**Project 3**

**Use a logical clock to impose an ordering on Event and Acknowledgement messages**

**Course** : CSE536

**Faculty** : Prof. Alan Skousen

**Name** : Ganesh Ramkrushna Zilpe

**ASUID** : 1207578251

**Index**

[**Goal:** 3](#_Toc405301619)

[System Description: 3](#_Toc405301620)

[**Project Description:** 3](#_Toc405301621)

[Important issues of project: 3](#_Toc405301622)

[TCP/IP Protocol Architecture Model 4](#_Toc405301623)

[Physical Network Layer 4](#_Toc405301624)

[Data-Link Layer 4](#_Toc405301625)

[Internet Layer 4](#_Toc405301626)

[Application Layer 4](#_Toc405301627)

[Project Implementation Steps 5](#_Toc405301628)

[**1.** **Create struct for packet** 5](#_Toc405301629)

[**2.** **Register protocol:** 5](#_Toc405301630)

[**3.** **Writing the message** 6](#_Toc405301631)

[**4.** **Send the message** 6](#_Toc405301632)

[**5.** **Receive the message** 7](#_Toc405301633)

[**6.** **Read the message** 8](#_Toc405301634)

[**7.** **Sending event and acknowledgement to the Monitor** 9](#_Toc405301635)

[Project Results 10](#_Toc405301636)

[1. IP addresses of both virtual machines: 10](#_Toc405301637)

[2. Make node for character device and compile the kernel module of character device. 11](#_Toc405301638)

[3. Send the .ko of compiled kernel module and object file of application program to other virtual machine as we did in project 1C. 12](#_Toc405301639)

[4. Run the application at both machine and set Monitor address 12](#_Toc405301640)

[5. Set the destination address in at both machines. 13](#_Toc405301641)

[6. Write the message at M1. 14](#_Toc405301642)

[7. Monitor get event from M1 14](#_Toc405301643)

[8. Read at M2 15](#_Toc405301644)

[9. Check acknowledgement at M1 15](#_Toc405301645)

[10. Check acknowledgement at Monitor 16](#_Toc405301646)

[11. Reverse communication from M2 to M1 16](#_Toc405301647)

[References: 19](#_Toc405301648)

**Goal:**

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

# 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:**

Project 3 uses project 2 to communicate between machines to implement a logical clock. When sending a message from user 1 to user 2, we want to have an ordering to determine among several messages which came first. As per class discussion, we can maintain a common clock for each host that will place an ordering on the messages that are communicated between the hosts.

## Important issues of project:

* Store the clock in the .ko
* For handling multiple apps on the same host, we decided to use synchronization to control access from multiple users.
* For making the communication reliable and not duplicated, we limit our reliability to waiting for an acknowledgement before retransmitting after 5 seconds. If a second timeout occurs discard the request as unobtainable. Protocol Number 234
* Transaction format – Store IP numbers in standard format (Big Endian in\_aton() at kernel level, inet\_aton() at the user level)
  + Ack format
    - 4 Record ID – ack=0 or event =1
    - 4 final clock
    - 4 original clock
    - 4 source IP
    - 4 destination IP
    - 236 string
  + Event format
    - 4 Record id – ack=0 or event=1
    - 4 final clock
    - 4 origonal clock
    - 4 source IP
    - 4 destination IP
    - 236 string

# 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. **Create struct for packet**

Create the packet of 256 byte. This struct packet is used for sending event and acknowledgement.

struct datagramBuffer

{

uint32\_t record\_id;

uint32\_t final\_clock;

uint32\_t original\_clock;

\_\_be32 source\_ip;

\_\_be32 destination\_ip;

uint8\_t data[236];

};

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 machines. For sending the message, we need to write the message to the character device first. Project 2 is already dealt with writing the message. But, for project 3, we need to write the message along with assigning written packet structure to linked list of cse536 buffer, so that we can read this filled packet structure in the application program and can be able to send to monitor.

Also, after sending the written message, increment the logical clock of the module.

if ( cse536buffhead == 0 )

{

cse536buffhead = temp;

}

if ( cse536bufftail != 0 )

{

cse536bufftail->next = temp;

}

cse536bufftail = temp;

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

cse536\_sendmsg((char \*)&tempDatagram, sizeof(tempDatagram));

//increment\_clock

increment\_clock();

//Check if you don't get ack in 5 sec then send event again semaphore // 5sec wait,

if(down\_timeout(&my\_sem, msecs\_to\_jiffies(5000)))

{

cse536\_sendmsg((char \*)&tempDatagram, sizeof(tempDatagram));

//increment\_clock

increment\_clock();

}

1. **Send the message**

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

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. We have already done this for project 2. For project 3, if received message is event, then we need to send back the acknowledgement to the sender. For checking whether received message is event or not, we can check the record\_id of message. For event, record\_id is 1.

Also, we need to synchronize the logical clock of module with the final clock set in the received message. If received message is event and module clock is less than the final clock of received event, then assign final clock of received event + 1 to the module logical clock.

// clock syncronization

/\* If received packet is event, then check the clock of receiver whether it is greater than event final clock\*/

if(item->datagram.record\_id == 1)

{

if(item->datagram.final\_clock >= moduleClock)

{

moduleClock = item->datagram.final\_clock + 1;

}

}

if(item->datagram.record\_id == 0)

{

up(&my\_sem);

return 0;

}

//create new datagramBuffer for acknowledgement

tempDatagram.record\_id = 0;

tempDatagram.original\_clock = item->datagram.original\_clock;

tempDatagram.final\_clock = moduleClock;

tempDatagram.source\_ip = item->datagram.source\_ip;

tempDatagram.destination\_ip = item->datagram.destination\_ip;

memcpy(tempDatagram.data, item->datagram.data, sizeof(item->datagram.data));

cse536\_daddr = item->datagram.source\_ip;

cse536\_sendmsg((char \*)&tempDatagram, 256);

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. If the received message is acknowledgement, then with reading the acknowledgement, we need to send the acknowledgement to the Monitor from application program.

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;

}

**From cse536app.c**

case 3:

printf("Data Read:\n");

fd = openfile("rb");

if (fd)

{

count = fread((&datagram3), 1, sizeof(datagram3),fd);

if (!count)

printf("No data read\n");

else

{

if(datagram3.record\_id == 1)

printf("%s\n", datagram3.data);

else

{

printf(" Got Acknowledgement and sent to Monitor \n ");

sendToMonitor((char \*)&datagram3);

}

}

fclose(fd);

}

1. **Sending event and acknowledgement to the Monitor**

If we sent event to other machine/user, the we need to send that event to Monitor also. Similarly, if we get the acknowledgement for the event we have sent in the past, then also we need to send the acknowledgement to the Monitor.

void sendToMonitor(char \*data)

{

struct sockaddr\_in client, server;

struct hostent \*hp;

char buf[MAX\_LINE];

int len, ret, n;

int s, new\_s;

bzero((char \*)&server, sizeof(server));

server.sin\_family = AF\_INET;

server.sin\_addr.s\_addr = INADDR\_ANY;

server.sin\_port = htons(0);

s = socket(AF\_INET, SOCK\_DGRAM, 0);

if (s < 0)

{

perror("simplex-talk: UDP\_socket error");

exit(1);

}

if ((bind(s, (struct sockaddr \*)&server, sizeof(server))) < 0)

{

perror("simplex-talk: UDP\_bind error");

exit(1);

}

if(monitorAddress == "")

hp = gethostbyname( "192.168.0.2" );// Monitor Address Change in class

else

hp = gethostbyname( monitorAddress );

if( !hp )

{

fprintf(stderr, "Unknown host %s\n", "localhost");

exit(1);

}

bzero( (char \*)&server, sizeof(server));

server.sin\_family = AF\_INET;

bcopy( hp->h\_addr, (char \*)&server.sin\_addr, hp->h\_length );

server.sin\_port = htons(SERVER\_PORT);

ret = sendto(s, data, 256, 0,(struct sockaddr \*)&server, sizeof(server));

if( ret <= 0)

{

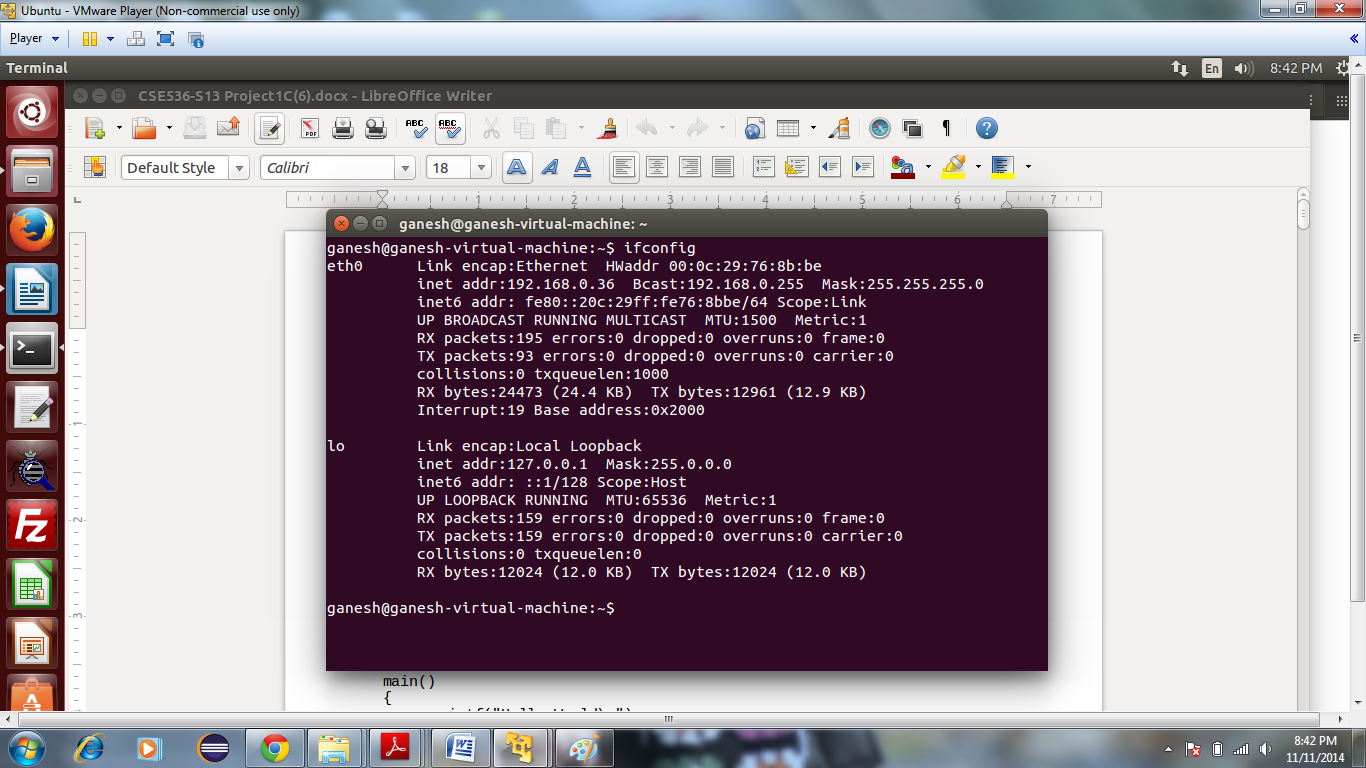
fprintf( stderr, "Datagram Send error %d\n", ret );

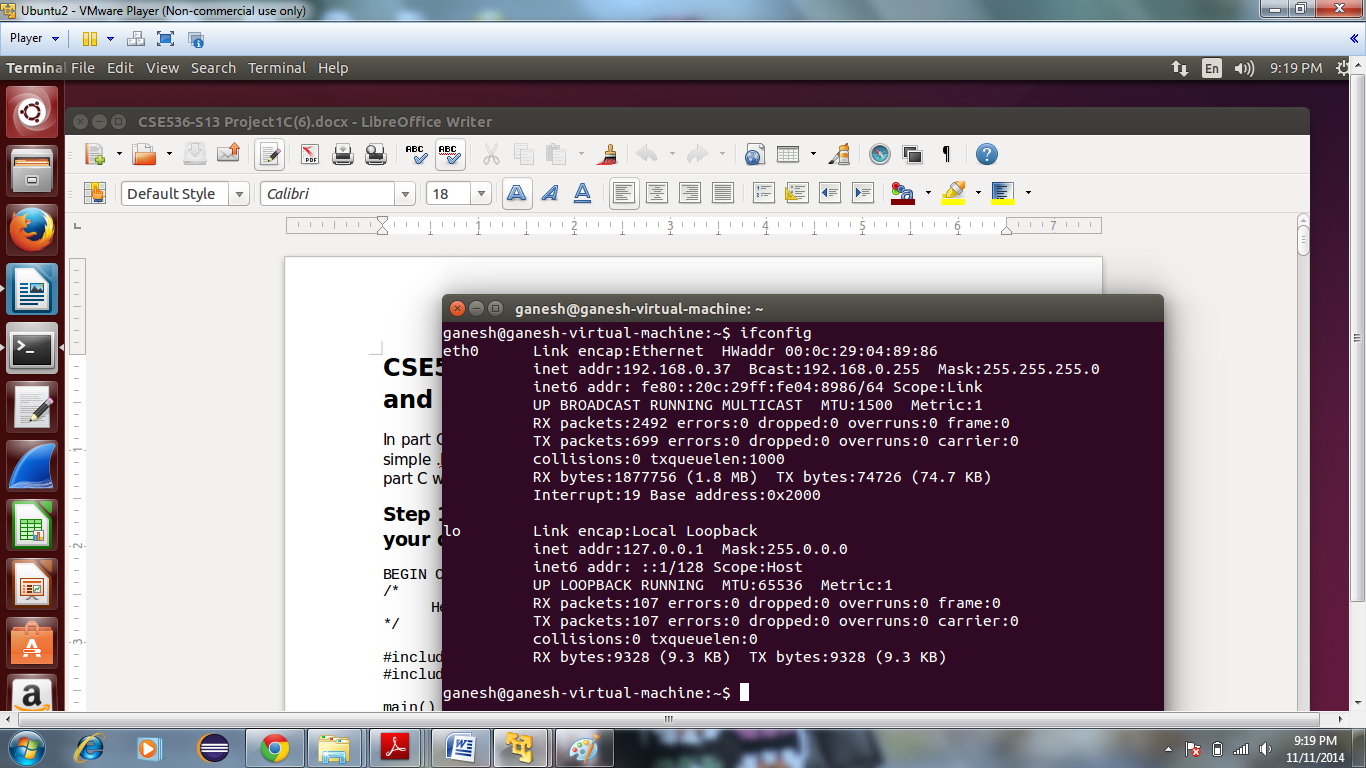
}

}

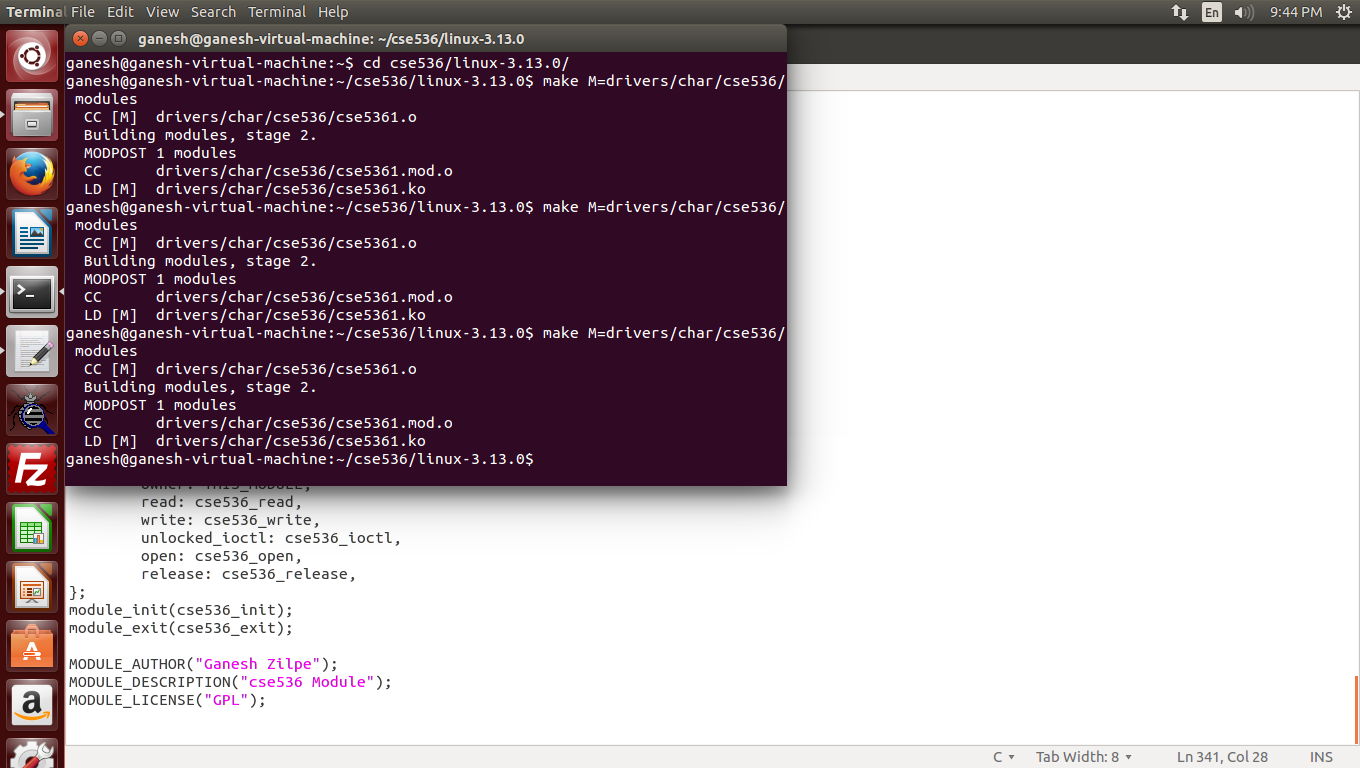
# Project Results

1. IP addresses of both virtual machines:

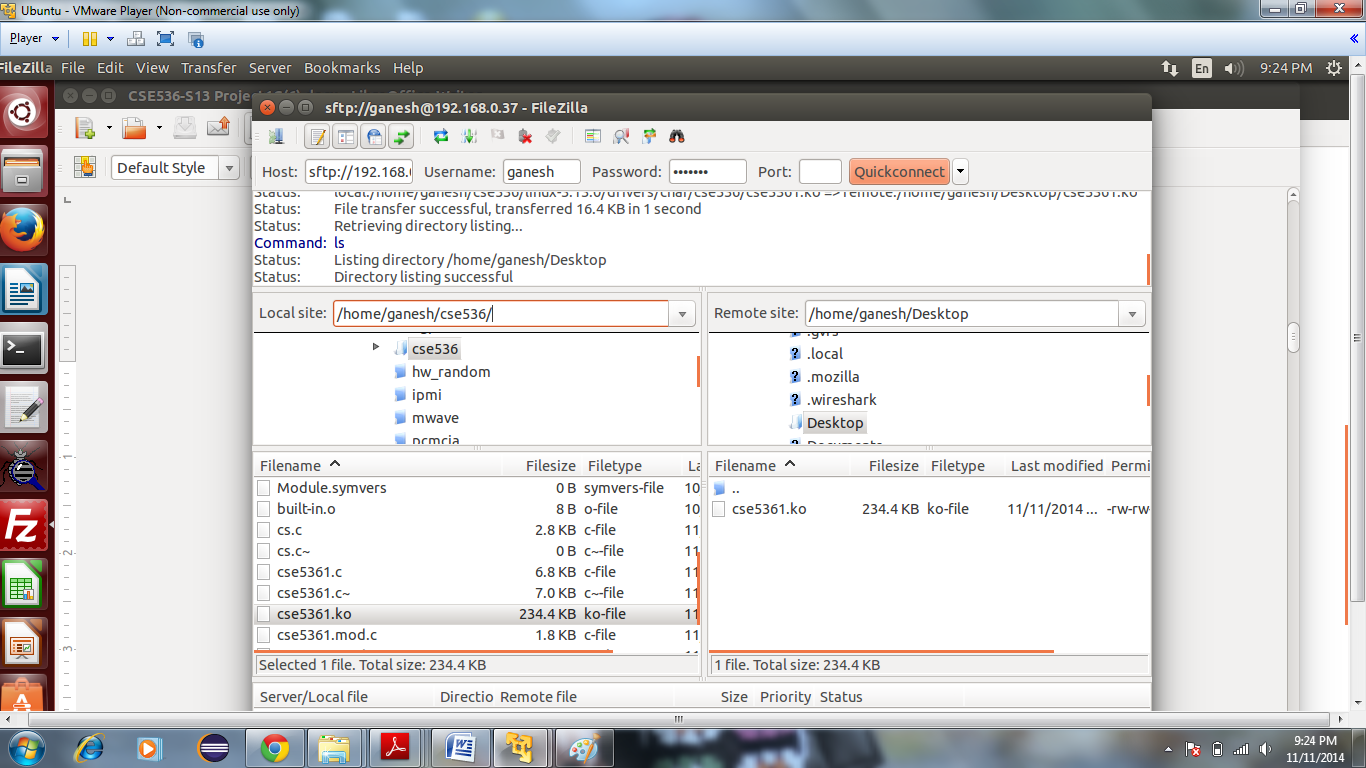




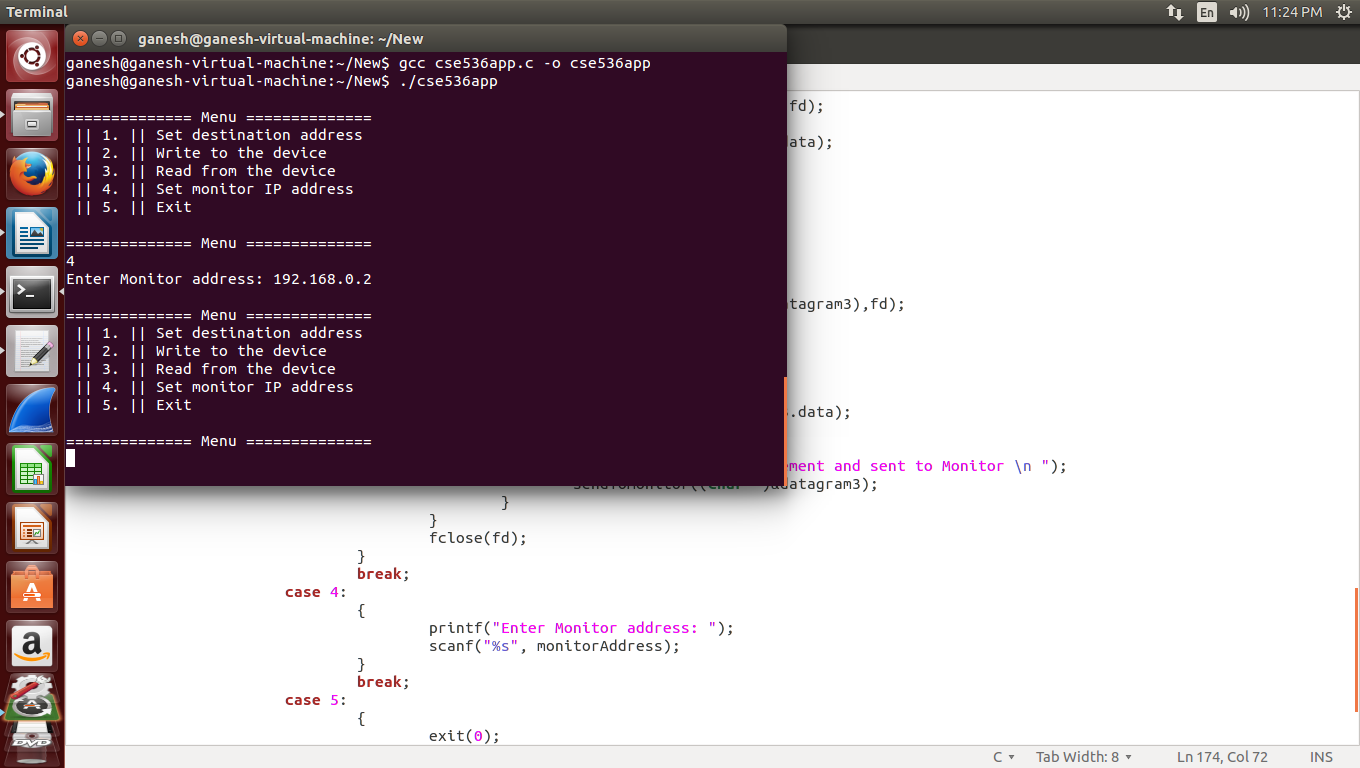
1. Make node for character device and compile the kernel module of character device.



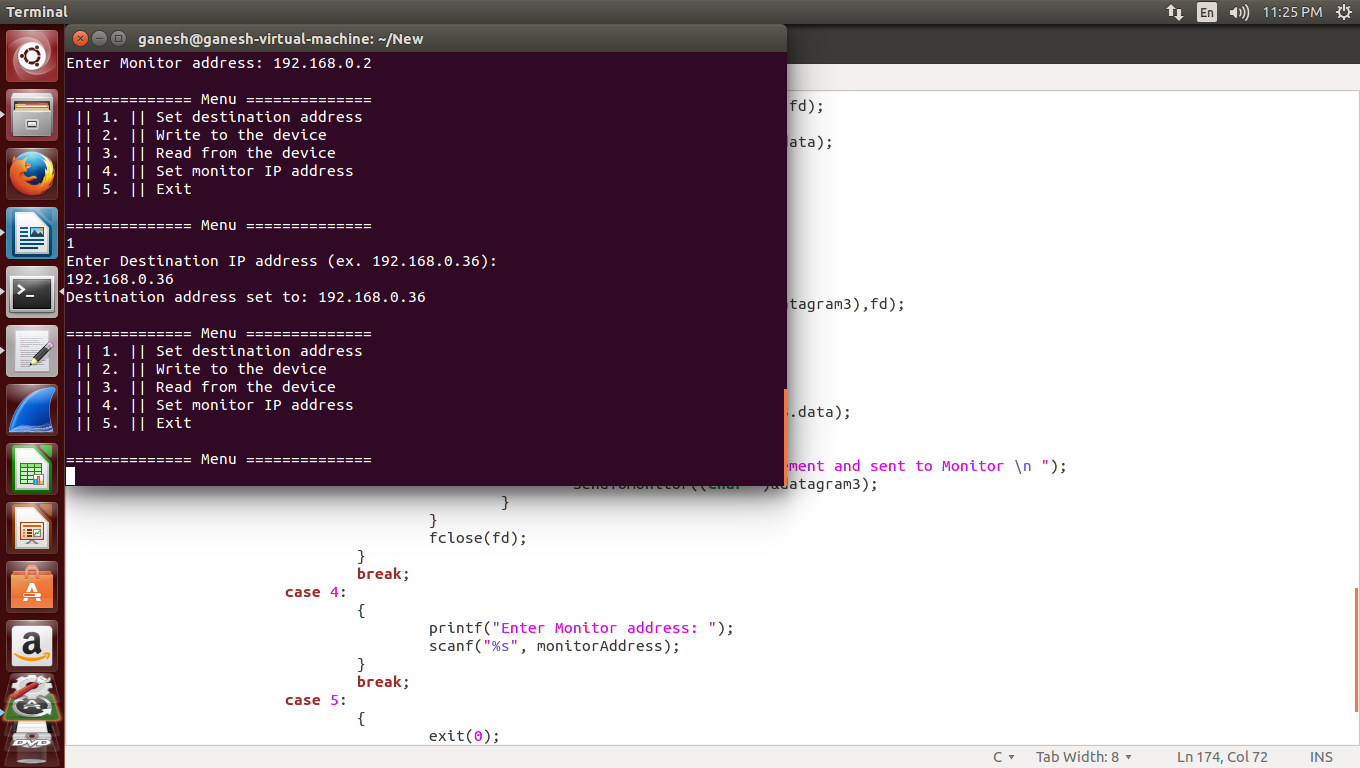
1. Send the .ko of compiled kernel module and object file of application program to other virtual machine as we did in project 1C.

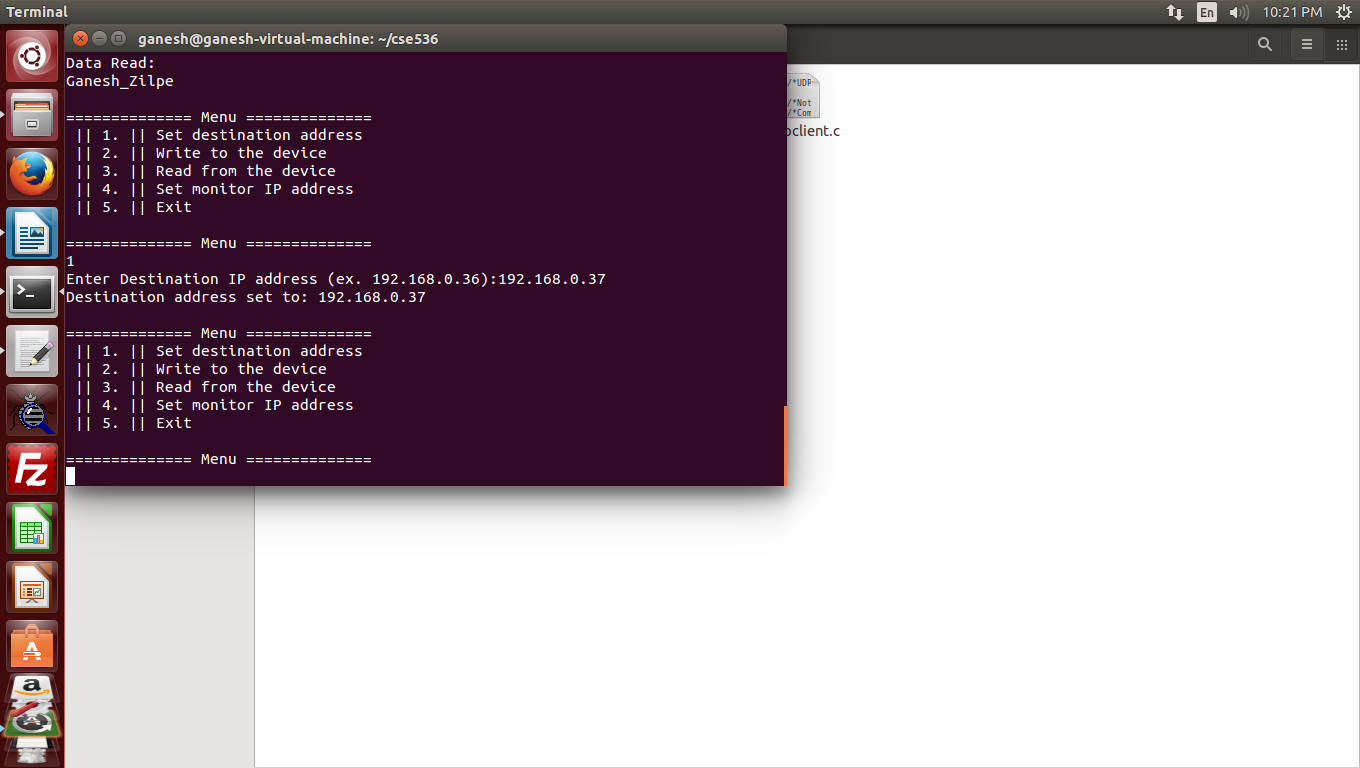


1. Run the application at both machine and set Monitor address where monitor.exe is running.

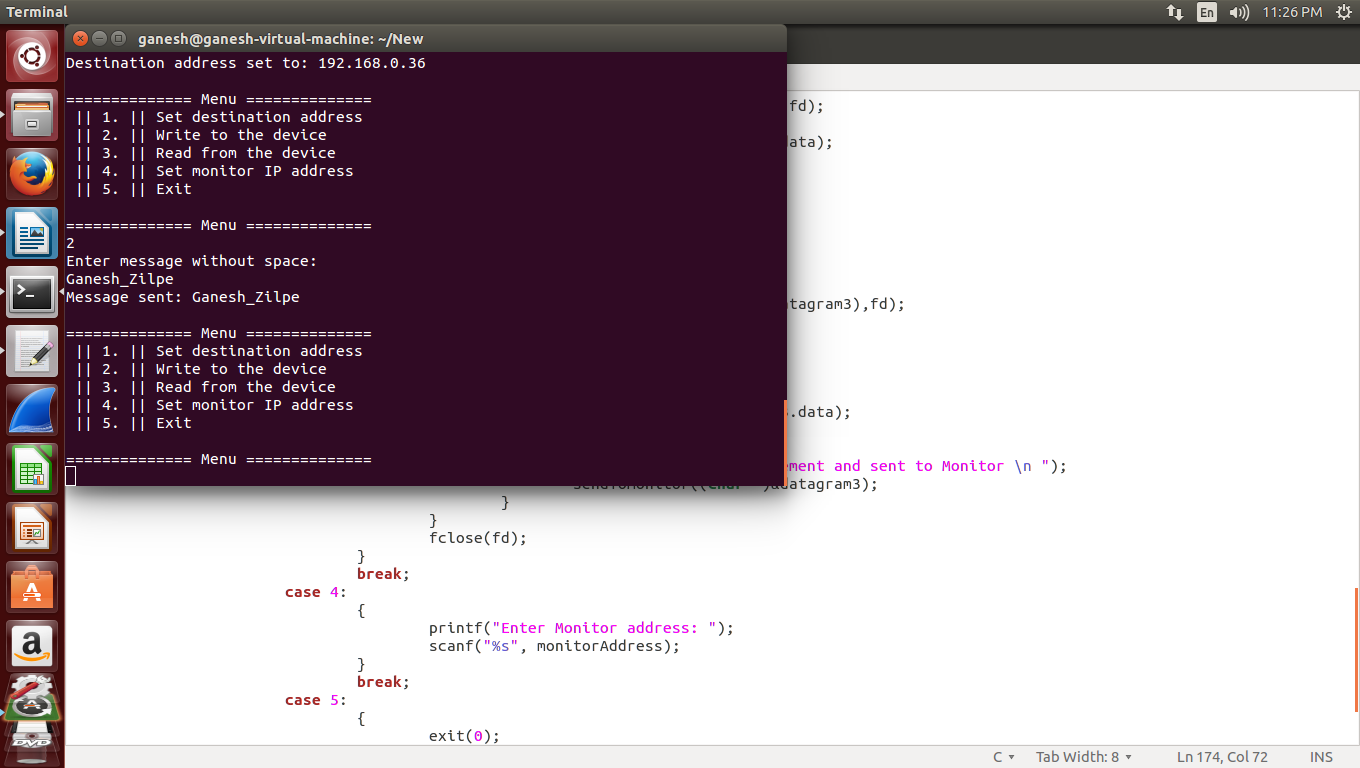


1. Set the destination address in at both machines. IP address of M1 is 192.168.0.37 and M2 is 192.168.0.36

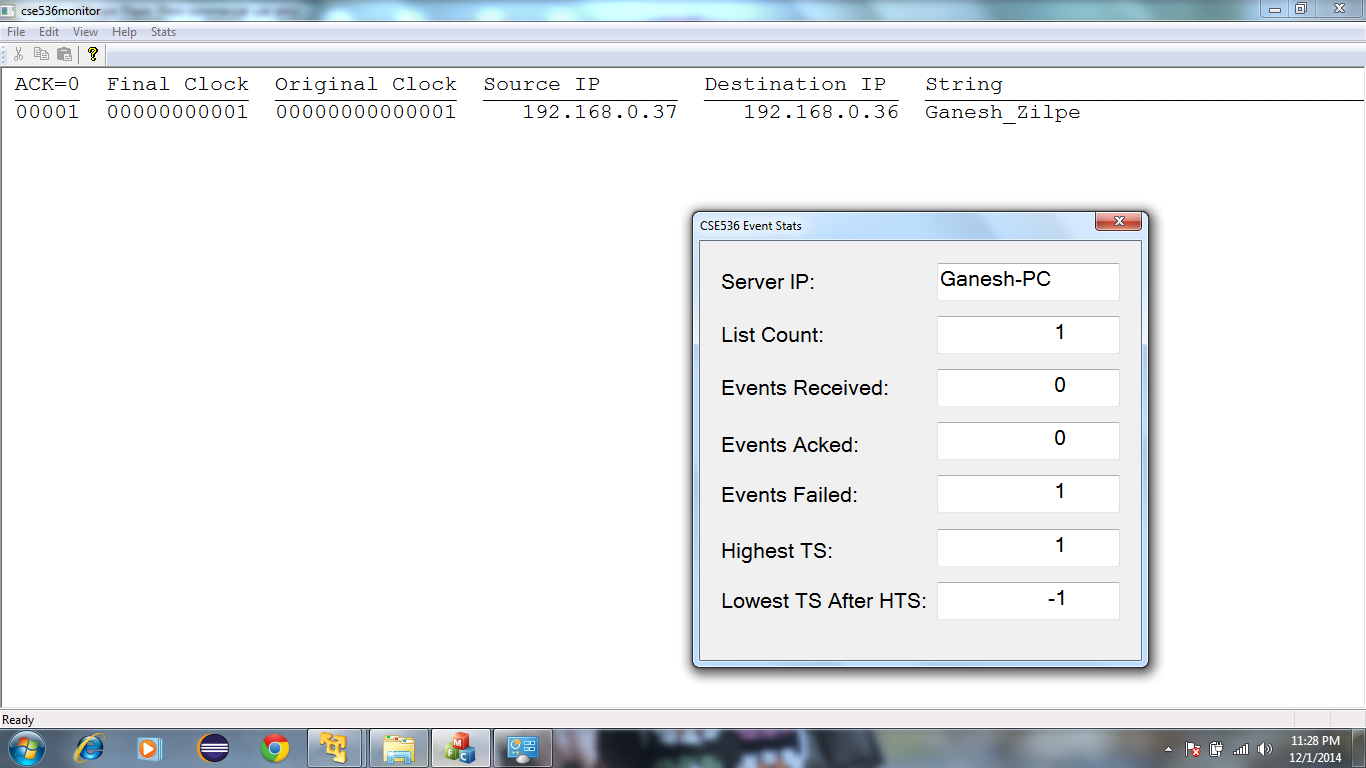




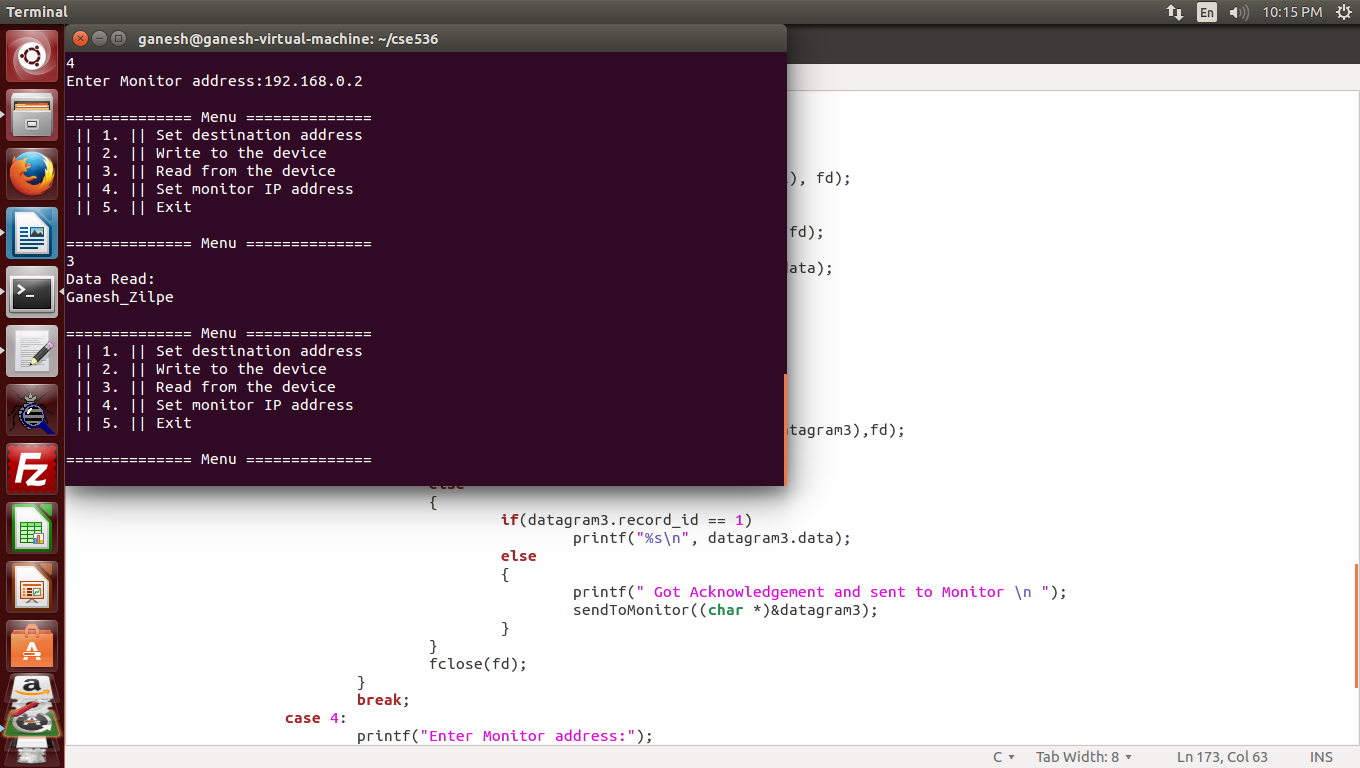
1. Write the message at M1.



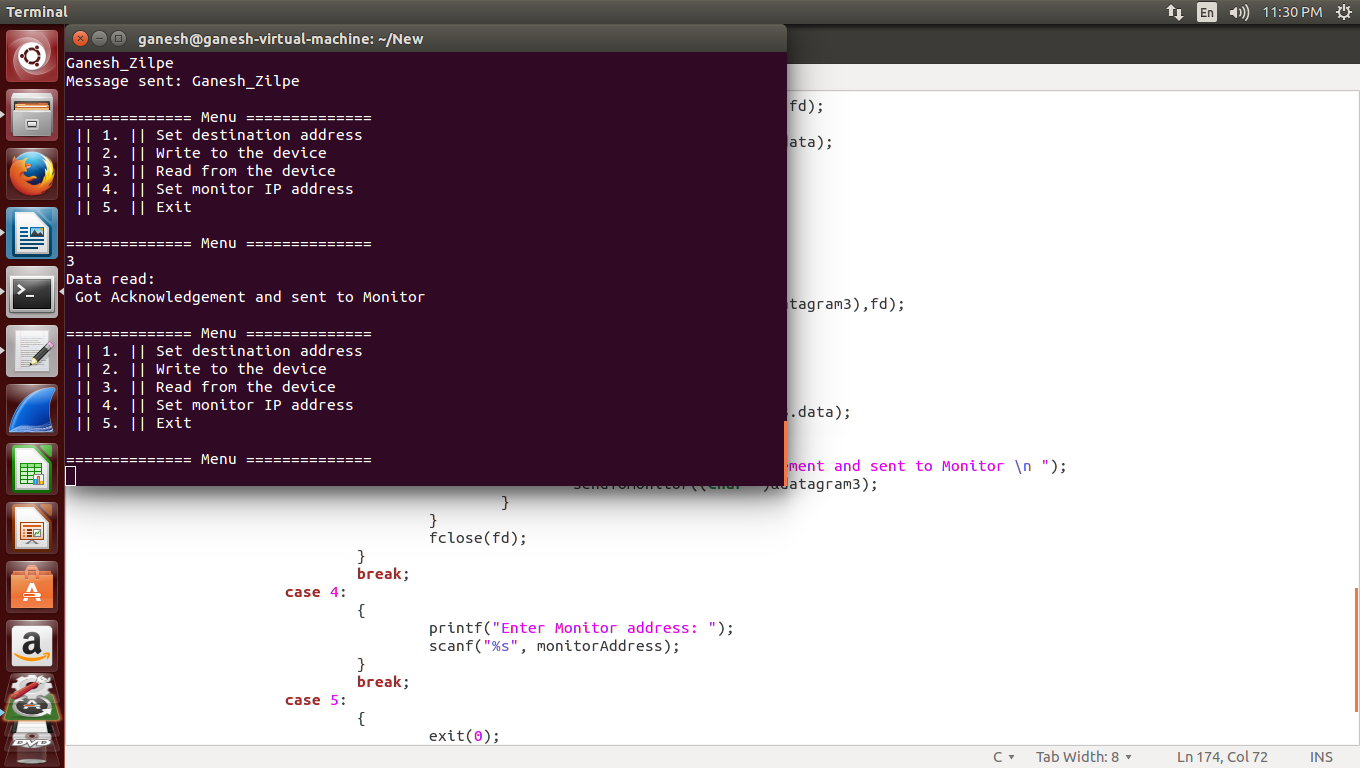
1. Monitor get event from M1



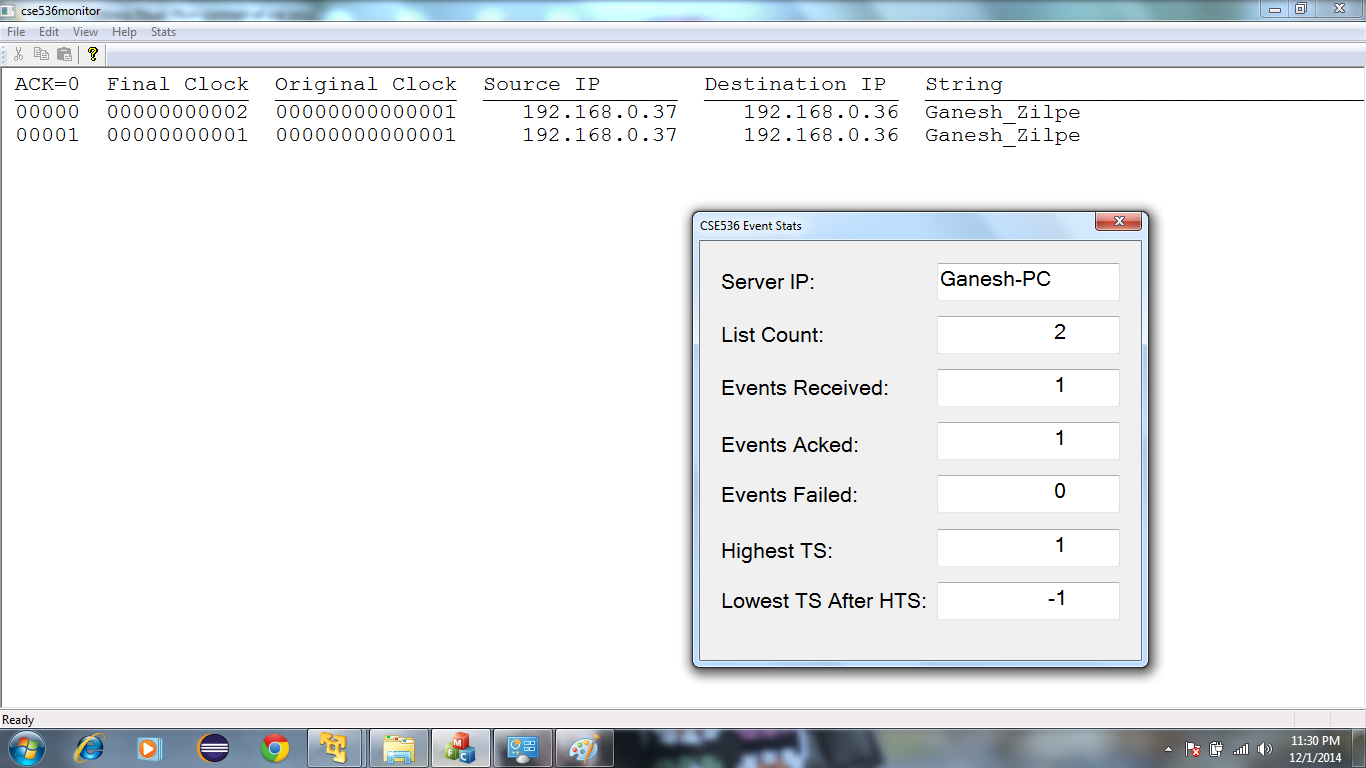
1. Read at M2



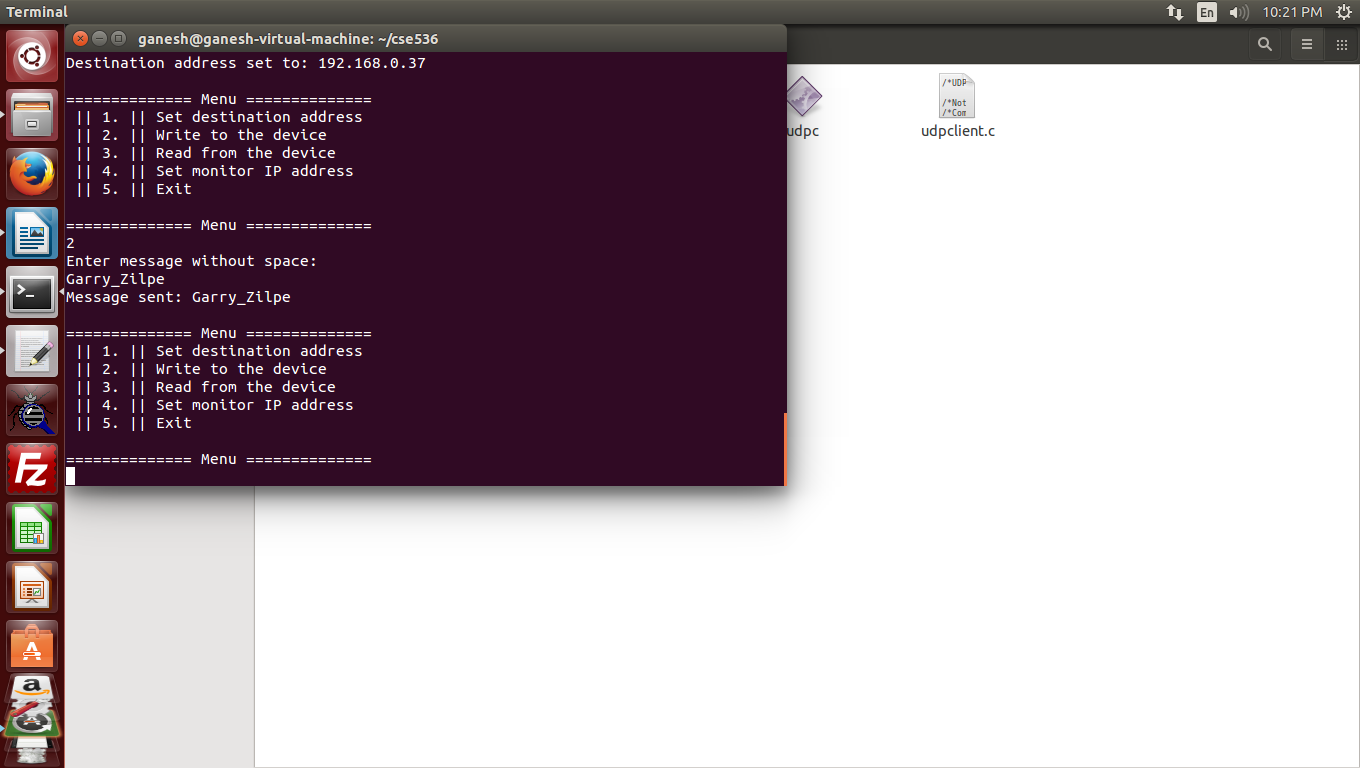
1. Check acknowledgement at M1



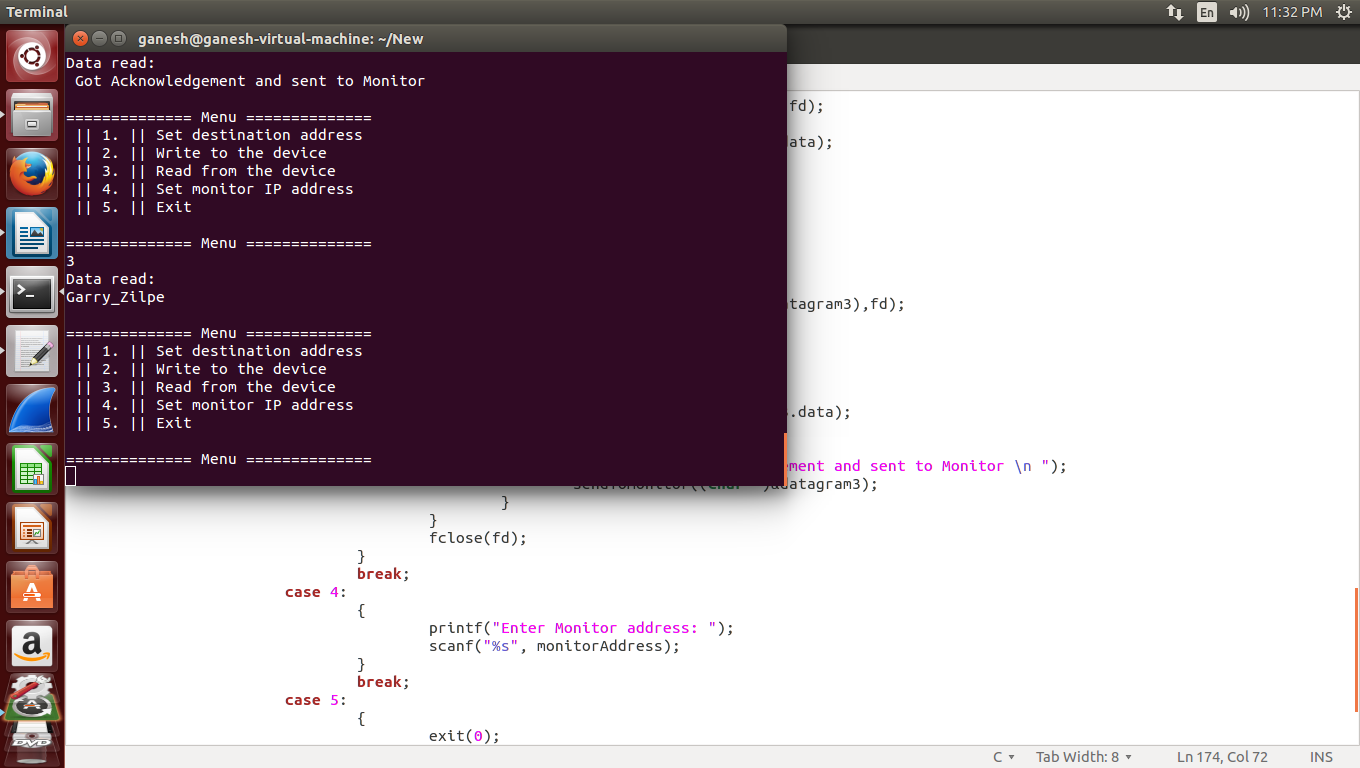
1. Check acknowledgement at Monitor



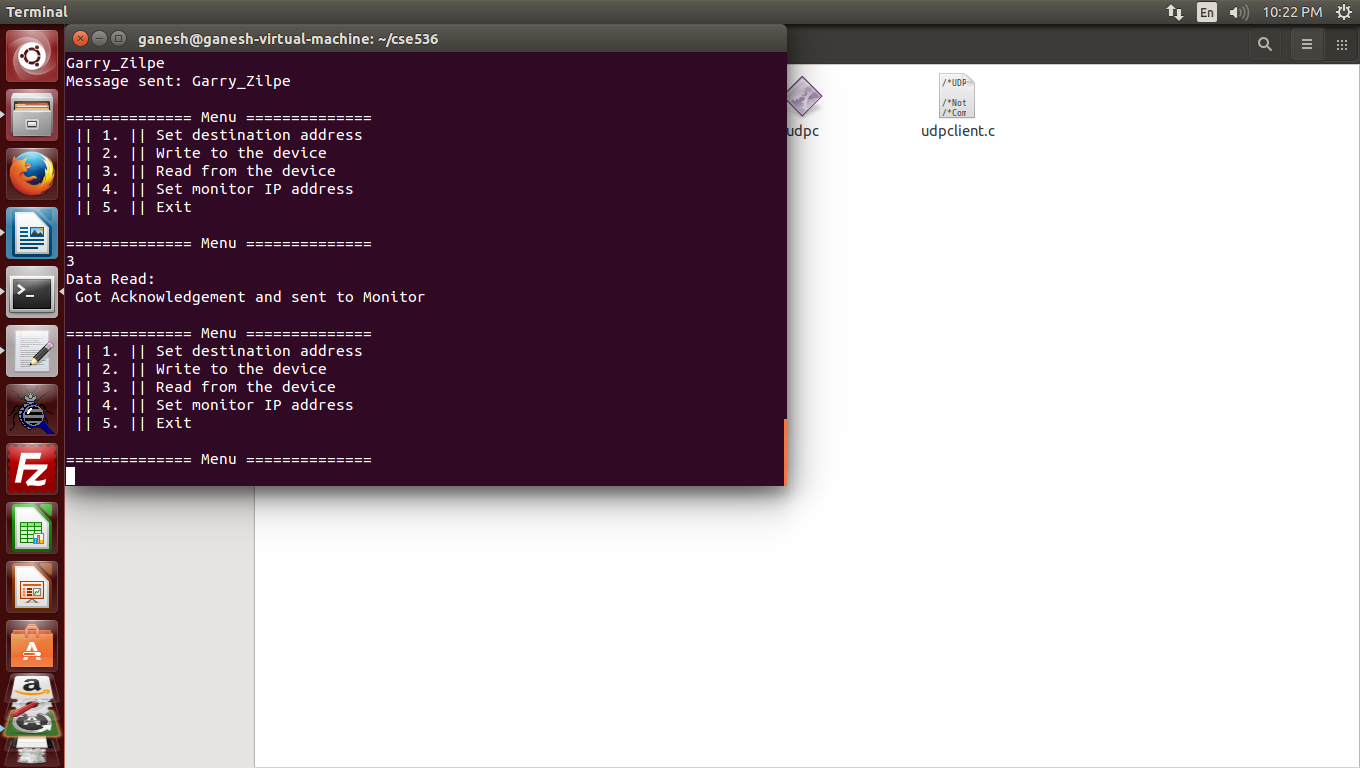
1. Reverse communication from M2 to M1
2. Write at M2



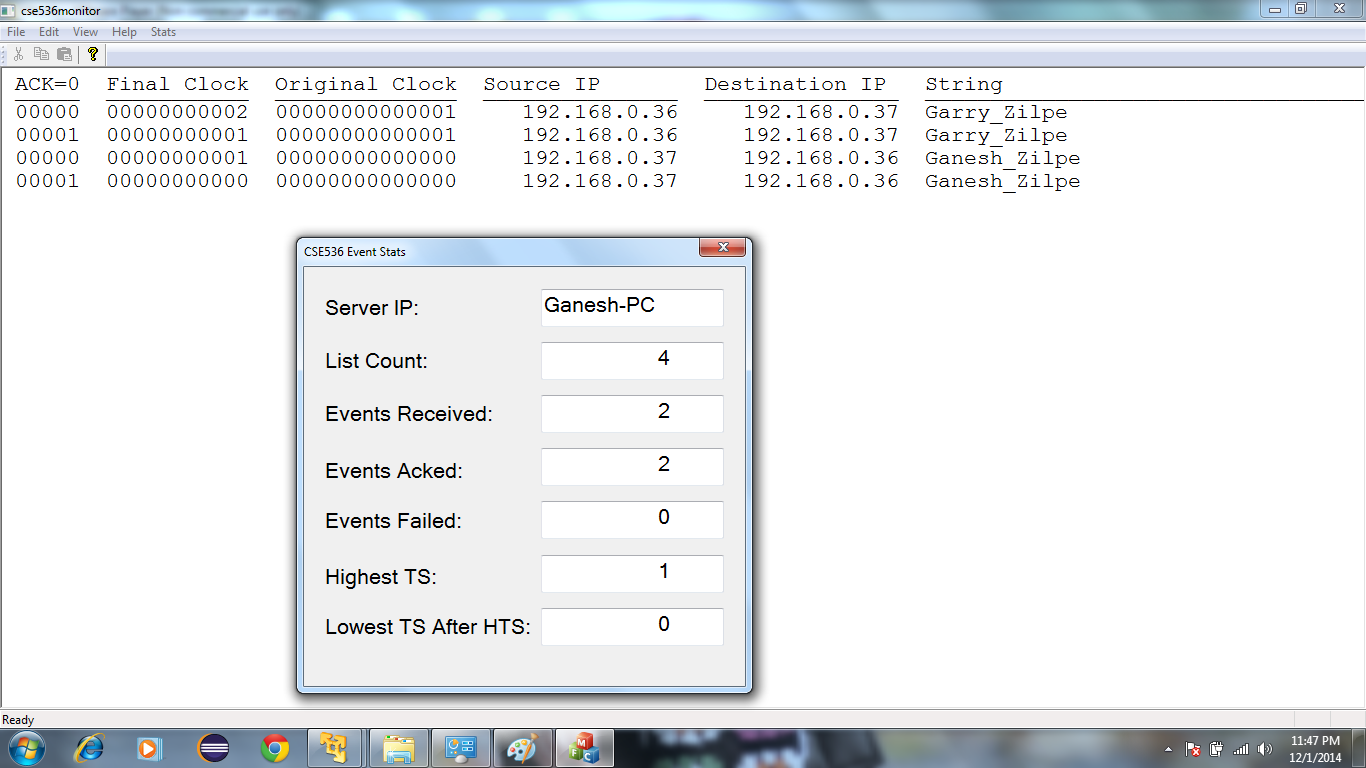
1. Read at M1



1. Read Ack at M2



1. Check the Event and Ack at Monitor



These snapshots show the duplex communication between two 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>