Netlink Operations for WTP-Station Control

Purpose:

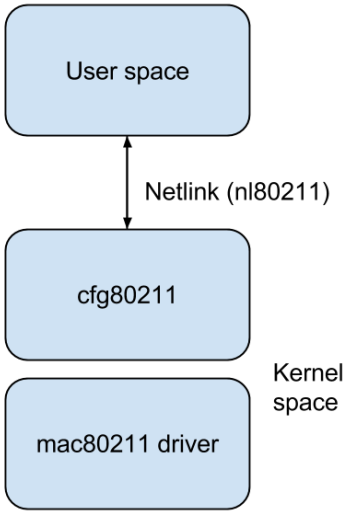
The Access Controller (AC) required information about the connected stations to manage new requests and balance the overall load on a single Access Point (AP) and thus subsets of the overall network. The aim of this activity is to fetch information of connected stations/clients from a Wireless Termination Point (WTP), and deliver it to a centralized management server for monitoring, control and management of the network. Depending on the requirement of the CAPWAP operations one might need to either view all the stations connected to the particular AP, view details of only one particular station connected to the AP identifying the station by its mac address or delete the connection between a particular station and AP again identifying it by its mac address.

Framework:

The wireless card is controlled through kernel space functionalities which are accessed in the user space through sockets. In the development framework since IOCTL socket are getting deprecated (Wireless extensions paradigm) and is being replaced by netlink socket, we are also using netlink sockets for controlling the wireless card (hardware). IOCTL are specific to a particular driver or hardware. They restricts the application to a single device. While using netlink library, our application will be interoperable will all wireless cards from any vendor.

Availability of user space commands:

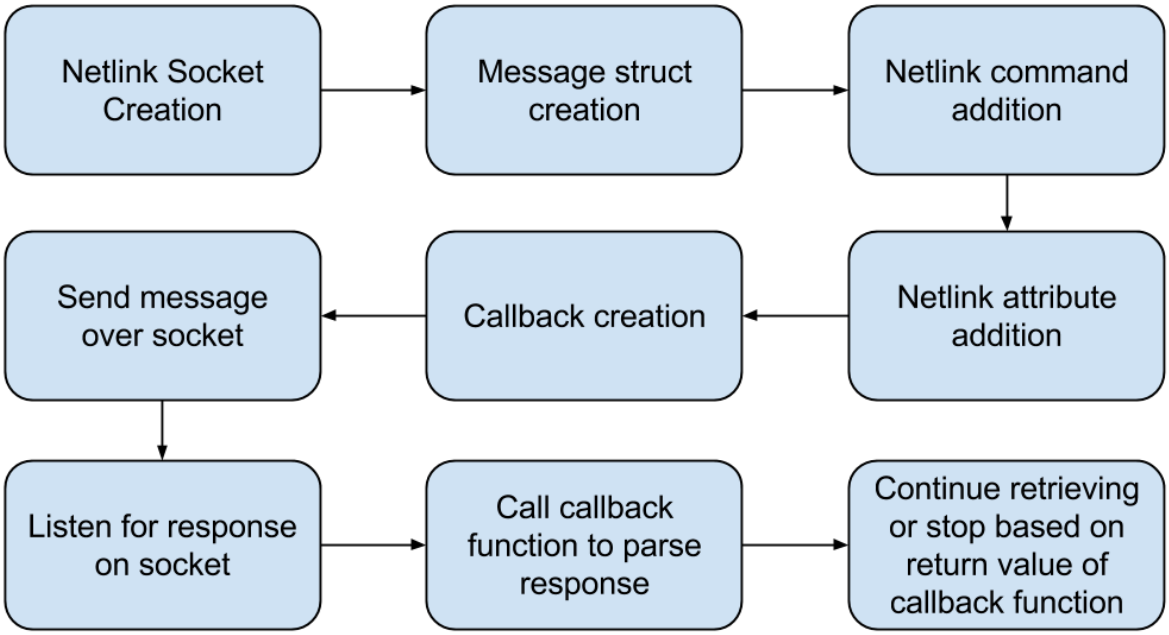
For the softmac i.e. user space control over hardware, that rely on mac80211 drivers have a header file nl80211.h which specifies the commands and the attributes that can be controlled through user space functions by means of netlink socket. We utilize the functionalities for the same in our code. The following is the organization of the modules:



Learning from the code of iw

Since netlink socket tutorials in regards to nl80211 are not widely available a good starting point was to understand the implementations in the iw. iw is a command line utility in Linux that allows users to perform operations on the wireless hardware utilizing a few operational commands and attributes from the nl80211.h header file. We also had the code of hostapd to understand from however that was too hard as a starting point.
As the purpose was to gain insight into the stations connected to the AP, a survey into the functions inside of iw pertaining to those was started. Understanding of the Netlink libraries and their functionalities - libnl-3 and libnl-genl-3 were linked to the main code.

Netlink operation in detail:



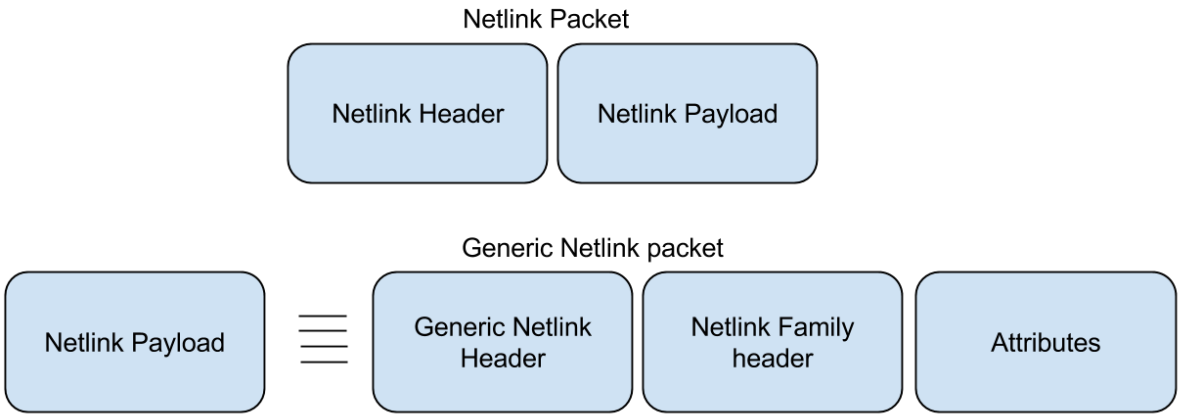
Overview of the netlink process: The details for each step in the flow-chart are highlighted separately. In this section, the entire process is summarized to give an idea about the entire socket communication process.

Netlink is a socket family in the linux kernel interface that is used for inter-process communication (IPC) between the kernel and user space processes or between the user processes. Netlink communication cannot traverse host boundaries because of its addressing scheme that is done through PIDs which are inherently local. In the our scheme of things we are utilizing the netlink socket to communicate with the mac80211 based wireless drivers in the kernel space from functions written in the user space to configure them.

Since the netlink process that happens at the kernel level is beyond the scope of discussion here, we will focus only on the user space utilities that are used in communication. Like any regular inter process socket, we create the netlink socket initialized with its AF_NETLINK

socket family. This initialization process ensures that the command functions sent through the socket in future would be mapped correctly to the kernel space drivers they are intended for. This establishes the framework for communication between the kernel and user space.

The initialization process ensures that we have set the family id to the one corresponding to nl80211 so that the commands are recognized properly. Netlink protocol requires a specific message structure. Since the protocol is generic netlink inside the Netlink construct, we have the following packet:



At the onset when we create a message element we get a netlink packet with a header and no netlink data. In order to fill the netlink data we require generic netlink headers and attributes. Therefore depending on the command and the resulting attribute requirement, we complete the creation of the entire message packet.

After the message is created, we need to build functions that are known as “call backs”. As the name suggests callbacks are the functions which come into picture when we receive netlink packets as responses to our message packets sent over the socket. The details of the functioning of callbacks is mentioned in subsequent sections.

After the creation of message and callbacks we are finally now in a position to send these messages over the netlink socket. Post the sending of request message packet, we wait for the response packet and bind the callback functions onto the socket to act upon the received messages. The callback functions through their return values, indicate to the socket if they should continue receiving further messages or stop. The details of the process in terms of the functions that are used is highlighted below:

Creation of socket: The nl80211_init takes care of this part initializing the created socket with the right nl80211 family.

Creation of message element: nlmsg_alloc() creates a blank netlink message that must be further filled with attributes and command to be able to carry out requisite operations.

Setting up the command in the message: genlmsg_put() which takes in the message, command and the flags as parameters and puts them in the genl message.

Attaching parameters to the message: Broadly there are two kinds of commands, ones that set parameters and ones that get them. Based on these qualifications commands may or may not require attributes to be passed on to the kernel space. For instance the get command to dump all stations does not require any additional attribute. However a single station get command requires the MAC attribute. These are added through the NLA_PUT functions, variations of which are NLA_PUT_U32 etc if the attribute size is known.

Callbacks: Callbacks are the single most important functions while reading the response from the driver in the kernel space. There are different kinds of callbacks namely:

- *Callback for ack responses*: If the response message is an ack response, then we need to carry on receiving further messages and hence direct the function that is receiving netlink messages to continue further. This is done through the ack callback which in our case is the **ack_handler** function.

- *Callback for finish responses*: If the message received is a finish message one needs to stop receiving any further messages. The finish callback does this which in our case is the **finish_handler** function.

- *Callback for error responses*: If we receive an error message, then too we need to skip reading that message and move on to the next message which is taken care by the return value of an error callback function. In our case this is taken care through **error_handler** function.

- *Callback for the general expected response message*: This callback is the custom callback that we design depending on the attributes expected in response message packed from the netlink socket. In the current case, it is the station_dump_handler function. This callback is called when we enter the **nl_recvmsgs() function** and the socket starts reading the message packets incoming from the kernel space.

Thus after creating a callback struct using **cb = nl_cb_alloc(NL_CB_DEFAULT),** one needs to assign different types of callbacks to different functions as mentioned above. Variations of nl_cb_set with different parameters are used to pair these different handlers with the callbacks associated with the main callback struct.

After assigning the callbacks, we are ready to send the messages to the socket which is done through **nl_send_auto_complete()**. In order to obtain the response messages now, we enter

into another function as described later- **nl_recvmsgs() function.** It is in this function that we attach the callback struct which contains the different callback functions to the socket we are listening on. As soon as some response is received the callback struct calls a function appropriate to the message received and begins the parsing process.

Inside the Callback function: While different kind of message packets are being received from the netlink socket, different callback functions are being called. Since we know the basic construction of the received message packet, we can parse it according to our requirement. So there are functions like nlmsg_hdr and nlmsg_data that return pointers to the header of the netlink message and the payload respectively.

```
static int station_dump_handler(struct nl_msg *msg, struct sta_info *arg)
{

        struct nlattr *tb[NL80211_ATTR_MAX + 1];
        struct genlmsghdr *gnlh = nlmsg_data(nlmsg_hdr(msg));
        struct nlattr *sinfo[NL80211_STA_INFO_MAX + 1];
```

Once we have entered the payload of the netlink message we are actually inside the genl message. Hence to parse that we need to strip the header and get to the attribute pointer which is done through genlmsg_attrdata after which nl_parse serially places the attribute values inside the array that is passed to it.

```
nla_parse(tb, NL80211_ATTR_MAX, genlmsg_attrdata(gnlh, 0),
                genlmsg_attrlen(gnlh, 0), NULL);
```

Once we have the array of attribute value (in this case tb) the process of netlink is complete and one is free to use the attribute values in their algorithm and code as required. In our particular case we have stored these attributes in a systematic way in an array of station info structs. In addition we have also computed the number of stations currently attached to a particular WTP which would help in future load management algorithms as valuable data points. The storage of station attributes in an array format helps in not only easy access but also in sending it systematically in packet format under the CAPWAP operation to the AC. The further processing of the same is carried out in the Client side of the WTP that reads from these functions and creates packets to be sent to the AC.

The Main block:

Firstly let us look at the environment the code module is operating in: There is an AP that is sending out beacons continuously (in the test bench this was created on laptop wireless hardware using hostapd). This AP is connected to one or more devices which will now be

referred to as Stations. The stations are using the internet via this AP and the AP is managing all these stations.

Once the configuration is established, we run our code, which depending on whether you want to get all station parameters, or a specific station parameter returns you the values accordingly. There is also a block of code which allows you to disassociate a particular station from the AP (station_delete).

Based on the particular operation one is undertaking, the function initializes the netlink message, its attributes and the callback functions. The callback functions for getting a single station and station dump are the same since we change the request command appropriately through flags so as to get a different response message to both the requests. Since the AP might require to process all the station attributes further, it is essential that it gets the number of stations that are connected information as well. Our function provides that data too.

The information that we seek is stored on the AP at the time of association of the station and is constantly updated. As a result we can retrieve these values if we can communicate with the drivers in the kernel space, which is exactly what we are undertaking through netlink sockets. As was highlighted before this information is to be used in load management and hence the code has created a special structure to organize the data in a single flow, making it easy to send it over to the AC (by copying the struct in a memory block)

As per the requirement, we need to know different station attributes dynamically. Some example station parameters that we obtain:

- inactive time: 452 ms
- rx bytes: 56953
- rx packets: 368
- tx bytes: 80120
- tx packets: 237
- tx retries: 4
- tx failed: 0
- signal: 22.2 dBm
- signal avg: 22.2 dBm
- tx bitrate: 72.2 MBit/s MCS 7 short GI
- rx bitrate: 12.0 MBit/s
- authorized: yes
- authenticated: yes
- preamble: short
- WMM/WME: yes
- MFP: no
- TDLS peer: no

Structures:

- **State maintaining structs:** We require some structs exclusively to store state information, which is used again and again in the form of parameters that are passed to functional calls. Example of one such state maintaining structure is **nl80211_state** that is used in passing the identifiers to every netlink socket call. Essentially the id is set to nl80211 socket type and the socket identifier is initialized using the function **nl80211_init()**

- **Nested Attribute storing structures:** Many attributes of the nl80211 family are nested within other attributes. Hence in order to retrieve them, one requires to parse the message packet received twice. Bit rate related attributes are one such kind of parameters. Hence in order to not crowd the main struct sta_info with too many parameters, a separate struct **bit_rate_info** was created which can now be independently placed in sta_info.

- **Station attribute storing structs:** In order to pass the information to the AC as a packet we require that all information related to a particular station can be packaged in a single struct which can then be used to create packed bits in the CAPWAP protocol message and sent over to AC. Writing independent parameters in the memory and creating the packet could be painful, and hence to simplify the implementation one can copy in a single memcpy operation the entire memory struct into the byte order. This would require that the struct be placed in the correct memory byte order which could be done by programs utilizing this code. The particular struct that stores the station information and is used both as an array of structs and single struct is **sta_info**

Functions:

Critical functions:

- **static int station_dump_handler:** Netlink message requires a callback function. Since it is required to read a lot of station attributes in response to the STATION_GET command sent with the FDUMP flag, we need a custom callback function. station_dump_handler is that callback function. It parses the received netlink message by stripping the header information and getting into the attribute space part of the message. For the attributes which required nested parsing (in our case the Bit rate information) the station_dump_handler() calls the parse_bitrate() function.
  By utilizing the attributes defined in the nl80211.h header file one can retrieve all the parameters for **all** the stations connected to the given AP

- **static int nl80211_init:** In order to perform communication over the netlink socket that connects the user space application with the kernel space function one needs to

create the socket first. This requires setting the appropriate family type of the socket and initializing it. This is performed with the init function. It takes in the state as parameter and fills its socket parameter with the newly created socket over which communication is to take place.

Requirement specific Main functions:

- **int station_dump:** Station dump function uses the GET_STATION command with the dump flag set on. Since no single mac address is provided the kernel space program returns details of all the stations connected with the AP. In order to parse the message one requires to call the station_dump_handler function from this function.

- **int station_get:** As mentioned previously the application may require that details of only a single station be procured. This function explicitly passes the mac address of the station as a parameter in the nl80211 GET_STATION command. However the handler required to parse the received netlink packet is the same as used before which is the station_dump_handler.

- **int station_delete:** In order to sever the connection between an AP and a particular station for functionalities ranging from load balancing to network management, this function provides for the delete operation which again uses the DEL_STATION command along with the mac address of the station that is to be deleted. Since delete function does not have any specific callbacks for it (as no response message is received), the current function reports success if we receive a return value of zero. Any intermediate error would give the error number as defined in the linux error header files.

Auxiliary functions:

- **static int error_handler / static int finish_handler / static int ack_handler:** Netlink socket requires different kind of callback functions for different kind of response messages. So while for the general case we defined the handler function to be station_dump_handler, in cases where the received message is an error/finish/ack message then we require a different kind of callback function to take care of such events. The respective functions handle these specific events and return value to the function receiving the netlink packets to either stop/skip or continue receiving further messages.

- **void parse_bitrate:** As is mentioned in the description previously that there are certain attributes that are nested, with bit rate info being one of them, we need to perform a 2-level parsing for such attributes. Once the primary level attributes are retrieved in the station_dump_handler function, the parse_bitrate function is called in

order to perform secondary level parsing of the NL80211_STA_INFO_TX_BITRATE attribute which yields values defined in the struct bit_rate_info.

- **void mac_addr_n2a / void mac_addr_a2n:** Readability of the mac address for the users is in the xx:xx:xx:xx:xx notation. However in order to provide the mac address into the netlink command packets, a different kind of notation/format is required. The interchange between the usage format and the user display format is handled via these two functions.

Future Scope:

- As I mentioned previously, currently no retrieval mechanism has been implemented for getting the chain level signal strength in a PPDU. Although it is beyond the scope of current build requirements, its future use could be helpful if optimization has to be done within a PUDM as well.

- The memory management for the current case is static in the sense that a fixed number of stations are assumed at the start and the memory is allocated once for all. The dynamic memory management code has memory leak issues for now and hence has not been implemented for now. The idea about dynamic memory would be to provide a framework where in APs with few 10s of stations and those with few 100s of them could both work on a single platform without worrying about memory optimization issues. Each new station reading in the station dump would allocate new memory for itself in order to store the contents of its parameters.

- The interaction between the WTP and Stations has been well captured through the information that is present in the hardware regarding the bitrates/inactive time and other flags, currently there is no way to directly access the packets that are coming in from the stations to the WTP. If we could achieve this, it would be a matter of just forwarding these packets to the AC enabling the ACs direct control over applications being run over the stations. This would help in classifying the kind of services running on the station since the AC would have higher processing power as compared to an individual AP.

- The WTP side would eventually require a mechanism to control its frequency and power parameters as well currently which are not controlled through netlink parameters. Since we are using a hostapd framework to create AP mode on the wireless device, changes in frequency and power require a restart of the entire hostapd system. What needs to be achieved is an independent system, that does not affect the hostapd operations and restarts and resets the power/frequency parameters.

Conclusion:

The following were the key learnings from the entire process

- **Inter Process Communication (IPC):** At the onset, there was no experience in socket programming. However with sockets being at the core of the entire exercise, a large set of literature was surveyed on them. While learning simple code snippets available on the internet connecting a server and a client were examined. Once the familiarity with PIDs, sockets and port terminologies was achieved it was easier to understand the advanced implementations focusing on kernel userspace communication.

- **Driver operations in Linux:** A successful implementation required an understanding of the driver level operations happening at the different kernel modules. Although a function level analysis was not required, a broad insight into the kernel space programs was achieved.

- **Netlink Sockets:** With the ioctl paradigm being deprecated in all modern linux wireless driver interaction utilities, a learning of the netlink sockets was a must. Since the entire family and its usage programs are still under development it was hard to find tutorials and codes. A deep dive into the primary netlink functions from the libnl and lib-genl libraries was undertaken. With the help of the iw utility which was the only simple to understand code block available (hostapd had multiple layers above netlink sockets), a fair understanding of the entire netlink operation was acquired. This would be utilized in furthering many other wireless module related projects since it is the current development paradigm.

- **Object programming:** Prior to the project the experience was mainly in algorithmic programming which required no or very little usages of structs. The use of pointers was also limited to arrays that are easily hidden in c programming. With this project which involved a lot of nested function calls, use of pointers to structs became mandatory. Memory management and custom requirements of the project made it essential to create and use structures. The entire exercise introduced many different c built in functions required in memory allocation and cleaning up. A fair amount of expertise was gained in building structures from scratch, allocating memory space, modifying their variable values, creating arrays of structs by pointer manipulations and eventually cleaning them up.