CSE536 Project 2 – Build a communication channel that uses normal read/write system calls to communicate over IP.

Goal:- Mission is to replace (or at least bury) the standard socket interface used by most network programs and allow cse5361app to communicate in full duplex with another cse5361app at an arbitrary network end location via ipv4 using standard read/write system calls. Ipv4 is not to be altered. A write on system A should fill a read buffer on system B and visa-versa. The calls can be blocking or non-blocking.

Implementation:-

1. **Registration and deregistration of new IP Protocol**
   a. As the goal suggest we have to register new protocol with the IPv4. This can be achieved using following method
      *int inet_add_protocol(const struct net_protocol *prot, unsigned char num);*
   The second argument which is protocol number should be unused by existing protocol in IPv4 protocol array. [This was hash table, called as 'inet_protos' till linux 2.4.]
   In this project we use '234' as our protocol number.
   b. Similarly, after unloading the module we should deregister the protocol with following method
      *int inet_del_protocol(const struct net_protocol *prot, unsigned char num);*

2. **net_protocol datastructure** [1]
   Protocols that rest on IPv4 are defined by net_protocol data structures, defined in *include/net/protocol.h*, which consist of the following three fields:
   *int (*handler)(struct sk_buff *skb)*
   Function registered by the protocol as the handler for incoming packets. It is possible to have protocols that share the same handler for both IPv4 and IPv6 (e.g., SCTP).
   *void (*err_handler)(struct sk_buff *skb, u32 info)*
   Function used by the ICMP protocol handler to inform the L4 protocol about the reception of an ICMP UNREACHABLE message.
   int no_policy
   This field is consulted at certain key points in the network stack and is used to exempt protocols from IPsec policy checks: 1 means that there is no need to check the IPsec policies for the protocol.

```
static const struct net_protocol cse536_protocol = {
        .handler     = cse536_rcv,
        .err_handler = cse536_err,
        .no_policy   = 1,
        .netns_ok    = 1,
};
```

3. **file_operations** [2]
    a. An open device is identified internally by a *file structure*, and the kernel uses the *file_operations* structure to access the driver's functions.
    b. The *file_operations* structure, defined in <*linux/fs.h*>, is an array of function pointers.
    c. Each field in the structure must point to the function in the driver that implements a specific operation, or be left *NULL* for unsupported operations.
    d. You can implements only the most important device methods, and uses the tagged format to declare its *file_operations* structure.

```
struct file_operations cse536_fops = {
        owner: THIS_MODULE,
        read: cse536_read,
        write: cse536_write,
        unlocked_ioctl: cse536_ioctl,
        open: cse536_open,
        release: cse536_release,
};
```

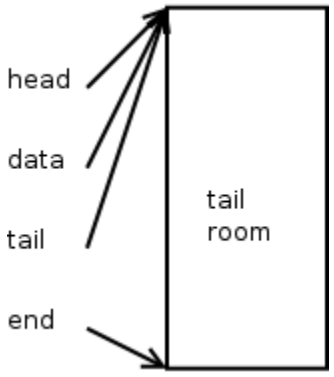
4. **Read method**

    Read is simpler as compared to write. It copies the data received into the buffer to display on the terminal by the application ‘cse536app’.

5. **Send method**

    In the write method, we pass the data to IPv4 through sk_buff structure.
    We first create and setup sk_buff [3]

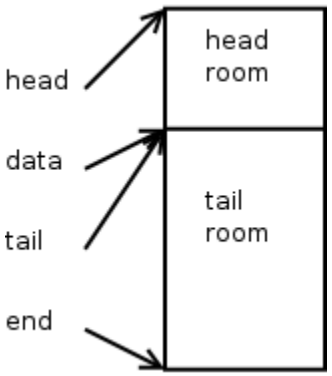
```
struct sk_buff *alloc_skb(unsigned int size,gfp_t priority)
```



This is what a new SKB looks like right after you allocate it using alloc_skb(). As you can see, the head, data, and tail pointers all point to the beginning of the data buffer. And the end pointer points to the end of it. Note that all of the data area is considered tail room. The length of this SKB is zero, since it doesn't contain any packet data at all.

Let's reserve some space for protocol headers using skb_reserve()

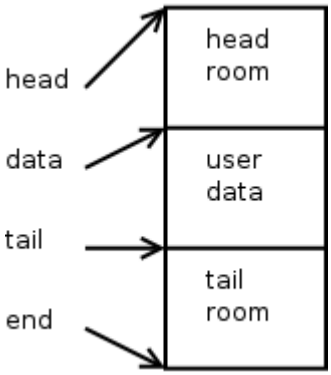
*void skb_reserve(struct sk_buff *skb, int len);*



This is what a new SKB looks like right after the skb_reserve() call.

Let's now add some user data to the packet.

skbdata = skb_put(skb, len);
skb->csum = csum_and_copy_from_user(data, skbdata, len, 0, &err);



This is what a new SKB looks like right after the user data is added. skb_put() advances 'skb->tail' by the specified number of bytes, it also increments 'skb->len' by that number of bytes as well. This routine must not be called on a SKB that has any paged data. You must also be sure that there is enough tail room in the SKB for the amount of bytes you are trying to put. Both of these conditions are checked for by skb_put() and an assertion failure will trigger if either rule is violated. The computed checksum is remembered in 'skb->csum'.

Now, it's time to build the protocol headers.

```
skb_push(skb, sizeof(struct iphdr));
skb_reset_network_header(skb);
iph = ip_hdr(skb);
iph->version  = 4;
iph->ihl      = 5;
iph->tos      = 0;
iph->frag_off = 0;
iph->ttl      = 64;
iph->daddr    = cse536_daddr;
iph->saddr    = cse536_saddr;
iph->protocol = IPPROTO_CSE536;          // my protocol number
iph->id       = htons(1);
iph->tot_len  = htons(skb->len);
```

Now, send the packet to IPv4 using

```
int ip_local_out(struct sk_buff *skb);
```

6. **Write method**
   In a 257 byte data, if first byte is 1 then it is an IP address and we assign it to 'cse536_daddr' else it is data and forwarded to send method for forwarding over network.

7. **Receive method**
   Here we add the data packet to linked list buffer.

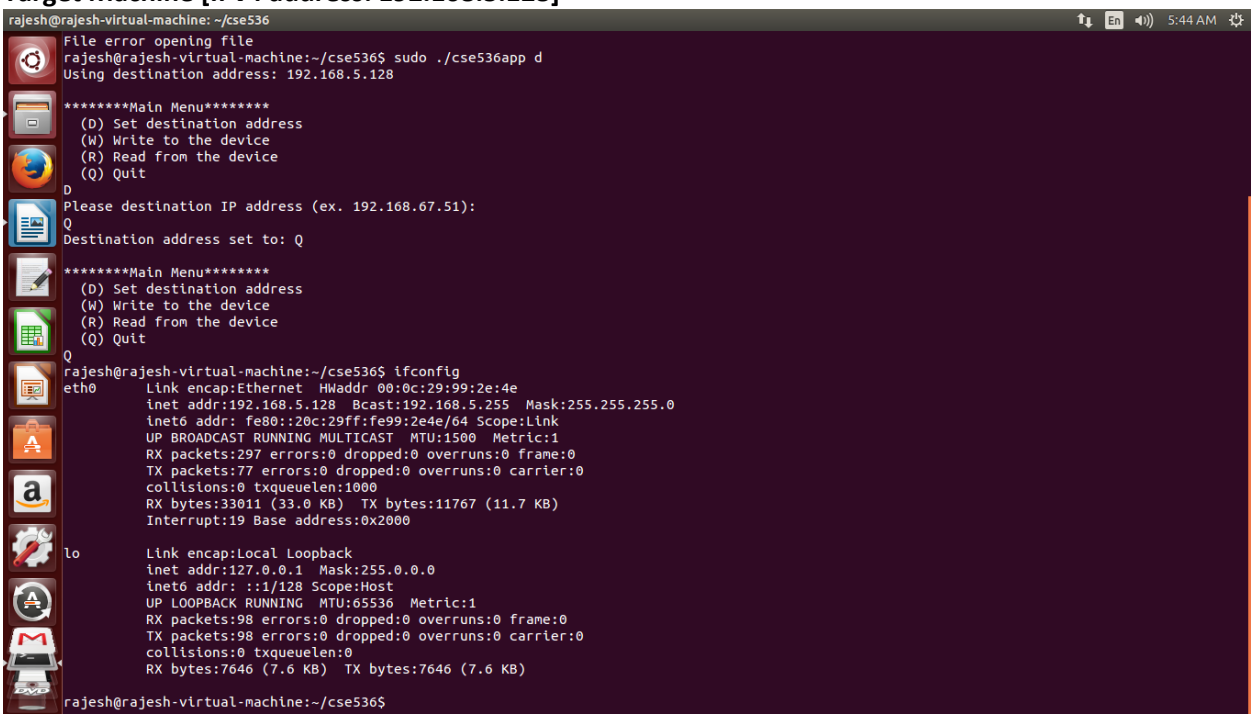
8. **Local Address Retrieval method**
   a. In this method first we obtain the Ethernet device using
      *struct net_device * dev_get_by_name (const char * name);*
   b. Now we cast net_device to in_device structure using
      *struct in_device *in_dev_get(const struct net_device *dev);*
      we need to do this because for_primary_ifa take in_device structure and not net_device
   c. Now, we take ip address and assign it to cse536_saddr

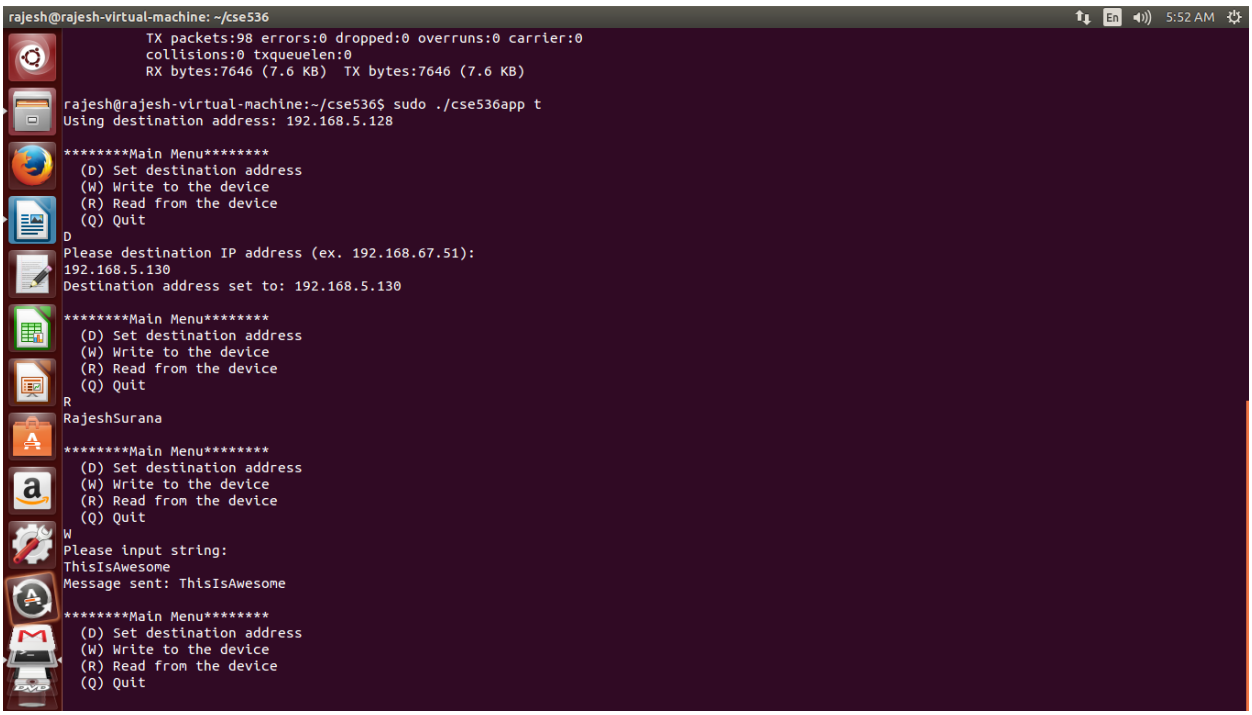
```
for_primary_ifa(ineth0){
        cse536_saddr = ifa->ifa_address;
} endfor_ifa(ineth0);
```

   For_primary_ifa is a macro that can be used to browse all of the in_ifaddr instances associated with a given in_device structure. for_primary_ifa considers only primary addresses though.[4]

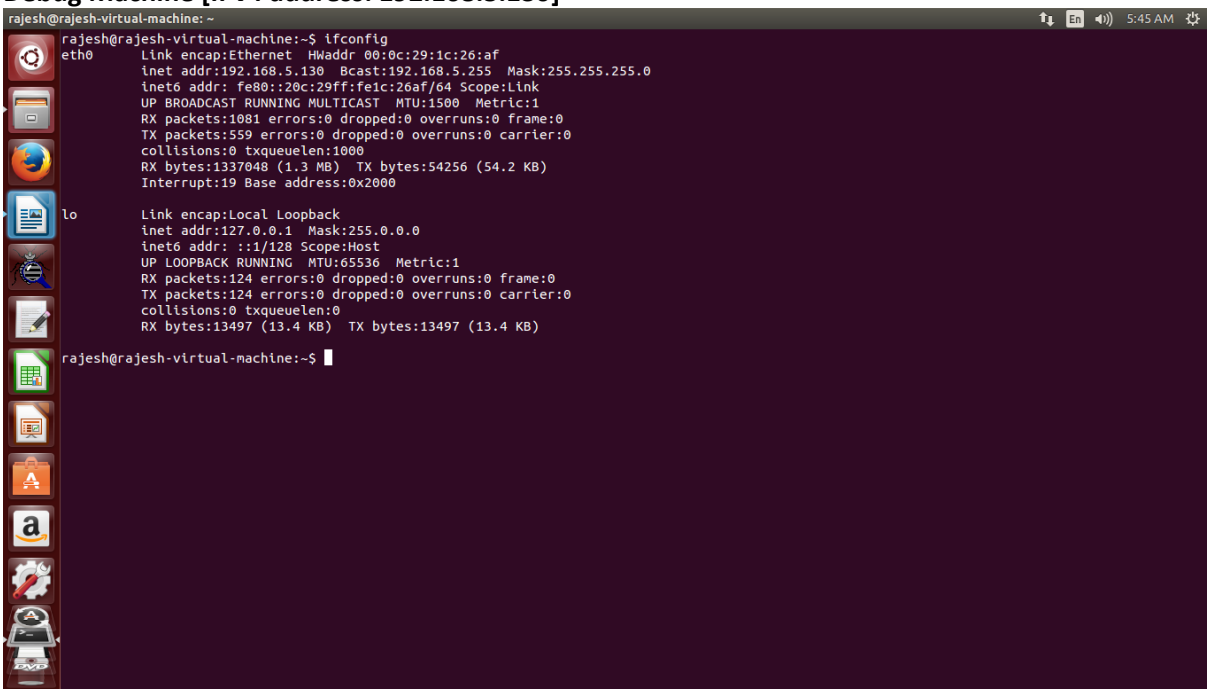
Screen Shots of application using new protocol:-

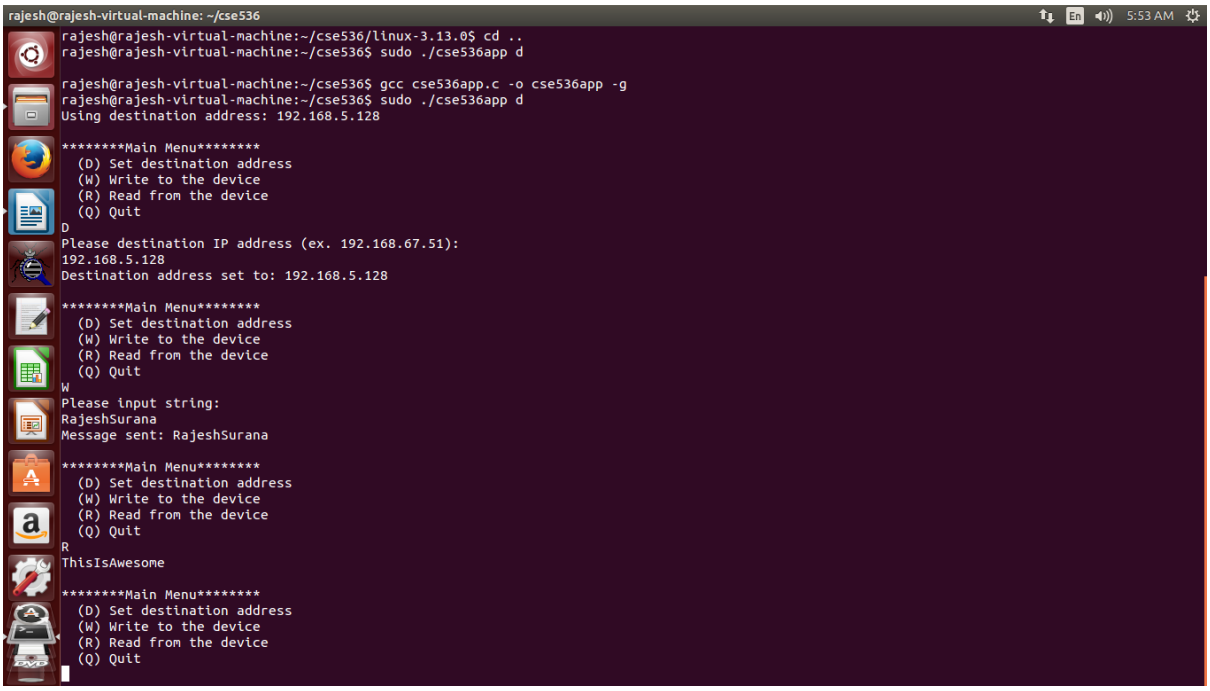
Target Machine [IPv4 address: 192.168.5.128]





Debug Machine [IPv4 address: 192.168.5.130]





References:-

[1] http://www.embeddedlinux.org.cn/linux_net/0596002556/understandlni-CHP-24-SECT-2.html
[2] Jonathan Corbet, Alessandro Rubini, “Linux Device Drivers”, O'Reilly Media, 2nd edition, June 2001
[3] http://vger.kernel.org/~davem/skb_data.html
[4] Christian Benvenut, “Understanding Linux Network Internals”, December 2005: First Edition