**Project 2**

**Build a communication channel that uses normal read/write system calls to communicate over IP**

**Course** : CSE536

**Faculty** : Prof. Alan Skousen

**Name** : Ganesh Ramkrushna Zilpe

**ASUID** : 1207578251

**Index**

[Goal: 3](#_Toc403510051)

[System Description: 3](#_Toc403510052)

[Project Description: 3](#_Toc403510053)

[TCP/IP Protocol Architecture Model 4](#_Toc403510054)

[Physical Network Layer 4](#_Toc403510055)

[Data-Link Layer 4](#_Toc403510056)

[Internet Layer 4](#_Toc403510057)

[Application Layer 4](#_Toc403510058)

[Project Implementation Steps 5](#_Toc403510059)

[**1.** **Register protocol:** 5](#_Toc403510060)

[**2.** **Writing the message** 5](#_Toc403510061)

[**3.** **Send the message** 6](#_Toc403510062)

[**4.** **Receive the message** 7](#_Toc403510063)

[**5.** **Read the message** 8](#_Toc403510064)

[**6.** **Problem faced during registering protocol** 9](#_Toc403510065)

[Project Results 10](#_Toc403510066)

[1. IP addresses of both virtual machines: 10](#_Toc403510067)

[2. Make node for character device and compile the kernel module of character device. 11](#_Toc403510068)

[3. Inserting the module to the device. 11](#_Toc403510069)

[4. Send the .ko of compiled kernel module and object file of application program to other virtual machine as we did in project 1C. 12](#_Toc403510070)

[5. Run application on both virtual machines. Set the destination address in virtual machine application. IP address of VM2 is 192.168.0.37. 12](#_Toc403510071)

[6. Write the message at VM1. 13](#_Toc403510072)

[7. Read the sent message at VM2. 13](#_Toc403510073)

[8. Write message at VM2. For this first set the destination address of VM1 at application running at VM2. 14](#_Toc403510074)

[9. Read the sent message at VM1 14](#_Toc403510075)

[References: 16](#_Toc403510076)

# Goal:

Our goal is to build a communication channel that uses normal read/write system calls to communicate over IP.

# System Description:

**VMware Player Information:** VMware Player 6.0.3

**Guest Operating System:** Ubuntu 14.04.1 LTS

**Host Operating System:** Windows 7 32-bit (Hardware support 32-bit OS)

**Project Description:**

In theory and in practice any Linux kernel module can interface with and use any other Linux kernel module as long as the interfacing module is designed to meet all of the existing module's requirements. In this case our cse5361 module will be modified to communicate with cse5361 modules on other computers by interfacing with standard IPv4. Internet Protocol (IP) normally operates in the standard internet network stack and is able to interface with a variety of transport layer modules as well as link layer modules. Typical transport layer modules include TCP and UDP. A typical link layer module would be Ethernet. In the current world of network technology IPv4 is the internet glue and the lower layer modules interface IPv4 with local hardware while the upper layer modules provide application programs access to IPv4.

Our 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 vice-versa. The calls can be blocking or non-blocking.

# TCP/IP Protocol Architecture Model

## Physical Network Layer

The physical layer of model governs the characteristics of the hardware to be used for the network. For example, physical network layer specifies the physical characteristics of the communications media. The physical layer of TCP/IP describes hardware standards such as IEEE 802.3, the specification for Ethernet network media, and RS-232, the specification for standard pin connectors.

## Data-Link Layer

The data-link layer identifies the network protocol type of the packet, in this instance TCP/IP. The data-link layer also provides error control and “framing.” Examples of data-link layer protocols are Ethernet IEEE 802.2 framing and Point-to-Point Protocol (PPP) framing.

## Internet Layer

The Internet layer, also known as the **network layer** or **IP layer**, accepts and delivers packets for the network. This layer includes the powerful Internet Protocol (IP), the Address Resolution Protocol (ARP), and the Internet Control Message Protocol (ICMP).

**IP Protocol**

The IP protocol and its associated routing protocols are possibly the most significant of the entire TCP/IP suite. IP is responsible for the IP addressing, Host-to-host communications, Packet formatting, Fragmentation.

#### Transport Layer

The TCP/IP transport layer ensures that packets arrive in sequence and without error, by swapping acknowledgments of data reception, and retransmitting lost packets. This type of communication is known as end-to-end. Transport layer protocols at this level are Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and Stream Control Transmission Protocol (SCTP). TCP and SCTP provide reliable, end-to-end service. UDP provides unreliable datagram service.

## Application Layer

The application layer defines standard Internet services and network applications that anyone can use. These services work with the transport layer to send and receive data. Many application layer protocols exist.

Table : TCP/IP Protocol Stack

|  |  |  |  |
| --- | --- | --- | --- |
| **OSI Ref. Layer No.** | **OSI Layer Equivalent** | **TCP/IP Layer** | **TCP/IP Protocol Examples** |
| 5,6,7 | Application, session, presentation | Application | NFS, NIS, DNS, LDAP, telnet, ftp, rlogin, rsh, rcp, RIP, RDISC, SNMP, and others |
| 4 | Transport | Transport | TCP, UDP, SCTP |
| 3 | Network | Internet | IPv4, IPv6, ARP, ICMP |
| 2 | Data link | Data link | PPP, IEEE 802.2 |
| 1 | Physical | Physical network | Ethernet (IEEE 802.3), Token Ring, RS-232, FDDI, and others |

# Project Implementation Steps

We are going to use existing character device i.e. cse5361. But, we need to add functionalities for sending message and receiving messages.

For networking, we need to use some functionality. For that, cse5361 has included following headers:

#include <linux/inet.h>

#include <linux/netdevice.h>

#include <linux/inetdevice.h>

#include <net/protocol.h>

#include <net/sock.h>

#include <net/ip.h>

1. **Register protocol:**

As mentioned in the project scenario, standard protocol is not going to be used. Hence, we need to register new protocol number to networking model. Be careful, choosing proper protocol number is very important, because we cannot use existing protocol number. All protocol numbers are given in **in.h** file.

For our protocol, we use protocol number = 234.

static int cse536\_add\_protocol(void)

{

/\* Register protocol with inet layer. \*/

if (inet\_add\_protocol(&cse536\_protocol, IPPROTO\_CSE536) < 0)

return -EAGAIN;

return 0;

}

Above method is going to add protocol number = 234 into networking model. For that, we use library function:

inet\_add\_protocol():

This function adds a protocol of inet family. It takes the protocol number as the only argument which defines the protocol to be added. It is found in **net/ipv4/protocol.c.**

1. **Writing the message**

We are using character device for communication between two virtual machines. For sending the message, we need to write the message to the character device first. I have modified the existing write() method of cse5361 device.

static ssize\_t cse536\_write(struct file \*file, const char \*buf,

size\_t count, loff\_t \* ppos)

{

char data[256], address[16];

if ( buf[0] == 1 )

{ // data contains ip address

// address starts at byte 2

memcpy(address, buf+1, 16);

printk("cse536\_write - setting address: %s\n", address);

cse536\_daddr = in\_aton(address);

}

else

{

// initialize data memory to zeros

memset(data, 0, 256);

// copy write buffer to data array, data starts at 2nd byte in buffer

memcpy(data, buf+1, count);

printk("cse536\_write - sending message: %s\n", data);

cse536\_sendmsg(data, count);

}

printk("cse536\_write1: %s\n", buf);

return -1;

}

In this code, when buf[0] = 1 means the data written on command prompt is destination address. If buf[0] = 2 or more then the data is actual message string need to be sent over the network. Before sending the data, the destination address needs to be entered. Otherwise, there will be no destination address, so message won’t be sent.

1. **Send the message**

Now we have destination address and also data to be sent. Now, we can send the message.

// this method will send the message to the destination machine using ipv4

static int cse536\_sendmsg(char \*data, size\_t len)

{

struct sk\_buff \*skb;

struct iphdr \*iph;

struct rtable \*rt;

struct net \*net = &init\_net;

unsigned char \*skbdata;

// create and setup an sk\_buff

skb = alloc\_skb(sizeof(struct iphdr) + 4096, GFP\_ATOMIC);

skb\_reserve(skb, sizeof(struct iphdr) + 1500);

skbdata = skb\_put(skb, len);

memcpy(skbdata, data, len);

// setup and add the ip header

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;** // new protocol number

iph->id = htons(1);

iph->tot\_len = htons(skb->len);

// get the route. this seems to be necessary, does not work without

rt = ip\_route\_output(net, cse536\_daddr, cse536\_saddr, 0,0);

skb\_dst\_set(skb, &rt->dst);

return ip\_local\_out(skb);

}

For this, we take help of socket buffer. Though we are not using socket for communication, it is the basic data structure which is used while sending packets over network. In the bold work, we can see that iph->version = 4; => this means we are using ipv4 and iph->protocol = IPPROTO\_CSE536; => we are using 234 protocol for sending message.

**sk\_buff:** All network-related queues and buffers in the kernel use a common data structure, struct sk\_buff. This is a large struct containing all the control information required for the packet (datagram, cell, whatever). The sk\_buff elements are organized as a doubly linked list, in such a way that it is very efficient to move a sk\_buff element from the beginning/end of a list to the beginning/end of another list. A queue is defined by struct sk\_buff\_head, which includes a head and a tail pointer to sk\_buff elements.

1. **Receive the message**

As the message is sent over the network with the destination address, it may arrive at the destination. If the message is arrived at the destination, then the message data needs to be stored at the buffer of character driver.

int cse536\_rcv(struct sk\_buff \*skb)

{

// setup a linked list item to add the data to the linked list buffer

struct cse536buffer \*item = kmalloc(sizeof(struct cse536buffer), GFP\_ATOMIC);

memset(item->data, 0, 256); // intialize data in buffer item

memcpy(item->data, skb->data, skb->len); // copy data to buffer item

// add to the buffer linked list

if ( cse536buffhead == 0 )

{

cse536buffhead = item;

}

if ( cse536bufftail != 0 )

{

cse536bufftail->next = item;

}

cse536bufftail = item;

item->next = 0;

printk("Receive handler called. Received: %d bytes: %s\n", skb->len, item->data);

return 0;

}

1. **Read the message**

If the message is arrived at the destination, then the message data gets stored at the buffer of character driver. We can read the data from the character driver.

static ssize\_t cse536\_read(struct file \*file, char \*buf, size\_t count,loff\_t \*ptr)

{

struct cse536buffer \*next;

ssize\_t retCount = count;

if (count > 256)

retCount = 256; // data buffer sizes standardized at 256, make sure not trying to read more

if ( cse536buffhead != 0 )

{

// copy data from linked list head to read buffer

memcpy(buf, cse536buffhead->data, retCount);

// remove the head from the list and point to the next item

next = cse536buffhead->next;

kfree(cse536buffhead);

// manage the link list pointers

if ( next == 0 )

{

cse536buffhead = 0;

cse536bufftail = 0;

}

else

{

cse536buffhead = next;

}

}

else

{

// nothing in buffer, return 0s

memset(buf, 0, retCount);

}

//retCount = sprintf(buf, "cse536");

printk("cse536\_read: returning %d bytes\n", retCount);

return retCount;

}

1. **Problem faced during registering protocol**

All by-default protocols are registered in the namespace. But, as we are added new protocol and we are not going to create separate file for its functionality because we are not following network layer of TCP/IP model.

Therefore, when I compile mu kernel module, I got following error:

Error: Could not insert module driver/char/cse536/cse5361.ko : resource temporarilly unavailable

Hence, we need to add protocol as non-namespace aware. For this, I have found a patch to add non-namespace aware protocol. Its modification in /net/ipv4/protocol.c.

int inet\_add\_protocol(const struct net\_protocol \*prot, unsigned char protocol)

{

+#ifdef CONFIG\_NAMESPACES

if (!prot->netns\_ok) {

pr\_err("Protocol %u is not namespace aware, cannot register.\n",protocol);

return -EINVAL;

}

+#endif

return !cmpxchg((const struct net\_protocol \*\*)&inet\_protos[protocol],

NULL, prot) ? 0 : -1;

After adding this change in above method, I compiled complete kernel. But, this did not work also. I got the same error also. Then after carefully studying what happens actually so that this error message is coming, it is coming because of ***.netns\_ok***. I have also posted the query in class discussion and with the help of answer of this post, I figured out the solution.

So I modified my protocol structure as follows:

static const struct net\_protocol cse536\_protocol = {  
     .handler     = cse536\_rcv,  
     .err\_handler = cse536\_err,  
     .no\_policy   = 1,

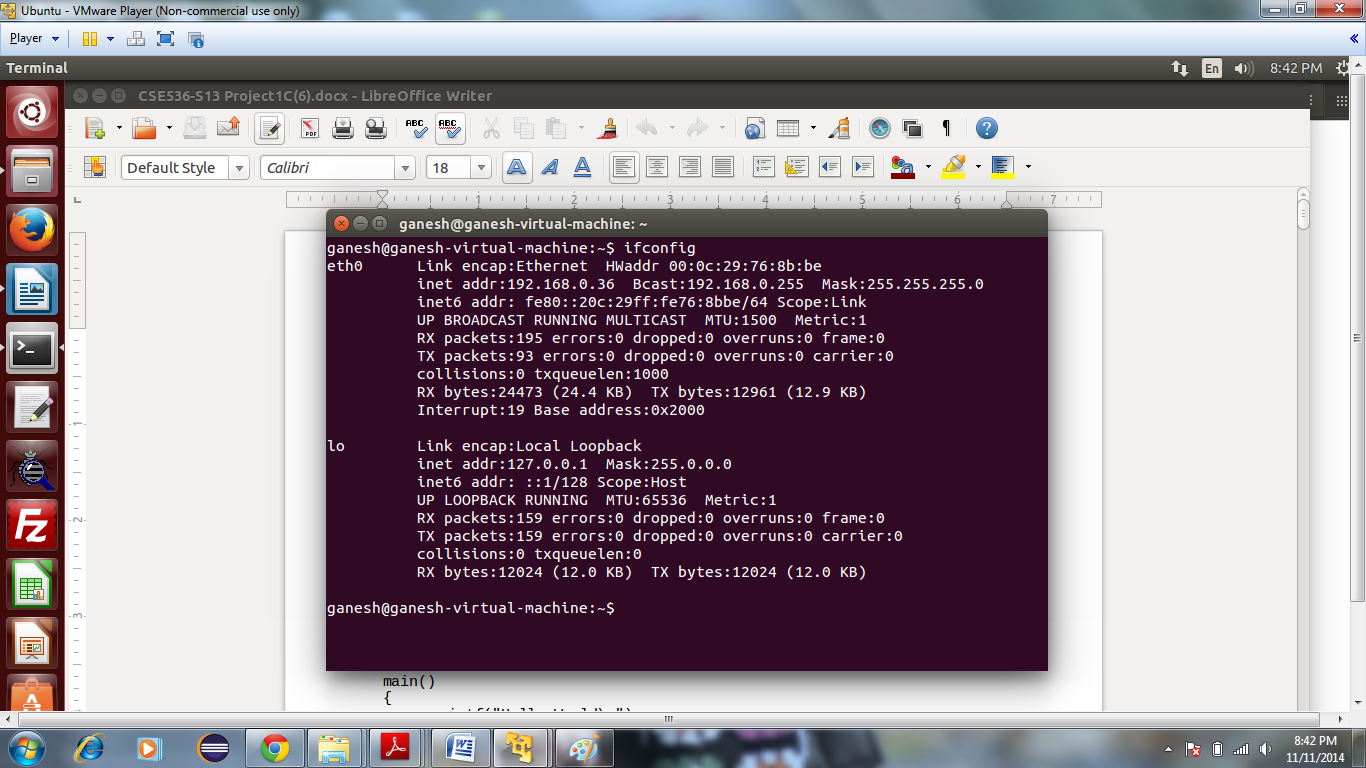
**>> .netns\_ok = 1,**

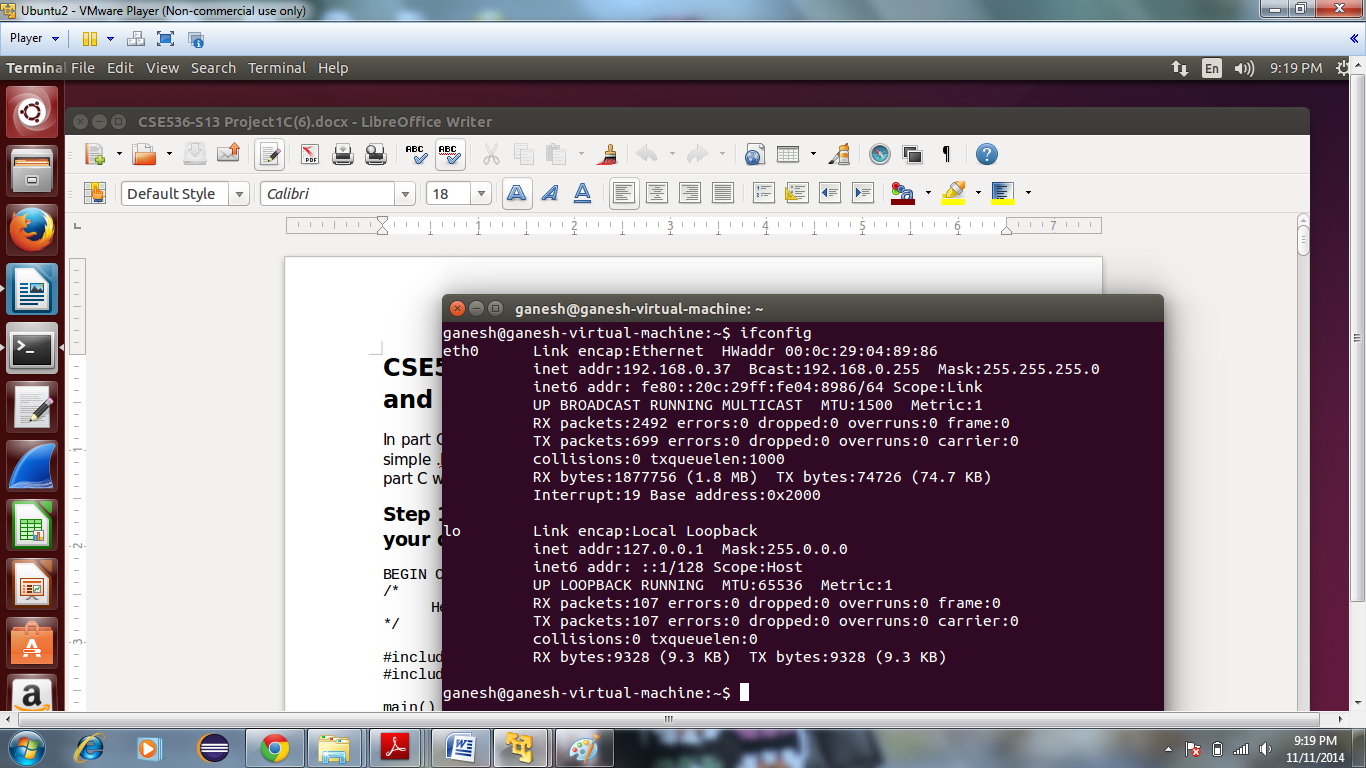
};

After this modification, I successfully added my protocol and able to send and receive the message over two virtual machine.

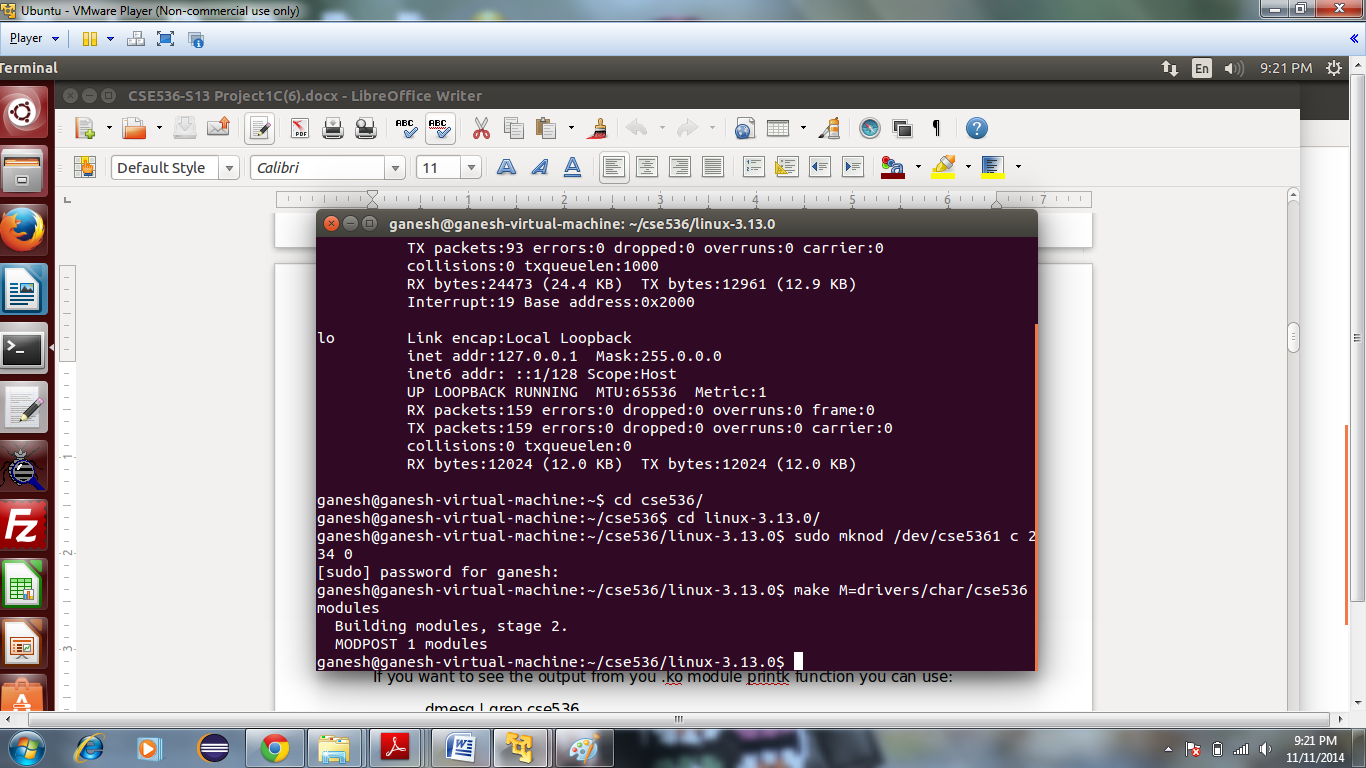
# Project Results

1. IP addresses of both virtual machines:

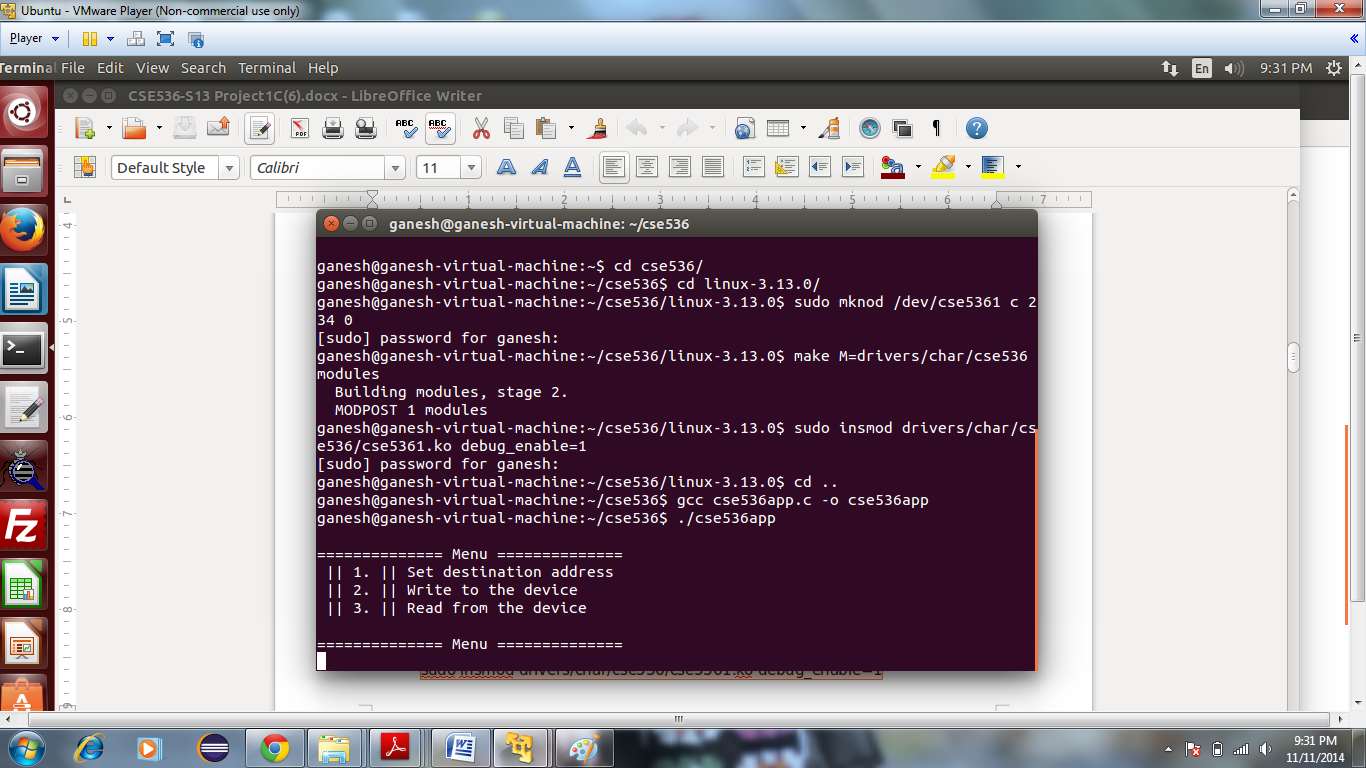




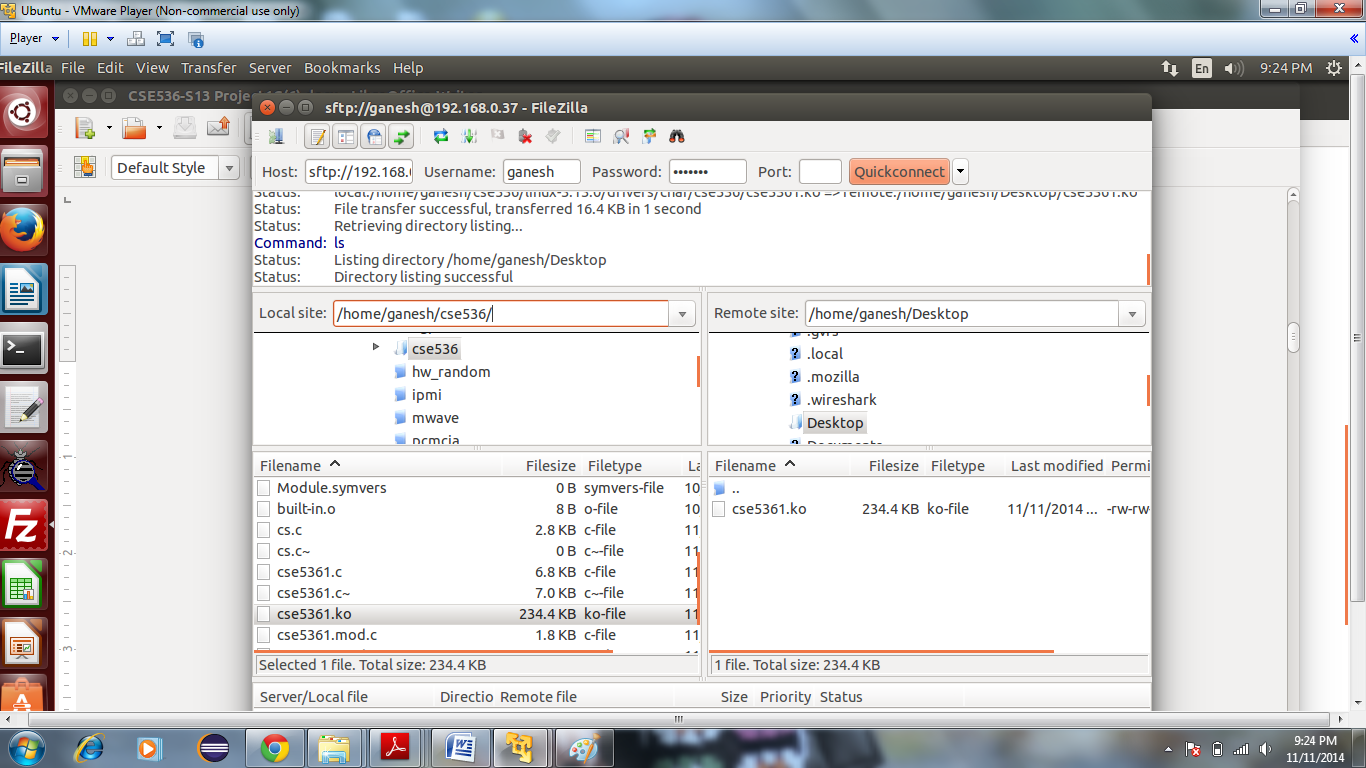
1. Make node for character device and compile the kernel module of character device.



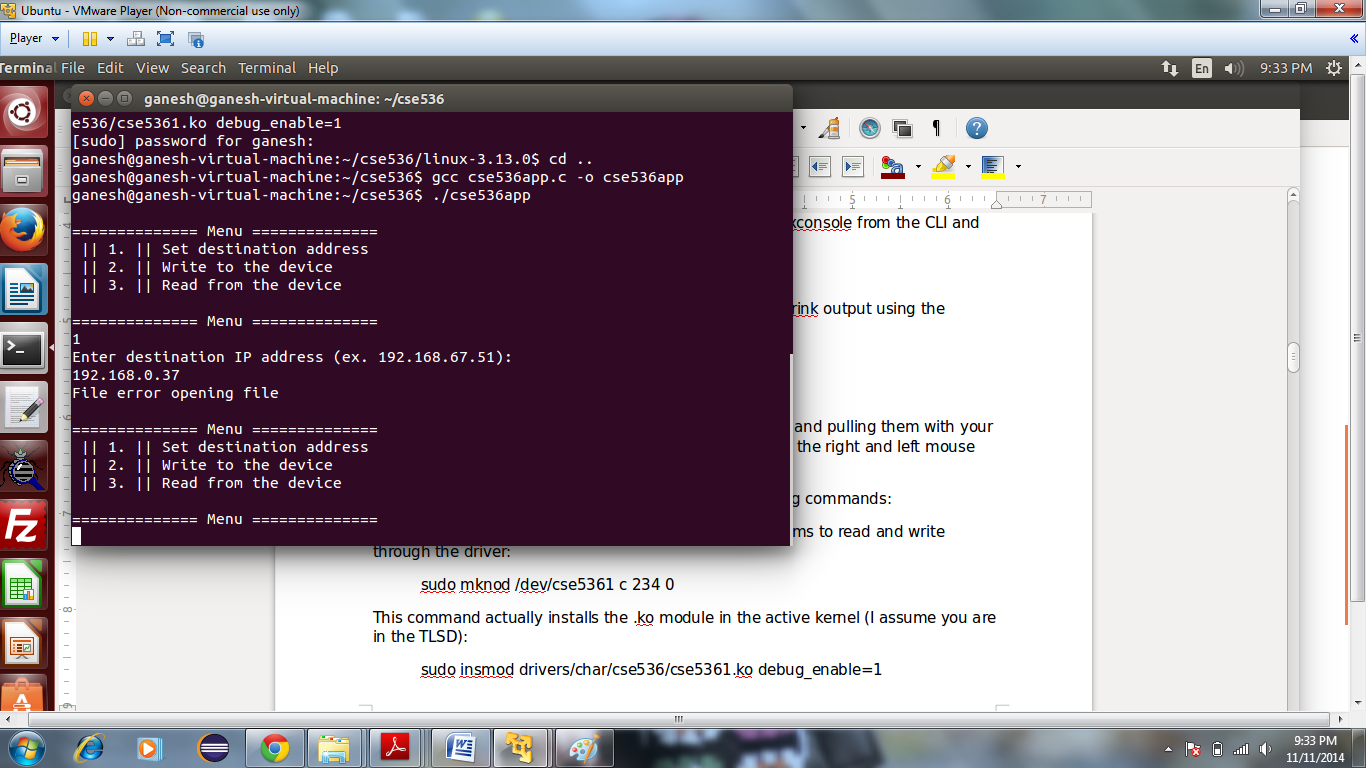
1. Inserting the module to the device.



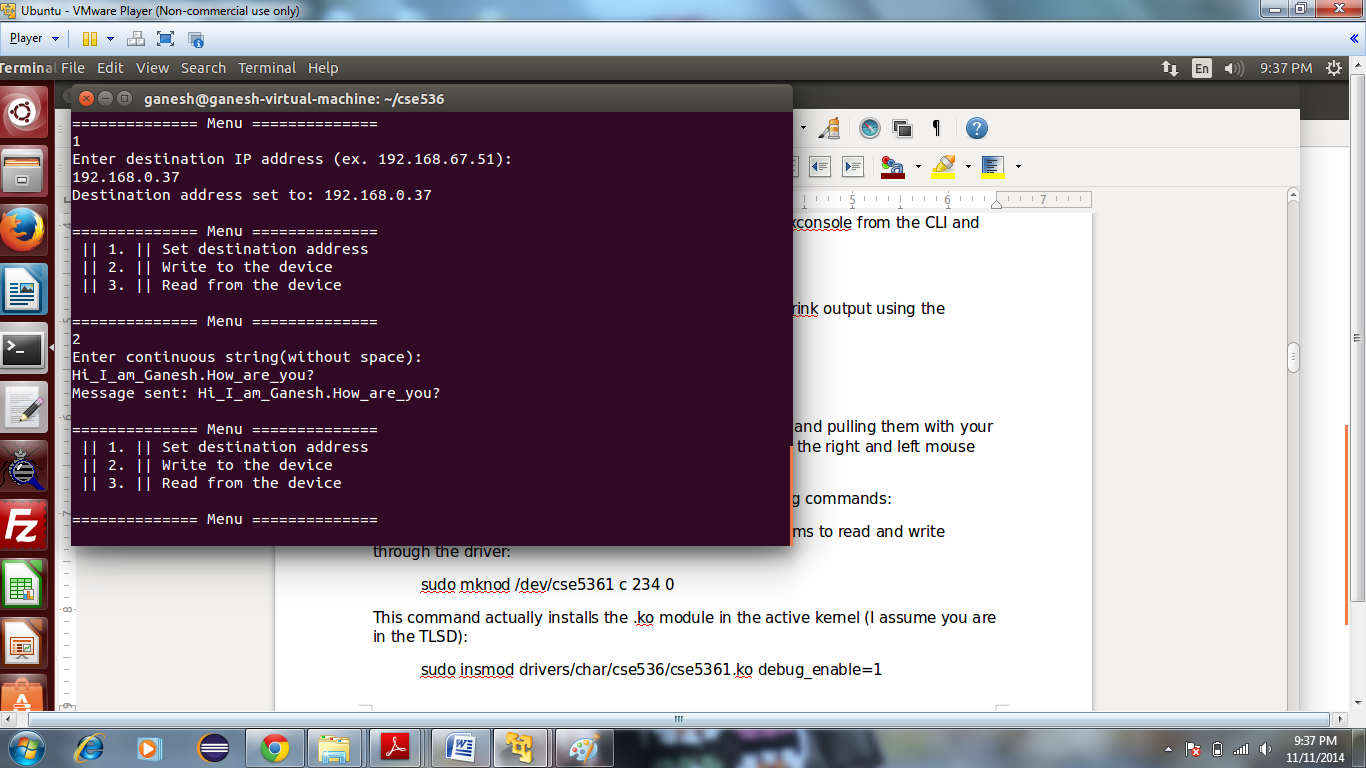
1. Send the .ko of compiled kernel module and object file of application program to other virtual machine as we did in project 1C.



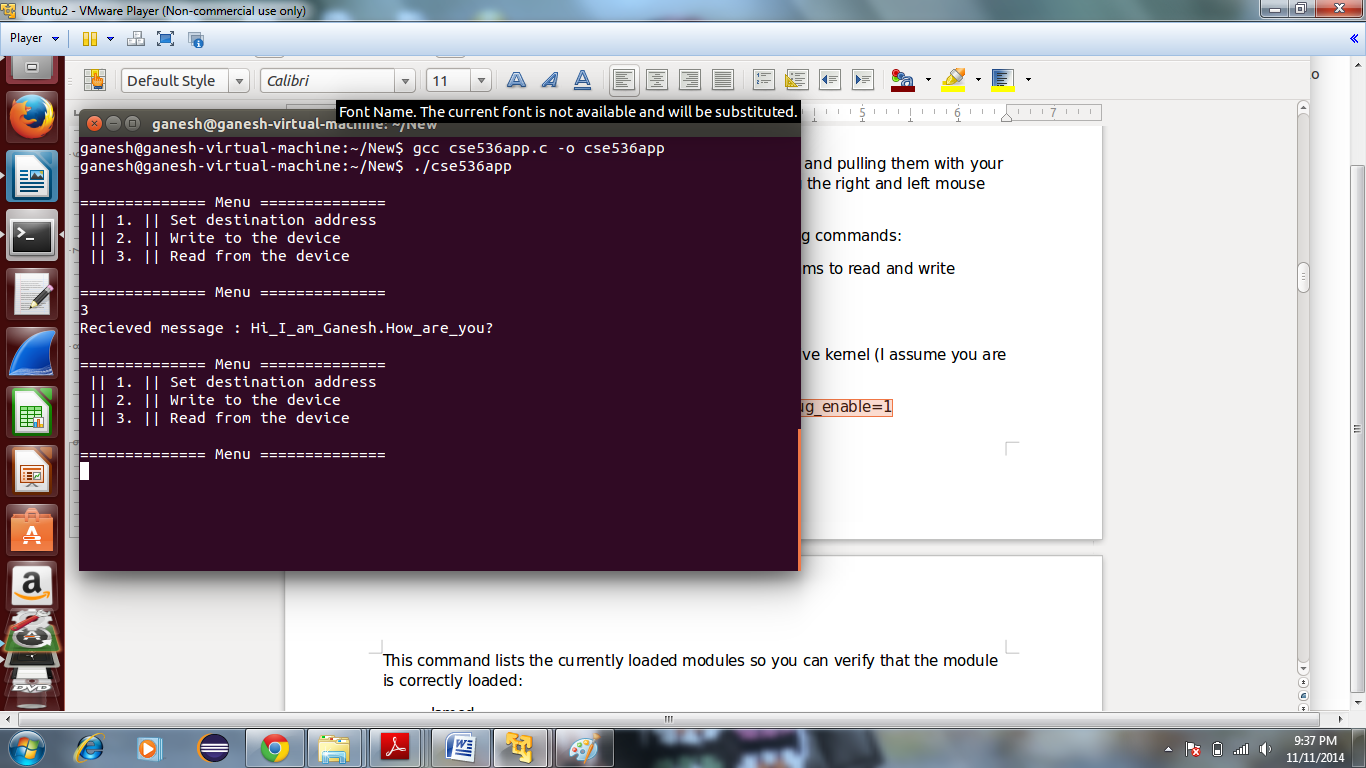
1. Run application on both virtual machines. Set the destination address in virtual machine application. IP address of VM2 is 192.168.0.37.



1. Write the message at VM1.

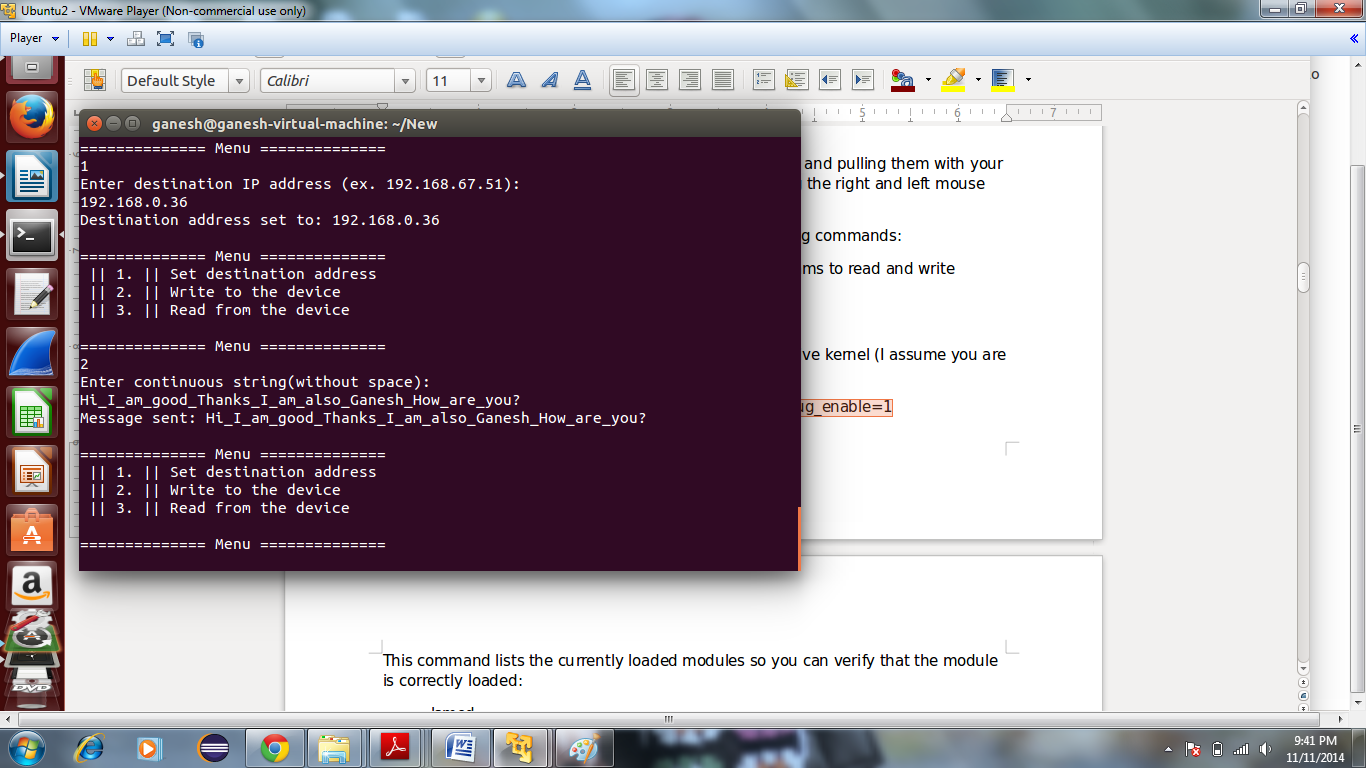


1. Read the sent message at VM2.

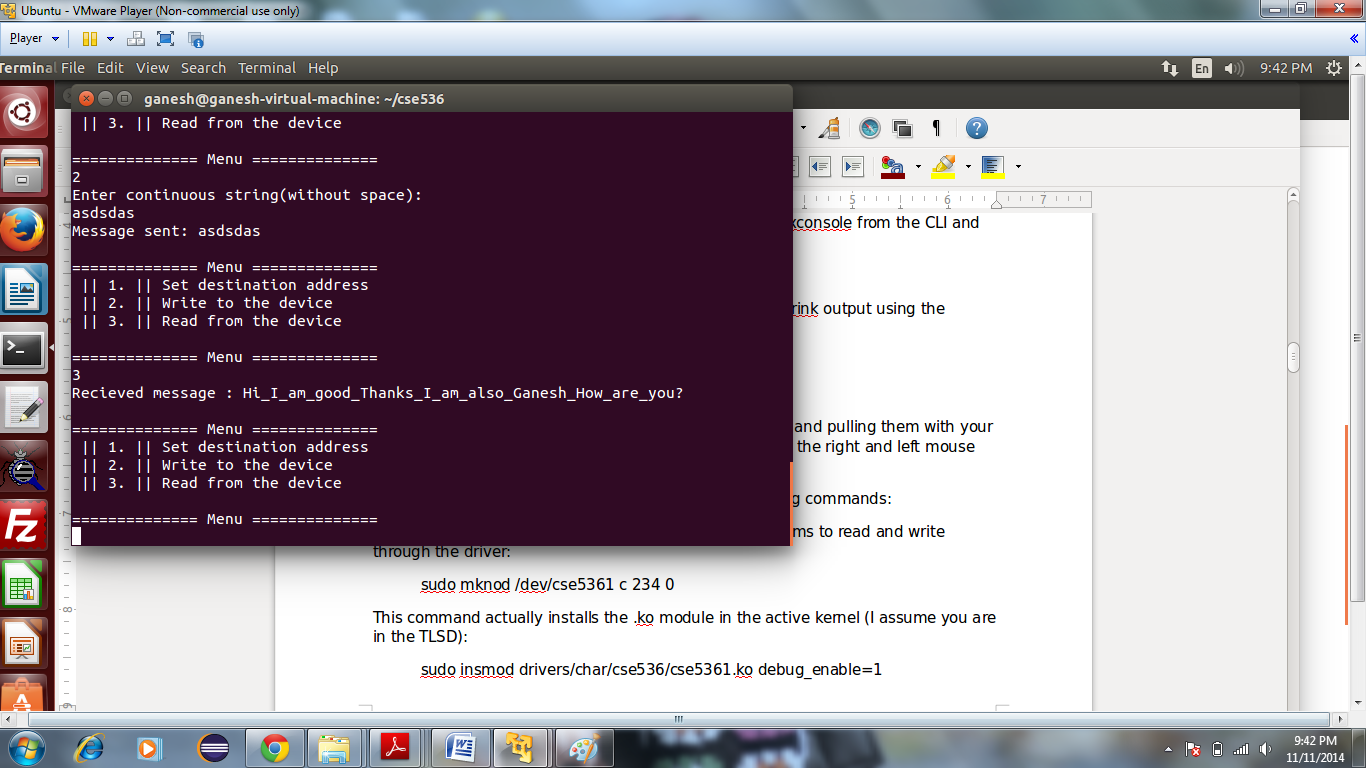


1. Write message at VM2. For this first set the destination address of VM1 at application running at VM2.

Destination address of VM1 is 192.168.0.36.



1. Read the sent message at VM1



These snapshots show the duplex communication between two virtual machines. We can send message to other machines in the network. For that, we need to set appropriate destination address and we can read the message at the receiving machine.

# References:

<https://docs.oracle.com/cd/E23823_01/html/816-4554/ipov-6.html>

<http://www.linuxfoundation.org/collaborate/workgroups/networking/sk_buff>

<http://www.spinics.net/lists/netdev/msg248984.html>