requests
- Notifications
- Replies

Message Types (datagram parameter)

- NLMSG_NOOP
  - No operation, message must be discarded
- NLMSG_ERROR
  - Error message or ACK
- NLMSG_DONE
  - End of multipart sequence
- NLMSG_OVERRUN
  - Overrun notification (Error)



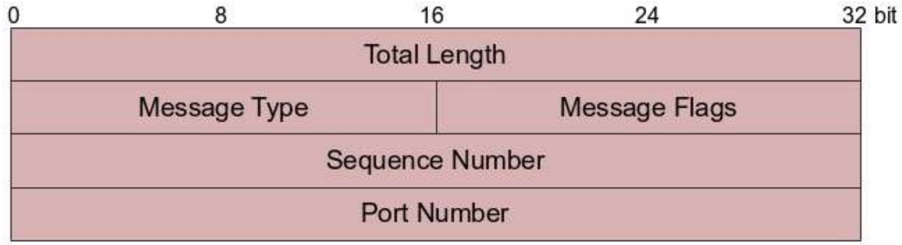
Total Length

Message Type | Message Flags

Sequence Number

Port Number

Message Flags



0 8	16 24 32 bit
Total Length	
Message Type	Message Flags
Sequence Number	
Port Number	

- NLM_F_REQUEST - Message is a request
- NLM_F_MULTI - Multipart message
- NLM_F_ACK - ACK message requested
- NLM_F_ECHO - Request to echo the request

- Universal flags for GET requests
  - NLM_F_ROOT - Return based on root of tree.
  - NLM_F_MATCH - Return all matching entries.
  - NLM_F_ATOMIC - Obsoleted, once used to request an atomic operation.
  - NLM_F_DUMP - Return a list of all objects (NLM_F_ROOT|NLM_F_MATCH).

Netlink Sockets

Netlink sockets

- Needed to use the Netlink protocol
- It’s where you send and receive protocol messages

Program example

- Let’s build a program that waits for NETLINK_ROUTE notifications. When one is received, it calls a user function (i.e., **callback function**)
- We need to:
  - Create a socket
  - Indicate which is the callback function
  - Connect the socket to NETLINK_ROUTE
  - Subscribe for receiving specific notifications, from a specific multicast group
  - Keep the program running, listening for the notifications
  - *(a notification is an event that is sent when some specific action occurred)*

Program example

//this program waits for NETLINK_ROUTE notifications. When one is received, it calls a user function (i.e., **callback function**)

```
//function
static int my_func(struct nl_msg *msg, void *arg){
        printf("A message was received! This is my function");
        return 0; }
```

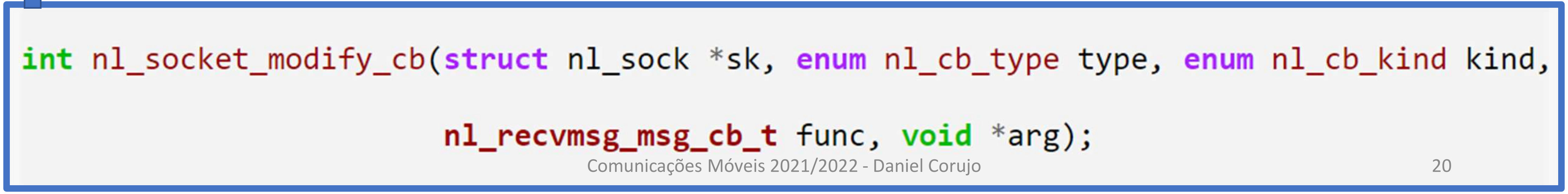
Program example

```
//Allocate a new socket
stuct nl_sock * sk = nl_socket_alloc();
```

```
//Disable sequence number checking, as we’re just using notifications
//Netlink automatically takes care of sequence numbers, when using the nl_send_auto() for sending messages.
//However, if we’re using a non request/reply netlink protocol, we must explicitly disable it:
nl_socket_disable_seq_check(sk);
```

Program example

```
//Callback function that calls "my_func"
nl_socket_modify_cb(sk, NL_CB_VALID, NL_CB_CUSTOM, my_func, null);

//Connect to routing netlink protocol
nl_connect(sk, NETLINK_ROUTE);
```



```
int nl_socket_modify_cb(struct nl_sock *sk, enum nl_cb_type type, enum nl_cb_kind kind,
                        nl_recvmsg_msg_cb_t func, void *arg);
```

Program example

```
//subscribe to link notifications multicast group
nl_socket_add_membership(sk, RTNLGRP_LINK,0);

//Listen to messages
//This will trigger the callback function, when a notification of
NETLINK_ROUTE protocol is received

while (1)
        nl_recvmsgs_default(sk);
```

Socket related functions

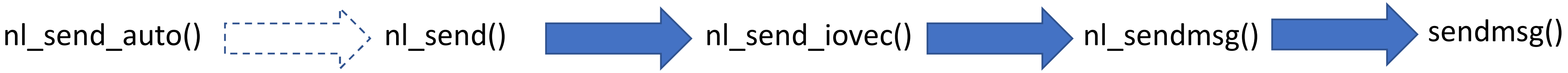
- nl_socket_set_local_port
- nl_socket_set_peer_port
- nl_socket_get_fd
- nl_socket_set_buffer_size
- …

Sending messages through a netlink socket

- Two ways
  - Automatic
    - nl_send_auto()
      - Automatically completes the missing bits of the netlink message header
      - Automatically deals with addressing based on current information set in the socket
      - Passes the message to nl_send()
      - Creates the final message
  - Manual (when automatic filling is not suitable)
    - Directly use nl_send()
      - Embeds the message into a ‘iovec’ structure and pass it to nl_send_iovec()
    - nl_send() can also be overwritten via nl_cb_overwrite_send()

Sending messages through a netlink socket

- Both ways lead to
  - nl_send_iovec()
    - Fills the message header – tries to find the peer, or leaves it to the kernel
    - Leads to:
  - nl_sendmsg()
    - Takes the final message and the optional header
    - Sends the final total message to sendmsg()



Receiving messages through a netlink socket

- Our program example received messages (Notifications) from the kernel using the following function:

```
while (1)
        nl_recvmsgs_default(sk);
```

- Easiest function
  - Receives messages based on how we configured the socket
  - Usually, the default behavior is enough
  - Fetches the callbackfunction (cb = nl_socket_get_cb(sk))
  - calls nl_recvmsgs()

Receiving messages through a netlink socket

- nl_recvmsgs()
  - Actual message reception loop
  - If we need specific reception characteristics, we can provide a complete own implementation of the reception mechanism
    - nl_cb_overwrite_recvmsgs()

Parsing messages

- Messages are 4-bytes aligned in all boundaries
- There are two methods of parsing
  - Low-level interface (manual parsing)
  - High-level interface (Implement a parser as part of cache operations)

- What is receiving a netlink protocol message on a netlink socket?
  - What you receive from a netlink socket is typically a stream of messages.
  - You will be given a buffer and its length
  - The buffer may contain any number of netlink messages.
  - The first message header starts at the beginning of the message stream
  - You can reach the next header by calling nlmsg_next() on the previous header
    - Position = Remaining_number_of_bytes – current_message_size

Parsing messages

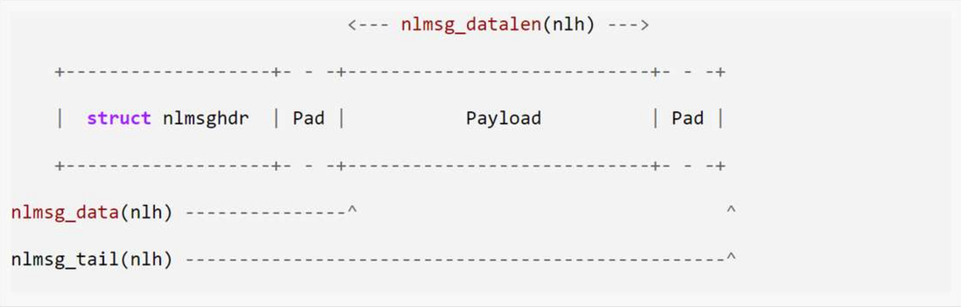
- Despite having nlmsg_next() we don’t know if there are more messages
- We must assume that more messages follow untill all bytes of the stream have been processed
  - nlmsg_ok()
    - Returns true if another message fits into the remaining number of bytes in the message stream
  - nlmsg_valid_hdr()
    - Checks if a message contains at least a minimum of payload

Creating a parsing message function: example

```
#include <netlink/msg.h>

void my_parse(void *stream, int length){

    struct nlmsghdr *hdr = stream;

    while (nlmsg_ok(hdr, length)) {

        // Parse message here

        hdr = nlmsg_next(hdr, &length);
    }
}
```

Parsing messages

- Accessing message payload
  - Remember that the header has alignment
  - Some of its fields, and the header itself, might have padding
  - nlmsg_data() returns a pointer to the start of the payload
  - nlmsg_datalen() returns the length of the message payload
  - nlmsg_tail() return a pointer to the end of the payload, including padding

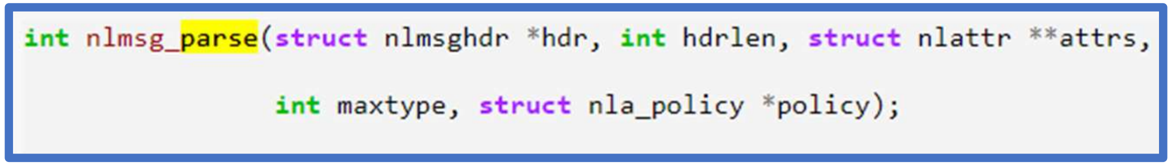


```
                                <--- nlmsg_datalen(nlh) --->

    +-------------------+- - -+-----------------------------+- - -+

    |  struct nlmsghdr  | Pad |           Payload           | Pad |

    +-------------------+- - -+-----------------------------+- - -+

nlmsg_data(nlh) ---------------^                                   ^

nlmsg_tail(nlh) ---------------------------------------------------^
```

Attributes

- Most netlink protocol messages use netlink attributes
- This means that, the message payload is composed of
  - Protocol Header (+ padding)
  - Attributes (+ padding)
- nlmsg_attrdata() returns a pointer to the beginning of the attributes section
- nlmsg_attrlen() returns the length of the attributes section

Parsing function



```
int nlmsg_parse(struct nlmsghdr *hdr, int hdrlen, struct nlattr **attrs,
                int maxtype, struct nla_policy *policy);
```

- nlmsg_parse()
  - Starts by validating the header
    - If hdrlen>0, calls nlmsg_valid_hdr()
    - Fills an array with pointers to each atribute
- nlmsg_validate()
  - Similar, but does not create the array
- There are also atribute variants of this function
  - nla_parse()
  - nla_validate() → does not create the array

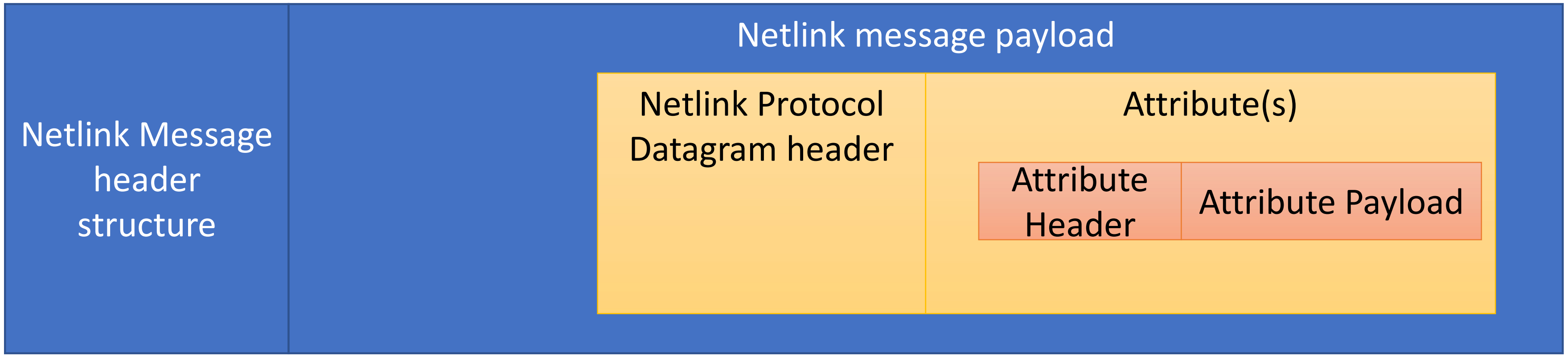
Creating a message

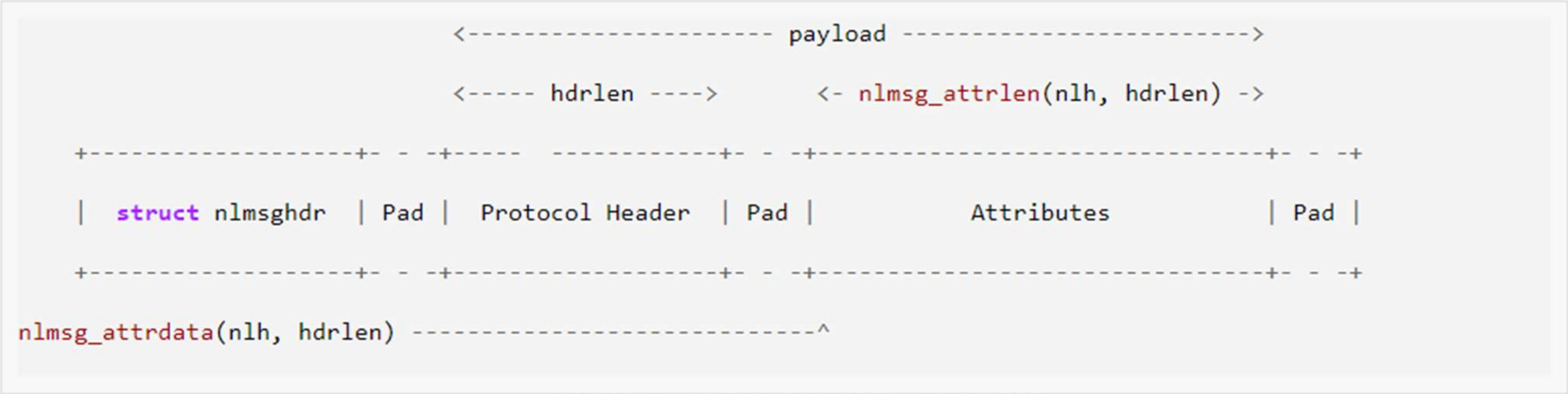
- We need to allocate a struct nl_message
    - Holds the message header and payload
- nlmsg_alloc() is the function used for message allocation
    - Uses the default maximum message size (one page, typically 4K)
        - Default size can be changed with nlmsg_set_default_size(size_t)
    - nlmsg_alloc_size() is a variant that allows definition of size at allocation time
    - Alternative #1: If we want to reuse an already known header, we can call nlmsg_inherit()
        - Appends header
    - Alternative #2: nmsg_alloc_simple() takes a message type and flag
        - Creates and appends header

Create a message

- If we don’t automatically create a header (nlmsg_alloc_simple() or nlmsg_inherit()) we need to add the header ourselves
- nlmsg_put()
    - nlmsg_type
    - nlmsg_flags
    - seqnr (NL_AUTO_SEQ)
    - Port (NL_AUTO_PORT)

Do not get confused!





```
                 <-------------------- payload ----------------------->
                 <----- hdrlen ---->       <- nlmsg_attrlen(nlh, hdrlen) ->
   +------------------+- - -+-----   -----------+- - -+--------------------------------+- - -+
   |  struct nlmsghdr  | Pad |  Protocol Header  | Pad |           Attributes            | Pad |
   +------------------+- - -+------------------+- - -+--------------------------------+- - -+
nlmsg_attrdata(nlh, hdrlen) ---------------------------^
```

Example

```
#include <netlink/msg.h>

struct nlmsghdr *hdr;  //header structure

struct nl_msg *msg;  //message structure

struct myhdr {

	uint32_t foo1, foo2;

} hdr = { 10, 20 }; //my header
```

Example (contd)

```
msg = nlmsg_alloc(); //allocate default maximum size message

// Create header with automatic fill of port and sequence number
//Notice that space for hdr is reserved in the message
hdr = nlmsg_put(msg, NL_AUTO_PORT, NL_AUTO_SEQ, MY_MSGTYPE,
sizeof(hdr), NLM_F_CREATE);

//copy costumised header into payload
memcpy(nlmsg_data(hdr), &hdr, sizeof(hdr));
```

What about the payload?

- Payload should be encoded as netlink attributes whenever possible
- Atributes are aligned at multiples of 4bytes position
  - nla_padlen() returns the number of padding bytes needed
- Atribute encoding
  - Length (16bit)
  - Type (16bit)
  - Payload

Parsing attributes

- You can obtain them as an array, when you parse the message with nlmsg_parse()
- We can also do it manually
  - Navigate individual attributes using nla_next() after getting a pointer to the first one using nlmsg_attrdata()
  - nla_ok() determines wether another atribute fits into the remaining number of bytes or not

Parsing Attributes

- Each atribute has a header and payload
- Access the header: nla_len()
- Access the type: nla_type()
- Access the payload: nla_data()
  - Avoid casting to anything larger than 4bytes due to the mandatory alignment

- Easier alternative: nla_parse()
  - Iterates all attributes in the stream
  - Validates each atribute
  - When validation succeeds, stores pointers to attributes in array

Attribute Validation

- We need to ensure that we get attributes formatted in the correct way
- We define structures that compose policies
  - Indicate what is the structure of the attribute's header

```
struct nla_policy {
        uint16_t type;          //NLA_U32/16/8/4
        uint16_t minlen;    //minimum payload length
        uint16_t maxlen; }; //maximum payload length
```

s

Attribute parsing example

```
//headers are ignored in this example
#include <netlink/msg.h>
#include <netlink/attr.h>

enum {
      MY_ATTR_FOO = 1, MY_ATTR_BAR,  __MY_ATTR_MAX,};

#define MY_ATTR_MAX (__MY_ATTR_MAX - 1)
```

Attribute parsing example (contd)

```
static struct nla_policy my_policy[MY_ATTR_MAX+1] = {
        //Validation policy
      [MY_ATTR_FOO] = { .type = NLA_U32 },

      [MY_ATTR_BAR] = { .type = NLA_STRING,

                    .maxlen = 16 },

};
```

Attribute parsing example (contd)

```
void parse_msg(struct nlmsghdr *nlh)

        struct nlattr *attrs[MY_ATTR_MAX+1];

        if (nlmsg_parse(nlh, 0, attrs, MY_ATTR_MAX, my_policy) < 0) /* error */

        if (attrs[MY_ATTR_FOO]) {
               /* MY_ATTR_FOO is present in message */
                printf("value: %u\n", nla_get_u32(attrs[MY_ATTR_FOO]));
        } }
```

I want to find a single attribute!

- There are functions that iterate over all attributes, search for a matching one and return a pointer to its header

- nla_find();
- nlmsg_find_attr

Nested attributes

- Attributes included as payload of container attributes
  - Type: NLA_NESTED
- Attributes can be stored inside a tree structure
- It is the common way to transmit lists of objects

Parsing Nested Attributes

- nla_parse_nested()
  - Identical to nla_parse(), but
    - Uses a struct nlattr as argument
    - Uses the payload as stream of attributes

Constructing Nested Attributes

- We can nest attributes by surrounding them with
  - nla_nest_start()
    - Add attribute header, without payload
    - All data added from this point on, will be part of the container
  - nla_nest_end()
    - Closes the container attribute

Create Netlink Message with Attributes

```
//We are going to put this behavior inside of a function
//So everything you’ll see next will be placed inside this function

struct nl_msg *build_msg(int ifindex, struct nl_addr *lladdr, int mtu)
{
        //Code from the next slides will be here
}
```

Create Netlink Message with Attributes (cont)

```
struct nl_msg *msg;

struct nlattr *info, *vlan;

struct ifinfomsg ifi = {

        .ifi_family = AF_INET,

        .ifi_index = ifindex,

};
```

Estrutura predefinida no netlink:

```
struct ifinfomsg {
 unsigned char ifi_family; /* AF_UNSPEC */
 unsigned short ifi_type; /* Device type */
 int ifi_index; /* Interface index */
 unsigned int ifi_flags; /* Device flags */
 unsigned int ifi_change; /* change mask */ };
```

Create Netlink Message with Attributes (cont)

```
/* Allocate a default sized netlink message */

       //ROUTING FAMILY NETLINK
     if (!(msg = nlmsg_alloc_simple(RTM_SETLINK, 0)))

          return NULL;
```

Create Netlink Message with Attributes (cont)

```
/* Append the protocol specific header (struct ifinfomsg)*/

    if (nlmsg_append(msg, &ifi, sizeof(ifi), NLMSG_ALIGNTO) < 0)

        goto nla_put_failure;
```

Create Netlink Message with Attributes (cont)

```
/* Append a 32 bit integer attribute to carry the MTU */

    NLA_PUT_U32(msg, IFLA_MTU, mtu);



    /* Append a unspecific attribute to carry the link layer address */

    NLA_PUT_ADDR(msg, IFLA_ADDRESS, lladdr);
```

Create Netlink Message with Attributes (cont)

```
/* Append a container for nested attributes to carry link information */
    if (!(info = nla_nest_start(msg, IFLA_LINKINFO)))
        goto nla_put_failure;


    /* Put a string attribute into the container */
    NLA_PUT_STRING(msg, IFLA_INFO_KIND, "vlan");
```

Create Netlink Message with Attributes (cont)

```
/* Append another container inside the open container to carry
 * vlan specific attributes  */

    if (!(vlan = nla_nest_start(msg, IFLA_INFO_DATA)))

        goto nla_put_failure;


    /* add vlan specific info attributes here... */
```

Create Netlink Message with Attributes (cont)

```
/* Finish nesting the vlan attributes and close the second container. */

    nla_nest_end(msg, vlan);


    /* Finish nesting the link info attribute and close the first container. */

    nla_nest_end(msg, info);


    return msg;
```

Create Netlink Message with Attributes (cont)

```
//code alias for failures:
nla_put_failure:
	nlmsg_free(msg);
	return NULL;

}
```

Parsing a Netlink Message with Attributes

```
//We are going to put this behavior inside of a function
//So everything you'll see next will be placed inside this function

int parse_message(struct nlmsghdr *hdr)
{
                // The code from the next slides will be here
}
```

Parsing a Netlink Message with Attributes

```
 /* The policy defines two attributes: a 32 bit integer and a container
for nested attributes.        */

      struct nla_policy attr_policy[] = {
            [ATTR_FOO] = { .type = NLA_U32 },
            [ATTR_BAR] = { .type = NLA_NESTED },       };

      struct nlattr *attrs[ATTR_MAX+1];
      int err;
```

Parsing a Netlink Message with Attributes

```
 /* The nlmsg_parse() function will make sure that the message
contains enough payload to hold the header (struct my_hdr), validates
any attributes attached to the messages and stores a pointer to each
attribute in the attrs[] array accessable by attribute type.        */

if ((err = nlmsg_parse(hdr, sizeof(struct my_hdr), attrs, ATTR_MAX,
attr_policy)) < 0)
            goto errout;
```

```
if (attrs[ATTR_FOO]) {

        /*It is safe to directly access the attribute payload without any
further checks since nlmsg_parse() enforced the policy. */

        uint32_t foo = nla_get_u32(attrs[ATTR_FOO]);

    }
```

```
if (attrs[ATTR_BAR]) {
        struct *nested[NESTED_MAX+1];
        /* Attributes nested in a container can be parsed the same way as top level attributes. */

        err = nla_parse_nested(nested, NESTED_MAX, attrs[ATTR_BAR], nested_policy);

        if (err < 0)
            goto errout;

        // Process nested attributes here.

    }
```

More details

- Core Library Developer’s Guide
  - https://www.infradead.org/~tgr/libnl/doc/core.html
- API Reference
  - https://www.infradead.org/~tgr/libnl/doc/api/group__core.html

Genery Netlink Library

Libn-genl

- One of the drawbacks of the Netlink protocol is that the number of protocol families is limited to 32 (MAX_LINKS)
- For this reason, the “Generic Netlink Family” was created
    - Provides support for adding a higher number of families
    - Acts as a Netlink multiplexer
    - Works with a single Netlink family: NETLINK_GENERIC
    - The generic Netlink protocol is based on the Netlink protocol and used its API

Generic Netlink Library – Controller API

- The controller is a component in the kernel that resolves Generic Netlink family names to their numeric identifiers. This module provides functions to query the controller to access the resolving functionality.

- Example:
  - expectedId = genl_ctrl_resolve(sk, "nl80211");

Generic Netlink Library – Controller API

- Genlmsg_put adds Generic Netlink headers to Netlink messages
    - genlmsg_put(msg, 0, 0, driver_id, 0, NLM_F_DUMP, NL80211_CMD_GET_SCAN, 0); // Setup which command to run.

Generic Netlink Library – Controller API

- Family and command registration

- Registers the specified Generic Netlink family definition together with all associated commands. After registration, received Generic Netlink messages can be passed to **genl_handle_msg()** which will validate the messages, look for a matching command and call the respective callback function automatically.

How can I use lib-nl and lib-genl

- You need to install the libs
  - libnl-3-dev
  - libnl-genl-3-dev

- The code you do will have to include related headers
  - #include "netlink/netlink.h"
  - #include "netlink/genl/genl.h"
  - #include "netlink/genl/ctrl.h“

To compile

- gcc example.c -o example -I/usr/include/libnl3/ -lnl-genl-3 -lnl-3

- The -I holds the location of Include header files, which are referenced in the #include directive. E.g. -I /home/myname/include will search for a header file in “/home/myname/include”.
- Since in Unix/Linux OSes library names begin with “lib” (e.g “libkuku.a”), then the “lib” prefix is omitted from the -l parameter: “-lkuku”.
- (Not used here, but the -L holds the location of library files, and -l holds the name of a library file, which should be included in the Link phase of the program’s build.)

What has this to do with Wi-Fi?

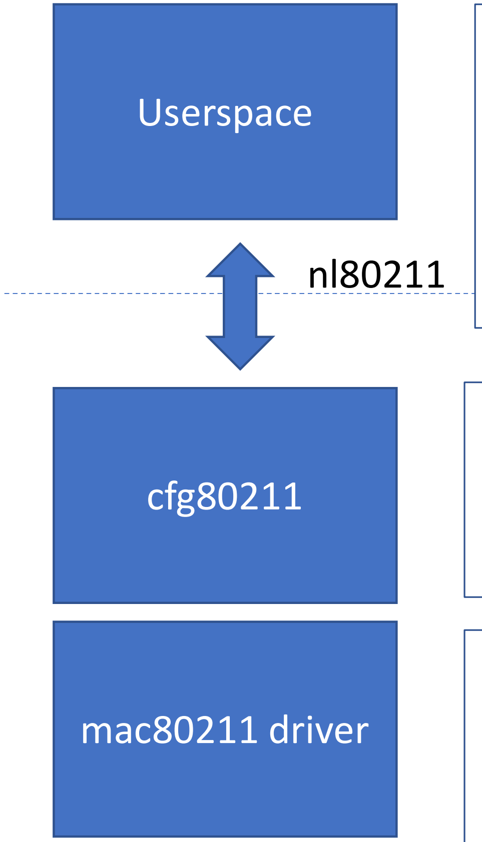
- Until now, we have been addressing how the netlink library works
- We will now adress a Wi-Fi related family…

NL80211

NL80211

- 802.11 userspace interface for Netlink
- Allows control of wireless drivers from userspace, using IPC Communications between kernel and userpace
- Before:
  - Input/output control (ioctl) system calls
  - Wireless Extensions generic API

NL80211



- nl80211 is the interface between user space software (iw, wpa_supplicant, etc.) and the kernel (cfg80211 and mac80211 kernel modules, and specific drivers)
- Register a generic netlink(nl) family “nl80211”
- Handle the messages for “nl80211” generic nl family from userpace via libnl
- Send messages to userspace program which monitors the generic nl family “nl80211”

- Registers and configures mac80211-based SoftMAC devices on the networking subsystem (unlike FullMAC which is done in HW/Firmware on the NIC)
- https://wireless.wiki.kernel.org/en/developers/Documentation/cfg80211

- Framework for writing drivers for wireless devices
- This is done via SoftMAC, where 802.11 frames can be managed via software (parsing and creation)
- Implements cfg80211 callbacks
- https://wireless.wiki.kernel.org/en/developers/Documentation/mac80211

Can I use this?

- Only works on network interfaces whose drivers are compatible with Netlink.
- Test this by running `iw list`
- Some of the commands used in the code require ‘root’ privileges
- The library’s header files already come installed with the kernel

Examples of NL80211-based apps

- iw
  - Wireless devices CLI configuration tool
- crda
  - Helper for regulatory domains compliance using udev device management
  - Country code, calibration channel, calibration bandwidth
- hostapd
  - Access Point connection and authentication daemon
- wpa_supplicant (-Dnl80211)
- iwd
  - Wireless management servisse
- NetworkManager
  - Connection manager based on iwd
- ConnMan
  - Another Connection Manager (CLI)

Role of nl80211.h

- Provides several enums related with 802.11
- You then use generic nl80211 constructs (i.e., sockets, messages, attributes, commands) to send commands to the wireless card

Generic structure of a nl80211 program

1 - Allocate new netlink socket in memory

```
struct nl_sock* sk = nl_socket_alloc();
```

2 – Create file descriptor and create socket

```
genl_connect(sk);
```

3 - Find the nl80211 driver ID.

```
int driver_id = genl_ctrl_resolve(socket, "nl80211");
```

Generic structure of a nl80211 program

- From here, you can do whatever you want
- Typically:
  - Create a callback function and attach it to the socket
  - Create a message with a command to be sent to the wifi card
    - Header + attribute(s)
  - Send the message and wait for answer
    - Might involve joining some multicast group
  - When reply arrives, your callback function is called
  - Inside of that function, you build code that processes the message payload and attributes (i.e., prints information in screen)

Create a callback function and attach it to the socket

- nl_socket_modify_cb(sk, NL_CB_VALID, NL_CB_CUSTOM, yourFunctionName, NULL);

Create a message with a command to be sent to the wifi card

```
 //allocate a message
   struct nl_msg* msg = nlmsg_alloc();

//request interface’s configuration
   enum nl80211_commands* cmd = NL80211_CMD_GET_INTERFACE;
   int ifIndex = if_nametoindex("wlan0");
   int flags = 0;

   // setup the message
   genlmsg_put(msg, 0, 0, driver_id, 0, flags, cmd, 0);

   //add message attributes
   NLA_PUT_U32(msg, NL80211_ATTR_IFINDEX, ifIndex);
```

Send the message and wait for answer

```
//send the messge (this frees it)
 ret = nl_send_auto_complete(sk, msg);

 //block for message to return
 nl_recvmsgs_default(sk);
```

Callback function

```
static int nlCallback(struct nl_msg* msg, void* arg) {
  struct nlmsghdr* ret_hdr = nlmsg_hdr(msg);
  struct nlattr *tb_msg[NL80211_ATTR_MAX + 1];

  if (ret_hdr->nlmsg_type != expectedId)    return NL_STOP;

struct genlmsghdr *gnlh = (struct genlmsghdr*) nlmsg_data(ret_hdr);
nla_parse(tb_msg, NL80211_ATTR_MAX, genlmsg_attrdata(gnlh, 0),  genlmsg_attrlen(gnlh, 0), NULL);

  if (tb_msg[NL80211_ATTR_IFTYPE]) {
    int type = nla_get_u32(tb_msg[NL80211_ATTR_IFTYPE]);
    printf("Type: %d", type);
  } }
```

Joining a multicast group

```
int mcid = nl_get_multicast_id(socket, "nl80211", "scan");
nl_socket_add_membership(socket, mcid);
```

Executing a scan

- Use the “NL80211_CMD_TRIGGER_SCAN” command to start a scan
  - If you try to start another one when one is running, this would fail
- Listen for the scan to complete when you get a NL80211_CMD_NEW_SCAN_RESULTS
- You can then send a NL80211_CMD_GET_SCAN command to ask for the results
  - You will get one message back for every station found, so be ready to handle multiple messages

Examples of commands

- Detailed explanation: nl80211.h
- Commands
  - NL80211_CMD_GET_INTERFACE
    - Request na interface’s configuration
      - For all interfaces or a specific one
      - For a single interface it is necessary to send the request with the atribute NL80211_ATTR_IFINDEX
  - NL80211_CMD_SET_BEACON
    - Change the beacon on na access point interface using attributes
  - NL80211_CMD_GET_STATION
    - Get attributes for station identified by NL80211_ATTR_MAC on the interface identified by NL80211_ATTR_IFINDEX
  - NL80211_CMD_GET_REG
    - Obtain regulatory domain information

More information

- Unfortunately, there is no developer’s guide for nl80211
- The best source of (properly coded) information, is to look at the reference implementations (iw)
- Some simplier code snips may be found in individual projects online
  - https://github.com/Robpol86/libnl