College Of Engineering Trivandrum

NETWORK PROGRAMMING LAB REPORT

CERTIFIED BONAFIDE RECORD OF WORK

done by

Albin Antony

*University Roll No:*TVE16CS010

*Roll No:*10

*Class:*S6 CSE

Supervisor: VIPIN VASU A.V
Associate Professor

Department Of Computer Science and Engineering

May 16, 2019

Contents

1 System Calls

1.1 Aim

To familiarize and understand the use and functioning of System Calls used for Operating system and network programming in Linux.

1.2 Theory

User programs sometimes need kernel services.These services are requested by the user programs via system calls.There are mainly 5 types of system calls

- Process control
- File management
- Device management
- Information maintenance.
- Communications

Some basic system calls used for Operating System and their usage are given below.

1.3 Process Control

1.3.1 fork()

Creates a duplicate of the process.The new process is called the child process which runs concurrently with process (which process called system call fork) and this process is called parent process.They both have different PID.
Return values:
Negative- Child creation failed
Zero- Return to child process
Positive- Return to parent process

1.3.2 exit()

The exit() function immediately terminates the process.Its children are inherited by init(1).

1.3.3 wait()

The function wait() pauses the calling process until its child terminates or a signal is received.Parent resumes from wait() when the child process calls its exit().

1.4 File Manipulation

1.4.1 open()

int open(const char *pathname, int flags);
open() system call opens the file in the ”pathname”.Flags specify the mode ie read-only,write etc.If no file is found in the path new file is created by open().

1.4.2 read()

size_t read (int fd, void* buf, size_t cnt);
Reads cnt bytes from the file pointed by fd to the memory location addressed by buf.Return number of bytes read on success,0 on reaching end,-1 on error.

1.4.3 write()

size_t write (int fd, void* buf, size_t cnt);
Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

1.4.4 close()

int close(int fd);
Tells the OS that you are done with file.Return 0 on success and -1 on error.

1.5 Device Manipulation

1.5.1 ioctl()

The ioctl() function manipulates the underlying device parameters of special files. The argument d is a file descriptor.The second argument is a device-dependent request code. The third argument is an untyped pointer to memory.

1.5.2 read() & write()

Same system call as those in File Manipulation,the files are the systems.

1.6 Information Maintenance

1.6.1 getpid()

Used to get the process id of the calling process.Never throws error.

1.6.2 alarm()

unsigned int alarm(unsigned int sec);
Arranges for sending a SIGALRM to the process in ’sec’ seconds

1.6.3 sleep()

unsigned int sleep(unsigned int sec);
Makes the calling process sleep for 'sec' seconds or until a signal arrives.

1.7 Communication

1.7.1 pipe()

Pipe is used for one way communication between processes.The standard input from one process becomes the standard output of the other.

1.7.2 shmget()

Used to create new message queues and access existing queues.It can access the shared memory used for ipc.

1.8 Program

File Manipulation

```
#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
#include<errno.h>
extern int errno;
int main()
{
        int fd = open("in.txt", O_RDWR | O_CREAT);

        int fd1 = open("out.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
        int sz;
        if (fd ==-1)
        {
                printf("Error Number % d\n", errno);
        }
        else {
                char *c = (char *) calloc(100, sizeof(char));
                sz = read(fd, c,10);
                c[sz] = '\0';
                sz = write(fd1,c, 8);
                if(close(fd) == 0 && close(fd1==0)) {
                        printf("File closed\n");
                }
        }
        return 0;
}
```

Output:-

```
~$cat in.txt

hello world
yoyoyoyo

~$cat out.txt
hello wo
```

Process Control

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int frk=fork();
    wait();
    printf("Fork Value : %d \t processId : %d\n",frk,getpid());
    exit(0);
    printf("Hello World");
    return 0;
}
```

Output:-

```
~$./a.out
Fork Value : 0    pid : 7857
Fork Value : 7857         pid : 7856
```

1.9 Result

Implemented system calls using C and compiled using gcc 6.3.0 on Debian 4.9.0 Kerenel 4.9.0 and the above output where obtained.

2 Basics of Network configurations files and Networking Commands in Linux.

2.1 Aim

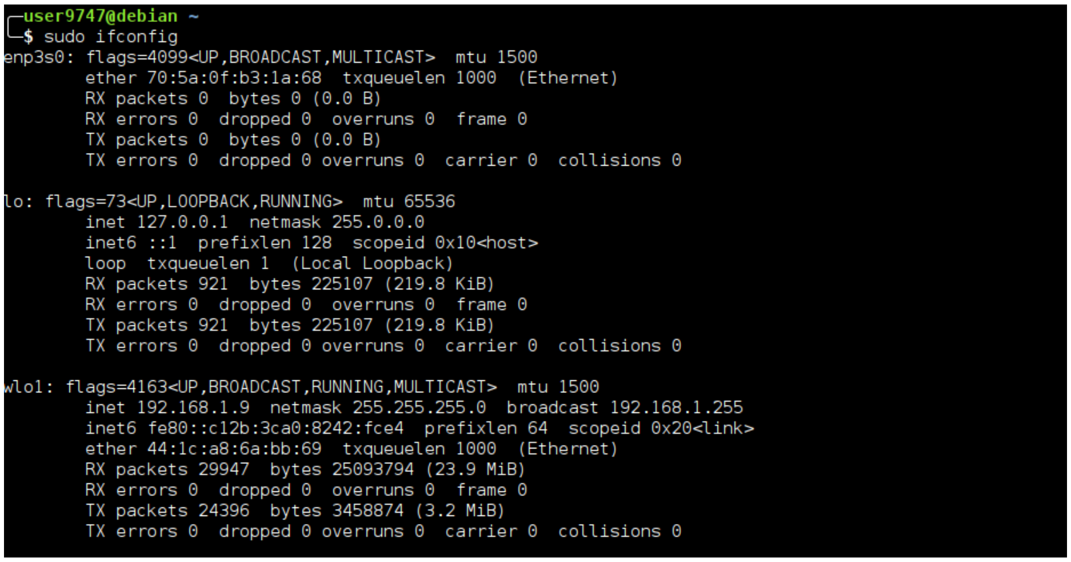
Getting started with Basics of Network configurations files and Networking Commands in Linux.And understand the usage of these commands.

2.2 Theory

Following is a list of basic Linux commands used for networking.

2.2.1 ifconfig

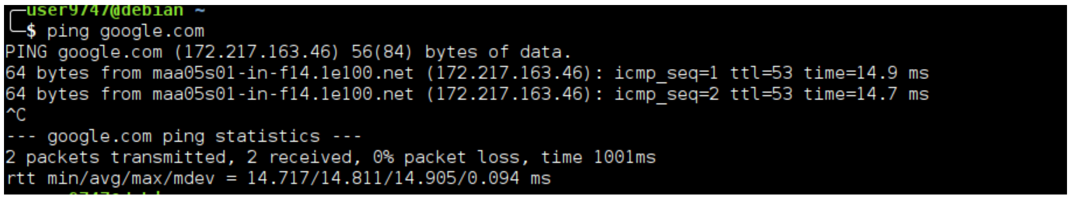
ifconfig stands for ”interface configuration”. It is used to view and change the configuration of the network interfaces on your system.



```
┌─user9747@debian ~
└─$ sudo ifconfig
enp3s0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether 70:5a:0f:b3:1a:68  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1  (Local Loopback)
        RX packets 921  bytes 225107 (219.8 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 921  bytes 225107 (219.8 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

wlo1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.9  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::c12b:3ca0:8242:fce4  prefixlen 64  scopeid 0x20<link>
        ether 44:1c:a8:6a:bb:69  txqueuelen 1000  (Ethernet)
        RX packets 29947  bytes 25093794 (23.9 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 24396  bytes 3458874 (3.2 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

2.2.2 ping

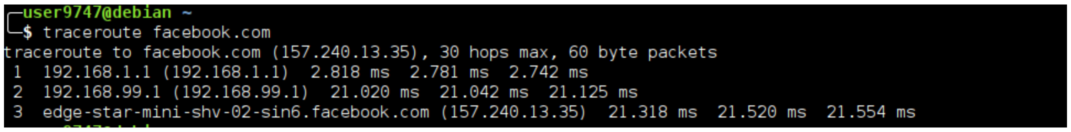
Ping is used to check the connectivity status between a source and a destination computer/device over an IP network. It also helps you assess the time it takes to send and receive a response from the network.



```
┌─user9747@debian ~
└─$ ping google.com
PING google.com (172.217.163.46) 56(84) bytes of data.
64 bytes from maa05s01-in-f14.1e100.net (172.217.163.46): icmp_seq=1 ttl=53 time=14.9 ms
64 bytes from maa05s01-in-f14.1e100.net (172.217.163.46): icmp_seq=2 ttl=53 time=14.7 ms
^C
--- google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 14.717/14.811/14.905/0.094 ms
```

2.2.3 traceroute

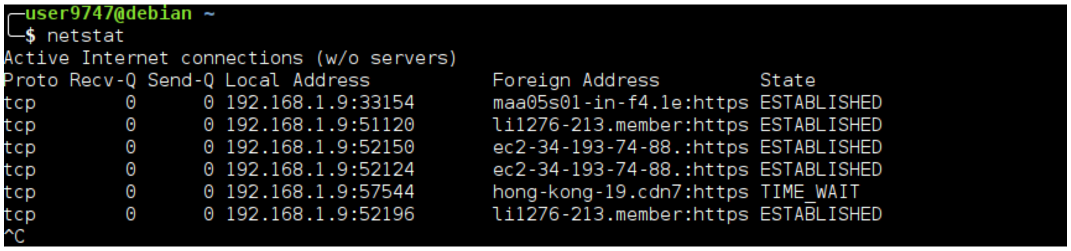
is a tool used to find the route taken by packets over an IP network and the delays it has.



```
┌─user9747@debian ~
└─$ traceroute facebook.com
traceroute to facebook.com (157.240.13.35), 30 hops max, 60 byte packets
 1  192.168.1.1 (192.168.1.1)  2.818 ms  2.781 ms  2.742 ms
 2  192.168.99.1 (192.168.99.1)  21.020 ms  21.042 ms  21.125 ms
 3  edge-star-mini-shv-02-sin6.facebook.com (157.240.13.35)  21.318 ms  21.520 ms  21.554 ms
```

2.2.4 netstat

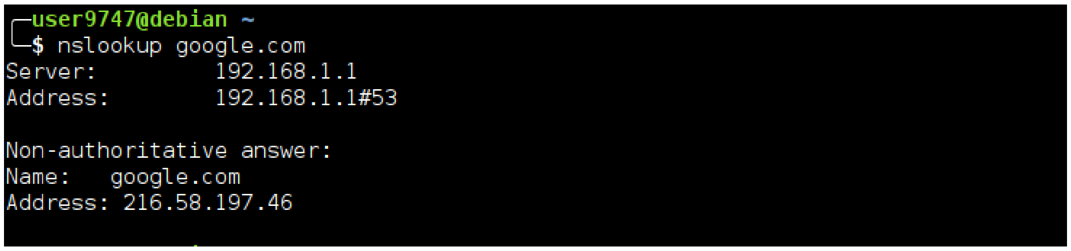
Prints network connection,ip tables and network interface statistics and information about the Linux networking subsystem.



```
┌─user9747@debian ~
└─$ netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 192.168.1.9:33154       maa05s01-in-f4.1e:https ESTABLISHED
tcp        0      0 192.168.1.9:51120       li1276-213.member:https ESTABLISHED
tcp        0      0 192.168.1.9:52150       ec2-34-193-74-88.:https ESTABLISHED
tcp        0      0 192.168.1.9:52124       ec2-34-193-74-88.:https ESTABLISHED
tcp        0      0 192.168.1.9:57544       hong-kong-19.cdn7:https TIME_WAIT
tcp        0      0 192.168.1.9:52196       li1276-213.member:https ESTABLISHED
^C
```

2.2.5 nslookup

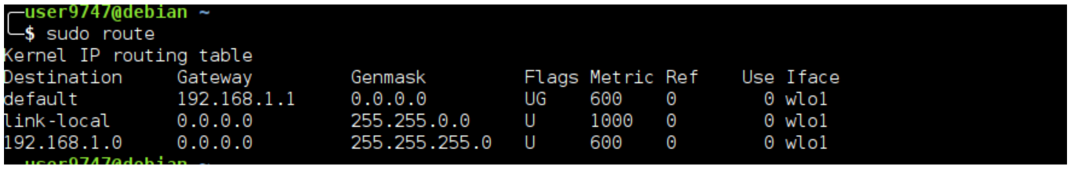
is a tool used to find DNS lookups in Linux and prints ip address of the specific device,MX records and domain name ip address mapping.



```
┌─user9747@debian ~
└─$ nslookup google.com
Server:         192.168.1.1
Address:        192.168.1.1#53

Non-authoritative answer:
Name:   google.com
Address: 216.58.197.46
```

2.2.6 route

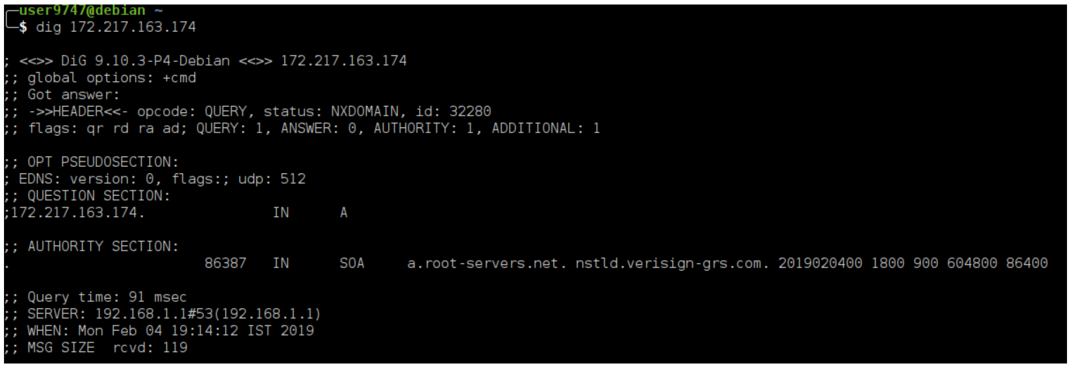
used to show and manipulate routing tables.When the add or del options are used, route modifies the routing tables. Without these options, route displays the current con-tents of the routing tables.



```
┌─user9747@debian ~
└─$ sudo route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         192.168.1.1     0.0.0.0         UG    600    0        0 wlo1
link-local      0.0.0.0         255.255.0.0     U     1000   0        0 wlo1
192.168.1.0     0.0.0.0         255.255.255.0   U     600    0        0 wlo1
```

2.2.7 dig

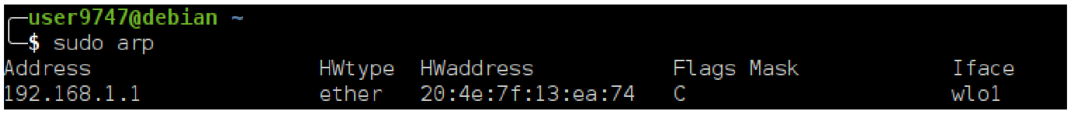
A DNS lookup tool used to query and troubleshoot DNS problems.



```
┌─user9747@debian ~
└─$ dig 172.217.163.174

; <<>> DiG 9.10.3-P4-Debian <<>> 172.217.163.174
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 32280
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;172.217.163.174.               IN      A

;; AUTHORITY SECTION:
.                       86387   IN      SOA     a.root-servers.net. nstld.verisign-grs.com. 2019020400 1800 900 604800 86400

;; Query time: 91 msec
;; SERVER: 192.168.1.1#53(192.168.1.1)
;; WHEN: Mon Feb 04 19:14:12 IST 2019
;; MSG SIZE  rcvd: 119
```

2.2.8 arp

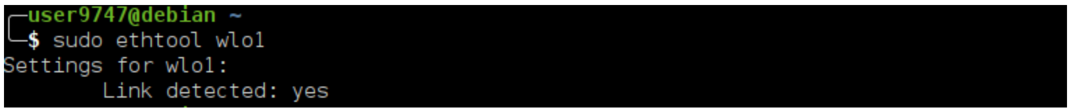
Tool used to show and modify ARP(Address resolution protocol) cache.An ARP cache is a simple mapping of IP addresses to MAC addresses.



```
┌─user9747@debian ~
└─$ sudo arp
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   20:4e:7f:13:ea:74   C                     wlo1
```

2.2.9 ethtool

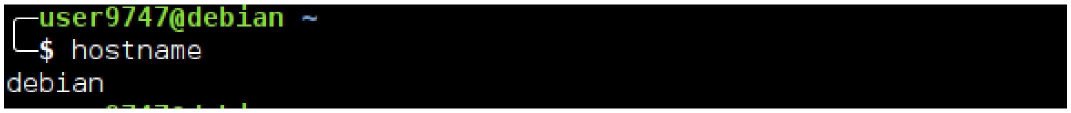
is Network Interface card configuration tool.Used to change NIC settings.



```
┌─user9747@debian ~
└─$ sudo ethtool wlo1
Settings for wlo1:
        Link detected: yes
```

2.2.10 hostname

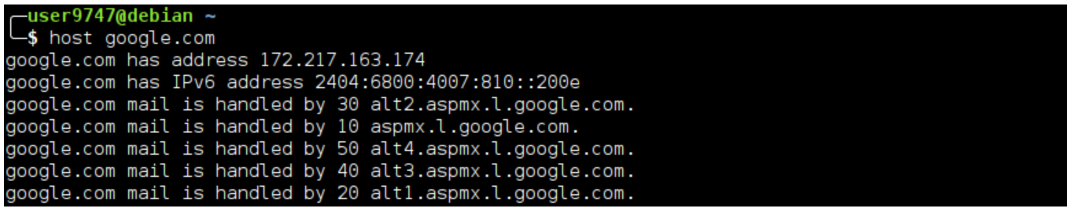
used to display systems DNS name and display and change its hostname.



```
┌─user9747@debian ~
└─$ hostname
debian
```

2.2.11 host

A DNS lookup tool.Normally used to convert names to ip addresses.When no arguments or options are given, host prints a short summary of its command line arguments and options.



```
┌─user9747@debian ~
└─$ host google.com
google.com has address 172.217.163.174
google.com has IPv6 address 2404:6800:4007:810::200e
google.com mail is handled by 30 alt2.aspmx.l.google.com.
google.com mail is handled by 10 aspmx.l.google.com.
google.com mail is handled by 50 alt4.aspmx.l.google.com.
google.com mail is handled by 40 alt3.aspmx.l.google.com.
google.com mail is handled by 20 alt1.aspmx.l.google.com.
```

2.3 Result

Successfully executed above commands and output verified using (zsh) Debian 4.9 Kernel4.9.0-6-amd64

3 Process And Thread

3.1 Aim

To familiarize and implement programs related to Process and thread.

3.2 Theory

A thread is a path of execution within a process. A process can contain multiple threads.A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads.A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section and OS resources like open files and signals. But, like process, a thread has its own program counter (PC), a register set, and a stack space

3.3 Algorithm

Algorithm 1 Algorithm for creating N threads

```
START
Let NUM_THREAD=10
Procedure *MyFunction(void * tid)
Begin
        Print("Thread : " tid)
End    MyFunction
create pthread thread
f=fork() //create fork
IF(f==0)
        PRINT "child : pid"
        Begin For i<NUM_THREAD
                call pthread_create(&thread,NULL,MyFunction,tid)
        i++
        END FOR
ELSE
        PRINT "parent : pid"
        Begin For i<NUM_THREAD
                call pthread_create(&thread,NULL,MyFunction,tid)
        i++
        END FOR
ENDIF
STOP
```

3.4 Program

```
#include<iostream>
#include<cstdlib>
#include<pthread.h>
#include<sys/types.h>
#include<unistd.h>
using namespace std;

#define NUM_THREAD 4

void *Printer(void *threadId){
        printf("Thread %d of process %d\n",threadId,getpid());
        pthread_exit(NULL);
}

int main(){
    pthread_t thread;
    int rc;
    int f=fork();
    if(f==0){
        printf("cHilD is %d \n",getpid());
        for(int i=0;i<NUM_THREAD;i++){
                rc = pthread_create(&thread,NULL,Printer,(void *)i);
                if(rc){
                        cout<<"ErrOR";
                }
        }
    }
    else{
        printf("pAreNt is %d \n",getpid());
        for(int i=0;i<NUM_THREAD;i++){
                rc = pthread_create(&thread,NULL,Printer,(void *)i);
                if(rc){
                        cout<<"ErrOR";
                }
        }

    }
    pthread_exit(NULL);

}
```

3.5 Output

```
pAreNt is 4105
cHilD is 4106
```

```
Thread 0 of process 4105
Thread 1 of process 4106
Thread 1 of process 4105
Thread 0 of process 4106
Thread 3 of process 4106
Thread 2 of process 4106
Thread 3 of process 4105
Thread 2 of process 4105
```

3.6 Result

Implemented thread in C compiled on gcc 6.3.0 and executed on Debian 4.9 Kernel 4.9 and outputs were verified.

4 First Readers-Writers Problem

4.1 Aim

To implement First Readers-Writers Problem.

4.2 Theory

4.2.1 Introduction

It is possible to protect the shared data behind a mutual exclusion mutex, in which case no two threads can access the data at the same time. However, this solution is suboptimal, because it is possible that a reader R1 might have the lock, and then another reader R2 requests access. It would be foolish for R2 to wait until R1 was done before starting its own read operation; instead, R2. This is the motivation for the first readers-writers problem, in which the constraint is added that no reader shall be kept waiting if the share is currently opened for reading. This is also called readers-preference

4.2.2 Solution

In this solution of the readers/writers problem, the first reader must lock the resource (shared file) if such is available. Once the file is locked from writers, it may be used by many subsequent readers without having them to re-lock it again.

Before entering the critical section, every new reader must go through the entry section. However, there may only be a single reader in the entry section at a time. This is done to avoid race conditions on the readers (e.g. two readers increment the readcount at the same time, and both try to lock the resource, causing one reader to block). To accomplish this, every reader which enters the ¡ENTRY Section¿ will lock the ¡ENTRY Section¿ for themselves until they are done with it. At this point the readers are not locking the resource. They are only locking the entry section so no other reader can enter it while they are in it. Once the reader is done executing the entry section, it will unlock it by signalling the mutex. Signalling it is equivalent to: mutex.V() in the above code. Same is valid for the ¡EXIT Section¿. There can be no more than a single reader in the exit section at a time, therefore, every reader must claim and lock the Exit section for themselves before using it.

4.3 Algorithm

Algorithm 2 Algorithm First Readers Writers Problem

[h]

```
semaphore resource=1;
semaphore rmutex=1;
readcount=0;
procedure WRITER
resource.P(); . resource.P() is equivalent to wait(resource)
<CRITICAL Section>
<EXIT Section>
resource.V();
end procedure
procedure READER
rmutex.P(); . rmutex.P() is equivalent to wait(rmutex)
<CRITICAL Section>
readcount++;
if readcount == 1 then
resource.P();
end if
<EXIT CRITICAL Section>
rmutex.V(); . resource.V() is equivalent to signal(resource)
rmutex.P();
<CRITICAL Section>
 readcount ;
if readcount == 0) then
resource.V();
end if
<EXIT CRITICAL Section>
rmutex.V(); . rmutex.V() is equivalent to signal(rmutex)
end procedure
```

4.4 Program

```
#include<iostream>
#include<semaphore.h>
#include<cstdlib>
#include<pthread.h>
#include<unistd.h>
#include<time.h>

using namespace std;
pthread_mutex_t rmutex;
pthread_mutex_t resources;
int readcount=0;

void *Writer(void* arg){
    long t=(long)arg;
    pthread_mutex_lock(&resources);
    printf("Thread %d writing ....\n",t);
    printf("Thread %d finished writing\n",t);
    sleep(2);
    pthread_mutex_unlock(&resources);
    pthread_exit(NULL);
}

void *Reader(void* arg){
    long t=(long)arg;
    pthread_mutex_lock(&rmutex);
    readcount++;
    if(readcount==1){
        pthread_mutex_lock(&resources);
    }
    pthread_mutex_unlock(&rmutex);
    printf("Thread %d reading .....\n",t);
    sleep(3);
    printf("Thread %d finished reading\n",t);
    pthread_mutex_lock(&rmutex);
    readcount--;
    if(readcount==0){
        pthread_mutex_unlock(&resources);
    }
    pthread_mutex_unlock(&rmutex);
    pthread_exit(NULL);
}


int main(){
    pthread_t t1,t2,t3,t4;
    int rc;
    srand(time(0));
    for(int i=1;i<10;i++){
        if(rand()%2==0)
            rc=pthread_create(&t1,NULL,Reader,(void *)i);
        else
             rc=pthread_create(&t1,NULL,Writer,(void *)i);
    }

    pthread_exit(NULL);
}
```

4.5 Output

```
Thread 1 writing....
Thread 1 finished writing
Thread 2 reading.....
Thread 4 reading.....
Thread 5 reading.....
Thread 7 reading.....
Thread 8 reading.....
Thread 9 reading.....
Thread 2 finished reading
Thread 8 finished reading
Thread 5 finished reading
Thread 4 finished reading
Thread 7 finished reading
Thread 9 finished reading
Thread 3 writing....
Thread 3 finished writing
Thread 6 writing....
Thread 6 finished writing
```

4.6 Result

Implemented First Readers Writers Problem in C ,compiled on gcc 6.3.0 and executed on Debian 4.9 Kernel 4.9 and outputs were verified.

5 Second Readers-Writers problem

5.1 Aim

To Implement the Second Readers-Writers problem.

5.2 Theory

The first solution is suboptimal, because it is possible that a reader R1 might have the lock, a writer W be waiting for the lock, and then a reader R2 requests access. It would be unfair for R2 to jump in immediately, ahead of W; if that happened often enough, W would starve. Instead, W should start as soon as possible. This is the motivation for the second readers-writers problem, in which the constraint is added that no writer, once added to the queue, shall be kept waiting longer than absolutely necessary. This is also called writers-preference.

In this solution, preference is given to the writers. This is accomplished by forcing every reader to lock and release the readtry semaphore individually. The writers on the other hand don't need to lock it individually. Only the first writer will lock the readtry and then all subsequent writers can simply use the resource as it gets freed by the previous writer. The very last writer must release the readtry semaphore, thus opening the gate for readers to try reading.

5.3 Algorithm

Algorithm 3 Algorithm for second Readers Writers Problem

```
semaphore resource=1;
semaphore rmutex=1;
semaphore wmutex=1;
semaphore resource=1;
readcount=0;
writecount = 0;
procedure READER
<ENTRY Section>
readTry.P()
rmutex.P()
readcount++
if readcount == 1 then
resource.P()
end if
rmutex.V()
readTry.V()
<CRITICAL Section>
<EXIT Section>
rmutex.P()
readcount
if readcount == 0 then
resource.V()
end if
rmutex.V()
end procedure
```

5.4 Program

```
#include<iostream>
#include<semaphore.h>
#include<cstdlib>
#include<pthread.h>
#include<unistd.h>
#include<time.h>

using namespace std;
pthread_mutex_t rmutex;
pthread_mutex_t wmutex;
pthread_mutex_t readTry;
pthread_mutex_t resources;
int readcount=0;
int writecount=0;

void *Writer(void* arg){
    long t=(long)arg;
    pthread_mutex_lock(&wmutex);
    writecount++;
    if (writecount==1)
        pthread_mutex_lock(&readTry);
    pthread_mutex_unlock(&wmutex);
    pthread_mutex_lock(&resources);
    printf("Thread %d writing....\n",t);
    printf("Thread %d finished writing\n",t);
    sleep(2);
    pthread_mutex_unlock(&resources);
    pthread_mutex_lock(&wmutex);
    writecount--;
    if(writecount==0)
        pthread_mutex_unlock(&readTry);
    pthread_mutex_unlock(&wmutex);
    pthread_exit(NULL);
}

void *Reader(void* arg){
    long t=(long)arg;
    pthread_mutex_lock(&readTry);
    pthread_mutex_lock(&rmutex);
    readcount++;
    if(readcount==1){
        pthread_mutex_lock(&resources);
    }
    pthread_mutex_unlock(&rmutex);
    pthread_mutex_unlock(&readTry);
    printf("Thread %d reading.....\n",t);
    sleep(3);
    printf("Thread %d finished reading\n",t);
    pthread_mutex_lock(&rmutex);
    readcount--;
    if(readcount==0){
        pthread_mutex_unlock(&resources);
    }
    pthread_mutex_unlock(&rmutex);
    pthread_exit(NULL);
}


int main(){
    pthread_t t1,t2,t3,t4;
    int rc;
    srand(time(0));
    for(int i=1;i<10;i++){
        if(rand()%2==0)
            rc=pthread_create(&t1,NULL,Reader,(void *)i);
        else
             rc=pthread_create(&t1,NULL,Writer,(void *)i);
    }

    pthread_exit(NULL);
}
```

5.5 Output

```
Thread 1 reading.....
Thread 2 reading.....
Thread 1 finished reading
Thread 2 finished reading
Thread 3 writing....
Thread 3 finished writing
Thread 4 writing....
Thread 4 finished writing
Thread 5 writing....
Thread 5 finished writing
Thread 8 writing....
Thread 8 finished writing
Thread 9 writing....
Thread 9 finished writing
Thread 6 reading.....
Thread 7 reading.....
Thread 7 finished reading
Thread 6 finished reading
```

5.6 Result

Implemented second Readers Writers problem in cpp compiled on g++ 6.3.0 and executed on Debian 4.9 Kernel 4.9 and outputs were verified.

6 Inter Process Communication

6.1 Aim

Implement programs for Inter Process Communication using PIPE, Message Queue and Shared Memory.

6.2 Theory

6.2.1 Pipe

pipe() is used for passing information from one process to another. pipe() is unidirectional therefore, for two-way communication between processes, two pipes can be set up, one for each direction.

6.2.2 Named Pipe

Named Pipes : Named pipes provide a much more powerful communication tool. Communication can be bidirectional, and no parent-child relationship is required. Once a named pipe is established, several processes can use it for communication. In fact, in a typical scenario, a named pipe has several writers. Additionally, named pipes continue to exist after communicating processes have finished.

6.2.3 Message Queues

Message queues provide an asynchronous communications protocol, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time. Messages placed onto the queue are stored until the recipient retrieves them. Message queues have implicit or explicit limits on the size of data that may be transmitted in a single message and the number of messages that may remain outstanding on the queue.

6.2.4 Shared Memory

Inter Process Communication through shared memory is a concept where two or more process can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by anther process. A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

6.3 Program

6.3.1 Pipe

```
#include<iostream>
#include <sys/types.h>
#include<unistd.h>
using namespace std;

int main(){
    int pid,pip[2];
    char string[20];
    pipe(pip);

    pid =fork();
    if(pid==0){
        write(pip[1],"hello\0",6);
    }
    else{
        read(pip[0],string,6);
        printf("%s",string);
    }
}
```

Named Pipe

Writer

```
#include<iostream>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<string.h>
#include<unistd.h>

using namespace std;
int main(){
    int fd;
    mkfifo("fifo",0666);
    char arr[10],arr1[10];
    while(1){
        fd=open("fifo",O_WRONLY);
        fgets(arr, 10, stdin);
        write(fd,arr,strlen(arr));
        close(fd);

    }
}
```

Reader

```
#include<iostream>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<string.h>
#include<unistd.h>

using namespace std;
int main(){
    int fd;
    mkfifo("fifo",0666);
    char arr[10],arr1[10];
    while(1){
        fd=open("fifo",O_RDONLY);
        read(fd,arr,10);
        printf("%s",arr);
        close(fd);
    }
}
```

6.3.2 Message Queue

Writer

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct mesg_buffer {
        long mesg_type;
        char mesg_text[100];
} message;

int main()
{
        key_t key;
        int msgid;
        key = ftok("progfile", 65);

        msgid = msgget(key, 0666 | IPC_CREAT);
        message.mesg_type = 1;

        printf("Write Data : ");
        gets(message.mesg_text);

        msgsnd(msgid, &message, sizeof(message), 0);
        printf("Data send is : %s \n", message.mesg_text);

        return 0;
}
```

Reader

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct mesg_buffer {
        long mesg_type;
        char mesg_text[100];
} message;

int main()
{
        key_t key;
        int msgid;

        key = ftok("progfile", 65);

        msgid = msgget(key, 0666 | IPC_CREAT);
        msgrcv(msgid, &message, sizeof(message), 1, 0);
        printf("Data Received is : %s \n",message.mesg_text);
        msgctl(msgid, IPC_RMID, NULL);

        return 0;
}
```

6.3.3 Shared Memory

Writer

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
        key_t key = ftok("shmfile",65);
        int shmid = shmget(key,1024,0666|IPC_CREAT);
        char *str = (char*) shmat(shmid,(void*)0,0);

        printf("Data read from memory: %s\n",str);

        shmdt(str);
        shmctl(shmid,IPC_RMID,NULL);

        return 0;
}
```

Reader

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
        key_t key = ftok("shmfile",65);
        int shmid = shmget(key,1024,0666|IPC_CREAT);
        char *str = (char*) shmat(shmid,(void*)0,0);

        cout<<"Write Data : ";
        cin>>str;

        printf("Data written in memory: %s\n",str);
                shmdt(str);

        return 0;
}
```

6.4 Output

6.4.1 Pipe

```
$ g++ pipe.cpp -lpthread
$ ./a.out
hello from child
```

6.4.2 Named Pipe

```
$ g++ npipeReader.cpp -o reader
$ ./reader
```

```
hello
hey

$ g++ npipeWriter.cpp -o writer
$ ./writer
hello
hey
```

6.4.3 Message Queue

```
$ gcc msgWriter.c -o w
$ ./w
Write Data : hello
Data send is : hello

$ gcc msgReader.c -o r
$ ./r
Data Received is : hello
```

6.4.4 Shared Memory

```
$ g++ sharedMemWriter.cpp -o w
$ ./w
Write Data : hello
Data written in memory: hello

$ g++ sharedMemReader.cpp -o r
$ ./r
Data read from memory: hello
```

6.5 Result

Implemented programs for IPC using pipes,Message queue and Shared Memory in C++ compiled on g++ 6.3.0 and executed on Debian 4.9 Kernel 4.9 and outputs were verified.

7 Socket Programming-TCP

7.1 Aim

To Implement Client-Server communication using Socket Programming and TCP as transport layer protocol.

7.2 Theory

7.2.1 TCP

TCP (Transmission Control Protocol) is a standard that defines how to establish and maintain a network conversation via which application programs can exchange data. TCP works with the Internet Protocol (IP), which defines how computers send packets of data to each other. Together, TCP and IP are the basic rules defining the Internet. TCP is defined by the Internet Engineering Task Force (IETF) in the Request for Comment (RFC) standards document number 793.

7.2.2 Client, Server and Socket

• Server-A server is a software that waits for client requests and serves or processes them accordingly.
• Client- a client is requester of this service. A client program request for some resources to the server and server responds to that request.
• Socket- Socket is the endpoint of a bidirectional communications channel between server and client. Sockets may communicate within a process, between processes on the same machine, or between processes on different machines. For any communication with a remote program, we have to connect through a socket port.

7.3 Algorithm

7.3.1 Server

Algorithm 4 Algorithm for creating a tcp server

```
START
Create TCP SOCKET
Bind SOCKET to a PORT
Start listing at the binded PORT for connection from CLIENT
WHILE TRUE:
    ACCEPT connection from CLIENT
    RECEIVE message from CLIENT
    SEND message to CLIENT
STOP
```

7.3.2 Client

Algorithm 5 Algorithm for creating a tcp client

```
START
CREATE TCP SOCKET
Connect to server using IP and PORT
SEND message to server
RECEIVE message from server
STOP
```

7.4 Program

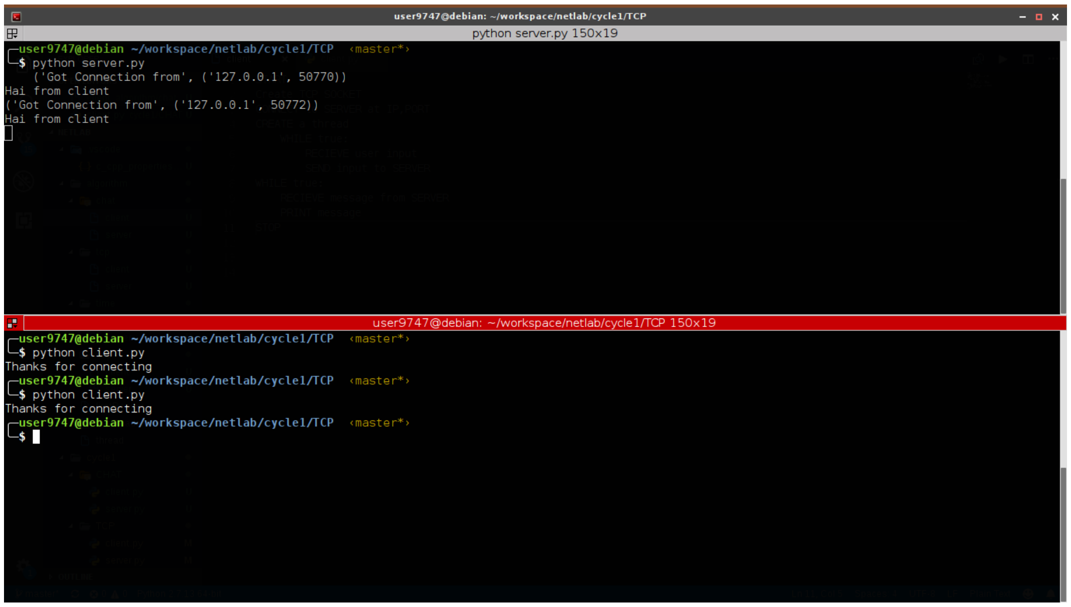
7.4.1 Server

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

port=8080

s.bind(('',port))

s.listen(5)

while True:
    c,addr=s.accept()
    message=c.recv(1024)
    print ('Got Connection from',addr)
    print message
    c.send('Thanks for connecting')
    c.close()
```

7.4.2 Client

```
import socket

s= socket.socket(socket.AF_INET, socket.SOCK_STREAM)

port = 8080

s.connect(('127.0.0.1',port))
s.send('Hai from client')
print s.recv(1024)

s.close()
```

7.5 Output



```
user9747@debian: ~/workspace/netlab/cycle1/TCP
python server.py 150x19
┌─user9747@debian ~/workspace/netlab/cycle1/TCP  ‹master*›
└─$ python server.py
    ('Got Connection from', ('127.0.0.1', 50770))
Hai from client
('Got Connection from', ('127.0.0.1', 50772))
Hai from client
```

```
user9747@debian: ~/workspace/netlab/cycle1/TCP 150x19
┌─user9747@debian ~/workspace/netlab/cycle1/TCP  ‹master*›
└─$ python client.py
Thanks for connecting
┌─user9747@debian ~/workspace/netlab/cycle1/TCP  ‹master*›
└─$ python client.py
Thanks for connecting
┌─user9747@debian ~/workspace/netlab/cycle1/TCP  ‹master*›
└─$
```

7.6 Result

Implemented TCP Socket Communication on Python 2.7.13 and executed on Debian 4.9 Kernel 4.9 and outputs were verified.
Server code creates a TCP socket using the socket library.Then binds the server to port 8080.The socket then listens for communications to this port.In a infinite while loop the server accepts the connection and address from connecting clients.Then it receives message from this connection and sends message on the same.After that the connection is closed.
Client code creates a TCP socket same as above.Then connects to the ip and port of the server.It then sends a message to the server.Then displays the message from server.Then closes the socket.

8 Socket Programming-UDP

8.1 Aim

To Implement Client-Server communication using Socket Programming and UDP as transport layer protocol.

8.2 Theory

8.2.1 UDP

UDP (User Datagram Protocol) is an alternative communications protocol to Transmission Control Protocol (udp) used primarily for establishing low-latency and loss-tolerating connections between applications on the internet. It is a process to process communication. It is unreliable.

8.2.2 Client, Server and Socket

• Server-A server is a software that waits for client requests and serves or processes them accordingly.
• Client- a client is requester of this service. A client program request for some resources to the server and server responds to that request.
• Socket- Socket is the endpoint of a bidirectional communications channel between server and client. Sockets may communicate within a process, between processes on the same machine, or between processes on different machines. For any communication with a remote program, we have to connect through a socket port.

8.3 Algorithm

8.3.1 Server

Algorithm 6 Algorithm for creating a udp server

```
START
Create UDP SOCKET
Bind SOCKET to a PORT
WHILE TRUE:
    RECEIVE message and address(IP,PORT) from CLIENT
    SEND message to CLIENT address
STOP
```

8.3.2 Client

Algorithm 7 Algorithm for creating a udp client

```
START
CREATE UDP SOCKET
SEND message to server at IP and PORT
RECEIVE message and address from server
STOP
```

8.4 Program

8.4.1 Server

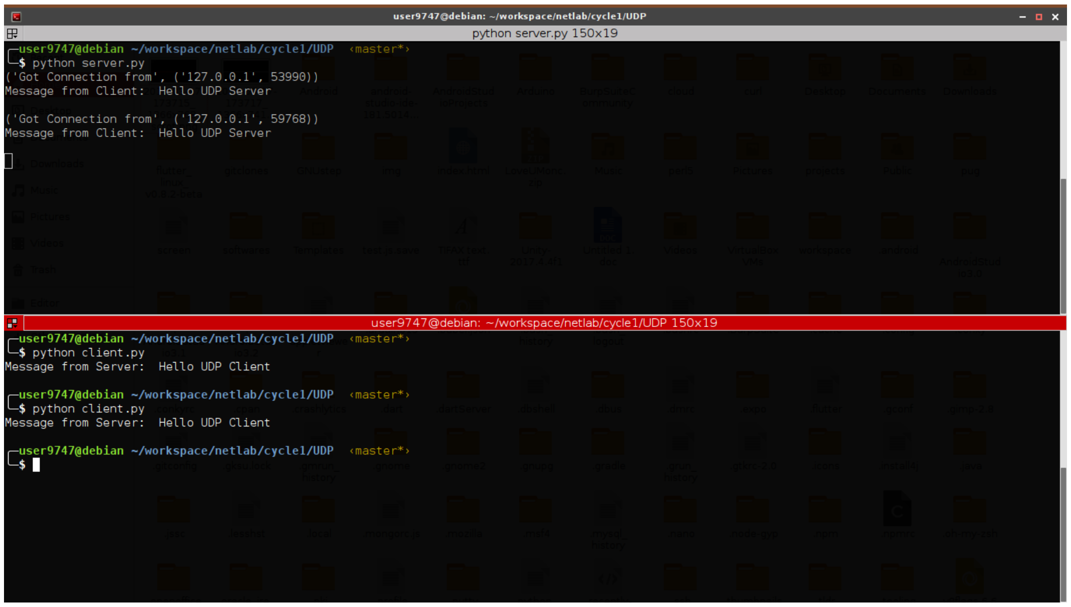
```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

port=8080

s.bind(('',port))


msgFromServer        = "Hello UDP Client\n"

bytesToSend          = str.encode(msgFromServer)

while True:
    msg,addr=s.recvfrom(1024)
    print ('Got Connection from',addr)
    print 'Message from Client: ',msg
    s.sendto(bytesToSend,addr)
```

8.4.2 Client

```
import socket

msgFromClient        = "Hello UDP Server\n"

bytesToSend          = str.encode(msgFromClient)

s= socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

port = 8080
```

```
s.sendto(bytesToSend,('127.0.0.1',port))
msgFromServer=s.recvfrom(1024)
# msg = "Message from Server:  {}".format(msgFromServer[0])
print"Message␣from␣Server:␣",msgFromServer[0]
s.close()
```

8.5 Output



user9747@debian: ~/workspace/netlab/cycle1/UDP
python server.py 150x19
user9747@debian: ~/workspace/netlab/cycle1/UDP 150x19

```
┌user9747@debian ~/workspace/netlab/cycle1/UDP  ‹master*›
└$ python server.py
('Got Connection from', ('127.0.0.1', 53990))
Message from Client:  Hello UDP Server

('Got Connection from', ('127.0.0.1', 59768))
Message from Client:  Hello UDP Server
```

```
┌user9747@debian ~/workspace/netlab/cycle1/UDP  ‹master*›
└$ python client.py
Message from Server:  Hello UDP Client

┌user9747@debian ~/workspace/netlab/cycle1/UDP  ‹master*›
└$ python client.py
Message from Server:  Hello UDP Client

┌user9747@debian ~/workspace/netlab/cycle1/UDP  ‹master*›
└$
```

8.6 Result

Implemented UDP Socket Communication on Python 2.7.13 and executed on Debian 4.9 Kernel 4.9 and outputs were verified.
Server code creates a udp socket using the socket library.Then binds the server to port 8080.In a infinite while loop the server receives message and address from sending clients.Then it sends message to this address.
Client code creates a UDP socket same as above.Then sends a message to the ip and port of the server.It then receives a message from the server.Then displays the message from the server.Then closes the socket.

9 Multi User Chat Server

9.1 Aim

Implement a multi user chat server using TCP as transport layer protocol.

9.2 Theory

9.2.1 Client, Server and Socket

• Server-A server is a software that waits for client requests and serves or processes them accordingly.
• Client- a client is requester of this service. A client program request for some resources to the server and server responds to that request.
• Socket- Socket is the endpoint of a bidirectional communications channel between server and client. Sockets may communicate within a process, between processes on the same machine, or between processes on different machines. For any communication with a remote program, we have to connect through a socket port.

9.2.2 Multithreading

A thread is sub process that runs a set of commands individually of any other thread. So, every time a user connects to the server, a separate thread is created for that user and communication from server to client takes place along individual threads based on socket objects created for the sake of identity of each client. We will require two scripts to establish the chat room. One to keep the serving running, and another that every client should run in order to connect to the server.

9.2.3 Server Side Script

The server side script will attempt to establish a socket and bind it to an IP address and port specified by the user. The script will then stay open and receive connection requests, and will append respective socket objects to a list to keep track of active connections. Every time a user connects, a separate thread will be created for that user. In each thread, the server awaits a message, and sends that message to other users currently on the chat. If the server encounters an error while trying to receive a message from a particular thread, it will exit that thread.

9.2.4 Client Side Script

The client side script will simply attempt to access the server socket created at the specified IP address and port. Once it connects, it will continuously check as to whether the input comes from the server or from the client, and accordingly redirects output. If the input is from the server, it displays the message on the terminal. If the input is from the user, it sends the message that the users enters to the server for it to be broadcasted to other users.This is the client side script, that each user must use in order to connect to the server.

9.3 Algorithm

9.3.1 Server

Algorithm 8 Algorithm for creating a tcp chat server

```
START
 Create TCP SOCKET
 Bind SOCKET to a PORT
 Start listing at the binded PORT for connection from CLIENTs
 append new CLIENT to list_clients
CREATE a new thread for each CLIENT which accepts messages from CLIENT and broad
STOP
```

9.3.2 Client

Algorithm 9 Algorithm for creating a tcp chat client

```
START
 Create TCP SOCKET
CONNECT to SERVER at IP ,PORT
CREATE a thread
    WHILE true:
        RECEIVE user input
        SEND input to SERVER
WHILE true:
    RECEIVE message from SERVER
    PRINT message
STOP
```

9.4 Program

9.4.1 Server

```
import socket
import select
import sys
from thread import *

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

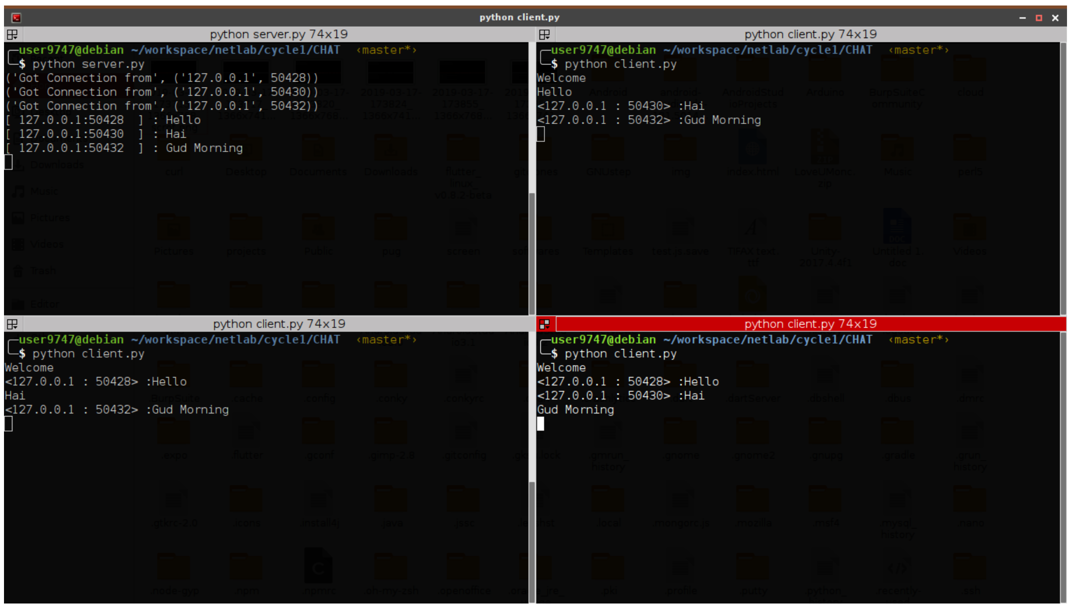
```
port=8000

s.bind(('',port))

s.listen(5)
list_of_clients = []
def clientThread(conn,addr):
    conn.send('Welcome')
    while True:
        message=conn.recv(1024)
        if message:
            print"[ "+ addr[0]+":"+str(addr[1]) +"  ] : "+message
            message_to_send = "<" + addr[0] +" : "+ str(addr[1]) + "> :" + messa
            broadcast(message_to_send,conn)
        else:
            remove(conn)

def broadcast(message,conn):
    for client in list_of_clients:
        if client!=conn:
            client.send(message)

def remove(conn):
    if conn in list_of_clients:
        list_of_clients.remove(conn)

while True:
    conn,addr=s.accept()
    print ('Got Connection from',addr)
    list_of_clients.append(conn)
    start_new_thread(clientThread,(conn,addr))


conn.close()
```

9.4.2 Client

```
import socket
import select
import sys
from thread import *

s= socket.socket(socket.AF_INET, socket.SOCK_STREAM)

port = 8000
s.connect((sys.argv[1],port))
```

```
def sendMessage(so):
    while True:
        message=raw_input()
        so.send(message)

start_new_thread(sendMessage,(s,))
while True:
    data = s.recv(1024)
    if data:
        print data


s.close()
```

9.5 Output



```
user9747@debian ~/workspace/netlab/cycle1/CHAT  ‹master*›
$ python server.py
('Got Connection from', ('127.0.0.1', 50428))
('Got Connection from', ('127.0.0.1', 50430))
('Got Connection from', ('127.0.0.1', 50432))
[ 127.0.0.1:50428  ] : Hello
[ 127.0.0.1:50430  ] : Hai
[ 127.0.0.1:50432  ] : Gud Morning
```

```
user9747@debian ~/workspace/netlab/cycle1/CHAT  ‹master*›
$ python client.py
Welcome
Hello
<127.0.0.1 : 50430> :Hai
<127.0.0.1 : 50432> :Gud Morning
```

```
user9747@debian ~/workspace/netlab/cycle1/CHAT  ‹master*›
$ python client.py
Welcome
<127.0.0.1 : 50428> :Hello
Hai
<127.0.0.1 : 50432> :Gud Morning
```

```
user9747@debian ~/workspace/netlab/cycle1/CHAT  ‹master*›
$ python client.py
Welcome
<127.0.0.1 : 50428> :Hello
<127.0.0.1 : 50430> :Hai
Gud Morning
```

9.6 Result

Implemented TCP Multi Client Chat Server on Python 2.7.13 and executed on Debian 4.9 Kernel 4.9 and outputs were verified.
Server code creates a TCP socket using the socket library.Then binds the server to port 8080.The socket then listens for communications to this port.In a infinite while loop the server accepts the connection and address from connecting clients.These connections appended to list_clients.A thread is created for each of the clients which accepts messages from clients in its own thread and then broadcasts the message to all clients in list_clients.

Client code creates a TCP socket same as above.Then connects to the ip and port of the server.It then creates a thread for an infinite loop for getting user input and sending this message to server.The parent thread is also a infinite while loop that displays messages it receives from the server.

10 Concurrent Time Server-UDP

10.1 Aim

Implement Concurrent Time Server application using UDP to execute the program at remote server. Client sends a time request to the server, server sends its system time back to the client. Client displays the result.

10.2 Theory

10.2.1 Time Server

Implement Concurrent Time Server application using UDP to execute the program at remote server. Client sends a time request to the server, server sends its system time back to the client. Client displays the result. We have a UDP based application which sends back current system time to the server indicating that some operation has been performed at the server.

10.2.2 Client, Server and Socket

• Server-A server is a software that waits for client requests and serves or processes them accordingly.
• Client- a client is requester of this service. A client program request for some resources to the server and server responds to that request.
• Socket- Socket is the endpoint of a bidirectional communications channel between server and client. Sockets may communicate within a process, between processes on the same machine, or between processes on different machines. For any communication with a remote program, we have to connect through a socket port.

10.3 Algorithm

10.3.1 Server

Algorithm 10 Algorithm for creating a concurrent server

```
START
Create UDP SOCKET
Bind SOCKET to a PORT
WHILE TRUE:
    RECEIVE time request and address(IP,PORT) from CLIENT
    SEND time to CLIENT address
STOP
```

10.3.2 Client

Algorithm 11 Algorithm for creating a udp client

```
START
CREATE UDP SOCKET
SEND time request to server at IP and PORT
RECEIVE time and address from server
STOP
```

10.4 Program

10.4.1 Server

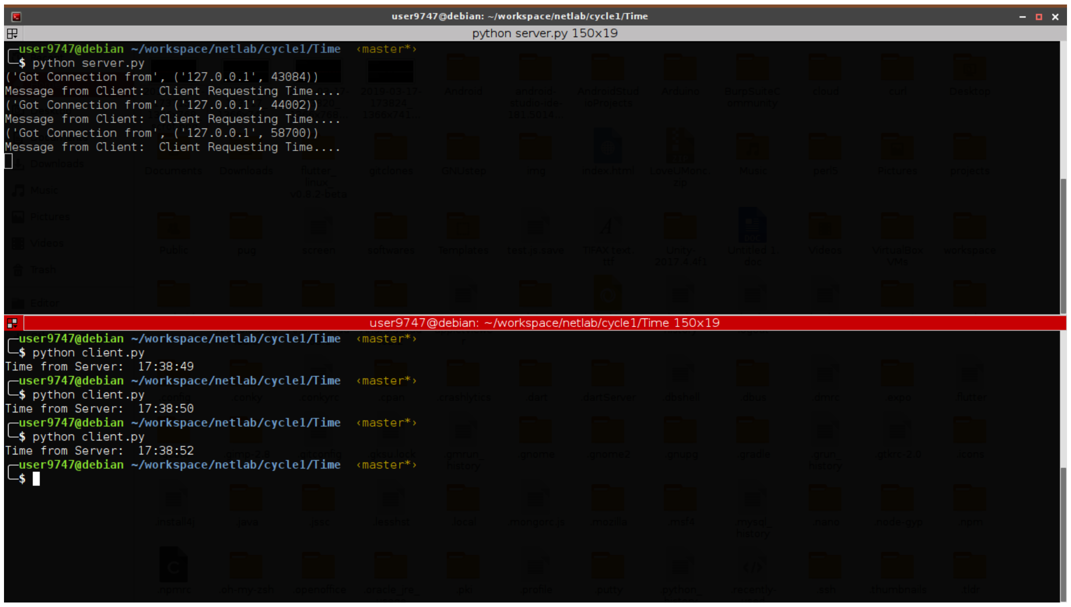
```
import socket
import datetime


s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

port=8080

s.bind(('',port))
while True:
    msg,addr=s.recvfrom(1024)
    print ('Got Connection from',addr)
    print 'Message from Client: ',msg
    now = datetime.datetime.now()
    time = now.strftime("%H:%M:%S")
    bytesToSend    = str.encode(time)
    s.sendto(bytesToSend,addr)
```

10.4.2 Client

```
import socket

msgFromClient        = "Client Requesting Time...."

bytesToSend          = str.encode(msgFromClient)

s= socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

port = 8080

s.sendto(bytesToSend,('127.0.0.1',port))
```

```
msgFromServer=s.recvfrom(1024)
# msg = "Message from Server: {}".format(msgFromServer[0])
print"Time from Server: ",msgFromServer[0]
s.close()
```

10.5 Output



user9747@debian: ~/workspace/netlab/cycle1/Time

python server.py 150x19

```
┌user9747@debian ~/workspace/netlab/cycle1/Time  ‹master*›
└$ python server.py
('Got Connection from', ('127.0.0.1', 43084))
Message from Client:  Client Requesting Time....
('Got Connection from', ('127.0.0.1', 44002))
Message from Client:  Client Requesting Time....
('Got Connection from', ('127.0.0.1', 58700))
Message from Client:  Client Requesting Time....
```

user9747@debian: ~/workspace/netlab/cycle1/Time 150x19

```
┌user9747@debian ~/workspace/netlab/cycle1/Time  ‹master*›
└$ python client.py
Time from Server:  17:38:49
┌user9747@debian ~/workspace/netlab/cycle1/Time  ‹master*›
└$ python client.py
Time from Server:  17:38:50
┌user9747@debian ~/workspace/netlab/cycle1/Time  ‹master*›
└$ python client.py
Time from Server:  17:38:52
┌user9747@debian ~/workspace/netlab/cycle1/Time  ‹master*›
└$
```

10.6 Result

Implemented UDP Concurrent time server on Python 2.7.13 and executed on Debian 4.9 Kernel 4.9 and outputs were verified.
Server code creates a udp socket using the socket library.Then binds the server to port 8080.In a infinite while loop the server receives time request and address from sending clients.Then it sends the current time to this address.
Client code creates a UDP socket same as above.Then sends a time request to the ip and port of the server.It then receives the server time from the server.Then displays the time from the server.Then closes the socket.

11 Distance vector routing protocol

11.1 Aim

Implement and simulate algorithm for Distance vector routing protocol.

11.2 Theory

A distance-vector routing (DVR) protocol requires that a router inform its neighbors of topology changes periodically. Each router maintains a Distance Vector table containing the distance between itself and all possible destination nodes. Distances,based on a chosen metric, are computed using information from the neighbors' distance vectors. DVR uses Bellman-Ford's Algorithm.

11.3 Algorithm

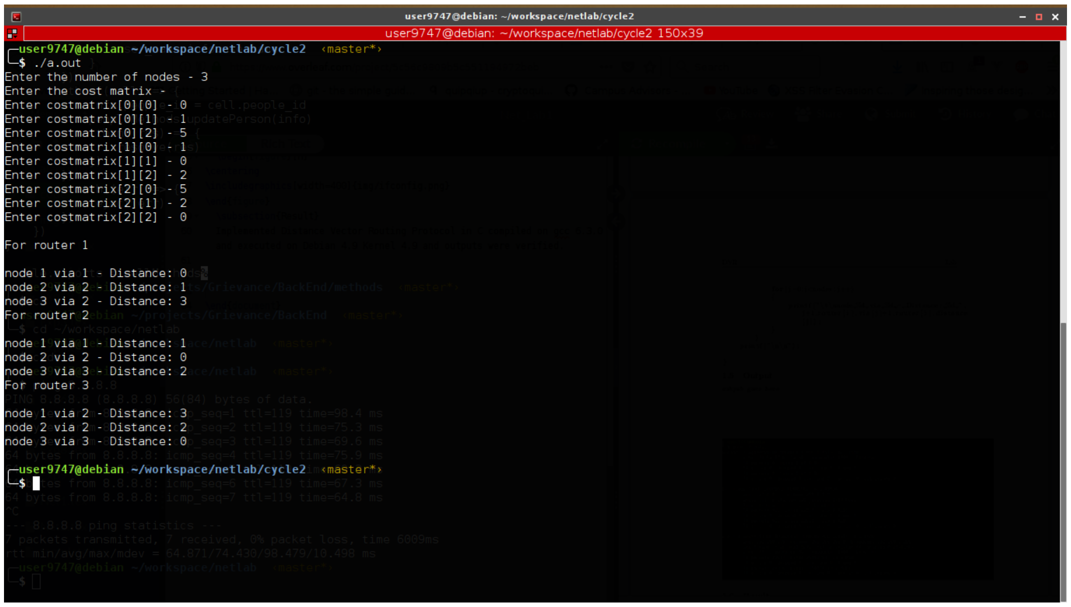
Algorithm 12 Algorithm for Distance Vector Routing Protocol

```
Input a matrix, cost of size N X N.
Initialize a matrix, d of size N X N with values of cost.
Iterate through each node i.
    Iterate through each node j.
* Iterate through each node k.
       d[i][j]=min(d[i][j],cost[i][k]+d[k][j])

Display d[i][1..N] for all node i.
```

11.4 Program

```c
#include<stdio.h>
struct node
{
    unsigned distance[20];
    unsigned via[20];
}router[10];
int main()
{
    int costmatrix[20][20];
    int nodes,i,j,k,count=0;
    printf("Enter␣the␣number␣of␣nodes␣−␣");
    scanf("%d",&nodes);//Enter the nodes
    printf("Enter␣the␣cost␣matrix␣−\n");
    for(i=0;i<nodes;i++)
    {
        for(j=0;j<nodes;j++)
        {
            printf("Enter␣costmatrix[%d][%d]␣−␣",i,j);
            scanf("%d",&costmatrix[i][j]);
            costmatrix[i][i]=0;
            router[i].distance[j]=costmatrix[i][j];//initialising the distance e
            router[i].via[j]=j;//initialising the via part
        }
    }
        do
        {
            count=0;
            for(i=0;i<nodes;i++)
            for(j=0;j<nodes;j++)
            for(k=0;k<nodes;k++)
                if(router[i].distance[j]>costmatrix[i][k]+router[k].distance[j])
                {//Calculating the minimum distance
                    router[i].distance[j]=router[i].distance[k]+router[k].distan
                    router[i].via[j]=k;
                    count++;
                }
        }while(count!=0);
        for(i=0;i<nodes;i++)
        {
            printf("\nFor␣router␣%d\n",i+1);
            for(j=0;j<nodes;j++)
            {
                printf("\t\nnode␣%d␣via␣%d␣−␣Distance:␣%d␣",j+1,router[i].via[j]
            }
        }
    printf("\n\n");

}
```

11.5 Output



```
user9747@debian: ~/workspace/netlab/cycle2
user9747@debian: ~/workspace/netlab/cycle2 150x39
┌user9747@debian ~/workspace/netlab/cycle2  ‹master*›
└$ ./a.out
Enter the number of nodes - 3
Enter the cost matrix -
Enter costmatrix[0][0] - 0
Enter costmatrix[0][1] - 1
Enter costmatrix[0][2] - 5
Enter costmatrix[1][0] - 1
Enter costmatrix[1][1] - 0
Enter costmatrix[1][2] - 2
Enter costmatrix[2][0] - 5
Enter costmatrix[2][1] - 2
Enter costmatrix[2][2] - 0

For router 1

node 1 via 1 - Distance: 0
node 2 via 2 - Distance: 1
node 3 via 2 - Distance: 3
For router 2

node 1 via 1 - Distance: 1
node 2 via 2 - Distance: 0
node 3 via 3 - Distance: 2
For router 3

node 1 via 2 - Distance: 3
node 2 via 2 - Distance: 2
node 3 via 3 - Distance: 0

┌user9747@debian ~/workspace/netlab/cycle2  ‹master*›
└$
```

11.6 Result

Implemented Distance Vector Routing Protocol in C compiled on gcc 6.3.0 and executed on Debian 4.9 Kernel 4.9 and outputs were verified.

12 Link state routing protocol

12.1 Aim

Implement and simulate algorithm for Link state routing protocol.

12.2 Theory

Link-State Routing protocol is a main class of routing protocols. It is performed by every switching node/router in the network. The basic concept of link-state routing is that every node constructs a map of the connectivity to the network, in the form of a Graph, showing which nodes are connected to which other nodes. Each node then independently calculates the next best logical path from it to every possible destination in the network. The collection of best paths will then form the node's routing table.

12.3 Algorithm

Algorithm 13 Algorithm for Link State Protocol

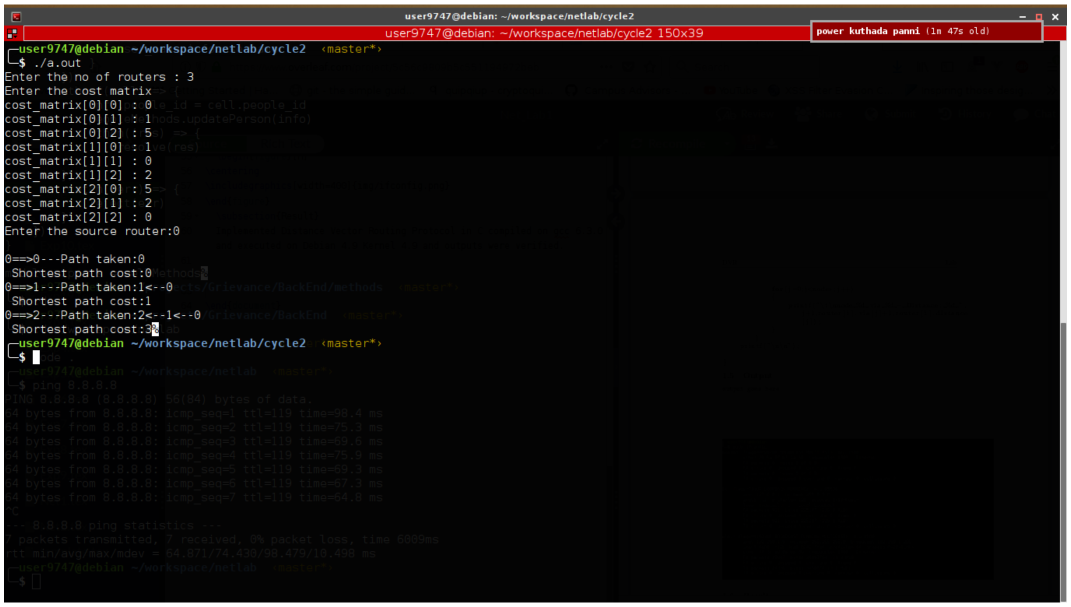
```
Add u to vector,  N  .
Input cost matrix, c.
for all node v
    if v is a neighbour of u
    * D[v]=c[u][v]
    else
    * D[v]=
Iterate till size of  N   becomes N (no of nodes in network)
    Find a node w not in  N   such that D(w) is minimum
    Add w to  N
    Update D[v] for each neighbour v of w and not in  N
* D[v]=min(D[v],D[w]+c[w][v])
print d[v] for all node v
```

12.4 Program

```
#include <stdio.h>
#include <string.h>
int main()
{
    int count,source,i,j,k,w,v,min;
    int cost_matrix[100][100],distance[100],last[100];
    int flag[100];
    printf("Enter the no of routers : ");
    scanf("%d",&count);
    printf("Enter the cost matrix\n");
    for(i=0;i<count;i++)
    {
        for(j=0;j<count;j++)
        {
            printf("cost_matrix[%d][%d] : ",i,j);
            scanf("%d",&cost_matrix[i][j]);
            if(cost_matrix[i][j]<0)cost_matrix[i][j]=1000;
        }
    }
    printf("Enter the source router:");
    scanf("%d",&source);
    for(v=0;v<count;v++)
    {
        flag[v]=0;
        last[v]=source;
        distance[v]=cost_matrix[source][v];
    }
    flag[source]=1;
    for(i=0;i<count;i++)
    {
        min=1000;
        for(w=0;w<count;w++)
        {
            if(!flag[w])
                if(distance[w]<min)
                {
                    v=w;
                    min=distance[w];
                }
            }
            flag[v]=1;
            for(w=0;w<count;w++)
            {
                if(!flag[w])
```

```
                    if(min+cost_matrix[v][w]<distance[w])
                    {
                        distance[w]=min+cost_matrix[v][w];
                        last[w]=v;
                    }
                }
            }
            for(i=0;i<count;i++)
            {
                printf("\n%d==>%d---Path taken:%d",source,i,i);
                w=i;
                while(w!=source)
                {
                    printf("<--%d",last[w]);w=last[w];
                }
                printf("\n Shortest path cost:%d",distance[i]);
            }
        }
```

12.5 Output



```
user9747@debian ~/workspace/netlab/cycle2  ‹master*›
$ ./a.out
Enter the no of routers : 3
Enter the cost matrix
cost_matrix[0][0] : 0
cost_matrix[0][1] : 1
cost_matrix[0][2] : 5
cost_matrix[1][0] : 1
cost_matrix[1][1] : 0
cost_matrix[1][2] : 2
cost_matrix[2][0] : 5
cost_matrix[2][1] : 2
cost_matrix[2][2] : 0
Enter the source router:0

0==>0---Path taken:0
 Shortest path cost:0
0==>1---Path taken:1<--0
 Shortest path cost:1
0==>2---Path taken:2<--1<--0
 Shortest path cost:3%
user9747@debian ~/workspace/netlab/cycle2  ‹master*›
$
```

12.6 Result

Implemented Link State Routing Protocol in C compiled on gcc 6.3.0 and executed on Debian 9.4 Kernel 4.9 and outputs were verified.

13 Simple Mail Transfer Protocol

13.1 Aim

To implement a subset of Simple Mail transfer Protocol using TCP.

13.2 Theory

SMTP is a simple ASCII protocol. This is not a weakness but a feature. Using ASCII text makes protocols easy to develop, test, and debug. They can be tested by sending commands manually, and records of the messages are easy to read. Most application-level Internet protocols now work this way (e.g., HTTP). After establishing the TCP connection to port 25, the sending machine, operating as the client, waits for the receiving machine, operating as the server, to talk first. The server starts by sending a line of text giving its identity and telling whether it is prepared to receive mail. If it is not, the client releases the connection and tries again later. If the server is willing to accept email, the client announces whom the email is coming from and whom it is going to. If such a recipient exists at the destination, the server gives the client the go-ahead to send the message. Then the client sends the message and the server acknowledges it. No checksums are needed because TCP provides a reliable byte stream. If there is more email, that is now sent. When all the email has been exchanged in both directions, the connection is released.

13.2.1 Protocol Overview

An SMTP Session consits of commands originated by an SMTP client and corresponding responses from the SMTP server so that the session is opened and session parameters are exchanged. A session may include zero or more SMTP transactions. A typical SMTP transaction consists of three command/reply sequences.

- 1. HELO command, to establish connection with server.
- 2. MAILFROM command, to establish the return address, also called return-path, reverse-path, bounce address, mfrom, or envelope sender.
- 3. RCPTTO command, to establish a recipient of the message. This command can be issued multiple times, one for each recipient. These addresses are also part of the envelope.
- 4. DATA to signal the beginning of the message text; the content of the message, as opposed to its envelope. It consists of a message header and a message body separated by an empty line. DATA is actually a group of commands, and the server replies twice: once to the DATA command itself, to acknowledge that it is ready to receive the text, and the second time after the end-of-data sequence, to either accept or reject the entire message.

13.3 Algorithm

Algorithm 14 Algorithm for SMTP server

```
START
Create TCP SOCKET
Bind SOCKET to a PORT
Start listing at the binded PORT for connection from CLIENT
WHILE TRUE:
    ACCEPT connection from CLIENT
    RECEIVE commands from CLIENT
    IF COMMAND == HELO
        RESPONSE:250
    ELSE IF COMMAND == MAILFROM
        RECEIVE mailid from CLIENT
        validate mailid
        IF valid
            RESPONSE:250 SENDER OK
        ELSE
            RESPONSE:421 SERVICE UNAVILABLE
   ELSE IF COMMAND == RCPTTO
        RECEIVE mailid from CLIENT
        validate mailid
        IF valid
            RESPONSE:250 RECIPIENT OK
        ELSE
            RESPONSE:421 RECIPIENT UNAVILABLE
    ELSE IF COMMAND == DATA
        IF HELO==250 & MAILFROM==250 &RCPTTO==250
            RESPONSE:354 Go Ahead, Enter data ending with <CRLF>.<CRLF>
            BREAK
        ELSE
            RESPONSE:421 SERVICE UNAVILABLE
READ message from CLIENT
IF COMMAND==QUIT
    STOP
```

Algorithm 15 Algorithm for SMTP client

```
START
CREATE TCP SOCKET
Connect to server using IP and PORT
WHILE INPUT != 'QUIT'
    READ commands from INPUT
    SEND commands to server
    RECEIVE response from server
STOP
```

13.4 Program

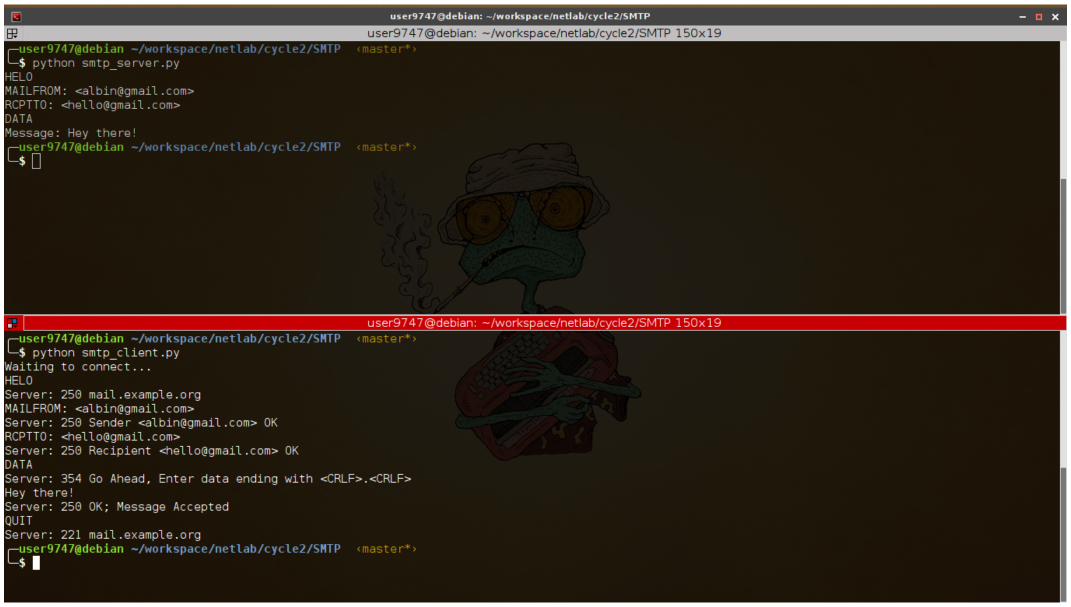
13.4.1 Server

```
import socket
import datetime
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.bind(("127.0.0.1",8080))
x=["albin@gmail.com","hello@gmail.com","jhondoe@gmail.com"]
s.listen(5)
conn,addr=s.accept()
while 1:
        mail=conn.recv(1024)
        tok=mail.split()
        print(mail)
        if tok[0]=='HELO':
                conn.send("250 mail.example.org")
        elif tok[0]=='MAILFROM:':
                flag1=0
                for i in x: #checking if mail is in the list
                        if tok[1]=="<"+i+">":
                                conn.send("250 Sender "+tok[1]+" OK")
                                flag1=1
                                break
                if flag1==0:
                        conn.send("421 Service Unavailable")
        elif tok[0]=='RCPTTO:':
                flag2=0
                for i in x: #checking if mail is in the list
                        if tok[1]=="<"+i+">":
                                conn.send("250 Recipient "+tok[1]+" OK")
                                flag2=1
                                break
                if flag2==0:
                    conn.send("421 Service Unavailable")
          elif tok[0]=='DATA':
                    flag3=0
                    if flag1==1 and flag2==1:
                              flag3=1
                              conn.send("354 Go Ahead, Enter data ending with <CRLF>.<
                              break
                    else:
                              conn.send("421 Service Unavailable")




if flag3==1:
          buff=conn.recv(2048)
          print("Message: "+buff)
          conn.send("250 OK; Message Accepted")
mail=conn.recv(1024)
tok=mail.split()
if tok[0]=='QUIT':
          conn.send("221 mail.example.org")
```

13.4.2 Client

```
import socket
import sys
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect(("127.0.0.1",8080))
print("Waiting to connect...")
while 1:
          comm=raw_input()
          s.send(comm)
          print("Server: "+s.recv(1024))
          if comm=='QUIT':
                    s.close()
                    break
```

13.5 Output



```
user9747@debian: ~/workspace/netlab/cycle2/SMTP
user9747@debian: ~/workspace/netlab/cycle2/SMTP 150x19
┌─user9747@debian ~/workspace/netlab/cycle2/SMTP  ‹master*›
└─$ python smtp_server.py
HELO
MAILFROM: <albin@gmail.com>
RCPTTO: <hello@gmail.com>
DATA
Message: Hey there!
┌─user9747@debian ~/workspace/netlab/cycle2/SMTP  ‹master*›
└─$
```

```
user9747@debian: ~/workspace/netlab/cycle2/SMTP 150x19
┌─user9747@debian ~/workspace/netlab/cycle2/SMTP  ‹master*›
└─$ python smtp_client.py
Waiting to connect...
HELO
Server: 250 mail.example.org
MAILFROM: <albin@gmail.com>
Server: 250 Sender <albin@gmail.com> OK
RCPTTO: <hello@gmail.com>
Server: 250 Recipient <hello@gmail.com> OK
DATA
Server: 354 Go Ahead, Enter data ending with <CRLF>.<CRLF>
Hey there!
Server: 250 OK; Message Accepted
QUIT
Server: 221 mail.example.org
┌─user9747@debian ~/workspace/netlab/cycle2/SMTP  ‹master*›
└─$
```

- Client sends a HELO command server replies with 250
- Client sends from address with MAILFROM command server verifies and sends 250 if everything is ok.
- Client sends to address like above using RCPTO command.
- Client sends DATA command server when its ready to recieve message with a 354 response.
- Client sends message to server.Server saves it and sends 250 response.

13.6 Result

Implemented SMTP using TCP in Python2.7 and executed on Debian 9.4 Kernel 4.9 and outputs were verified.

14 File Transfer Protocol

14.1 Aim

To implement a subset of File Transfer Protocol using TCP/IP.

14.2 Theory

In a typical FTP session, the user is sitting in front of one host (the local host) and wants to transfer files to or from a remote host. In order for the user to access the remote account, the user must provide a user identification and a password. After providing this authorization information, the user can transfer files from the local file system to the remote file system and vice versa. The user interacts with FTP through an FTP user agent. The user first provides the hostname of the remote host, causing the FTP client process in the local host to establish a TCP connection with the FTP server process in the remote host. The user then provides the user identification and password, which are sent over the TCP connection as part of FTP commands. Once the server has authorized the user, the user copies one or more files stored in the local file system into the remote file system (or vice versa).

Throughout a session, the FTP server must maintain state about the user. In par- ticular, the server must associate the control connection with a specific user account, and the server must keep track of the users current directory as the user wanders about the remote directory tree. Keeping track of this state information for each ongoing user session significantly constrains the total number of sessions that FTP can maintain simultaneously.

14.3 Algorithm

Algorithm 16 Algorithm for FTP server

```
START
Create TCP SOCKET
Bind SOCKET to a PORT
Start listing at the binded PORT for connection from CLIENT
WHILE TRUE:
    ACCEPT connection from CLIENT
    RECEIVE Filename from CLIENT
    IF FILE Found
        SEND FOUND
        READ FILE AND SEND IT TO CLIENT
STOP
```

Algorithm 17 Algorithm for FTP client

```
START
CREATE TCP SOCKET
Connect to server using IP and PORT
SEND Filename to server
IF response= FOUND
    CREATE A FILE
    RECEIVE packets from server until no more packets are send.
    WRITE packets to a FILE
STOP
```

14.4 Program

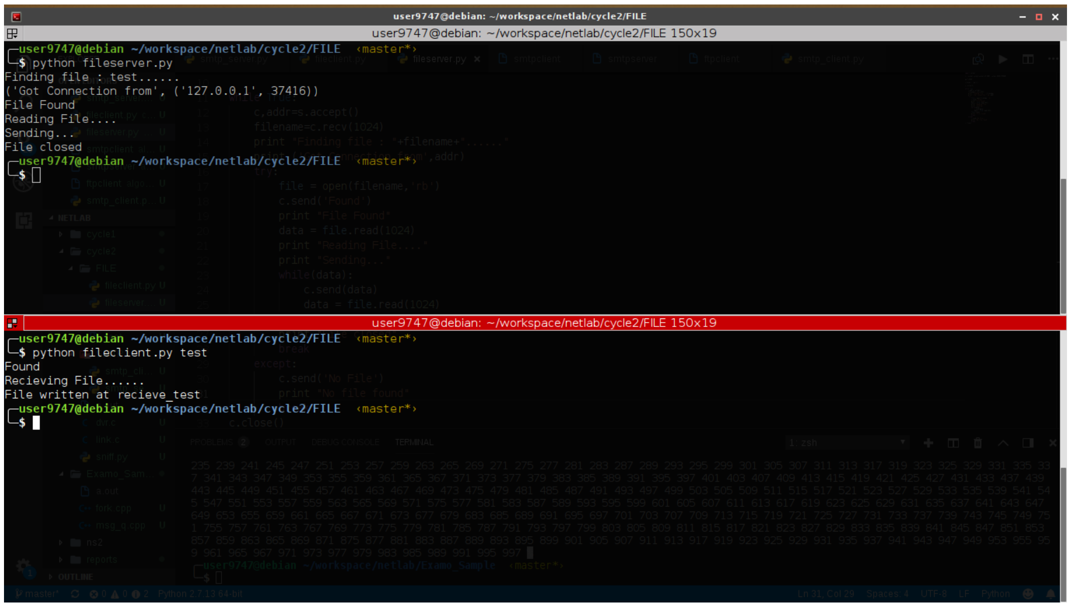
14.4.1 Server

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

port=8080

s.bind(('',port))

s.listen(5)

while True:
    c,addr=s.accept()
    filename=c.recv(1024)
    print "Finding file : "+filename+"......"
    print ('Got Connection from',addr)
    try:
        file = open(filename,'rb')
        c.send('Found')
        print "File Found"
        data = file.read(1024)
        print "Reading File...."
        print "Sending..."
        while(data):
            c.send(data)
            data = file.read(1024)
        file.close()
        print "File closed"
        break
    except:
        c.send('No File')
        print "No file found"
        break
c.close()
```

14.4.2 Client

```
import socket
import sys

s= socket.socket(socket.AF_INET, socket.SOCK_STREAM)

port = 8080
s.connect(('localhost',port))
s.send(sys.argv[1])
response = s.recv(1024)
print response
if(response =='Found'):
    file = open('recieve_'+ sys.argv[1] ,'wb')
    print "Recieving File......"
    while True:
        data = s.recv(1024)
        if not data:
            break
        file.write(data)
    file.close()
    print "File written at recieve_"+sys.argv[1]
else:
    print "File Not Found"

s.close()
```

14.5 Output



```
user9747@debian: ~/workspace/netlab/cycle2/FILE
user9747@debian: ~/workspace/netlab/cycle2/FILE 150x19
┌user9747@debian ~/workspace/netlab/cycle2/FILE  ‹master*›
└$ python fileserver.py
Finding file : test......
('Got Connection from', ('127.0.0.1', 37416))
File Found
Reading File....
Sending...
File closed
┌user9747@debian ~/workspace/netlab/cycle2/FILE  ‹master*›
└$
```

```
user9747@debian: ~/workspace/netlab/cycle2/FILE 150x19
┌user9747@debian ~/workspace/netlab/cycle2/FILE  ‹master*›
└$ python fileclient.py test
Found
Recieving File......
File written at recieve_test
┌user9747@debian ~/workspace/netlab/cycle2/FILE  ‹master*›
└$
```

- Client sends filename to Server
- Server replies with file found
- Server reads file and sends it.
- Client receives data from server and writes it to a file

14.6 Result

Implemented FTP using TCP in Python2.7 and executed on Debian 9.4 Kernel 4.9 and outputs were verified.

15 Wireshark : UDP

15.1 Aim

Using Wireshark observe data transferred in client server communication using UDP and identify the UDP datagram.

15.2 Theory

15.2.1 Wireshark

Wireshark, a network analysis tool formerly known as Ethereal, captures packets in real time and display them in human-readable format. Wireshark includes filters, color coding, and other features that let you dig deep into network traffic and inspect individual packets.

15.2.2 Capturing Packets

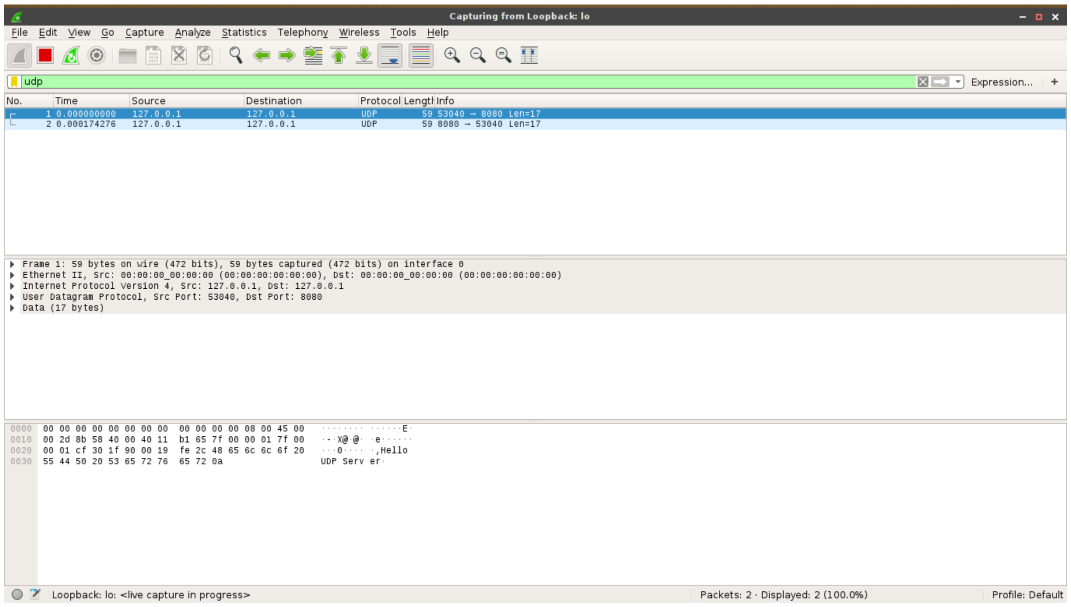
After downloading and installing Wireshark, you can launch it and double-click the name of a network interface under Capture to start capturing packets on that interface. For example, if you want to capture traffic on your wireless network, click your wireless interface.

15.2.3 Filtering Packets

The most basic way to apply a filter is by typing it into the filter box at the top of the window and clicking Apply (or pressing Enter). For example, type "dns" and you'll see only DNS packets. When you start typing, Wireshark will help you autocomplete your filter.

15.3 Output

Captured two packets, one from client to server and the other from server to client.The packet has data like source and destination IP,PORT,Data send etc.


Capturing from Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Expression...

udp

```
No.  Time         Source     Destination  Protocol Length Info
   1 0.000000000  127.0.0.1  127.0.0.1    UDP          59 53040 → 8080 Len=17
   2 0.000174276  127.0.0.1  127.0.0.1    UDP          59 8080 → 53040 Len=17
```

```
▸ Frame 1: 59 bytes on wire (472 bits), 59 bytes captured (472 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ User Datagram Protocol, Src Port: 53040, Dst Port: 8080
▸ Data (17 bytes)
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  00 2d 8b 58 40 00 40 11  b1 65 7f 00 00 01 7f 00   ·-·X@·@· ·e······
0020  00 01 cf 30 1f 90 00 19  fe 2c 48 65 6c 6c 6f 20   ···0···· ·,Hello 
0030  55 44 50 20 53 65 72 76  65 72 0a                  UDP Serv er·
```

Loopback: lo: <live capture in progress>   Packets: 2 · Displayed: 2 (100.0%)   Profile: Default

15.4 Result

Installed wireshark and captured UDP packets on localhost.The packets were filtered as udp.

16 Wireshark : TCP

16.1 Aim

Using Wireshark observe Three Way Handshaking Connection Establishment, Data Transfer and Three Way Handshaking Connection Termination in client server communication using TCP.

16.2 Theory

16.2.1 Wireshark

Wireshark, a network analysis tool formerly known as Ethereal, captures packets in real time and display them in human-readable format. Wireshark includes filters, color coding, and other features that let you dig deep into network traffic and inspect individual packets.

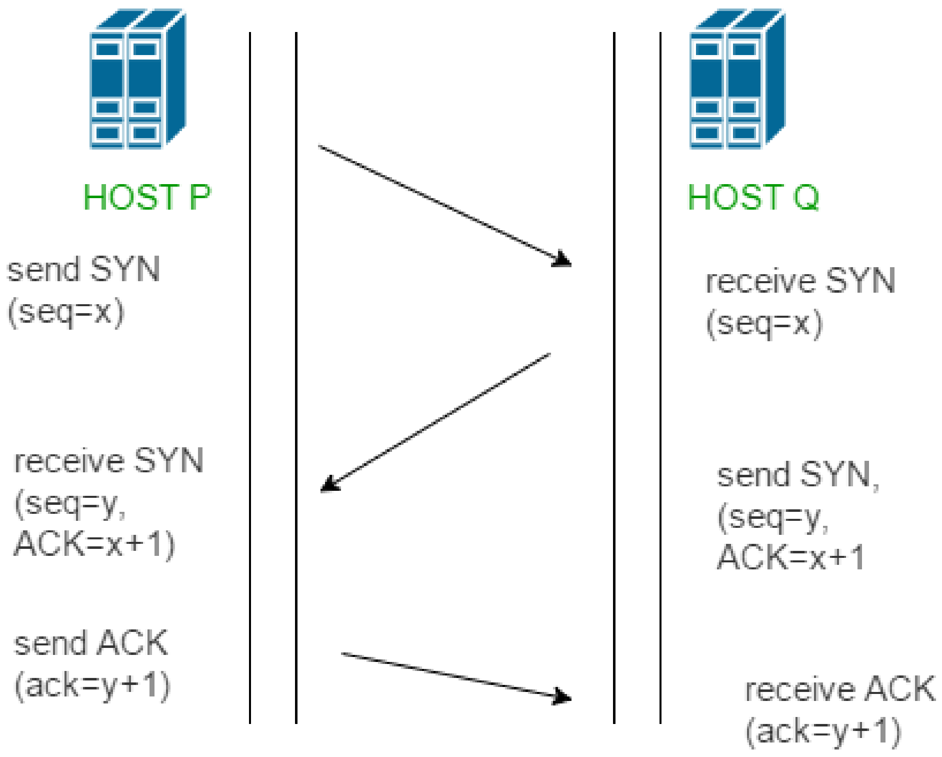
16.2.2 Capturing Packets

After downloading and installing Wireshark, you can launch it and double-click the name of a network interface under Capture to start capturing packets on that interface. For example, if you want to capture traffic on your wireless network, click your wireless interface.

16.2.3 Filtering Packets

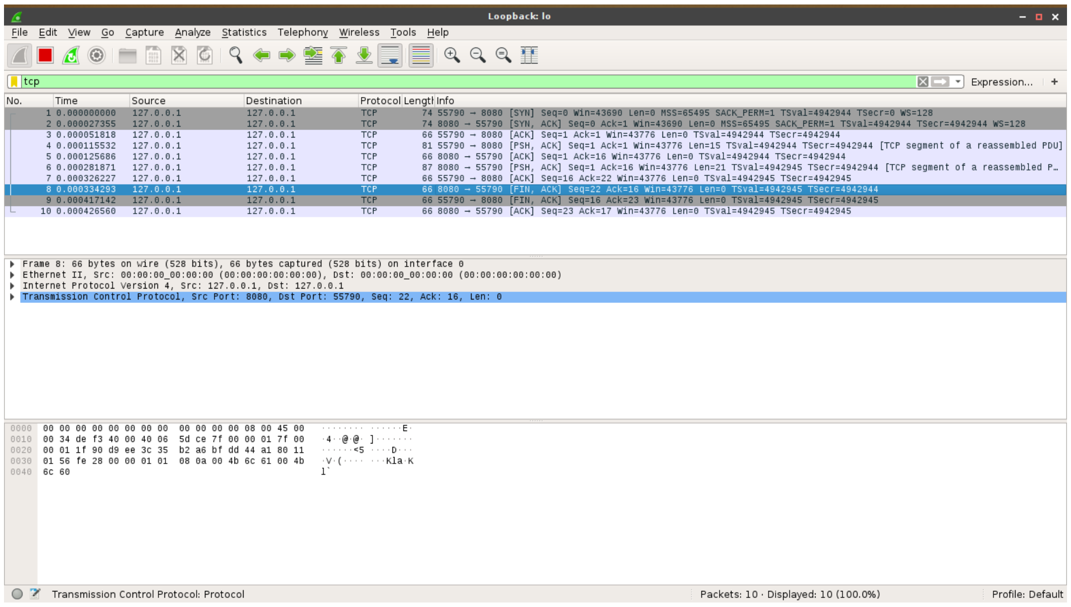
The most basic way to apply a filter is by typing it into the filter box at the top of the window and clicking Apply (or pressing Enter). For example, type “dns” and you’ll see only DNS packets. When you start typing, Wireshark will help you autocomplete your filter.

16.2.4 Three Way Hand Shake



- (SYN) : In the first step, client wants to establish a connection with server, so it sends a segment with SYN(Synchronize Sequence Number) which informs server that client is likely to start communication and with what sequence number it starts segments.
- (SYN + ACK): Server responds to the client request with SYN-ACK signal bits set. Acknowledgement(ACK) signifies the response of segment it received and SYN signifies with what sequence number it is likely to start the segments
- (ACK) : In the final part client acknowledges the response of server and they both establish a reliable connection with which they will start eh actual data transfer

16.3 Output


Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp
Expression...

```
No.  Time         Source      Destination  Protocol Length Info
   1 0.000000000  127.0.0.1   127.0.0.1    TCP      74 55790 → 8080 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=4942944 TSecr=0 WS=128
   2 0.000027355  127.0.0.1   127.0.0.1    TCP      74 8080 → 55790 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=4942944 TSecr=4942944 WS=128
   3 0.000051818  127.0.0.1   127.0.0.1    TCP      66 55790 → 8080 [ACK] Seq=1 Ack=1 Win=43776 Len=0 TSval=4942944 TSecr=4942944
   4 0.000115532  127.0.0.1   127.0.0.1    TCP      81 55790 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=43776 Len=15 TSval=4942944 TSecr=4942944 [TCP segment of a reassembled PDU]
   5 0.000125686  127.0.0.1   127.0.0.1    TCP      66 8080 → 55790 [ACK] Seq=1 Ack=16 Win=43776 Len=0 TSval=4942944 TSecr=4942944
   6 0.000281871  127.0.0.1   127.0.0.1    TCP      87 8080 → 55790 [PSH, ACK] Seq=1 Ack=16 Win=43776 Len=21 TSval=4942945 TSecr=4942944 [TCP segment of a reassembled P…
   7 0.000326227  127.0.0.1   127.0.0.1    TCP      66 55790 → 8080 [ACK] Seq=16 Ack=22 Win=43776 Len=0 TSval=4942945 TSecr=4942945
   8 0.000334293  127.0.0.1   127.0.0.1    TCP      66 8080 → 55790 [FIN, ACK] Seq=22 Ack=16 Win=43776 Len=0 TSval=4942945 TSecr=4942944
   9 0.000417142  127.0.0.1   127.0.0.1    TCP      66 55790 → 8080 [FIN, ACK] Seq=16 Ack=23 Win=43776 Len=0 TSval=4942945 TSecr=4942945
  10 0.000426560  127.0.0.1   127.0.0.1    TCP      66 8080 → 55790 [ACK] Seq=23 Ack=17 Win=43776 Len=0 TSval=4942945 TSecr=4942945

▶ Frame 8: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶ Transmission Control Protocol, Src Port: 8080, Dst Port: 55790, Seq: 22, Ack: 16, Len: 0

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  00 34 de f3 40 00 40 06  5d ce 7f 00 00 01 7f 00   ·4··@·@· ]·······
0020  00 01 1f 90 d9 ee 3c 35  b2 a6 bf dd 44 a1 80 11   ······<5 ····D···
0030  01 56 fe 28 00 00 01 01  08 0a 00 4b 6c 61 00 4b   ·V·(···· ···Kla·K
0040  6c 60                                              l`

Transmission Control Protocol: Protocol        Packets: 10 · Displayed: 10 (100.0%)        Profile: Default
```

16.4 Result

Installed wireshark and captured TCP packets on localhost.The packets were filtered as tcp. Three Way Handshaking Connection Establishment, Data Transfer and Three Way Handshaking Connection Termination was observed and deduced.

17 Packet capturing and filtering application

17.1 Aim

To develop a packet capturing and filtering application using raw sockets.

17.2 Theory

17.2.1 Network packets and packet sniffers

When an application sends data into the network, it is processed by various network layers. Before sending data, it is wrapped in various headers of the network layer. The wrapped form of data, which contains all the information like the source and destination address, is called a network packet

17.3 Algorithm

Algorithm 18 Algorithm for creating N threads

```
START
CREATE SOCKET
RECEIVE all packets
UNPACK the packets
FORMAT the packets
PRINT the filtered data
STOP
```

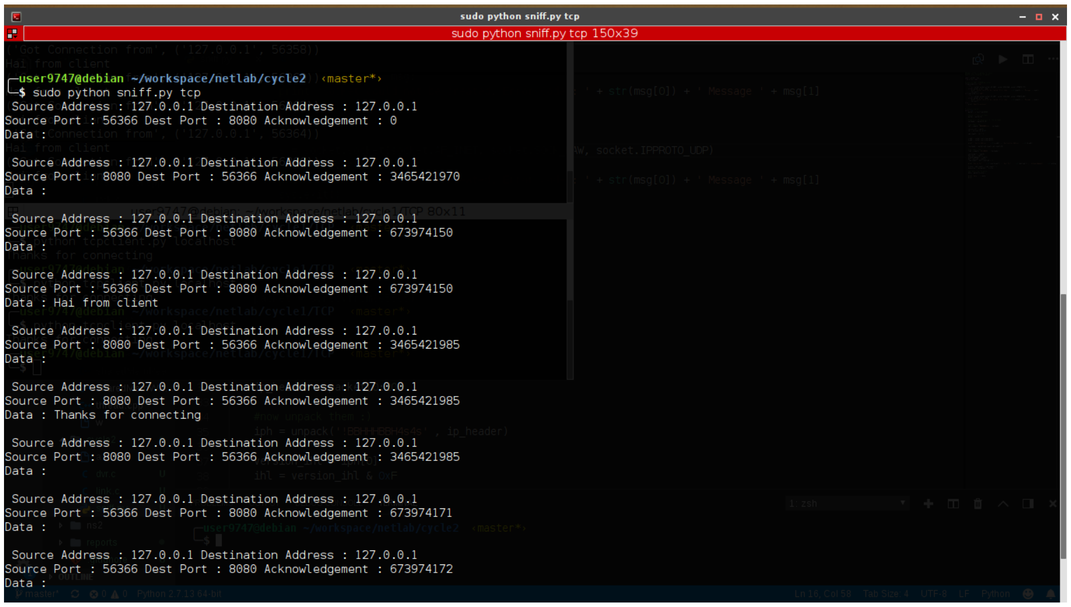
17.4 Program

```
import socket, sys
from struct import *

if(sys.argv[1]=="tcp"):
        try:
                s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROT
        except socket.error , msg:
                print 'Socket␣could␣not␣be␣created.␣Error␣Code␣:␣' + str(msg[0])
                sys.exit()
elif(sys.argv[1]=="udp"):
        try:
                s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROT
        except socket.error , msg:
                print 'Socket␣could␣not␣be␣created.␣Error␣Code␣:␣' + str(msg[0])
                sys.exit()
else:
        print "Specify␣protocol"


# receive a packet
while True:
        packet = s.recvfrom(65565)

        packet = packet[0]

        ip_header = packet[0:20]

        iph = unpack('!BBHHHBBH4s4s' , ip_header)

        version_ihl = iph[0]
        ihl = version_ihl & 0xF

        iph_length = ihl * 4


        s_addr = socket.inet_ntoa(iph[8]);
        d_addr = socket.inet_ntoa(iph[9]);

        print  '␣Source␣Address␣:␣' + str(s_addr) + '␣Destination␣Address␣:␣' +

        tcp_header = packet[iph_length:iph_length+20]
```

```
    tcph = unpack('!HHLLBBHHH' , tcp_header)

    source_port = tcph[0]
    dest_port = tcph[1]
    acknowledgement = tcph[3]
    doff_reserved = tcph[4]
    tcph_length = doff_reserved >> 4

    print 'Source Port : ' + str(source_port) + ' Dest Port : ' + str(dest_p

    h_size = iph_length + tcph_length * 4
    data_size = len(packet) - h_size

    #get data from the packet
    data = packet[h_size:]

    print 'Data : ' + data
    print
```

17.5 Output

All TCP packets were captured and their headers were displayed to the terminal.


sudo python sniff.py tcp

sudo python sniff.py tcp 150x39

```
┌user9747@debian ~/workspace/netlab/cycle2 ‹master*›
└$ sudo python sniff.py tcp
 Source Address : 127.0.0.1 Destination Address : 127.0.0.1
Source Port : 56366 Dest Port : 8080 Acknowledgement : 0
Data :

 Source Address : 127.0.0.1 Destination Address : 127.0.0.1
Source Port : 8080 Dest Port : 56366 Acknowledgement : 3465421970
Data :

 Source Address : 127.0.0.1 Destination Address : 127.0.0.1
Source Port : 56366 Dest Port : 8080 Acknowledgement : 673974150
Data :

 Source Address : 127.0.0.1 Destination Address : 127.0.0.1
Source Port : 56366 Dest Port : 8080 Acknowledgement : 673974150
Data : Hai from client

 Source Address : 127.0.0.1 Destination Address : 127.0.0.1
Source Port : 8080 Dest Port : 56366 Acknowledgement : 3465421985
Data :

 Source Address : 127.0.0.1 Destination Address : 127.0.0.1
Source Port : 8080 Dest Port : 56366 Acknowledgement : 3465421985
Data : Thanks for connecting

 Source Address : 127.0.0.1 Destination Address : 127.0.0.1
Source Port : 8080 Dest Port : 56366 Acknowledgement : 3465421985
Data :

 Source Address : 127.0.0.1 Destination Address : 127.0.0.1
Source Port : 56366 Dest Port : 8080 Acknowledgement : 673974171
Data :

 Source Address : 127.0.0.1 Destination Address : 127.0.0.1
Source Port : 56366 Dest Port : 8080 Acknowledgement : 673974172
Data :
```

17.6 Result

Implemented a program to capture and filter packets in python 2.7 and executed on Debian 9.4 Kernel 4.9 and outputs were verified.

18 Process And Thread

18.1 Aim

Design and configure a network with multiple subnets with wired and wireless LANs using required network devices. Configure the following services in the network - TELNET, SSH, FTP server, Web server, File server, DHCP server and DNS server.

18.2 Theory

18.2.1 Subnet

A subnet is a logical partition of an IP network into multiple, smaller network segments. It is typically used to subdivide large networks into smaller, more efficient subnetworks. The internet is composed of many networks that are run by many organizations. In turn, each organization's network can be composed of many smaller networks, or subnets. Each subnet allows its connected devices to communicate with each other, and routers are used to communicate between subnets. The size of a subnet depends on the connectivity requirements and the network technology employed. A point-to-point subnet allows two devices to connect, while a data center subnet might be designed to connect many more devices.

18.3 Configuring the Services

The following shows how the different services can be configured in an Ubuntu PC:

18.3.1 Telnet

Telnet is a user command and an underlying TCP/IP protocol for accessing remote computers. Through Telnet, an administrator or another user can access someone else's computer remotely. On the Web, HTTP and FTP protocols allow you to request specific files from remote computers, but not to actually be logged on as a user of that computer. With Telnet, you log on as a regular user with whatever privileges you may have been granted to the specific application and data on that computer. Telnet is most likely to be used by program developers and anyone who has a need to use specific applications or data located at a particular host computer.
Take the following steps to configure Telnet:

- Install Telnet

```
sudo apt install telnet xinetd
```

- Edit /etc/inetd.conf with root permission,add this line:

```
telnet stream tcp nowait telnetd /usr/sbin/tcpd /usr/sbin/in.telnetd
```

- Edit /etc/xinetd.conf, copy the following configuration:

```
# Simple configuration file for xinetd
#
# Some defaults , and include /etc/xinetd.d/
defaults
{
# Please note that you need a log_type line to be able to use log_on_success
# and log_on_failure. The default is the following :
# log_type = SYSLOG daemon info
instances = 60
log_type = SYSLOG authpriv
log_on_success = HOST PID
log_on_failure = HOST
cps = 25 30
}
```

- Change telnet port by using the following command in the terminal:

```
telnet 23/tcp
```

- Then restart the service:

```
sudo /etc/init.d/xinetd restart
```

18.3.2 SSH

The SSH protocol (also referred to as Secure Shell) is a method for secure remote login from one computer to another. It provides several alternative options for strong authentication, and it protects the communications security and integrity with strong encryption. It is a secure alternative to the non-protected login protocols (such as telnet, rlogin) and insecure file transfer methods (such as FTP).
Take the following steps to configure SSH:

- Install SSH:

```
sudo apt-get install openssh-server
```

  (Installing the client can be done by replacing openssh-server by openssh-client)

- Configure SSH:

```
sudo nano /etc/ssh/sshd_config
```

  Then make the changes you want to make.

- Restart SSH:

```
sudo systemctl restart ssh
```

We can login to the SSH server from an SSH client.

18.3.3 FTP Server

File Transfer Protocol (FTP) is the commonly used protocol for exchanging files over the Internet. FTP uses the Internet's TCP/IP protocols to enable data transfer. FTP uses a client-server architecture, often secured with SSL/TLS. FTP promotes sharing of files via remote computers with reliable and efficient data transfer. FTP uses a client-server architecture. Users provide authentication using a sign-in protocol, usually a username and password, however some FTP servers may be configured to accept anonymous FTP logins where you don't need to identify yourself before accessing files. Most often, FTP is secured with SSL/TLS.
The following steps show setting up an FTP server on the computer:

- Install FTP daemon:

```
sudo apt install vsftpd
```

- Configuring FTP can be done by editing the following file:

```
/etc/vsftpd.conf
```

- Restart the service:

```
sudo systemctl restart vsftpd.service
```

18.3.4 Web Server

A Web server is a program that uses HTTP (Hypertext Transfer Protocol) to serve the files that form Web pages to users, in response to their requests, which are forwarded by their computers' HTTP clients. Dedicated computers and appliances may be referred to as Web servers as well. The process is an example of the client/server model. All computers that host Web sites must have Web server programs. Leading Web servers include Apache (the most widely-installed Web server), Microsoft's Internet Information Server (IIS) and nginx (pronounced engine X) from NGNIX. Other Web servers include Novell's NetWare server, Google Web Server (GWS) and IBM's family of Domino servers. A web server can be hosted on the localhost of the PC by following the following steps:

- Installing the server: The most common server on Linux systems and it is called the LAMP server. It can be installed on Ubuntu by:

```
sudo apt install lamp-server^
```

- Hosting a website: By creating a *.conf* file in the */etc/apache2/sites-available/* folder, we inform the server of the location of the code for our website.

- Enabling the website by using the command:

```
sudo a2ensite <nameOfFile.conf>
```

- By editing */etc/hosts* file, we can give the domain name for the website
- The configuration the server is in the file: */etc/apache2/apache2.conf*
- Restart the server by the command:

```
sudo systemctl restart apache2.service
```

18.3.5 File Server

In the client/server model, a file server is a computer responsible for the central storage and management of data files so that other computers on the same network can access the files. A file server allows users to share information over a network without having to physically transfer files by floppy diskette or some other external storage device. Any computer can be configured to be a host and act as a file server. In its simplest form, a file server may be an ordinary PC that handles requests for files and sends them over the network. In a more sophisticated network, a file server might be a dedicated network-attached storage (NAS) device that also serves as a remote hard disk drive for other computers, allowing anyone on the network to store files on it as if to their own hard drive.
The following steps can be followed to setup a file server:

- Installing Samba File Server:

```
sudo apt install samba
```

- Configuring the file server by editing */etc/samba/smb.conf*

  First, edit the following key/value pairs in the [global] section of /etc/samba/smb.conf:

```
workgroup = EXAMPLE
...
security = user
```

  Create a new section at the bottom of the file, or uncomment one of the examples, for the directory to be shared:

```
[share]
 comment = Ubuntu File Server Share
 path = /srv/samba/share
 browsable = yes
 guest ok = yes
 read only = no
 create mask = 0755
```

- Make a directory for hosting files and setting permission for the directory:

  ```
  sudo mkdir -p /srv/samba/share
  sudo chown nobody:nogroup /srv/samba/share/
  ```

- Restart Samba service:

  ```
  sudo systemctl restart smbd.service nmbd.service
  ```

18.3.6 DHCP Server

DHCP (Dynamic Host Configuration Protocol) is a network management protocol used to dynamically assign an Internet Protocol (IP) address to any device, or node, on a network so they can communicate using IP. DHCP automates and centrally manages these configurations rather than requiring network administrators to manually assign IP addresses to all network devices. DHCP can be implemented on small local networks as well as large enterprise networks. DHCP will assign new IP addresses in each location when devices are moved from place to place, which means network administrators do not have to manually initially configure each device with a valid IP address or reconfigure the device with a new IP address if it moves to a new location on the network. Versions of DHCP are available for use in Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6).
The following steps shows how DHCP server can be run:

- Install DHCP server:

  ```
  sudo apt-get install isc-dhcp-server
  ```

- Configure DHCP server, the config file is */etc/dhcp/dhcpd.conf*:

  ```
  # Sample /etc/dhcpd.conf
  # (add your comments here)
  default-lease-time 600;
  max-lease-time 7200;
  option subnet-mask 255.255.255.0;
  option broadcast-address 192.168.1.255;
  option routers 192.168.1.254;
  option domain-name-servers 192.168.1.1, 192.168.1.2;
  option domain-name "mydomain.example";

  subnet 192.168.1.0 netmask 255.255.255.0 {
  range 192.168.1.10 192.168.1.100;
  range 192.168.1.150 192.168.1.200;
  }
  ```

- Starting and stopping services can be achieved using:

```
sudo service isc-dhcp-server restart
sudo service isc-dhcp-server start
sudo service isc-dhcp-server stop
```

After editing configuration files, we have to restart the service.

18.3.7 DNS Server

The Domain Name Systems (DNS) is the phonebook of the Internet. Humans access information online through domain names, like nytimes.com or espn.com. Web browsers interact through Internet Protocol (IP) addresses. DN S translates domain names to IP addresses so browsers can load Internet resources.

Each device connected to the Internet has a unique IP address which other machines use to find the device. DNS servers eliminate the need for humans to memorize IP addresses such as 192.168.1.1 (in IPv4), or more complex newer alphanumeric IP addresses such as 2400:cb00:2048:1::c629:d7a2 (in IPv6).
The following steps show the setup:

- Installing:

  ```
  sudo apt install bind9
  ```

- The configuration is in the */etc/bind* folder

- Setting as a catching name server by
  editing the file */etc/bind/named.conf.options*:

  ```
  forwarders {
  1.2.3.4; # replace with the ip address
  5.6.7.8; # of the name servers
  };
  ```

- BIND9 can be configured with the primary and the secondary master as a custom DNS server to access all the subnets.

- Restarting bind9:

  ```
  sudo systemctl restart bind9.service
  ```

18.4 Result

For accessing the different nodes in the subnet, TELNET, SSH, FTP server, Web server, File server, DHCP server and DNS server have been configured and runs successfully in an Ubuntu 16.04 PC.

19 Network simulator NS-2

19.1 Aim

Install network simulator NS-2 in any of the Linux operating system and simulate wired and wireless scenarios.

19.2 Theory

19.2.1 NS-2

NS2 is an open-source simulation tool that runs on Linux. It is a discreet event simulator targeted at networking research and provides substantial support for simulation of routing, multicast protocols and IP protocols, such as UDP, TCP, RTP and SRM over wired and wireless (local and satellite) networks. It has many advantages that make it a useful tool, such as support for multiple protocols and the capability of graphically detailing network traffic. Additionally, NS2 supports several algorithms in routing and queuing. LAN routing and broadcasts are part of routing algorithms. Queuing algorithms include fair queuing, deficit round-robin and FIFO.

19.2.2 Installation

Step 1: Download ns2 from https://sourceforge.net/projects/nsnam/files/latest/downloadhere The package downloaded will be named "ns-allinone-2.35.tar.gz". Copy it to the home folder. Then in a terminal use the following two commands to extract the contents of the package.:

```
cd ~/
tar -xvzf ns-allinone-2.35.tar.gz
```

Step 2:Building the dependencies

```
sudo apt-get install build-essential autoconf automake libxmu-dev
sudo apt-get install gcc-4.4
```

Now open the file named "ls.h" and scroll to the 137th line. In that change the word "erase" to "this->erase". The image below shows the line 137 (highlighted in the image below) after making the changes to the ls.h file.To open the file use the following command:

```
cd ~/ns-allinone-2.35/ns-2.35/linkstate
gedit ls.h
```

Now there is one more step that has to be done. We have to tell the ns which version of GCC will be used. To do so, go to your ns folder and type the following command:

```
   sudo gedit ns-allinone-2.34/otcl-1.13/Makefile.in
```

In the file, change Change CC= @CC@ to CC=gcc-4.4, as shown in the image below.
Step 3:Installation

```
  sudo su cd ~/ns-allinone-2.35/./install
```

Step 4:Setting up the environment path The final step is to tell the system, where the files for ns2 are installed or present. To do that, we have to set the environment path using the ".bashrc" file. In that file, we need to add a few lines at the bottom. The things to be added are given below. But for the path indicated below, many of those lines have "/home/albin/ns-allinone-2.35/...." , but that is where I have my extracted folder. Make sure you replace them with your path. For example, if you have installed it in a folder "/home/abc", then replace "/home/albin/ns-allinone-2.35/otcl-1.14" with "/home/abc/ns-allinone-2.35/otcl-1.14".
Step 5:Running NS-2

```
        ns
```

19.2.3 TCL

Tcl (pronounced "tickle" or tee cee ell, /ti si l/) is a high-level, general-purpose, interpreted, dynamic programming language. It was designed with the goal of being very simple but powerful.Tcl casts everything into the mold of a command, even programming constructs like variable assignment and pro- cedure definition.

19.2.4 C++

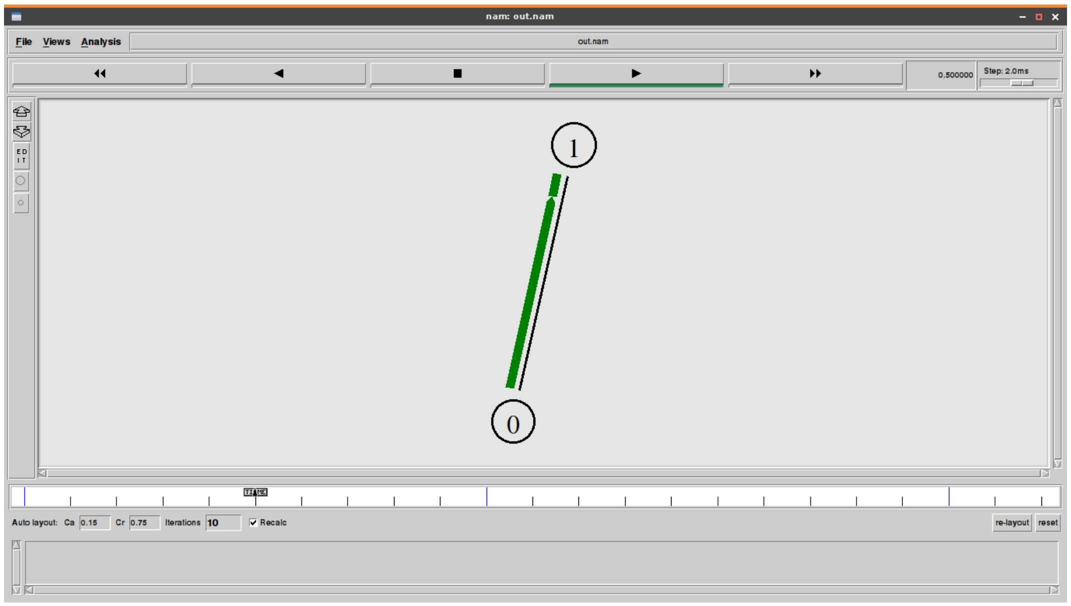
NS2 uses OTcl to create and configure a network, and uses C++ to run simulation.

19.3 Program

19.3.1 Wired Network Simulation

```
# Create scheduler
set ns [new Simulator]


$ns color 1 darkmagenta
$ns color 2 yellow
$ns color 3 blue
$ns color 4 green
$ns color 5 black

#Tracing simulation results
set fin [open result.dat w]
$ns trace-all $fin

#Tracing for NAM(Network Animator)
set nfin [open out.nam w]
$ns namtrace-all $nfin
# Create a node
set n0 [$ns node]

# Create another node
set n1 [$ns  node]
# Create TCP agent and attach to first node
set tcp0 [new Agent/TCP]
$ns attach-agent $n0 $tcp0

# Create TCP receiver and attach to second node
set tcp1 [new Agent/TCPSink]
$ns attach-agent $n1 $tcp1
# Connect both nodes with 1.5Mbps bandwidth, 5 milli seconds delay and DropTai
$ns duplex-link $n0 $n1 1Mb 5ms DropTail

# Connect tcp0 and tcp1 agents
$ns connect $tcp0 $tcp1

$tcp0 set fid_ 4
$tcp1 set fid_ 2
# Create a FTP object
set ftp [new Application/FTP]
# Attach ftp application with agent tcp0
 $ftp attach-agent $tcp0
 $ftp set fid_ 3
```

```
# Schedule events
$ns at 0.2 "$ftp start"
$ns at 2.0 "$ftp stop"
# Finish procedure to perform operation at the end of simulation

proc finish { } {
      # Mapping of global variable into local variable
      global ns fin nfin
        # Flush all buffers
        $ns flush-trace
        #close file objects
        close $fin
        close $nfin
        #Execute nam process in background
        exec nam out.nam &
        #Terminate current process
        exit 0
}
$ns at 2.2 "finish"
$ns run
```



Auto layout: Ca 0.15 Cr 0.75 Iterations 10 ✓ Recalc

Figure 1: Wired Network

19.3.2 Wireless Network Simulation

#initialize the variables

```
set val(chan)          Channel/WirelessChannel    ;#Channel Type
set val(prop)          Propagation/TwoRayGround   ;# radio-propagation model
set val(netif)         Phy/WirelessPhy            ;# network interface type WA
set val(mac)           Mac/802_11                 ;# MAC type
set val(ifq)           Queue/DropTail/PriQueue    ;# interface queue type
set val(ll)            LL                         ;# link layer type
set val(ant)           Antenna/OmniAntenna        ;# antenna model
set val(ifqlen)        50                         ;# max packet in ifq
set val(nn)            6                          ;# number of mobilenodes
set val(rp)            AODV                       ;# routing protocol
set val(x)  500   ;# in metres
set val(y)  500   ;# in metres
#Adhoc OnDemand Distance Vector

#creation of Simulator
set ns [new Simulator]

#creation of Trace and namfile
set tracefile [open wireless.tr w]
$ns trace-all $tracefile

#Creation of Network Animation file
set namfile [open wireless.nam w]
$ns namtrace-all-wireless $namfile $val(x) $val(y)

#create topography
set topo [new Topography]
$topo load_flatgrid $val(x) $val(y)

#GOD Creation - General Operations Director
create-god $val(nn)

set channel1 [new $val(chan)]
set channel2 [new $val(chan)]
set channel3 [new $val(chan)]

#configure the node
$ns node-config -adhocRouting $val(rp) \
  -llType $val(ll) \
  -macType $val(mac) \
  -ifqType $val(ifq) \
  -ifqLen $val(ifqlen) \
  -antType $val(ant) \
  -propType $val(prop) \
  -phyType $val(netif) \
  -topoInstance $topo \
  -agentTrace ON \
  -macTrace ON \
  -routerTrace ON \
  -movementTrace ON \
  -channel $channel1

set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]

$n0 random-motion 0
$n1 random-motion 0
$n2 random-motion 0
$n3 random-motion 0
$n4 random-motion 0
$n5 random-motion 0

$ns initial_node_pos $n0 20
$ns initial_node_pos $n1 20
$ns initial_node_pos $n2 20
$ns initial_node_pos $n3 20
$ns initial_node_pos $n4 20
$ns initial_node_pos $n5 50

#initial coordinates of the nodes
$n0 set X_ 10.0
$n0 set Y_ 20.0
$n0 set Z_ 0.0

$n1 set X_ 210.0
$n1 set Y_ 230.0
$n1 set Z_ 0.0

$n2 set X_ 100.0
$n2 set Y_ 200.0
$n2 set Z_ 0.0

$n3 set X_ 150.0
$n3 set Y_ 330.0
$n3 set Z_ 0.0

$n4 set X_ 430.0
$n4 set Y_ 320.0
```

```
$n4 set Z_ 0.0

$n5 set X_ 270.0
$n5 set Y_ 120.0
$n5 set Z_ 0.0
#Dont mention any values above than 500 because in this example, we use X and Y

#mobility of the nodes
#At what Time? Which node? Where to? at What Speed?
$ns at 1.0 "$n0␣setdest␣490.0␣340.0␣35.0"
$ns at 1.0 "$n1␣setdest␣490.0␣340.0␣5.0"
$ns at 1.0 "$n2␣setdest␣330.0␣100.0␣10.0"
$ns at 1.0 "$n3␣setdest␣300.0␣100.0␣8.0"
$ns at 1.0 "$n4␣setdest␣300.0␣130.0␣5.0"
$ns at 1.0 "$n5␣setdest␣190.0␣440.0␣15.0"
#the nodes can move any number of times at any location during the simulation (r
$ns at 20.0 "$n5␣setdest␣100.0␣200.0␣30.0"

#creation of agents
set tcp [new Agent/TCP]
set sink [new Agent/TCPSink]
$ns attach-agent $n0 $tcp
$ns attach-agent $n5 $sink
$ns connect $tcp $sink
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ns at 1.0 "$ftp␣start"

set udp [new Agent/UDP]
set null [new Agent/Null]
$ns attach-agent $n2 $udp
$ns attach-agent $n3 $null
$ns connect $udp $null
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
$ns at 1.0 "$cbr␣start"

$ns at 30.0 "finish"

proc finish {} {
 global ns tracefile namfile
 $ns flush-trace
 close $tracefile
 close $namfile
 exit 0
}
```

```
puts "Starting␣Simulation"
$ns run
```

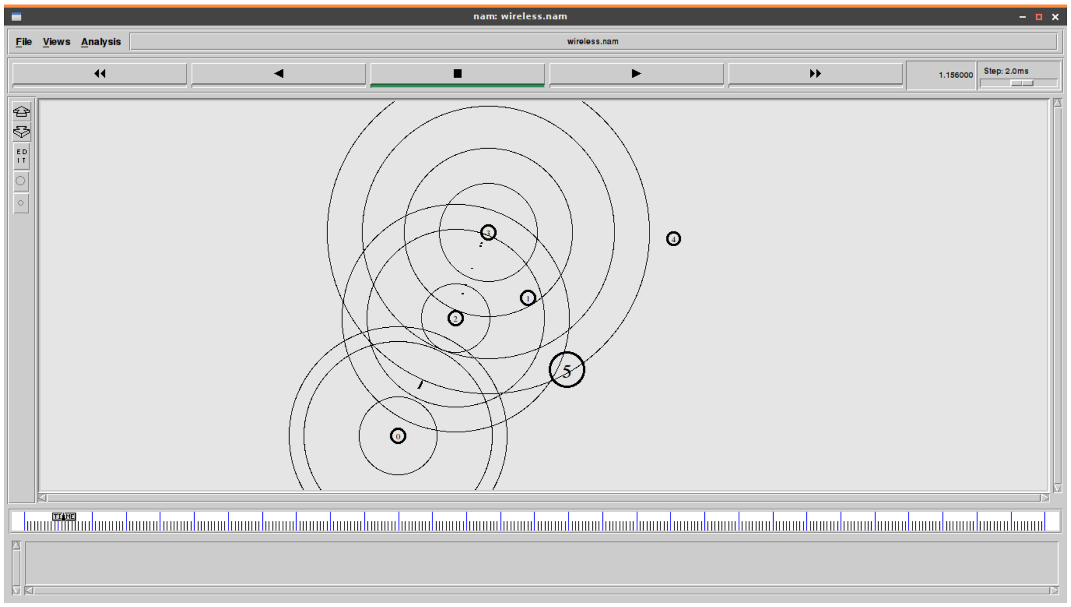


Figure 2: Wireless Network

19.4 Result

Installed NS-2 and implemented wired and wireless network simulation on NS-2 compiled on gcc 4.4 and executed on Debian 4.9 Kernel 4.9 and outputs were verified.