# Developing Protocol Handlers in Linux

by
**Asanga Udugama**
Communication Networks Group
University of Bremen
Germany
adu@comnets.uni-bremen.de
http://www.comnets.uni-bremen.de/~adu/
July 2007

# 1 Introduction

The Internet Engineering Task Force (IETF) defines a number of protocol standards called Request for Comments (RFC). Its multitude of working groups focusing on different areas propose, discuss and ratify many different RFCs that are extensions to the protocols of the Internet Protocol (IP) suit.

The Linux operating system provides an ideal platform to develop most of these protocols with relative ease. Linux has a number of capabilities that makes it possible to develop many of these protocols solely on the user space. This article looks at some of these user space and kernel level capabilities that can be used to develop protocol handlers. The article initially explains some basic aspects and moves into explaining the different capabilities.

# 2 Protocol Handler Requirements

A protocol handler requires to perform many different tasks depending on what the protocol aims to achieve. Following is a list of some possible tasks that any protocol handler may have to perform. This article elaborates on how these tasks can be done using the different facilities available in Linux. Linux has many alternative ways to achieve similar objectives though each alternative may do things a little differently.

**Getting access to packets that traverse a host** – Protocol handlers may require to know about the packet flow in a host. Sometimes, it would also require altering the flow of packets based on different conditions. In both cases, the protocol handler will require to know about the contents of the packets to make decisions. The contents that need to be investigated can be either in the link header, the IP header, the transport header or even in the payload of the packet. This section looks at some of the facilities in Linux to know and control the packets that traverse a host.

**Manipulating the networking environment** – A protocol handler may require to modify the networking environment to control the packet flow of a host. For example, new routes may have to be added, network interfaces might have to be configured, etc. Additionally, it would also require to know the current state of the networking environment, which might have to be monitored constantly as there would be other processes that may change this environment.

**Messaging between user space and kernel space** – A user space protocol handler may need to receive or send information to the kernel space. The kernel space entity that you

want to communicate with, might be a module that you have developed or an existing module that you have modified.

**Sending and receiving messages** – Some protocol handlers need to message with other instances of the protocol handler that may run on other hosts. This would require to send packets that contain different information. This information might reside anywhere in the packet starting from the IP header to the packet payload.

**Communicating with other user space processes** – A protocol handler might consist of multiple processes that run on the same host. These independent processes will require to communicate with others to exchange information.

**Performing tasks simultaneously** – A protocol handler might have multiple tasks that have to be done concurrently. These tasks might be waiting for different external triggers such as receiving a packet or for internal triggers such as expiration of a timer.

# 3 First things First

There are a couple of initial stuff that you need to know and do when developing system software on the Linux platform.

## 3.1 Kernel Headers

The first thing that you must do is to get the kernel headers installed. The kernel headers are the include files that were used when your current kernel was compiled. These headers are required in addition to the user space includes that get installed with the C compiler. Usually, running the following command will tell you whether you have the kernel headers installed or not (unless specifically installed in a different location).

```
ls -l /lib/modules/$(uname -r)/build
```

The user space headers can be found (usually) in the /usr/include directory. The kernel headers in particular are required if you plan to develop any kernel modules. Kernel module development is explained later.

## 3.2 Important IP Related Headers

One of the important things that a protocol handler does is to work with packets. Packets have to be created and/or contents of packets have to be read. To do this, the protocol handler must know the formats of the different IP packets. This is done using the header files that defines the packet formats. Assuming that a packet was captured using the libpcap library (explained later), and that we know that this is an IPv4 packet, the contents can be checked in the following manner.

```
#include <linux/ip.h>
#include <linux/in.h>
...
struct iphdr *ip_pkt;
ip_pkt=(struct iphdr *) captured_raw_pkt;

if(ip_pkt->protocol == IPPROTO_UDP) {
...
```

The important include files that define the basic packet headers are available in the /usr/include/linux directory. They are,

```
#include <linux/ip.h>
#include <linux/ipv6.h>
#include <linux/icmp.h>
#include <linux/icmpv6.h>
#include <linux/udp.h>
#include <linux/tcp.h>
#include <linux/in.h>
#include <linux/in6.h>
#include <linux/if_ether.h>
```

## *3.3 Kernel Modules*

Sometimes, code in the user space alone will not be sufficient. Even if you don't change any code in the kernel itself, kernel modules would have to be written. A kernel module is a software that runs in the kernel space. It is like any other user space program but it resides in the memory until it is unloaded. They are started and stopped using system commands. Since kernel modules work in the kernel space, it can hang the system if your kernel module has a bug. Here is an example of a simple kernel module that prints a message in the system log (/var/log/messages).

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/version.h>

/
/**
* Kernel module init & install
*/
int init_module()
{
        printk(KERN_INFO "my kernel module loaded\n");

        return 0;
}

/**
* Kernel module removal
*/
void cleanup_module()
{
        printk(KERN_INFO "my kernel module unloaded\n");
}

MODULE_LICENSE("GPL");
```

To compile this module, a Makefile has to be made. The kernel module formats are different for 2.6 and 2.4 based kernels (see accompanying sample code). So the Makefile should reflect this and should also point to where the kernel headers are located. To load and unload modules, simply type,

```
matale:~# insmod ./kmod.ko
matale:~# rmmod kmod
```

A kernel module can print information out like any other user space program. In user space we use printf() but in kernel space we use the function printk(). The definitions related to printk() can be found in the linux/kernel.h include. The output of kernel space programs get printed in the /var/log/messages file. To see the output of the above module, type,

```
matale:~# tail -f /var/log/messages
```

Kernel modules, like any program that you develop can also accept command line parameters. An example is,

```
matale:~# insmod ./pktmon.ko ifc=eth2 timeout=50
```

A difference compared to normal user space programs is that the name of the parameter must be given together with the equal sign for each parameter. To have parameters, 2 statements must be inserted at a minimum for each variable. Here are the definitions for each of the above command line parameters.

```
....

#include <linux/moduleparam.h>

....

// load with insmod ./pktmon.ko ifc=eth5 timeout=1000
// define parameters with default values
static char *ifc = "eth0"; // monitored interface
static int timeout = 6000; // reporting time in millisec

// identification of the parameters
module_param(ifc, charp, 0);
module_param(timeout, int, 0);

/**
* Kernel module init & install
*/
int init_module()
{

        ....

        // use of the parameter
        printk("kmod: parameters %s %d \n", (ifc == 0 ? " " : ifc), timeout);

        ....

}
```

If a user is not aware of what a certain module does, what parameters it takes, etc., this information can be listed my running the modinfo command. Here is an example.

```
matale:~# modinfo ./pktmon.ko
filename:       ./pktmon.ko
license:        GPL
description:    Packet Monitoring Module
author:         Asanga Udugama
vermagic:       2.6.8-2-686 preempt 686 gcc-3.3
depends:
parm:           timeout:timeout period
parm:           ifc:interface name
```

Most of this information must be set by the developer of the module and here is the code related to how some of this information is made available.

```
....
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Packet Monitoring Module");
MODULE_AUTHOR("Asanga Udugama");
MODULE_PARM_DESC(ifc, "interface name");
MODULE_PARM_DESC(timeout, "timeout period");
....
```

## *3.4 Background Processes*

Normally, the protocol handlers get started automatically when a machine boots up. To do this to your protocol handler, you must make the program a daemon. A daemon is a user space program that runs in the background. This is done using the daemon() function in the unistd.h file. A code snippet of a process that is placed in the background follows.

```
int main(int argc, char *argv[])
{
        // check correctness of command line
        // parameters
        if(argc != 3 || strstr(argv[1], “-c”) == NULL) {
                // print error, show usage and exit
                ....
                exit(1);
        }

        // perform initialization work
        read_config_file(argv[2]);

        // create/open and start appending to logs
        start_logging();

        // background process; change to root
        // and close standard in, out & error
        daemon(0, 0);

        // start main thread of the protocol
        // handler
        ....
}
```

Before placing the program in the background, it usually closes the file pointers that are open for standard input, standard output and standard error. Further, it also changes the current directory with which it started to the root (/) directory. The 2 parameters that are

given to this function, can change this behavior. The first parameter when true (i.e. non-zero), will not change the directory to root while the second parameter when true will redirect the above file pointers to the /dev/null device. The /dev/null device in Linux is the black-hole equivalent where nothing is requested from or shown to the user.

The general policy in protocol handlers is to send it to background only after performing all the initialization work. Initialization work such as reading configuration files, opening logs must use the standard in, out and error to inform the user about any problems encountered. Once the logs are open, all errors and warnings can be sent to the logs.

## *3.5 Activity Logging*

A protocol handler performs many activities and it is the usual practice to log all the activities done to a log file so that you can use this information to figure out problems. On of the questions that you must ask yourself is how much of activity logging should my protocol handler perform. Because, logging means that you are writing a lot of information to a file that can get bigger over time and result in other problems.

A general solution adopted by many protocol handlers is to use logging levels. With logging levels the user specifies what volume of information is logged. As an example consider a protocol handler that has 3 logging levels.

Critical errors – Log only critical errors, which are catastrophic in nature. These errors result in the protocol handler simply terminating being unable to proceed any further. An example is,

```
rreqp = (RREQ_MSG *) malloc(sizeof(RREQ));
if(rreqp == NULL) {
        WRITE_LOG(CRITICAL, “Cannot allocate memory for RREQ message”);
        ....
        exit(1);
}
```

Warning errors – These errors are informing the user that something is wrong, but still the protocol handler can continue without the internal state of the protocol handler being invalid. An example is,

```
if((rreqp->flags & 0x20 == 0x20) && (rreqp->flags & 0x10 == 0x10)) {
        WRITE_LOG(WARNING, “Destination Only & Gratuitous flag problem. Flags = %02x”,
                                                                        rreqp->flags);
        ....
        return -INVAILD_RREQ;
}
```

Informative information – These are simply messages about the activities of the protocol handler for the user to know what is going on. An example is,

```
if(*(ptr + sizeof(struct iphdr) + sizeof(struct udphdr)) == RREQ_TYPE) {
        rreqp = (struct RREQ_MSG) (ptr + sizeof(struct iphdr) + sizeof(struct udphdr));
        WRITE_LOG(INFO, “Got a RREQ message from %s”,
                                inet_ntoa((struct in_addr) iphdrp->saddr));
```

```
        process_rreq(rreqp);
}
```

You must decide how the WRITE_LOG() define is called in each place of your protocol handler. The function that performs the logging can decide to log the message passed based on the current level of logging the user has specified. What is usually a practice is where a higher level of logging will automatically include the lower levels. As an example, if the user sets the current level to be INFO, then all messages including CRITICAL and WARNING should be logged. The WRITE_LOG() define may look as follows.

```
#define WRITE_LOG(log_level, fmt...)        \
                                        { \
                                        char buf[MAX_BUF]; \
                                        sprintf(buf, ## fmt); \
                                        write_log(log_level, buf); \
                                        }
```

The write_log() function logs the message with the time.

```
void write_log(int log_level, char *msg)
{
        struct timeval time_now;
        char time_str[24];

        // check whether logging possible
         if(user_log_level >= log_level) {

                // get time & date
                gettimeofday(&time_now, NULL);
                sprintf(time_str, "%08d:%03d - ",
                                time_now.tv_sec,
                                (time_now.tv_usec - (time_now.tv_usec % 1000)) / 1000);

                // log
                fprintf(lfp, time_str);
                fprintf(lfp, msg);
                fprintf(lfp, "\n");
                fflush(lfp);
         }

         return;
}
```

This is one way of logging activities of a protocol handler. This method is a dynamic way in that the user does not need to recompile the protocol handler to set a different logging level. But, simply change it in a configuration file.

Another method adopted regularly is to enable logging through compiler directives. An example is as follows.

```
// user required debug level
#define DEBUG_LEVEL 1

// enable level 1 debugging
#if DEBUG_LEVEL >= 1
#define DBG dbg
#else
#define DBG(...)
#endif

// enable level 2 debugging
#if MD_DEBUG_LEVEL >= 2
```

```
#define MDBG2 dbg
#else
#define MDBG2(...)
#endif
```

In this way, only the debug code related to the user specified level gets enabled and compiled into the executable. There are advantages and disadvantages of both methods.

## *3.6 /proc File System*

The /proc file system is a virtual file system that contains information related to the current configuration and state of the system. This file system is created by the kernel at boot up. Among other information, it contains a heap of information related to the state of the networking environment. An example is the information related to wireless LAN network interfaces connected to the system. This is held in the /proc/net/wireless file. If you list the contents of this file, you will see an output as following.

```
matale:~# cat /proc/net/wireless
Inter-| sta-|   Quality        |   Discarded packets               | Missed | WE
 face | tus | link level noise |  nwid  crypt   frag  retry   misc | beacon | 16
  eth1: 03bf   14.  199.  161.   42968      0      0      0  89011        0
```

This is the current state of the WLAN interfaces. As you see above, these files can be read as text files, for example using FILE type streams defined in the <stdio.h>.

Another set of /proc files contain configuration information. An example is the /proc/sys/net/ipv4/ip_forward which holds whether the system operates as a router or not. A value "0" indicates that it is not a router, i.e. does not perform any packet forwarding while a value of "1" indicates that it will forward any packet destined to other hosts. The files related to configuration are held under the /proc/sys. To see the contents and change them a programmer can use the sysctl system command or simply the echo command. An example of a piece of code that uses sysctl to set the packet forwarding state given by the caller of the function is as follows.

```
int set_forwarding(int state)
{
        char cmd[64];
        int rtn;

        sprintf(cmd, "sysctl -w net.ipv4.ip_forward=%d", state);
        rtn = system(cmd);
        if(rtn < 0)
                return FAILURE;

        return SUCCESS;
}
```

Another alternative is to use the echo command. You could replace the sprintf() statement with the following statement.

```
        sprintf(cmd, "echo %d > /proc/sys/net/ipv4/ip_forward", state);
```

The /proc file system mechanism can also be used by yourself to create you own /proc files

to exchange information between the kernel and the protocol handler. This is explained later. To know about what files are available in the /proc file system and their contents, read proc(5).

## *3.7 Byte Order*

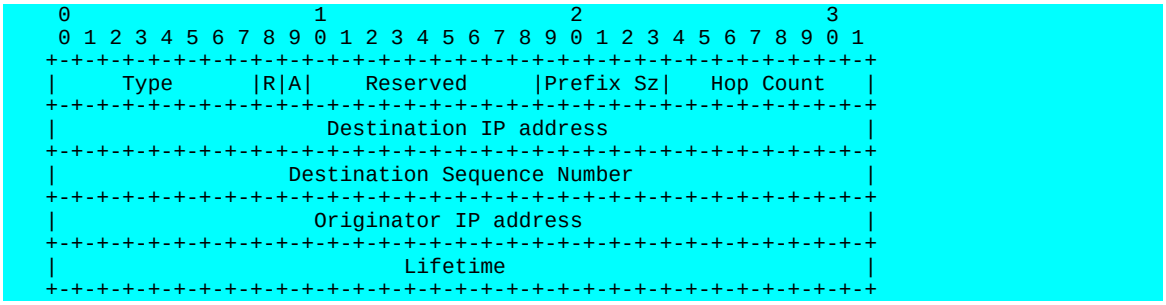
Every operating system stores the bytes of numeric values in a particular order based on the architecture of the processor. As an example, in the Intel 386 architecture, an integer is stored with higher order and lower order bits switched. The following code will show this switch when the data type int is checked to see how it is stored internally.

```
int main(int argc, char *argv[])
{
        unsigned int num;
        unsigned char *num_ptr;

        num = 275;
        num_ptr = (unsigned char *) &num;

        printf("Size of integer %d \n", sizeof(num));
        printf("Integer in hex %08x \n", num);
        printf("Bytes in hex %02x%02x%02x%02x\n",
                              *num_ptr, *(num_ptr + 1),
                              *(num_ptr + 2), *(num_ptr + 3));

}
```

This will given the following output.

```
matale:~# ./byte-conv
Size of integer 4
Integer in hex 00000113
Bytes in hex 13010000
```

But in networking this will be unacceptable since different hosts running different processor architectures have to communicate. This is especially valid for messaging between 2 hosts that use some protocol. The convention and also what is defined in protocol specifications is that the higher part of the number will be represented in the higher order bits and the lower part, in the lower order bits. For example, here is a message of an IETF protocol specification. The field “Destination Sequence Number” has to be set as an unsigned integer in the previously mentioned order.




Since this number has to be processed in the host system in its native order there needs to

be a conversion. To do this we use a set of functions provided in the following include file.

```
#include <netinet/in.h>
```

The functions are htonl(), htons(), ntohl() and ntohs(). These functions simply convert an integer (32 bit) or a short integer (16 bit) from host format to the network format and vise-versa. Here is the above example changed.

```
int main(int argc, char *argv[])
{
        unsigned int num;
        unsigned char *num_ptr;

        num = 275;
        num_ptr = (unsigned char *) &num;

        printf("Bytes in hex (before) %02x%02x%02x%02x\n",
                              *num_ptr, *(num_ptr + 1),
                              *(num_ptr + 2), *(num_ptr + 3));

        num = htonl(num);

        printf("Bytes in hex (after) %02x%02x%02x%02x\n",
                              *num_ptr, *(num_ptr + 1),
                              *(num_ptr + 2), *(num_ptr + 3));

}
```

The output of this code shows how the integer is stored, before and after the conversion.

```
matale:~# ./byte-conv2
Bytes in hex (before) 13010000
Bytes in hex (after) 00000113
```

For more information on these functions, check byteorder(3).

## *3.8 Sleep Duration*

Some protocol handlers may have tasks that need to be performed at different times. One example is the removal of a route entry after a certain period has elapsed. To do this, timers have to be used that wait for a given time and then run some code at the end. The sleep() or the alarm() functions in the <unistd.h> include file are suitable functions to perform this task.

But a drawback of these functions is the limitation of the duration of wait that you can specify. In both functions the wait time is given in seconds and hence the minimum will be 1 second. But protocol handlers may need to perform tasks in lower time durations such as 500 milliseconds. This can be done using the select() function define in the <sys/select.h> include file. Here is an example.

```
struct timeval tv;
int retval, milisec;
```

```
// wait time given in milliseconds
milisec = 500;

/* Convert the milliseconds to the timeval strcture values */
tv.tv_sec = (milisec - (milisec % 1000)) / 1000;
tv.tv_usec = (milisec - (tv.tv_sec * 1000)) * 1000;

// sleep for the given period
retval = select(0, NULL, NULL, NULL, &tv);
```

The select() function is usually used to receive notifications on any input/output activities in a given set of descriptors. Using the timeout variable in this function, you can setup a timer. The timeout value is specified using a struct timeval structure. So a conversion has to be done to set the appropriate values, viz., seconds and micro-seconds. With this, you can setup timers not only for milliseconds, but also for other time units. For more information, check select(2) and gettimeofday(2).

# 4 Getting Hold of Packets

In Linux, there are a couple of facilities that can be used to obtain access to the packets that traverse a host. I explain the following 3 here.

1. NETFILTER modules
2. PCAP library
3. Packet Sockets

## 4.1 NETFILTER Modules

NETFILTER is an interface that you can use to control the packet flow at the network layer of the protocol stack. This is the same interface that is used by iptables/ip6tables to perform NAT, firewalling, etc. It consist of a number of decision points in the path of a packet that traverses the host system. You can write code for each decision point to let the packet pass through or to take some other action. These points and the code that is written are called hooks. There are 5 hooks defined in NETFILTER.

- Pre-routing – A hook for an incoming packet that arrives from the link layer (NF_IP_PRE_ROUTING)
- Local in – A hook that is placed before the local host receives a packet destined to itself (NF_IP_LOCAL_IN)
- Forward – A hook that is placed for a packet that is not to the local host (NF_IP_FORWARD)
- Local out – A hook that is placed after a packet is generated by a local host (NF_IP_LOCAL_OUT)
- Post-routing – A hook that is placed after a packet has consulted the routing table (NF_IP_POST_ROUTING)

As you would derive from this, a single packet can pass through a couple of decision points. For example, a packet that is destined to the local host can be seen at the “Pre-routing” as well as the “Local in” hooks.

The code that is associated with the decision point makes decisions on the progress of the packet. These are identified as verdicts. The decisions that can be made are as follows.

- Drop the packet without letting it go any further (NF_DROP)
- Accept the packet letting it go further (NF_ACCEPT)
- Steal the packet without letting it go any further. Similar to drop, but the stealing code has to look after the release of resources associated with packet (NF_STOLEN)
- Queue the packet to user space for it to be released at a later time (NF_QUEUE)
- Repeat the flow of the packet, thereby the code associated with the hook will be called again (NF_REPEAT)

To use NETFILTER capabilities, a kernel module has to be written that provides code to handle the hooks of interest. Following is the code for a hook that captures packets that originate from the local host (NF_IP_LOCAL_OUT). The module initialization code should register the hook using the NETFILTER structure. The termination code should un-register the hook. The code for module setup and termination is as follows.

```
static struct nf_hook_ops nfho;
int init_module()
{
        nfho.hook     = hook_function;
        nfho.hooknum  = NF_IP_LOCAL_OUT;
        nfho.pf       = PF_INET;
        nfho.priority = NF_IP_PRI_LAST;
        nf_register_hook(&nfho);

        printk(KERN_INFO "my netfilter module loaded\n");

        return 0;
}

void cleanup_module()
{
        nf_unregister_hook(&nfho);

        printk(KERN_INFO "my netfilter module unloaded\n");
}
```

The definitions related to NETFILTER are defined in the following includes that need to be included at a minimum.

```
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/netfilter_ipv6.h>
```

Specific definitions related to IPv6 and IPv4 are defined in 2 separate include files. The hook function given when registering the hook can be as follows.

```
static unsigned int hook_function(unsigned int hooknum,
                          struct sk_buff **skb,
                          const struct net_device *in,
                          const struct net_device *out,
```

```
                            int (*okfn)(struct sk_buff *))
{
        printk(KERN_INFO "got packet num %d on NF_IP_LOCAL_OUT hook \n", ++count);
        return NF_ACCEPT;
}
```

This code will get control every time a packet is received at the associated decision point (e.g. NF_IP_LOCAL_OUT). The code here returns accept to each of the packets. The variables passed to the hook function provides information about the hook and the packet. Following is a description of these variables.

hooknum – The hook number (i.e. NF_IP_LOCAL_OUT, etc.) related to the hook that got executed
skb – A pointer to a pointer of the packet and the details related to the packet
in – The network device information related to the device through which this packet arrived into this host (only if applicable)
out – The network device information related to the device from which this packet will be sent out (only if applicable)
okfn – The OK function that is executed when all the registered hook functions accept the packet (i.e. returns NF_ACCEPT)

Using the skb variable, a programmer can get to the contents of the IP packet to investigate the contents. Assuming the receipt of an IPv4 packet, an example is as follows.

```
if((*skb)->nh.iph != NULL && (*skb)->nh.iph->protocol == IPPROTO_UDP)
{
        printk(KERN_INFO "got UDP packet with destination port %d \n",
                ntohs((*skb)->h.uh->dest));
}
```

## *4.2 PCAP Library*

Another way of getting access to the packets that come to or go out of a host is through the pcap library. pcap is a packet capture facility available in Linux. Unlike NETFILTER, it is a simple, solely user space based, but a more restricted mechanism. The NETFILTER is able to make decisions on the packets it receives. But with pcap, you can only see the packets that is seen by a network interface. pcap is the same capability that is used by tools such as wireshark and ethereal

To use pcap, a programmer must install the pcap development files. This includes header files and the pcap library. The include files that define functions and macros to use pcap functionality are the following

```
#include <pcap.h>
#include <pcap-namedb.h>
```

The pcap library consist of many functions. A simple application that needs to know of the packets that traverse a network interface requires to implement a packet handler. A

packet handler is a function that gets control every time a packet is captured from the network interface. Following is a packet handler that simply prints out the version and the size of the packet size from the IP header.

```
// identifies the positions of the link protocol and the data positions
// depending on the type of captured packet defined in net/bpf.h
int link_types_offset[] = {0, 12, -1, -1, -1, -1, 20, -1, -1, 2, 19, 6, -1 };
int data_start_offset[] = {4, 14, -1, -1, -1, -1, 22, -1, 16, 4, 21, 8,  0 };

void pkt_handler(u_char *passed_var_ptr, const struct pcap_pkthdr *header, const u_char
*pkt_data)
{
	u_short link_proto;
	struct iphdr *ipv4_pkt_ptr;
	struct ipv6hdr *ipv6_pkt_ptr;
	int i;

	// get link protocol
	link_proto = ntohs(*((u_short *) (pkt_data + link_types_offset[link_type])));

	// check the type & print info
	switch(link_proto)
	{
		case ETH_P_IP: // IPv4 packet
			ipv4_pkt_ptr = (struct iphdr *)
					(pkt_data + data_start_offset[link_type]);
			printf("Got IPv%d packet with length %d\n",
				ipv4_pkt_ptr->version, ntohs(ipv4_pkt_ptr->tot_len));
			break;

		case ETH_P_IPV6: // IPv6 packet
			ipv6_pkt_ptr = (struct ipv6hdr *)
					(pkt_data + data_start_offset[link_type]);
			printf("Got IPv%d packet with length %d\n",
				ipv6_pkt_ptr->version, ntohs(ipv6_pkt_ptr->payload_len));
			break;

		default:
			printf("Some other protocol packet\n");
			break;
	}
}
```

The variable pkt_data contains a pointer to the captured packet that also includes the link header. The positions of the link header and the IP packet header depends on the type of link packet (e.g. Ethernet, PPP, etc.). The link_proto variable which is picked from the link header, defines the type of IP packet in this captured packet (i.e. IPv4 or IPv6). The packet capture handler set up code is as follows.

```
pcap_t *cap_hndl;
char pcap_errbuf[PCAP_ERRBUF_SIZE];
int link_type;
u_char passed_var;

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

	// open the live capture
	cap_hndl = pcap_open_live(argv[1], 1000, 1, 20, pcap_errbuf);
	if(cap_hndl == NULL)
	{
		printf("Something is wrong: %s \n", pcap_errbuf);
		exit(1);
	}

	// get the type of device
	link_type = pcap_datalink(cap_hndl);
```

```
        // setup the packet handler
        pkt_cnt = pcap_loop(cap_hndl, -1, pkt_handler, &passed_var);
}
```

The code opens a live capture on the given device and sets up the function to handle each packet.

## *4.3 Packet Sockets*

Another possibility of capturing packets that traverse a host is by using Packet sockets. Packet sockets allow you to capture any packet or a selected set of packets. When opening the socket, you can specify what type of packets you will capture. To open a packet socket, you must call the socket function in the following manner.

```
        sock_id = socket(AF_PACKET, int socket_type, int protocol);
```

The socket_type can be either SOCK_RAW or SOCK_DGRAM. SOCK_RAW indicates that you want to capture raw packets. If SOCK_DGRAM is used, the packets sent will be minus the link layer header. The protocol can be any number as defined in the include file <linux/if_ether.h> that has the ETH_P_ prefix. An example would be.

```
        sock_id = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

In this example, a packet socket is open where all the packets that traverse a host (i.e. incoming and outgoing) are captured and returned with the link layer header.

A point that you should remember is that though this socket captures the packet, it will not disrupt the movement of the packet. This is similar to capturing packets with the pcap library. But unlike in pcap, where any packet that is on the wire is seen, packet sockets only capture packets that are destined to your host or packets that originate from your host.

Here is a code snippet from a program where a packet socket captures raw packets and prints the link layer (i.e. MAC) header contents.

```
// open the socket
sock_id = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
if(sock_id < 0) {
        printf("Error in socket open\n");
        exit(1);
}

while(i++ < MAX_PKT_CAPTURE) {
        // listen for a packet
        rtn = recvfrom(sock_id, pkt_buffer, 2048, 0, (struct sockaddr *) &from, &fromlen);
        if(rtn <= 0) {
                exit(1);
        }

        // show Ethernet header
        ethhdrp = (struct ethhdr *) pkt_buffer;
        printf("pkt details:\n");
        ptr = ethhdrp->h_source;
        printf(" Srce MAC %02x:%02x:%02x:%02x:%02x:%02x \n",
                        ptr[0], ptr[1], ptr[2], ptr[3], ptr[4], ptr[5]);
```

```
        ptr = ethhdrp->h_dest;
        printf(" Dest MAC %02x:%02x:%02x:%02x:%02x:%02x \n",
                        ptr[0], ptr[1], ptr[2], ptr[3], ptr[4], ptr[5]);
        printf(" Proto 0x%04x \n", ntohs(ethhdrp->h_proto));

}
```

In this example, the packet that is captured will have a header that contains the Ethernet related information which is followed by the payload of the packet. The payload protocol type is also specified in this header information. The header is in the struct ethhdr format and is defined in the linux/if_ether.h header file. It has the following variables.

```
struct ethhdr
{
        unsigned char   h_dest[ETH_ALEN];       /* destination eth addr */
        unsigned char   h_source[ETH_ALEN];     /* source ether addr    */
        unsigned short  h_proto;                /* packet type ID field */
} __attribute__((packed));
```

There are more possibilities of capturing different packets using packets sockets. Read packet(7) to get more information about packet sockets. One point to mention is the struct sockaddr_ll structure. This structure has the following format.

```
struct sockaddr_ll {
        unsigned short  sll_family;    /* Always AF_PACKET */
        unsigned short  sll_protocol;  /* Physical layer protocol */
        int             sll_ifindex;   /* Interface number */
        unsigned short  sll_hatype;    /* Header type */
        unsigned char   sll_pkttype;   /* Packet type */
        unsigned char   sll_halen;     /* Length of address */
        unsigned char   sll_addr[8];   /* Physical layer address */
};
```

When this structure is given in the recvfrom() function, it will return additional information related to the packet captured, that might not be available in the packet itself. For example, the sll_ifindex will give the index of the network interface related to this packet.

# 5 ***Manipulating the Networking Environment***

Obtaining information and controlling the routing environment can be done using the following 2 capabilities in Linux.

1. RTNETLINK sockets
2. IOCTL calls

## *5.1 RTNETLINK Sockets*

RTNETLINK sockets are an implementation of NETLINK sockets to retrieve information and modify the networking environment of a host. This is a facility where you send information to the kernel using a socket and the kernel identifies the message sent to act accordingly. There are a number of networking information types that can be

manipulated by RTNETLINK. Here is a list of these.

1. Link level information of network interfaces
2. IP address information of network interfaces
3. Routing information (routing table)
4. Neighbor information
5. Rule based routing information
6. Traffic class information for traffic shaping
7. Queuing discipline information for traffic shaping
8. Filter information for traffic shaping

You are able to add, delete and get information related to each of the above areas. Further, some of them allow you to monitor for changes done by other processes. To use RTNETLINK sockets you must include the following files that contain all the definitions related to RTNETLINK.

```
#include <linux/socket.h>
#include <linux/rtnetlink.h>
```

The general sequence of actions that need to be done in using a RTNETLINK socket is as follows.

1. Open RTNETLINK socket
2. Bind RTNETLINK socket
3. Build message to send
4. Send message over the socket
5. Read message from the socket
6. Process message
7. Close RTNETLINK socket

The socket that is opened has to be a socket with the type SOCK_RAW and protocol NETLINK_ROUTE. Here is an example.

```
// open the NETLINK socket
fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
if(fd < 0) {
        printf("Error in sock open\n");
        usage();
        exit(1);
}
```

Once the socket is opened, it has to be bound to a local address in the following manner. The local address in this case is the process ID of the current process.

```
// setup the local socket & bind
bzero(&local, sizeof(local));
local.nl_family = AF_NETLINK;
local.nl_pad = 0;
local.nl_pid = getpid();
local.nl_groups = 0;
if(bind(fd, (struct sockaddr*) &local, sizeof(local)) < 0) {
        printf("Error in sock bind\n");
        usage();
```

```
        exit(1);
}
```

The other peer of this socket is the kernel space and the following code sets up the peer address.

```
// setup the peer socket
bzero(&peer, sizeof(peer));
peer.nl_family = AF_NETLINK;
peer.nl_pad = 0;
peer.nl_pid = 0;
peer.nl_groups = 0;
```

Setup the message based on the RTNETLINK service required. The following code shows the setting up of a message to add a route entry to the main routing table of the host.

```
// initialize & setup the message IO structs
// sent to sendmsg
bzero(&msg_info, sizeof(msg_info));
msg_info.msg_name = (void *) &peer;
msg_info.msg_namelen = sizeof(peer);

// setup the NETLINK, RTNETLINK messages
bzero(&netlink_req, sizeof(netlink_req));
rtmsg_len = sizeof(struct rtmsg);

// add destnation IP address
rtattr_ptr = (struct rtattr *) netlink_req.buffer;
rtattr_ptr->rta_type = RTA_DST;
rtattr_ptr->rta_len = sizeof(struct rtattr) + 4;
inet_pton(AF_INET, dst_str, ((char *)rtattr_ptr) + sizeof(struct rtattr));
rtmsg_len += rtattr_ptr->rta_len;

// add gateway (nexthop) IP address, if given
if(strlen(gw_str) > 0) {
        rtattr_ptr = (struct rtattr *) (((char *)rtattr_ptr)
                                        + rtattr_ptr->rta_len);
        rtattr_ptr->rta_type = RTA_GATEWAY;
        rtattr_ptr->rta_len = sizeof(struct rtattr) + 4;
        inet_pton(AF_INET, gw_str, ((char *)rtattr_ptr)
                                        + sizeof(struct rtattr));
        rtmsg_len += rtattr_ptr->rta_len;
}

// add interface index
rtattr_ptr = (struct rtattr *) (((char *)rtattr_ptr)
                                    + rtattr_ptr->rta_len);
rtattr_ptr->rta_type = RTA_OIF;
rtattr_ptr->rta_len = sizeof(struct rtattr) + 4;
memcpy(((char *)rtattr_ptr) + sizeof(struct rtattr), &ifc_val, 4);

rtmsg_len += rtattr_ptr->rta_len;

// setup the NETLINK header to idetify the type of
// NETLINK call & the related settings
netlink_req.nlmsg_info.nlmsg_len = NLMSG_LENGTH(rtmsg_len);
netlink_req.nlmsg_info.nlmsg_flags = NLM_F_REQUEST | NLM_F_CREATE;
netlink_req.nlmsg_info.nlmsg_type = RTM_NEWROUTE;

netlink_req.rtmsg_info.rtm_family = AF_INET;
netlink_req.rtmsg_info.rtm_table = RT_TABLE_MAIN;
netlink_req.rtmsg_info.rtm_dst_len = prefix_val;

netlink_req.rtmsg_info.rtm_protocol = RTPROT_STATIC;
netlink_req.rtmsg_info.rtm_scope = RT_SCOPE_UNIVERSE;
netlink_req.rtmsg_info.rtm_type = RTN_UNICAST;

// setup the rest of the IO struct send to sendmsg
```

```
iov_info.iov_base = (void *) &netlink_req.nlmsg_info;
iov_info.iov_len = netlink_req.nlmsg_info.nlmsg_len;
msg_info.msg_iov = &iov_info;
msg_info.msg_iovlen = 1;
```

Since this example related to routing table related functions, the struct rtmsg and a number of struct rtattr are used. The struct nlmsghdr is the NETLINK header that identifies what type of NETLINK message follows. In this case, it is a RTM_NEWROUTE type message. The variables that are used in this code is defined in the following manner.

```
struct {
       struct nlmsghdr       nlmsg_info;
       struct rtmsg          rtmsg_info;
       char                  buffer[2048];
} netlink_req;

int fd;
struct sockaddr_nl local;
struct sockaddr_nl peer;
struct msghdr msg_info;
struct iovec iov_info;

struct rtattr *rtattr_ptr;
int rtmsg_len;
```

Once the message is created, it has to be sent to the kernel using the sendmsg() function. Here is the code of that call.

```
// send the RTNETLINK message
rtn = sendmsg(fd, &msg_info, 0);
if(rtn < 0) {
       printf("Error in sendmsg\n");
       usage();
       exit(1);
}
```

In this example, we are simply sending a RTNETLINK message to modify the routing environment, but do not expect to receive anything in return. But there are RTNETLINK services that need to call the recvmsg() function to get messages. Once the message is received, it has to be processed to obtain the information. An example of the processing of a message received after a request get the routing table would be in the following manner.

```
char *buf; // ptr to RTNETLINK data
int nll; // byte length of all data
struct nlmsghdr *nlp;
struct rtmsg *rtp;
int rtl;
struct rtattr *rtap;
nlp = (struct nlmsghdr *) buf;
for(;NLMSG_OK(nlp, nll); nlp=NLMSG_NEXT(nlp, nll))
{
  // get RTNETLINK message header
  rtp = (struct rtmsg *) NLMSG_DATA(nlp);
  // get start of attributes
  rtap = (struct rtattr *) RTM_RTA(rtp);
  // get length of attributes
  rtl = RTM_PAYLOAD(nlp);
  // loop & get every attribute
  for(;RTA_OK(rtap, rtl); rtap=RTA_NEXT(rtap, rtl))
  {
    // check and process every attribute
```

```
  }
}
```

More information on using RTNETLINK services can be found at netlink(7) and rtnetlink(7). These services are extensively used in the IPROUTE2 program suite through the programs ip and tc. You can look at the code of these programs to see how each of these RTNETLINK services are called.

## *5.2 IOCTL Calls*

Networking related IOCTL calls allow you to retrieve and modify the networking information of a host. This is similar to RTNETLINK sockets, but was there before RTNETLINK and also contains less functionality than RTNETLINK. To use these IOCTLs, you must include the following include files.

```
#include <linux/socket.h>
#include <linux/in.h>
#include <linux/route.h>
```

Using these IOCTL calls, you can manipulate the different types of information related to networking including the following.

1. Network interface information
2. Routing information
3. ARP information

The available services are defined in the <linux/sockios.h> include file. This include file is referenced in the above mentioned include files. The general procedure in calling these services is as follows.

1. Open the socket
2. Set the message to be sent with the IOCTL call
3. Perform the IOCTL call
4. Process the returned message
5. Close the socket

The opening of the socket to perform IOCTL calls are dome in the following manner.

```
// open the IOCTL socket
if((sfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket()");
        usage();
        exit(1);
}
```

Considering an example of adding a route to the routing table, the first thing that must be done is to fill the structure struct rtentry with the required information. Here is the code for setting the struct rtentry.

```
struct rtentry rtent;
struct sockaddr_in  *psaddr;
```

```
// initialize the route entry structure
bzero(&rtent, sizeof(rtent));

// set the destination IP address
psaddr = (struct sockaddr_in *) &rtent.rt_dst;
psaddr->sin_family = AF_INET;
inet_pton(AF_INET, dst_str, &psaddr->sin_addr);

// set the netmask
psaddr = (struct sockaddr_in *) &rtent.rt_genmask;
psaddr->sin_family = AF_INET;
inet_pton(AF_INET, mask_str, &psaddr->sin_addr);

// set gateway IP address, if given
if(strlen(gw_str) > 0) {
       psaddr = (struct sockaddr_in *) &rtent.rt_gateway;
       psaddr->sin_family = AF_INET;
       inet_pton(AF_INET, gw_str, &psaddr->sin_addr);
       rtent.rt_flags = rtent.rt_flags | RTF_GATEWAY;
}

// set the name of interface & other settings
rtent.rt_dev = ifc_str;
rtent.rt_metric = 1;
rtent.rt_flags = rtent.rt_flags | RTF_UP;
```

The IOCTL for adding route entries is called in the following manner.

```
// call the relevent IOCTL
if (ioctl(sfd, SIOCADDRT, &rtent) < 0) {
       perror("ioctl()");
       usage();
       exit(1);
}
```

The SIOCADDRT is the IOCTL request related to adding a route entry. The different IOCTL calls related to networking and the structure that should follow can be found in ioctl_list(2) under section // <include/linux/sockios.h>.

# 6 Messaging between Kernel Space and User Space

There are a number of ways in which user space programs can communicate with the kernel space. Here, I will explain the following methods.

1. NETLINK Sockets
2. Device Files
3. /proc Files

## 6.1 NETLINK Sockets

The socket interface allows the communication between 2 endpoints. NETLINK sockets are an extension of this socket interface to communicate between user space and the kernel space. The 2 endpoints in this case are the user space and the kernel space. RTNETLINK is an implementation of NETLINK socket.

To create your own NETLINK socket, a kernel module has to be developed that performs the tasks that you want in the kernel. But first, you must first define the following 3 types of information.

1 A unique NETLINK socket identifier
2 The message types
3 The structures of the messages

Every NETLINK message must be prefixed by a NETLINK header. The header identifies the message type and optionally attached are your own messages. When opening the NETLINK socket, you must provide the socket identifier. All the NETLINK related definitions and the macros to process these messages are provided in a include file. This include file must be included in the kernel module as well as the user space program.

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

When defining the socket identifier, you must take care not to use a identifier that is already being used by another NETLINK socket. The known identifiers (i.e. used) are defined in the above include file with the NETLINK_ prefix. The known identifies are,

```
...
#define NETLINK_ROUTE           0       /* Routing/device hook                          */
#define NETLINK_SKIP            1       /* Reserved for ENskip                         */
#define NETLINK_USERSOCK        2       /* Reserved for user mode socket protocols     */
#define NETLINK_FIREWALL        3       /* Firewalling hook                            */
#define NETLINK_TCPDIAG         4       /* TCP socket monitoring                       */
#define NETLINK_NFLOG           5       /* netfilter/iptables ULOG */
#define NETLINK_XFRM            6       /* ipsec */
#define NETLINK_ARPD            8
#define NETLINK_ROUTE6          11      /* af_inet6 route comm channel */
#define NETLINK_IP6_FW          13
#define NETLINK_DNRTMSG         14      /* DECnet routing messages */
#define NETLINK_TAPBASE         16      /* 16 to 31 are ethertap */
...
```

Another point to mention is the message types to use. You must start the numbering of message types at least from 16 onwards as some of the the lower numbers are used by NETLINK. RTNETLINK for example, has the following message type definitions.

```
...
#define RTM_BASE        0x10

#define RTM_NEWLINK     (RTM_BASE+0)
#define RTM_DELLINK     (RTM_BASE+1)
#define RTM_GETLINK     (RTM_BASE+2)
#define RTM_SETLINK     (RTM_BASE+3)

#define RTM_NEWADDR     (RTM_BASE+4)
...
```

Now that we know some of basic information related NETLINK sockets let us proceed and develop the kernel module required to handle your own NETLINK socket. The first thing to define is our socket identifier, message types and the data structures.

```
#define NETLINK_TEST                    24
```

```
#define NETLINK_TEST_BASE                    0x20
#define NETLINK_TEST_GETMSG                  (NETLINK_TEST_BASE + 1)
#define NETLINK_TEST_SETMSG                  (NETLINK_TEST_BASE + 2)
#define MSG_SIZE                             256
struct nl_test_msg {
       char msg[MSG_SIZE];
};
```

To init_module() function in the kernel module executed when the module is loaded, firstly opens the NETLINK socket to accept incoming connections.

```
// create the socket to accept incoming user space
// NETLINK_TEST connections
#if (LINUX_VERSION_CODE < KERNEL_VERSION(2,6,14))
       netlink_test_sock = netlink_kernel_create(NETLINK_TEST,
                                          netlink_test_socket_recv);
#else
       netlink_test_sock = netlink_kernel_create(NETLINK_TEST, 2,
                                          netlink_test_socket_recv, THIS_MODULE);
#endif
if(netlink_test_sock == NULL) {
       printk(KERN_ERR "kmod_netlink: could not create netlink socket\n");
       return -1;
}
```

The netlink_kernel_create() must have the socket identifier and the function which is called for incoming messages. There are some changes since kernel 2.6.14 and this is also taken into consideration here. The cleanup_module() function which called when the module is unloaded should close the incoming connection socket.

```
// close socket handling incoming NETLINK_TEST connections
sock_release(netlink_test_sock->sk_socket);
```

To handle the functionality of your own NETLINK socket, the following include files are required at a minimum in your kernel module.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/skbuff.h>
#include <linux/netlink.h>
#include <net/sock.h>
#include <linux/types.h>
```

Every time the user application sends a message to the kernel module, the netlink_test_socket_recv() will be called. This function will be given the socket buffer that hold a queue that holds the messages. You must get each of these messages and process them.

```
static void netlink_test_socket_recv(struct sock *sk, int len)
{
       // do only if the incomming listener socket
       // is open and message queue has at least
       // one message
       do {
              struct sk_buff *skb;
              struct nlmsghdr *nlh;

              // get lock to process multiple requests sequentially
              if(down_trylock(&netlink_lock))
                     return;

              // loop around the message queue
              while((skb = skb_dequeue(&sk->sk_receive_queue)) != NULL) {
```

```
                        nlh = (struct nlmsghdr *) skb->data;

                        // check message type and call approprite function
                        switch(nlh->nlmsg_type) {
                                case NETLINK_TEST_GETMSG:
                                        get_msg(nlh);
                                        break;

                                case NETLINK_TEST_SETMSG:
                                        set_msg(nlh);
                                        break;
                        }

                        // free the message buffer
                        kfree_skb(skb);
                }

                up(&netlink_lock);

        } while (netlink_test_sock && netlink_test_sock->sk_receive_queue.qlen);

        return;
}
```

The netlink_lock mutex handles multiple simultaneous calls in a sequential manner. The get_msg() function sets a message buffer and sends a NETLINK message to the user space. In this example, a simple array is used to send a different message every time a NETLINK request is made.

```
static char msgs[5][MSG_SIZE] = {"Buy", "Sell", "Keep", "Donate", "Destroy"};
static int msg_set_pos = 0, msg_get_pos = 0;

static void get_msg(struct nlmsghdr *recv_nlh_ptr)
{
        struct nl_test_msg *test_msg_ptr;
        size_t size;
        struct sk_buff *skb;
        struct nlmsghdr *send_nlh_ptr;

        // create socket buffer
        size = NLMSG_SPACE(sizeof(*test_msg_ptr));
        skb = alloc_skb(size, GFP_ATOMIC);

        // get pointer to place the message
        send_nlh_ptr = NLMSG_PUT(skb, 0, 0, NLMSG_DONE, size - sizeof(*send_nlh_ptr));
        test_msg_ptr = (struct nl_test_msg *) NLMSG_DATA(send_nlh_ptr);

        // copy message
        memcpy(test_msg_ptr->msg, msgs[msg_get_pos], MSG_SIZE);
        printk("kmod_netlink: sending  - %s\n", test_msg_ptr->msg);

        msg_get_pos = (msg_get_pos >= 4 ? 0 : (msg_get_pos + 1));

        // set socket buffer parameters
        NETLINK_CB(skb).pid = 0;  /* from kernel */
        NETLINK_CB(skb).dst_pid = recv_nlh_ptr->nlmsg_pid;

        // send message to original caller
        netlink_unicast(netlink_test_sock, skb, recv_nlh_ptr->nlmsg_pid, MSG_DONTWAIT);

nlmsg_failure:
        return;
}
```

A socket buffer must be created to send our message back and attach our message using the macros provided by NETLINK. The message that is sent can be either sent to the original caller using the netlink_unicast() function or to multiple users using the netlink_broadcast() function. The set_msg() function gets a pointer to the sent message and updates the

internal array.

```
static void set_msg(struct nlmsghdr *nlh)
{
        struct nl_test_msg *test_msg_ptr;

        // gets the pointer to the message
        test_msg_ptr = (struct nl_test_msg *) NLMSG_DATA(nlh);

        // set the message
        strcpy(msgs[msg_set_pos], test_msg_ptr->msg);
        printk("kmod_netlink: received - %s\n", test_msg_ptr->msg);

        msg_set_pos = (msg_set_pos >= 4 ? 0 : (msg_set_pos + 1));
}
```

The user space program can now call the 2 types of NETLINK requests implemented in the kernel module. To communicate over NETLINK sockets the following include files have to be included at a minimum.

```
#include <linux/socket.h>
#include <linux/types.h>
#include <linux/netlink.h>
```

The NETLINK socket is opened and bound to a local address in the following manner.

```
// open the NETLINK socket
fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_TEST);
if(fd < 0) {
        printf("Error in sock open\n");
        usage();
        exit(1);
}

// setup the local socket & bind
bzero(&local, sizeof(local));
local.nl_family = AF_NETLINK;
local.nl_pad = 0;
local.nl_pid = getpid();
local.nl_groups = 0;
if(bind(fd, (struct sockaddr*) &local, sizeof(local)) < 0) {
        printf("Error in sock bind\n");
        usage();
        exit(1);
}
```

The socket message buffers to call both sendmsg() and recv() have to set in the following manner.

```
// setup the peer socket
bzero(&peer, sizeof(peer));
peer.nl_family = AF_NETLINK;
peer.nl_pad = 0;
peer.nl_pid = 0;
peer.nl_groups = 0;

// initialize & setup the message IO structs
// sent to sendmsg
bzero(&msg_info, sizeof(msg_info));
msg_info.msg_name = (void *) &peer;
msg_info.msg_namelen = sizeof(peer);

bzero(msg_buffer, 2048);
nlmsg_ptr = (struct nlmsghdr *) msg_buffer;
len = NLMSG_LENGTH(sizeof(struct nl_test_msg));
```

```
nlmsg_ptr->nlmsg_len = len;
nlmsg_ptr->nlmsg_pid = getpid();

// setup the rest of the IO struct sent to sendmsg
iov_info.iov_base = (void *) nlmsg_ptr;
iov_info.iov_len = nlmsg_ptr->nlmsg_len;
msg_info.msg_iov = &iov_info;
msg_info.msg_iovlen = 1;
```

To get a message from the NETLINK socket, you must first send a request using the sendmsg() function and then wait for the reply with the recv() function.

```
int get_msg()
{

        // set for a get request
        nlmsg_ptr->nlmsg_flags = NLM_F_REQUEST;
        nlmsg_ptr->nlmsg_type = NETLINK_TEST_GETMSG;

        // send the RTNETLINK message
        rtn = sendmsg(fd, &msg_info, 0);
        if(rtn < 0) {
                printf("Error in sendmsg\n");
                usage();
                exit(1);
        }

        bzero(msg_buffer, 2048);

        // wait for the reply
        rtn = recv(fd, msg_buffer, 2048, 0);
        if(rtn < 0) {
                printf("Error in recv\n");
                usage();
                exit(1);
        }

        // get and show the received message
        nlmsg_ptr = (struct nlmsghdr *) msg_buffer;
        nl_test_msg_ptr = (struct nl_test_msg *) NLMSG_DATA(nlmsg_ptr);
        printf("%s \n", nl_test_msg_ptr->msg);
}
```

To send a message to the kernel module, the buffer is set with the message and set using the sendmsg() function.

```
int set_msg()
{
        // get a pointer to the message buffer
        // and set the message
        nl_test_msg_ptr = (struct nl_test_msg *) NLMSG_DATA(nlmsg_ptr);
        strcpy(nl_test_msg_ptr->msg, msg_str);

        // set parameters for a set request
        nlmsg_ptr->nlmsg_flags = NLM_F_REQUEST | NLM_F_CREATE;
        nlmsg_ptr->nlmsg_type = NETLINK_TEST_SETMSG;

        // send the NETLINK message
        rtn = sendmsg(fd, &msg_info, 0);
        if(rtn < 0) {
                printf("Error in sendmsg\n");
                usage();
                exit(1);
        }
}
```

You can communicate multiple messages using NETLINK sockets in the same request or

as different messages. You could also develop your own macros to process this messages similar to the macros given in RTNETLINK (i.e. linux/rtnetlink.h). To get more information about using NETLINK sockets check netlink(3) and netlink(7).

## *6.2 Device Files*

Another alternative to communicate between the kernel space and the user space is through /dev files. There are many of these files that are being used for different purposes in the Linux kernel. These device files are read from, written to by many processes running in a host. As an example, the /dev/tty1 is the device that handles the display of the fist console of a host. Typing the following command on a shell prompt, will make it appear in the first console.

```
matale:~# echo "Echo to first Console" > /dev/tty1
```

These are different kinds of device types. The console device above is a character device. We too will use character device to communicate between the kernel and the user spaces. Usually all device files are held in the /dev directory. Following is a listing of a directory contents of the /dev/tty1.

```
matale:~# ls -l /dev/tty1
crw-------  1 root root 4, 1 2007-06-24 18:08 /dev/tty1
```

The “c” indicates that it is a character device. Every device has a major number and a minor number. The number 4 is the major device number while the 1 that follows is the minor device number.

Device files require a kernel module to handle the different functions that are allowed on a device file. The following include files are required in the kernel module at a minimum.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
```

The structure struct file_operations is populated and passed to the device registration function to indicate what functions are allowed. The init_module() function of the kernel module in which the device file is registered is as follows.

```
#define MAJOR_NUM                   100
#define DEVICE_NAME                 "/dev/test"

static struct file_operations file_ops;

int init_module()
{
       // set functions implemented
       file_ops.read = device_read;
       file_ops.write = device_write;
       file_ops.ioctl = device_ioctl;
```

```
        file_ops.open = device_open;
        file_ops.release = device_release;
        file_ops.owner = THIS_MODULE;

        // register device with the given name
        rtn = register_chrdev(MAJOR_NUM, DEVICE_NAME, &file_ops);
        if(rtn < 0) {
                printk("kmod-devfile : device registratiom failed\n");
                return -1;
        }

        printk("kmod-devfile : my ioctl kernel module loaded.\n");

        return 0;
}
```

The name of the device here is /dev/test. The file_ops variable initialization indicates that only the following functions are implemented in the kernel module. These functions mean what is allowed for the user space program to call.

1. Open device file
2. Close device file
3. Read from device file
4. Write to device file
5. IOCTL calls

The cleanup_module() where the device is unregistered is as follows.

```
void cleanup_module()
{
        // unregister device
        rtn = unregister_chrdev(MAJOR_NUM, DEVICE_NAME);

        printk("kmod-devfile : my ioctl kernel module unloaded\n");
}
```

The open and close function on the kernel module simply handles the device file usage counts and the “Used by” information in the kernel. “Used by” information is displayed when you run a lsmod command on a shell prompt. The parameters passed are the information related to the file descriptor created on the user space. Here is the code for these 2 functions.

```
static int device_open(struct inode *inode, struct file *file)
{
        if(open_count > 0)
                return -EBUSY;

        // increment the usage count
        open_count++;
        try_module_get(THIS_MODULE);
        printk("kmod-devfile : device opened\n");

        return 0;
}

static int device_release(struct inode *inode, struct file *file)
{
        if(open_count <= 0)
                return -EPERM;

        // decrease the usage count
        open_count--;
```

```
        module_put(THIS_MODULE);
        printk("kmod-devfile : device closed\n");

        return 0;
}
```

The read and write functions are given the file descriptor information and information related to the message buffer. In this example we simply pick a message from an array and passes it to the user space in the case of a read and update the array with the user space message in the case of a write.

```
static char msgs[5][MSG_SIZE] = {"Buy", "Sell", "Keep", "Donate", "Destroy"};
static int msg_set_pos = 0, msg_get_pos = 0;

static ssize_t device_read(struct file *file, char *buf, size_t len, loff_t *offset)
{
        if(open_count <= 0)
                return -EPERM;

        // send a message from the array
        memcpy(buf, msgs[msg_get_pos], MSG_SIZE);
        printk("kmod_netlink: read  - %s\n", buf);
        msg_get_pos = (msg_get_pos >= 4 ? 0 : (msg_get_pos + 1));

        return strlen(buf);
}

static  ssize_t  device_write(struct  file  *file,  const  char  *buf,  size_t  len,  loff_t
*offset)
{
        if(open_count <= 0)
                return -EPERM;

        // update the user sent message
        memcpy(msgs[msg_set_pos], buf, MSG_SIZE);
        printk("kmod_netlink: write - %s\n", buf);
        msg_set_pos = (msg_set_pos >= 4 ? 0 : (msg_set_pos + 1));

        return strlen(buf);
}
```

IOCTL calls are also possible in this device file. To perform different activities, we define request types for IOCTL calls. The request types are codes that get generated by using the macros provided in the asm/ioctl.h include file. In this example, 2 types of requests. A get message and a set message.

```
#define IOCTL_GETMSG                _IOWR(MAJOR_NUM, 0, char *)
#define IOCTL_SETMSG                _IOWR(MAJOR_NUM, 1, char *)
```

The _IOWR indicates that this is a request where messages are passed in both directions (i.e. user space to kernel space and vice versa in the same IOCTL request). Though it is not so in this example, it is used only as a demonstration. To use single direction requests use _IOR for read only from user space and _IOW for writes only from user space. The number indicates the request type. The char * parameter is the type of the passed variable. The function code is as follows.

```
static int device_ioctl(struct inode *inode, struct file *file,
                            unsigned int ioctl_num, unsigned long ioctl_param)
{
        unsigned long rtn2;

        if(open_count <= 0)
```

```
                return -EPERM;

        switch (ioctl_num) {
                case IOCTL_GETMSG:
                        // copy the message to the user space buffer to array
                        rtn2 = copy_to_user((void *) ioctl_param,
                                        (void *) msgs[msg_get_pos], MSG_SIZE);

                        printk("kmod-devfile: get  - %s\n", msgs[msg_get_pos]);
                        msg_get_pos = (msg_get_pos >= 4 ? 0 : (msg_get_pos + 1));
                        break;
                case IOCTL_SETMSG:
                        // get the message from the user space buffer
                        rtn2 = copy_from_user(msgs[msg_set_pos],
                                        (void *) ioctl_param, MSG_SIZE);
                        printk("kmod-devfile: set  - %s\n", msgs[msg_set_pos]);
                        msg_set_pos = (msg_set_pos >= 4 ? 0 : (msg_set_pos + 1));
                        break;
                default:
                        break;
        }
        return 0;
}
```

Unlike in the read and write functions, with IOCTL calls the user space buffer itself is sent to the kernel module. Since there will be memory segment jumps, the 2 macros copy_to_user and copy_from_user are used.

The user space program will call the 5 functions to get and set messages through the file descriptor of the device file that was obtained through the open() function. To use device files, the following include files must be used at a minimum.

```
#include <linux/types.h>
#include <sys/stat.h>
#include <linux/ioctl.h>
#include <unistd.h>
#include <fcntl.h>
```

Before using the /dev/test file used in this example, it has to be created using the mknod() function. This function is given the device file name, the access rights and the major/minor numbers.

```
// create the device file
rtn = mknod(DEVICE_NAME,
                S_IFCHR | S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH,
                ((dev_t) MAJOR_NUM) << 8);
if(!(rtn == 0 || rtn < 0 && errno == EEXIST)) {
        printf("Error in device create (mknod) %d \n", errno);
        usage();
        exit(1);
}
```

The device file is opened and closed in the following manner. Since both reads and writes are done to this device file, the O_RDWR used as the opening mode.

```
// open the device file
fd = open(DEVICE_NAME, O_RDWR);
if(fd < 0) {
        printf("Error in device open\n");
        usage();
        exit(1);
}

...
```

```
// close the device file
close(fd);
```

The user space program can retrieve a message using either the read() or the ioctl() functions.

```
// get a message through the device file
if(mode == READ) {
       rtn = read(fd, msg_buffer, MSG_SIZE);
} else {
       rtn = ioctl(fd, IOCTL_GETMSG, msg_buffer);
}
if(rtn < 0) {
       printf("Error in getting/reading %d\n", rtn);
       usage();
       exit(1);
}
```

The user space program can send a message using either the write() or the ioctl() functions.

```
// send a message through the device file
if(mode == WRITE) {
       rtn = write(fd, msg_buffer, MSG_SIZE);
} else {
       rtn = ioctl(fd, IOCTL_SETMSG, msg_buffer);
}
if(rtn < 0) {
       printf("Error in setting/writing %d\n", rtn);
       usage();
       exit(1);
}
```

## *6.3 /proc Files*

The /proc file system is a virtual file system that is built by the Linux kernel at boot up. There are many files in this file system which are used to exchange information between the kernel space and the user space. In addition to the standard files that are created at boot up, you can also create your own /proc file to send/receive information to/from the kernel.

To have your own /proc file, you must write a kernel module that implements the functions required to exchange information. All the definitions related to using /proc files are held in the <linux/proc_fs.h> include file. At a minimum, the following include files must be used.

```
#include <linux/version.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>
```

The module initialization function must create a /proc file entry with the required name and provide the functions to handle input/output to this file. The module cleanup function has to remove the /proc file before the module is unloaded. Here is the code for init_module()

function..

```
static struct proc_dir_entry *proc_entry;

int init_module()
{
        // create the proc entry
        proc_entry = create_proc_entry("gratuitous_rrep", 0644, proc_net);

        if(proc_entry == NULL) {
                printk("kmod-procfile: create proc entry failed\n");
                return -1;
        }

        // set possible functions on the proc file
        proc_entry->read_proc = procfile_read;
        proc_entry->write_proc = procfile_write;
        proc_entry->owner = THIS_MODULE;

        printk("kmod-procfile: /proc file module loaded \n");

        return 0;
}
```

The functions that will implemented are the read and the write to the /proc file. The name of the file is “gratuitous_rrep”. The function create_proc_entry() creates the file in the /proc file system with the given access rights (0644) and under the /proc/net part of the file system. To create the file under the /proc/net we provide the system initialized variable proc_net which is the proc_dir_entry to the /proc/net directory. You could use the other system initialized variables to locate your /proc file in other places depending on your requirement. If this variable is NULL, then the /proc file will be created in the /proc directory itself.

Here is the code for the cleanup_module() function that removes the “gratuitous_rrep” file that was created.

```
void cleanup_module()
{
        // remove /proc file & unload module
        remove_proc_entry("gratuitous_rrep", proc_net);
        printk("kmod-procfile: /proc file module unloaded\n");
}
```

The read and the write functions to the /proc/net/gratuitous_rrep file is implemented in the following manner.

```
char buffer[MSG_SIZE] = "0";

int procfile_read( char *page, char **start, off_t off,
                   int count, int *eof, void *data )
{
        int len;

        // send the current contents of buffer to user space
        len = sprintf(page, "%s\n", buffer);
        printk("kmod-procfile: read  - %s \n", buffer);
        return len;
}

ssize_t procfile_write( struct file *filp, const char __user *buff,
                        unsigned long len, void *data )
{
        unsigned long rtn;

        // get what user space wrote
```

```
        memset(buffer, 0, MSG_SIZE);
        rtn = copy_from_user(buffer, buff, len);
        printk("kmod-procfile: write - %s \n", buffer);
        return len;
}
```

The read function sends the contents of the buffer and the write will update the buffer with the sent value. The following commands were executed on a shell prompt to check the operations of the /proc/net/gratuitous_rrep file.

```
matale:~# insmod ./kmod-procfile.ko
matale:~# cat /proc/net/gratuitous_rrep
0
matale:~# echo "1" >  /proc/net/gratuitous_rrep
matale:~# cat /proc/net/gratuitous_rrep
1
matale:~# rmmod kmod-procfile
```

# 7 Messaging with other hosts

Certain protocol handlers, as part of its operation require to communicate with other hosts to send/receive control information. The socket API allows us to do this.

## 7.1 INET Sockets

Messaging between protocol handlers may use different types of IP packets with many different message formats. The information that is carried can be in the IP header, in other headers or even in the payload area. There are 3 types of sockets that you can open depending on the requirement.

1. SOCK_STREAM – Connection oriented socket where message packets are assured of being delivered such as a TCP connection
2. SOCK_DGRAM – Connectionless sockets where no acknowledgments are received (i.e. fire and forget type socket) such as a UDP connection
3. SOCK_RAW – A socket for packets where you are responsible for filling in all the values of a packet and this could be a TCP packet, UDP packet, ICMP packet or even some other IP packet which for example may contain multiple headers.

Connection oriented sockets work in a client-server manner where a listener waits for incoming connections (i.e. server) from another host (i.e. client). The general procedure for making connections with SOCK_STREAM sockets are as follows.

```
+-------------------------------------------------------------------------------+
|             Server                        |      Client                       |
+-------------------------------------------+-----------------------------------+
| Open socket (socket())                    | Open socket (socket())            |
| Bind to local port (bind())               | -                                 |
| Set max connection queue size (accept()   | -                                 |
| Wait for connections (accept())           | -                                 |
| -                                         | Make connection (connect())       |
```

```
| Communicate (send()/recv())                | Communicate (recv()/send())                |
| Close socket (close())                     | Close socket (close())                     |
+--------------------------------------------+--------------------------------------------+
```

With connectionless sockets, the general procedure is as follows.

```
+-------------------------------------------------------------------------------------+
|              Peer                          |              Peer                      |
+--------------------------------------------+----------------------------------------+
| Open socket (socket())                     | Open socket (socket())                 |
| Communicate (sendto()/recvfrom())          | Communicate (recvfrom()/sendto())      |
| Close socket (close())                     | Close socket (close())                 |
+--------------------------------------------+----------------------------------------+
```

The include files that are required to be included in your program vary depending on the type of processing you want to perform. Supposing a message containing IPv4 addresses have to be sent to another host, that the messaging is over UDP and that the IP header needs to be changed as well, then the following include files are required at a minimum.

```
#include <sys/socket.h>
#include <linux/ip.h>
#include <linux/udp.h>
#include <linux/in.h>
#include <errno.h>
```

I will use an example where a unicast message is required to be sent to another host to establish a route. The format of the message is as follows.

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |     Type      |R|A|    Reserved     |Prefix Sz|   Hop Count   |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                     Destination IP address                    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                  Destination Sequence Number                  |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Originator IP address                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                           Lifetime                            |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This message will be carried in the UDP payload. To handle this format, the following C structure is defined.

```
// message format of an AODV route reply
struct aodv_rrep
{
       unsigned char type;
       unsigned char flags;
       unsigned char prefix;
       unsigned char hopcount;
       struct in_addr dst_ipaddr;
       unsigned int dst_seqnum;
       struct in_addr orig_ipaddr;
       unsigned int lifetime;
};
```

To send this message, a UDP socket is opened. Since this message requires to also have a specific TTL value in the IP header and UDP port numbers have to be specified, a raw socket is opened. Following is the code for opening the socket and requesting that the program itself will set the contents of the IP header.

```
int flag = 1;

// open socket
sock_id = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
if(sock_id < 0) {
        perror("Socket open error: ");
        exit(1);
}

// indicate that we will set IP header
rtn = setsockopt(sock_id, IPPROTO_IP, IP_HDRINCL, &flag, sizeof(flag));
if(rtn < 0) {
        perror("setsockopt error: ");
        exit(1);
}
```

The setsockopt() function with the IP_HDRINCL indicates that the socket creator will set the contents of the IP header. To send this message to a host, a buffer is used to build the packet with the appropriate values. Here is the function to build the packet.

```
int build_pkt()
{
        memset(pkt_buffer, 0, 2048);

        // set packet pointers
        iph = (struct iphdr *) pkt_buffer;
        udph = (struct udphdr *) (pkt_buffer + sizeof(*iph));
        rrep = (struct aodv_rrep *) (pkt_buffer + sizeof(*iph) + sizeof(*udph));

        // fill the IP header
        iph->version = IPVERSION;
        iph->ihl = 5;
        iph->tos = 0;
        iph->tot_len = sizeof(*iph) + sizeof(*udph) + sizeof(*rrep);
        iph->id = 0;
        iph->frag_off = 0;
        iph->ttl = 1;
        iph->protocol = IPPROTO_UDP;
        iph->check = 0;
        memcpy(&iph->saddr, &src_ipaddr, sizeof(iph->saddr));
        memcpy(&iph->daddr, &dst_ipaddr, sizeof(iph->daddr));

        // fill the UDP header
        udph->source = htons(654);
        udph->dest = htons(654);
        udph->len = sizeof(*udph) + sizeof(*rrep);
        udph->check = 0;

        // fill the UDP payload (AODV message)
        rrep->type = 2;
        rrep->flags = 0;
        rrep->prefix = 24;
        rrep->hopcount = 1;
        memcpy(&rrep->dst_ipaddr, &src_ipaddr, sizeof(rrep->dst_ipaddr));
        rrep->dst_seqnum = 0;
        memcpy(&rrep->orig_ipaddr, &dst_ipaddr, sizeof(rrep->orig_ipaddr));
        rrep->lifetime = htonl(3000);
}
```

Here, using the IP and UDP header formats defined in the <linux/ip.h> and <linux/udp.h> include files, the IP and the UDP headers are filled. The UDP payload carries the message. The checksum fields in the IP as well as the UDP headers are initialized to zero as the socket API will compute them automatically. The built packet is now sent using the sendto() function of the socket API.

```
struct sockaddr_in sin;
```

```
// set sendto sockaddr
sin.sin_family = AF_INET;
sin.sin_port = udph->dest;
memcpy(&sin.sin_addr, &dst_ipaddr, sizeof(sin));

// send AODV route reply packet
rtn = sendto(sock_id, pkt_buffer, iph->tot_len, 0, (struct sockaddr *) &sin, sizeof(sin));
if(rtn < 0) {
        perror("Socket send error: ");
        exit(1);
}
```

To receive a similar packet sent by a different host, the recvfrom() function of the socket API is received.

```
sin_size = sizeof(sin);
rtn = recvfrom(sock_id, pkt_buffer, 2048, 0, (struct sockaddr *) &sin, &sin_size);
if(rtn < 0) {
        perror("Socket receive error: ");
        exit(1);
}

// check whether UDP packet
iph = (struct iphdr *) pkt_buffer;
if(iph->protocol != IPPROTO_UDP)
        continue;

// check whether AODV packet
udph = (struct udphdr *) (pkt_buffer + sizeof(*iph));
if(ntohs(udph->dest) != 654)
        continue;

// print AODV packet contents
rrep = (struct aodv_rrep *) (pkt_buffer + sizeof(*iph) + sizeof(*udph));
if(rrep->type == 2) {
        printf("Got AODV route reply\n");
        break;
}
```

After receiving the packet, a check is made to see if the packet is really the message expected. To get more information about the different sockets and the options that can be set, check socket(7), ip(7), udp(7), tcp(7), raw(7), getsockopt(2) and setsockopt(2).

# 8 Inter-process Communications

Protocol handlers may require to communicate with other processes running in the same host. Here we describe 2 possibilities for doing inter-process communications.

1. Unix sockets
2. INET sockets over loop back

## 8.1 Unix Sockets

A Unix socket is a means of communication for 2 processes that run on the same host. It uses the same socket API used by INET sockets, but here the 2 endpoints are linked through a file rather than the IP addresses and port numbers. There are other ways as

well of linking the endpoints over a Unix socket.

Both endpoints communicate using file specified in the struct sockaddr_un. The format of this structure is as follows.

```
struct sockaddr_un {
       sa_family_t  sun_family;              /* AF_UNIX */
       char         sun_path[UNIX_PATH_MAX]; /* pathname */
};
```

To use Unix sockets, the following include files are required at a minimum.

```
#include <sys/socket.h>
#include <linux/un.h>
```

I use an example here where a connected socket (i.e. SOCK_STREAM) is used to send messages between 2 processes. Since it is a connection oriented socket, one acts as the server accepting connections while the other acts as the client. On both, the socket is opened and the linking address is set in the following manner.

```
#define COMM_ADDRESS  "/tmp/UNIXSOCKET"
struct sockaddr_un ipc_saun, ipc_peer_saun;
....
// open socket
sock_id = socket(PF_UNIX, SOCK_STREAM, 0);
if(sock_id < 0) {
       perror("Socket open error: ");
       exit(1);
}

// setup the address
ipc_saun.sun_family = AF_UNIX;
strcpy(ipc_saun.sun_path, COMM_ADDRESS);
len = sizeof(ipc_saun.sun_family) + strlen(ipc_saun.sun_path);
```

The linking file here is the /tmp/UNIXSOCKET file. The process that is the server, will bind to the address and wait for incoming connections.

```
// remove link, if it is there
unlink(COMM_ADDRESS);

// bind to socket
rtn = bind(sock_id, (struct sockaddr *) &ipc_saun, len);
if(rtn < 0) {
       perror("Socket bind error: ");
       exit(1);
}

// set the max queued connection attempts
rtn = listen(sock_id, 2);
if(rtn < 0) {
       perror("Socket listen error: ");
       exit(1);
}

// wait for incoming connections
addrlen = sizeof(ipc_peer_saun);
peer_sock_id = accept(sock_id, (struct sockaddr *) &ipc_peer_saun, &addrlen);
if(peer_sock_id < 0) {
       perror("Socket accept error: ");
       exit(1);
}
```

Once a client makes a connection, the peer_sock_id can be used to communicate with the client. For example, the following code shows the server waiting to receive message.

```
// wait for an incomming message on the connected socket
bzero(msg_buffer, 2048);
rtn = recv(peer_sock_id, msg_buffer, 2048, 0);
if(rtn < 0) {
        perror("Socket recv error: ");
        exit(1);
}
```

On the client side, a connection has to be established using the connect() function of the socket API before any communications.

```
// connect with the server
if(connect(sock_id, (struct sockaddr *) &ipc_saun, len) < 0) {
        perror("Socket connect error: ");
        exit(1);
}


// send messagae
rtn = send(sock_id, msg_buffer, strlen(msg_buffer), 0);
if(rtn < 0) {
        perror("Socket send error: ");
        exit(1);
}
```

Once the communication is over, the socket can be closed in the following manner.

```
// close socket
close(sock_id);
```

More information regarding Unix sockets can be found at unix(7).

### *8.2 INET Sockets over Loop-back Device*

Another alternative for inter-process communications is the use of INET sockets that communicate over the loop-back network interface. With INET sockets, you specify an IP address with which to communicate. To communicate locally, you have to specify the address of the loop-back device. This is 127.0.0.1 in IPv4 and ::1 in IPv6.

This is a slightly better way to communicate than Unix sockets as a subsequent relocation of one of the processes in a different host will still allow the 2 processes to communicate without any change to the code. This is simply achieved by changing the communicating IP address to the IP address of the new host instead of the loop-back device address.

## *9 Performing Simultaneous Tasks*

Tasks need to be performed concurrently within the same process in many protocol handlers. This section explains 2 of the facilities available.

1 POSIX Threads
2 Signals

## *9.1 POSIX Threads*

Threads are independent flows of execution within the same process that run simultaneously. Using threads, a protocol handler can assign the different tasks that need to be done parelly to different threads. For example, a protocol handler may require to check life times of routes and remove them. At the same time, there may be other tasks such as waiting for control messages to establish these routes. They can all be done using threads. Linux provides a library called pthread which is an implementation of the POSIX thread standard. To use threads, the following includes files have to be included at a minimum.

```
#include <pthread.h>
```

Considering the example of establishment of routes and removal of them when the life time expires, the fist thing that needs to be done is to start the threads to perform these activities. The following code shows the creation and start of these threads.

```
....

// start route adding thread
rtn = pthread_create(&thread1, NULL, add_route, NULL);
if(rtn < 0) {
       perror("Thread creation error: ");
       exit(1);
}

// start route removal thread
rtn = pthread_create(&thread2, NULL, del_route, NULL);
if(rtn < 0) {
       perror("Thread creation error: ");
       exit(1);
}

....
```

The thread1 and thread2 are the handles to the threads which can be used later for performing other operations on the running threads. The third parameter to the create_thread() function is the name of the function that will be executed when the thread is created. Theses are the functions add_route() and del_route().

The function that created the threads will run to the end and possibly terminate the whole program without waiting for the threads to complete its operations. Therefore, the thread function pthread_join() has to be used for that function to wait until the threads do their tasks and return. Following is this code.

```
....
// wait for both threads to return, to exit the program
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
....
```

The pthread_join() function uses the thread handle as the input.

Threads that are created within one process share the same data area. This will be a problem due to different threads modifying common variables without considering what the other threads are doing at that same time. To avoid this problem of simultaneous update to common variables, you should use mutexes which are simply variables that enable mutual exclusion during updates. Said in another way, they are simply locks placed at certain areas of the code that will prevent other threads from entering or updating common variables. The following code shows the definition, initialization, locking and unlocking of a mutex.

```
// create & initialize the mutex
pthread_mutex_t route_lock = PTHREAD_MUTEX_INITIALIZER;

....

// lock before updating common variable
pthread_mutex_lock(&route_lock);

// perform the updates to common variables
....

// unlock after updates
pthread_mutex_unlock(&route_lock);

....
```

The thread function that is given to the pthread_create() function must be in the following format.

```
void * (*start_routine)(void *)
```

The void pointer parameter given as an input parameter to the thread function will carry the 4th parameter given to the pthread_create() function. An example of a thread function is as follows.

```
void *del_route(void *p)
{
	int sleep_time = 0;
	struct timeval current_time;

	// check the route in an endless loop
	while(1) {

		// lock mutex before update
		pthread_mutex_lock(&route_lock);

		// check with the current time & remove
		// route if time expired
		gettimeofday(&current_time, NULL);
		if(rt_ent.route_active) {
			if(rt_ent.expiry_time.tv_sec < current_time.tv_sec) {
				rt_ent.route_active = FALSE;
				printf("de-activating route... lifetime expired \n");
			} else
				printf("route lifetime not expired \n");
		}

		// determine next time to check
		if(rt_ent.route_active)
```

```
                        sleep_time = rt_ent.expiry_time.tv_sec - current_time.tv_sec;

                // determine the default time to wait if no route
                // present
                sleep_time = (sleep_time == 0 ? 1 : sleep_time);

                // unlock mutext after updates
                pthread_mutex_unlock(&route_lock);

                // wait for some time before checking
                sleep(sleep_time);
        }
}
```

When linking code that uses POSIX threads, the pthread library must be specified as an input library in the command line of the link statement in the following manner.

```
gcc -g -lpthread -o route-control route-control.o
```

More information about what facilities and what functions are available in the POSIX thread library and ho they can be used can be found at pthread(7).

## *9.2 Signals*

Another means in Linux to do different tasks independently is through the use of signals and their handlers. Signals are interrupts that can be scheduled to occur for different events. A handler (i.e. some code) is associated with the signal which will be executed when the event triggers. The signal handling facility is much limited that threads. But it can perform certain tasks that threads are unable to do. To use signals, the following file must be included at a minimum.

```
#include <signal.h>
```

To install a signal, signal() function together with with the type of signal you want to trap and the handler function. Here is an example where the pressing of the Ctrl and C are trapped to terminate a running program in a graceful manner without an abrupt ending.

```
// install the signal handler for Ctrl+C
signal(SIGINT, term_handler);
```

The term_handler() function can simply do a cleanup and exit the program. Here is an example code for the SIGINT signal handler.

```
void term_handler(int signal)
{
        // prepare for a graceful exit
        ....

        exit(1);
}
```

The signal variable that is a parameter to this function will carry the number of the signal that was triggered. In this case, it will be SIGINT.

Another type of signal that is useful in developing protocol handlers is the SIGALRM type signal. This is a timer signal where when a certain time period elapses, a signal handler is executed. To user SIGALRM type handlers, you must also include the following file.

```
#include <unistd.h>
```

With the SIGALRM type signal, an alarm must also me activated for the timer event to be executed. Following is an example of setting up this timer and the signal.

```
// install a timer handler
signal(SIGALRM, alarm_handler);
alarm(10);
```

Here, an alarm is set to trigger at 10 seconds and when it is triggered, the alarm_handler() function id executed. The alarm_handler() function has the same prototype as the term_handler() function described above.

There are many types of signals that you can trap and perform different actions. More information can be obtained at signal(2), signal(7) and alarm(2).

## *10 Conclusion*

This article looked at some of the activities that a protocol handler may need to perform and how they can be done using the facilities available in the Linux environment. The list of possible activities that were identified is not an exhaustive list but a probable list that was compiled from my experience of developing protocol handlers. Though the Linux environment provides a very rich set of facilities, only a selected set that I thought was appropriate was explained.

The sample code used in his article is available as `developing-protocol-handlers.tgz` to be downloaded.

## *11 Acknowledgment*

I wish to sincerely thank Professor Carmelita Goerg, Andreas Koensgen, Niko Fikouras and the myriad of Linux networking related code developers from whom I have learned immensely through their code.