**ABSTRACT**

The main aim of this project is to develop a packet sniffer which can be used to intercept and log traffic passing over a digital network or a part of a network. This tool would inject an IP packet into the available network interface and wait for the reply from the intermediate routers and the destination IP, giving information about the network setup and the firewall configuration on each intermediate node.

A packet sniffer analyses network behavior, performance and applications that generate or receive network traffic. It can also be used for analyzing the network infrastructure itself by determining whether all necessary [routing](http://en.wikipedia.org/wiki/Router_(computing)) is occurring properly, allowing the user to further isolate the source of a problem.

It is also possible to use a packet sniffer for the specific purpose of intercepting and displaying the communications of another user or computer. A user with the necessary privileges on a system acting as a [router](http://en.wikipedia.org/wiki/Router_(computing)) or [gateway](http://en.wikipedia.org/wiki/Gateway_(computer_networking)) through which unencrypted traffic such as [Telnet](http://en.wikipedia.org/wiki/Telnet) or [HTTP](http://en.wikipedia.org/wiki/HTTP) passes can use tcpdump to view login IDs, passwords, the [URLs](http://en.wikipedia.org/wiki/Uniform_Resource_Locator) and content of websites being viewed, or any other unencrypted information.

**Table of Contents**

**Acknowledgement i**

**Abstract ii**

**Table of Contents iii**

**List of Figures vi**

**List of Tables vii**

**Glossary viii**

|  |  |  |
| --- | --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8  9. | Introduction   * 1. Purpose   2. Scope   3. Motivation   4. Literature Survey   Software Requirement Specifications  2.1 Overall Description  2.2 Specific Requirements  2.2.1 Functionality  2.2.2 Design Constraints  2.2.3 Software Requirements  2.2.4 Hardware Requirements  2.3 Interfaces  2.3.1 User Interfaces  2.3.2 Hardware Interfaces  2.3.3 Using Traceroute  High Level Design  3.1. Modules  3.1.1 [Main](http://www.cmcrossroads.com/bradapp/docs/sdd.html#TOC_SEC7#TOC_SEC7) Trace Loop  3.1.2 [Packet](http://www.cmcrossroads.com/bradapp/docs/sdd.html#TOC_SEC8#TOC_SEC8) Injector  3.1.3 Module to Initiate Packet Capture Session  3.1.4 ICMP Packet Filter  3.1.5 Reply Packet Parser  3.2 Input  3.4 Output  Implementation  4.1 Concept  4.2 Protocols Used  4.3 Data Structures Used  4.4 Implementing Different Modules  4.4.1 Implementing the Trace Loop  4.3.2 Implementing the Sniffer Session Initiator  4.3.3 Implementing the Packet Injector  4.3.4 Implementing the Parser  Testing  5.1 Unit Testing  5.1.1 Unit Test Case1  5.1.2 Unit Test Case2  5.1.3 Unit Test Case3  5.1.4 Unit Test Case4  5.1.5 Unit Test Case5  5.1.6 Unit Test Case6  5.1.7 Unit Test Case7  5.2 Integration Testing  5.2.1 Integration Test Case1  5.2.2 Integration Test Case2  5.3 System Testing  5.3.1 System Test Case1  Conclusion  6.1 Summary  6.2 Limitations  6.3 Further Enhancement  References  Appendices  Appendix A – Listing of Tables and Source Code  Appendix B – Screenshots | 1  1  1  1  2  3  3  3  3  4  5  5  5  6  6  6    9  9  10  10  11  11  12  12  13    14  14  15  18  19  19  21  22  23  24  24  24  25  25  25  26  26  26  27  27  27  28  28  29  29  29  29  30  31  31  37 |

**List of Figures**

|  |  |  |
| --- | --- | --- |
|  | **Name of Figure** | **Page No.** |
| Fig. 3.1  Fig. 4.1  Fig. 4.2  Fig. 4.3  Fig. 4.4  Fig. 4.5  Fig. 8.1  Fig. 8.2  Fig. 8.3 | Implementation of Modules  Diagram showing concept behind Traceroute  IPv4 Traceroute option format RFC  IPv4 Header  ICMP packet dump format  ICMP Header format  Screenshot1  Screenshot 2  Screenshot 3 | 9  14  16  17  17  18  35  36  37 |
|  |  |  |

**List of Tables**

|  | **Name of Table** | **Page No.** |
| --- | --- | --- |
| Table 5.1 Table 5.2 Table 5.3 Table 5.4  Table 5.5 Table 5.6 Table 5.7 | Unit test case 1  Unit test case 2  Unit test case 3  Unit test case 4  Unit test case 5  IP options with the traceroute option highlighted  ICMP Message Type | 24  25  25  25  26  28  29 |
|  |  |  |

**Glossary**

|  |  |
| --- | --- |
| IP | : Internet Protocol |
|  |  |
| ICMP | : Internet Control Message Protocol |
| UDP | : Universal Datagram Protocol |
| TCP | : Transmission Control Protocol |
| GUI | : Graphical User Interface |
| NIC | : Network Interface Controller/Card |
| ISDN | : Integrated Services Digital Network |
| DNS | : Domain Name Server/System |
| SYN/ACK | : Synchronize and Acknowledge |
| OHC | : Outbound Hop Count |
| RHC | : Return Hope Count |
| SSL | : Secure Sockets Layer |

# Chapter 1

# Introduction

**1.1 Purpose:**

The main aim of this project is to develop a network based program known as a packet sniffer which that can intercept and log traffic passing over a digital [network](http://en.wikipedia.org/wiki/Telecommunications_network) or part of a network.. This tool is commercially available. For example, tcpdump is a commonly used packet analyser.

This tool logs the information flowing across the data stream of a network by analyzing the packets being passed through it. The packet sniffer is usually installed on one of the nodes of the network and then is used to observe and analyse traffic.

**1.2 Scope:**

This project will only be dealing with the following:

* Only one packet sniffer shall be used
* Analysis and the sniffing will be conducted on the same processor.

**1.3 Motivation:**

In terms of security measures across networks, the only way to guarantee the network is not being misused is to analyse each and every packet being carried across it. A packet sniffer is one such program with which all the packets travelling across a network can be logged and analysed. Along with general capture of packets, sniffing can be oriented to only a specific type of packets based on filters. This is a very useful technique as we can filter out packets working on specific protocols to analyse. Overall, this program is very useful in the realms of cyber security and thus we are motivated to develop one.

**1.4 Literature Survey:**

This project aims to create a system that links multiple, previously unconnected, packet

sniffers together to analyse network data in a more intelligent way, allowing the monitoring

of networks as a whole, rather than as multiple small segments. Before creating this system,

it was necessary to research how best to create the system, based on previous work in this

area.

From initial background research, it was quickly realised that the project would encompass

various fields of computer science and business, including distributed systems, network

analysis, interface design and algorithmic trading. Alongside this, a lot of the research

would need to be centered around the company and the hardware itself; information which

can only be easily obtained through meetings and discussions with the parties involved |

so called “grey literature".

# Chapter 2

# Software Requirement Specifications

**2.1 Overall Description**

All data sent over the Internet is sent in packets. Consider the following analogy. The idea behind packets is very similar to the idea of the capsules used to send checking and information from vehicle to tellers inside the bank via vacuum tubes. The emails sent and the files downloaded are all broken down into raw data and inserted into little packets. These packets are piped through the available Internet connection. When a packet arrives at a destination computer, the data is extracted and reassembled into a file.

A packet sniffer is a program that can be used to log and analyse these packets of information. In general a packet sniffer logs the packets travelling across a digital network and analyses the information being carried by it. These packets can also be filtered by the sniffer based on protocols and various other criteria.

**2.2 Specific Requirements**

The specific requirements are those that give the specifications to be satisfied by the project. They include the functionalities, formatting, design constraints and such specifications that are to be satisfied.

**2.2.1 Functionality**

For serving the purpose of being a successful tool for analyzing the kind of information traversing a network, a packet sniffer must be set up correctly. It should be located at a node in the network through which a majority of the data travels through.

An additional feature of the packet sniffer is to be able to filter through the packets based on various criteria mentioned by the user.

A standard packet sniffer tends to use pcap or libpcap which in the field of network communication is an API for capturing network traffic. Unix-like systems implement libpcap while Windows uses a port of it known as WinPcap.

So the output of the implementation will be details on the number of packets passing through the packet filter (and thus the network) and also the protocols they belong to.

**2.2.2 Design Constraints**

It has been strongly suggested that no analysis should be performed on the packet sniffers themselves, and the software upon them should not be modified beyond the configuration of the program “Sniffer Focused Intelligence" (SFI). In brief, the SFI application allows another machine to request packet traces from the packet sniffer and have the traces delivered to another machine. This has been suggested as the most effective way of performing analysis of network trace, since it will not place undue load on the packet sniffers and also allows analysis to be carried out in a central location.

It therefore follows that a suggested architecture would be to use a client-server model, with one analysis server and many packet sniffers. However, this does introduce further issues.

An alternative to processing the traces at another location would be to process the traces on the packet sniffer itself. This would all but eliminate the load placed upon the network. Although this method will not be explored in great detail (as this is not recommended or approved)

Thus all we need to implement is a module to run on the local machine which analyses the packets being passed from the machine to another server, say, a commonly used website.

**2.2.3 Software Requirement**

* Operating System : UNIX, Fedora 10 or higher versions
* Special Library Packages: libpcap 1.0.0 or higher versions which includes <pcap.h>

**2.2.4 Hardware Requirement**

* Processor: Pentium  3.0 GHz or higher
* RAM: 256 Mb or more
* Hard Drive: 10 GB or more

**2.3 Interfaces**

Interfaces are the go-between of the project and user and also project and hardware.

**2.3.1 User Interfaces**

A packet sniffer is usually a program which is implemented in a standard command line format for user interface. Output of the program will be details of the number of packets travelling through a node and their protocols which does not require any additional GUI features. And this tool is mostly used by a Network administrator or a person having enough knowledge about the basic networking. And hence a command line implementation will be the fastest and the simplest of the user interfaces. Instead of making the implementation code more complex by using any GUI libraries we have tried to implement a simple command line interface.

**2.3.2 Hardware Interfaces**

Implementation of a packet sniffer mainly depends on the network setup and its general topology. It also depends on the type of packets required to be analysed by the user of the software.

Thus the packet sniffer can be run on pretty much any and every hardware interface which consisits of a properly configured Ethernet card.

**2.3.3 Using a packet sniffer**

Using a packet sniffer usually can be explained in as the series of the following steps:

1. Recognising interfaces
2. Capturing our first packet in the network
3. Packet analysis

**Recognising interfaces**

Initially, the first step of a packet tracer is to recognize the interfaces present on the system it is being installed onto. We use libpcap to give the specs of some interfaces we can listen on, using our program. An interface, in layman’s terms, is the computer’s hardware connection to whatever network it is connected to. On Linux, eth0 denotes the first Ethernet card in your computer. (All the interfaces of the computer can be seen by using the **ipconfig** command).

**Grabbing packets**

After compiling the packet sniffer and recognizing the interfaces of the network, some packets are sent that from the computer that has the packet sniffer installed to a website using the **ping** command. The website can be mentioned using its URL or its IP address. The program is then extended in developing this into a packet grabbing engine which will grab as many packets as possible while filtering unnecessary packets.

**Packet analysis**

First, one must procure all the RFCs required by the user and then, after integrating them into their packet sniffer, they can get a large amount of information and even display the header information of the packets, if required.

# Chapter 3

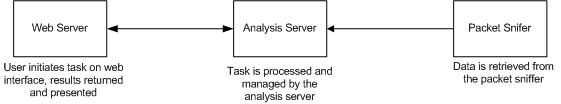
**High Level Design**

**3.1 Modules**

As already mentioned this implementation of the sniffer will have only one module which will be run on the local machine. This module in turn has many sub modules which have various responsibilities and purposes. They are discussed in detail. This implementation of the packet sniffer is simple and small covering all the basic functionalities of a packet sniffer, eliminating the complex filtering techniques and heavy scale packet analysis. Modules used are discussed below.

Different Modules implemented are:

1. Module to check the network interfaces
2. Module to initiate a packet sniffer session
3. Packet analysis

****

**Fig 3.1 Implementation of Modules**

**3.1.1 Check the Network Interfaces**

This is the module responsible for checking the network interfaces available on the node we are installing the packet sniffer onto. This helps us determine the type of connection in the network so as to make sure the packet sniffer is configured properly.

**3.1.2 Grabbing packets using the packet sniffer**

Once the packet sniffer program is run, we can ping a website using its IP address or URL or also just open a regular browser and start browsing some common websites. The packet sniffer will pick up the packets being sent to the server where the webpage is located.

**3.1.3 Packet Analysis**

This module is called once a packet is captured. It tells the user what protocol the packet was running under. For example, whether it’s an ICMP protocol (in the case of sending ping messages) or a TCP or UDP protocol. It also provides the header file of the packet which has been sniffed which obviously provides further information.

**3.2 Input**

As mentioned before entire user interface will be on the command line. The usage of the tool is very straightforward and as such requires no input. Once the program is run, the user needs to just ping a webpage or open one using a browser.

**3.3 Output**

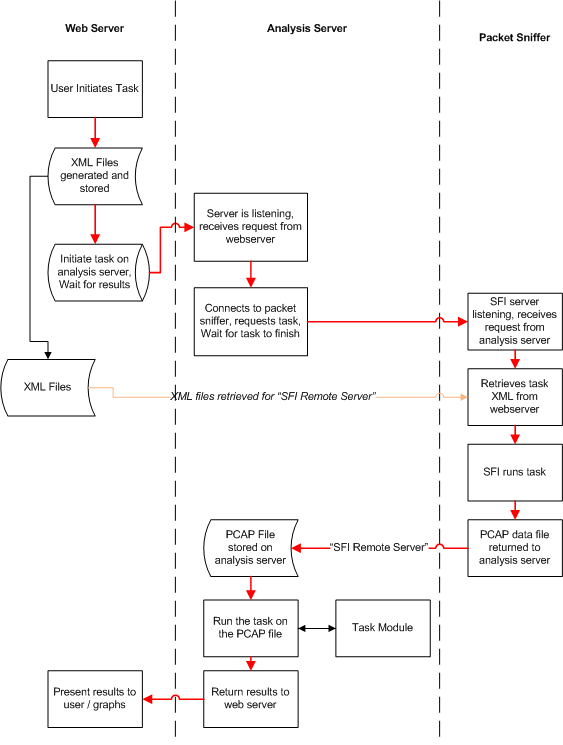
The output of the data sniffer usually tells us the number of packets crossing the network and also the protocols the packets are working under. If requested, the user can also be provided with the header of the packets sniffed, thus providing more information such as source IP, destination IP and various other details which can be derived from it.

**3.4.Data Flow Diagram**

To show how the system flow is ordered, a data flow model has been produced to illustrate which actions are performed by the user, the effect they have on the system and how data moves around the system

The process initially starts with the user initiating a task via the web browser. The packets running along the network are picked up by the packet sniffer and returned to the pcap program to analyse the packets.

The data flow diagram below shows the flow of data in a corporate level packet sniffer. Our implementation, while much simpler, follows a similar flow.



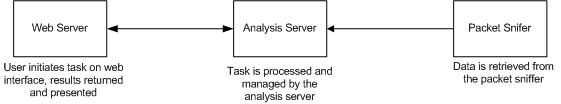
**Fig: Data Flow Diagram for a Packet Sniffer**

# Chapter 4

**Implementation**

**4.1 Concept**

* Given below in Figure 4.1, is the basic structure of a packet sniffer. At its most basic level, the architecture is very simple, as seen. A user will use the user interface provided by the web server to initiate a task. A web server was chosen to present the interface as it was a key user requirement that the system could be accessed from any machine without the installation of any specialist software. The web server is then responsible for initiating the task on an analysis server. The analysis server will then query the packet sniffer for packet traces and process this trace. When results are received they are to be returned to the web server for presentation to the user. In the given implementation, the analysis server and packet sniffer are present on the same machine and the

****

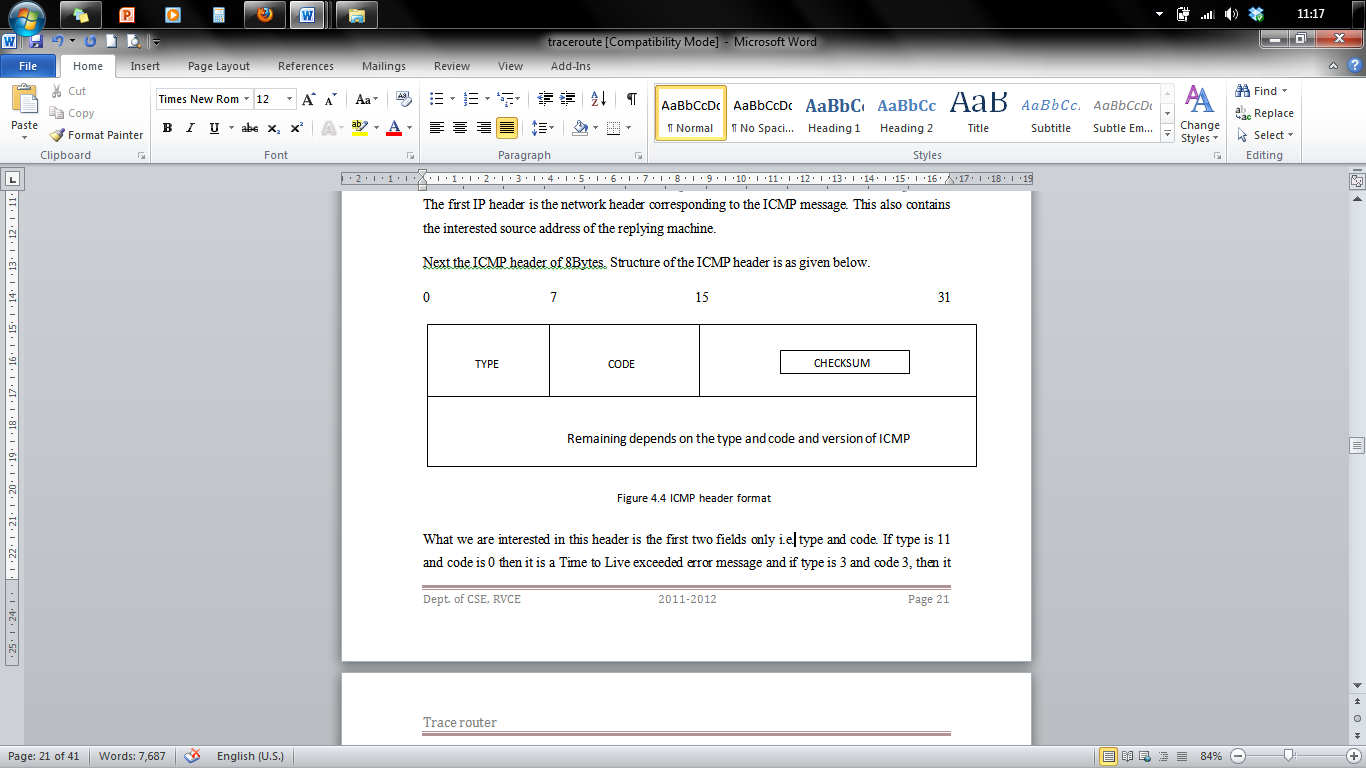
**Figure 4.1 Diagram showing the concept behind a packet sniffer**

**4.2 Protocols Used**

The packet sniffer is tested by using ping packets i.e. ICMP echo request and ICMP echo reply packets.

From the discussions till now we can say that the entire packet sniffer implementation depends on the request and reply ICMP messages. But nowadays the trend is to block unnecessary ICMP traffic by some of the Internet routers, this is because an ICMP message reveals lot of information about the Network setup, For example if we can do a probe-reply cycle on every port between 1 and 1024 we can easily know which all standard services are running on the remote machine. And thus the network administrator may configure the firewall such that it may block the transfer of ICMP messages which creates a problem in our implementation.

We in our implementation of the packet sniffer just use a server where the ICMP protocol is not blocked by a firewall as the receiving and passing of ICMP messages is critical to the working of most packet sniffers.



**Figure 4.2 ICMP header format**

What we are interested in this header is the first two fields only i.e. type and code.

**4.3 Data Structures Used**

As we need to parse the reply ICMP packet we need to have structures representing each layer i.e. IP header, UDP header, ICMP header and also the Ethernet header.

The structures declared in the source code are as given below:

/\* the 48 bit ethernet address available on many

systems. \*/

struct ether\_addr

{

u\_int8\_t ether\_addr\_octet[ETH\_ALEN];

} \_\_attribute\_\_ ((\_\_packed\_\_));

/\* 10Mb/s ethernet header \*/

struct ether\_header

{

u\_int8\_t ether\_dhost[ETH\_ALEN]; /\* destination eth addr \*/

u\_int8\_t ether\_shost[ETH\_ALEN]; /\* source ether addr \*/

u\_int16\_t ether\_type; /\* packet type ID field \*/

} \_\_attribute\_\_ ((\_\_packed\_\_));

/\*ICMP HEADER\*/

struct icmp\_hdr

{

unsigned char icmp\_type;

unsigned char icmp\_code;

unsigned short int icmp\_chksum;

int icmo\_nouse;

};

/\* UDP HEADER\*/

struct udp\_hdr

{

unsigned short int udp\_srcport;

unsigned short int udp\_destport;

unsigned short int udp\_len;

unsigned short int udp\_chksum;

};

Each field in the above structure definitions specifies a unique field in the respective IP header, ICMP header, UDP header, Ethernet header.

**4.4 Implementing different modules**

In this section let us see how the different modules have been implemented in the trace router.

**4.4.1 Implementing the test for checking the interfaces**

The general details of the implementation was defined earlier in the design section. The code responsible for the main loop is as given below:

int main(int argc, char \*\*argv)

{

char \*dev; /\* name of the device to use \*/

char \*net; /\* dot notation of the network address \*/

char \*mask;/\* dot notation of the network mask \*/

int ret; /\* return code \*/

char errbuf[PCAP\_ERRBUF\_SIZE];

bpf\_u\_int32 netp; /\* ip \*/

bpf\_u\_int32 maskp;/\* subnet mask \*/

struct in\_addr addr;

/\* ask pcap to find a valid device for use to sniff on \*/

dev = pcap\_lookupdev(errbuf);

/\* error checking \*/

if(dev == NULL)

{

printf("%s\n",errbuf);

exit(1);

}

/\* print out device name \*/

printf("DEV: %s\n",dev);

/\* ask pcap for the network address and mask of the device \*/

ret = pcap\_lookupnet(dev,&netp,&maskp,errbuf);

if(ret == -1)

{

printf("%s\n",errbuf);

exit(1);

}

/\* get the network address in a human readable form \*/

addr.s\_addr = netp;

net = inet\_ntoa(addr);

if(net == NULL)/\* thanks Scott :-P \*/

{

perror("inet\_ntoa");

exit(1);

}

printf("NET: %s\n",net);

/\* do the same as above for the device's mask \*/

addr.s\_addr = maskp;

mask = inet\_ntoa(addr);

if(mask == NULL)

{

perror("inet\_ntoa");

exit(1);

}

printf("MASK: %s\n",mask);

return 0;

}

We run the program from command line and qualify the input. The value for DEV is your default interface name (likely eth0 on linux, could be eri0 on solaris). The NET and MASK values are your primary interface's subnet and subnet mask.

**4.4.2 Implementing the packet sniffer:**

The part of the source code responsible for having the packet sniffer grab a packet is given below.

int main(int argc, char \*\*argv)

{

int i;

char \*dev;

char errbuf[PCAP\_ERRBUF\_SIZE];

pcap\_t\* descr;

const u\_char \*packet;

struct pcap\_pkthdr hdr; /\* pcap.h \*/

struct ether\_header \*eptr; /\* net/ethernet.h \*/

u\_char \*ptr; /\* printing out hardware header info \*/

/\* grab a device to peak into... \*/

dev = pcap\_lookupdev(errbuf);

if(dev == NULL)

{

printf("%s\n",errbuf);

exit(1);

}

printf("DEV: %s\n",dev);

/\* open the device for sniffing.

pcap\_t \*pcap\_open\_live(char \*device,int snaplen, int prmisc,int to\_ms,

char \*ebuf)

snaplen - maximum size of packets to capture in bytes

promisc - set card in promiscuous mode?

to\_ms - time to wait for packets in miliseconds before read

times out

errbuf - if something happens, place error string here

Note if you change "prmisc" param to anything other than zero, you will

get all packets your device sees, whether they are intendeed for you or

not!! Be sure you know the rules of the network you are running on

before you set your card in promiscuous mode!! \*/

descr = pcap\_open\_live(dev,BUFSIZ,0,-1,errbuf);

if(descr == NULL)

{

printf("pcap\_open\_live(): %s\n",errbuf);

exit(1);

}

/\*

grab a packet from descr (yay!)

u\_char \*pcap\_next(pcap\_t \*p,struct pcap\_pkthdr \*h)

so just pass in the descriptor we got from

our call to pcap\_open\_live and an allocated

struct pcap\_pkthdr \*/

packet = pcap\_next(descr,&hdr);

if(packet == NULL)

{/\* dinna work \*sob\* \*/

printf("Didn't grab packet\n");

exit(1);

}

/\* struct pcap\_pkthdr {

struct timeval ts; time stamp

bpf\_u\_int32 caplen; length of portion present

bpf\_u\_int32; lebgth this packet (off wire)

}

\*/

printf("Grabbed packet of length %d\n",hdr.len);

printf("Recieved at ..... %s\n",ctime((const time\_t\*)&hdr.ts.tv\_sec));

printf("Ethernet address length is %d\n",ETHER\_HDR\_LEN);

/\* lets start with the ether header... \*/

eptr = (struct ether\_header \*) packet;

/\* Do a couple of checks to see what packet type we have..\*/

if (ntohs (eptr->ether\_type) == ETHERTYPE\_IP)

{

printf("Ethernet type hex:%x dec:%d is an IP packet\n",

ntohs(eptr->ether\_type),

ntohs(eptr->ether\_type));

}else if (ntohs (eptr->ether\_type) == ETHERTYPE\_ARP)

{

printf("Ethernet type hex:%x dec:%d is an ARP packet\n",

ntohs(eptr->ether\_type),

ntohs(eptr->ether\_type));

}else {

printf("Ethernet type %x not IP", ntohs(eptr->ether\_type));

exit(1);

}

/\* copied from Steven's UNP \*/

ptr = eptr->ether\_dhost;

i = ETHER\_ADDR\_LEN;

printf(" Destination Address: ");

do{

printf("%s%x",(i == ETHER\_ADDR\_LEN) ? " " : ":",\*ptr++);

}while(--i>0);

printf("\n");

ptr = eptr->ether\_shost;

i = ETHER\_ADDR\_LEN;

printf(" Source Address: ");

do{

printf("%s%x",(i == ETHER\_ADDR\_LEN) ? " " : ":",\*ptr++);

}while(--i>0);

printf("\n");

return 0;

}

**4.4.3 Packet analysis**

This module is for the analysis of the packet. The code is given below for packet analysis:

int main(int argc,char \*\*argv)

{

char \*dev;

char errbuf[PCAP\_ERRBUF\_SIZE];

pcap\_t\* descr;

struct bpf\_program fp; /\* hold compiled program \*/

bpf\_u\_int32 maskp; /\* subnet mask \*/

bpf\_u\_int32 netp; /\* ip \*/

u\_char\* args = NULL;

/\* Options must be passed in as a string because I am lazy \*/

if(argc < 2){

fprintf(stdout,"Usage: %s numpackets \"options\"\n",argv[0]);

return 0;

}

/\* grab a device to peak into... \*/

dev = pcap\_lookupdev(errbuf);

if(dev == NULL)

{ printf("%s\n",errbuf); exit(1); }

/\* ask pcap for the network address and mask of the device \*/

pcap\_lookupnet(dev,&netp,&maskp,errbuf);

/\* open device for reading. NOTE: defaulting to

\* promiscuous mode\*/

descr = pcap\_open\_live(dev,BUFSIZ,1,-1,errbuf);

if(descr == NULL)

{ printf("pcap\_open\_live(): %s\n",errbuf); exit(1); }

if(argc > 2)

{

/\* Lets try and compile the program.. non-optimized \*/

if(pcap\_compile(descr,&fp,argv[2],0,netp) == -1)

{ fprintf(stderr,"Error calling pcap\_compile\n"); exit(1); }

/\* set the compiled program as the filter \*/

if(pcap\_setfilter(descr,&fp) == -1)

{ fprintf(stderr,"Error setting filter\n"); exit(1); }

}

/\* ... and loop \*/

pcap\_loop(descr,atoi(argv[1]),my\_callback,args);

fprintf(stdout,"\nfinished\n");

return 0;

}

# Chapter 5

**Testing**

Software Testing is the process used to help identify the correctness, completeness, security and quality of the developed computer software. Testing is the process of technical investigation and includes the process of executing a program or application with the intent of finding errors.

**Test Strategies**

Test strategy tells the test plan of the project. It also tells how to test and what to test. The testing done in this project are Unit testing and Integration testing.

* Features to be tested: Form navigation and generation of reports.
* Items to be tested: Functioning of forms and buttons.
* Purpose of testing: To check the effective working of MDMS.
* Pass / Fail Criteria: Changes made on the back end like recreation of tables should affect the front end as well. If so, the test is successful.
* Assumptions and Constraints: Tables should be created and values have to be entered at the back end before testing and entity integrity and referential integrity constraints should be taken care

**5.1 Unit Testing**

Unit testing is a software verification and validation method in which a programmer tests if individual units of source code are fit for use. Some of the tests performed in the project are insert, delete, retrieve and modify.

**5.1.1 Unit Test Case 1**

**Table 5.1 Unit test case for packet interface**

|  |  |
| --- | --- |
| Sl. No. of test case : | 1 |
| Name of test : | Interface test |
| Item / Feature being tested : | Ldev.c |
| Description : | The program which checks for the existence of network interfaces is tested |
| Sample Input : | gcc ldev.c  ./a.out. |
| Expected output : | The interfaces available on the system are listed. |
| Actual output : | List of interfaces, usually eth0(), is listed |
| Remarks : | Test succeeded |

**5.1.2 Unit Test Case 2**

**Table 5.2 Unit test case for packet sniffing and analysis**

|  |  |
| --- | --- |
| Sl. No. of test case : | 2 |
| Name of test : | Sniffing test |
| Item / Feature being tested : | Testpcap2.c |
| Description : | The program which sniffs the packets flowing through the network is tested |
| Sample Input : | gcc testpcap2.c  .ping www.google.com |
| Expected output : | The number of packets flowing through the network along with their protocols is listed. |
| Actual output : | No of packets and protocols listed along with headers |
| Remarks : | Test succeeded |

# Chapter 6

**Conclusion**

**6.1 Summary**

After running the implementation created, a user can browse a webpage and detect the packets being sent to the server containing the webpage requested. The user can also determine the protocols being used, (for example, UCMP protocols if the user decides to ping a webpage instead of browsing it) and also can request to

**6.2 Limitations**

* This implementation is Linux-only.
* Timeouts during the packet capture process are not supported on Unix.
* If the network has a firewall set up, the sniffer may malfunction depending on the protocols being blocked by the firewall.

**6.3 Further Enhancements**

* Implementing the program on Windows OS
* Implementing high level filters to provide better and more accurate results for packet analysis.
* Implementing a Graphical user interface.

**References**

[1] W. Richard Stevens, Bill Fenner and Andrew M. Rudoff, *UNIX Network Programming: The Sockets Networking API,* Volume 1, 3rd edition, 2007, Addison-Wesley, 2007, ISBN-10: 0-13-141155-1

[2] <http://www.tcpdump.org/pcap.htm>

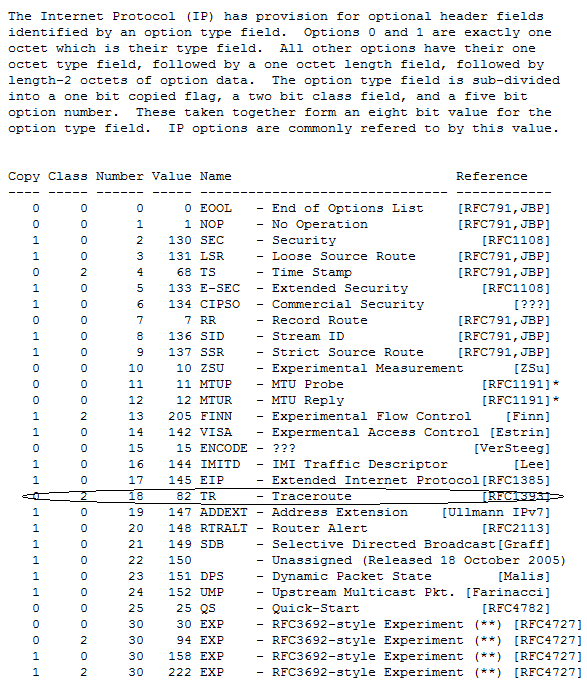
[3] <http://yuba.stanford.edu/~casado/pcap/section1.html>

[4] <http://stackoverflow.com/questions/10040883/linux-libpcap-programming>

**Appendices**

**Appendix A – Listing of tables and Source Code**

**List of Tables**

**Table 7.1: IP options with the traceroute option highlighted.** 

**Table 7.2: ICMP Message Types**

| **TYPE** | **CODE** | **Description** | **Query** | **Error** |
| --- | --- | --- | --- | --- |
| 0 | 0 | Echo Reply | X |  |
| 3 | 0 | Network Unreachable |  | x |
| 3 | 1 | Host Unreachable |  | x |
| 3 | 2 | Protocol Unreachable |  | x |
| 3 | 3 | Port Unreachable |  | x |
| 3 | 4 | Fragmentation needed but no frag. bit set |  | x |
| 3 | 5 | Source routing failed |  | x |
| 3 | 6 | Destination network unknown |  | x |
| 3 | 7 | Destination host unknown |  | x |
| 3 | 8 | Source host isolated (obsolete) |  | x |
| 3 | 9 | Destination network administratively prohibited |  | x |
| 3 | 10 | Destination host administratively prohibited |  | x |
| 3 | 11 | Network unreachable for TOS |  | x |
| 3 | 12 | Host unreachable for TOS |  | x |
| 3 | 13 | Communication administratively prohibited by filtering |  | x |
| 3 | 14 | Host precedence violation |  | x |
| 3 | 15 | Precedence cutoff in effect |  | x |
| 4 | 0 | Source quench |  |  |
| 5 | 0 | Redirect for network |  |  |
| 5 | 1 | Redirect for host |  |  |
| 5 | 2 | Redirect for TOS and network |  |  |
| 5 | 3 | Redirect for TOS and host |  |  |
| 8 | 0 | Echo request | X |  |
| 9 | 0 | Router advertisement |  |  |
| 10 | 0 | Route solicitation |  |  |
| 11 | 0 | TTL equals 0 during transit |  | x |
| 11 | 1 | TTL equals 0 during reassembly |  | x |
| 12 | 0 | IP header bad (catchall error) |  | x |
| 12 | 1 | Required options missing |  | x |
| 13 | 0 | Timestamp request (obsolete) | X |  |
| 14 |  | Timestamp reply (obsolete) | X |  |
| 15 | 0 | Information request (obsolete) | X |  |
| 16 | 0 | Information reply (obsolete) | X |  |
| 17 | 0 | Address mask request | X |  |
| 18 | 0 | Address mask reply | X |  |

**Source Code Listing**

#include<netinet/in.h>

#include<errno.h>

#include<netdb.h>

#include<stdio.h> //For standard things

#include<stdlib.h> //malloc

#include<string.h> //strlen

#include<netinet/ip\_icmp.h> //Provides declarations for icmp header

#include<netinet/udp.h> //Provides declarations for udp header

#include<netinet/tcp.h> //Provides declarations for tcp header

#include<netinet/ip.h> //Provides declarations for ip header

#include<netinet/if\_ether.h> //For ETH\_P\_ALL

#include<net/ethernet.h> //For ether\_header

#include<sys/socket.h>

#include<arpa/inet.h>

#include<sys/ioctl.h>

#include<sys/time.h>

#include<sys/types.h>

#include<unistd.h>

void ProcessPacket(unsigned char\* , int);

void print\_ip\_header(unsigned char\* , int);

void print\_tcp\_packet(unsigned char \* , int );

void print\_udp\_packet(unsigned char \* , int );

void print\_icmp\_packet(unsigned char\* , int );

void PrintData (unsigned char\* , int);

FILE \*logfile;

struct sockaddr\_in source,dest;

int tcp=0,udp=0,icmp=0,others=0,igmp=0,total=0,i,j;

int main()

{

int saddr\_size , data\_size;

struct sockaddr saddr;

unsigned char \*buffer = (unsigned char \*) malloc(65536); //Its Big!

logfile=fopen("log.txt","w");

if(logfile==NULL)

{

printf("Unable to create log.txt file.");

}

printf("Starting...\n");

int sock\_raw = socket( AF\_PACKET , SOCK\_RAW , htons(ETH\_P\_ALL)) ;

//setsockopt(sock\_raw , SOL\_SOCKET , SO\_BINDTODEVICE , "eth0" , strlen("eth0")+ 1 );

if(sock\_raw < 0)

{

//Print the error with proper message

perror("Socket Error");

return 1;

}

while(1)

{

saddr\_size = sizeof saddr;

//Receive a packet

data\_size = recvfrom(sock\_raw , buffer , 65536 , 0 , &saddr , (socklen\_t\*)&saddr\_size);

if(data\_size <0 )

{

printf("Recvfrom error , failed to get packets\n");

return 1;

}

//Now process the packet

ProcessPacket(buffer , data\_size);

}

close(sock\_raw);

printf("Finished");

return 0;

}

void ProcessPacket(unsigned char\* buffer, int size)

{

//Get the IP Header part of this packet , excluding the ethernet header

struct iphdr \*iph = (struct iphdr\*)(buffer + sizeof(struct ethhdr));

++total;

switch (iph->protocol) //Check the Protocol and do accordingly...

{

case 1: //ICMP Protocol

++icmp;

print\_icmp\_packet( buffer , size);

break;

case 2: //IGMP Protocol

++igmp;

break;

case 6: //TCP Protocol

++tcp;

print\_tcp\_packet(buffer , size);

break;

case 17: //UDP Protocol

++udp;

print\_udp\_packet(buffer , size);

break;

default: //Some Other Protocol like ARP etc.

++others;

break;

}

printf("TCP : %d UDP : %d ICMP : %d IGMP : %d Others : %d Total : %d\r", tcp , udp , icmp , igmp , others , total);

}

void print\_ethernet\_header(unsigned char\* Buffer, int Size)

{

struct ethhdr \*eth = (struct ethhdr \*)Buffer;

fprintf(logfile , "\n");

fprintf(logfile , "Ethernet Header\n");

fprintf(logfile , " |-Destination Address : %.2X-%.2X-%.2X-%.2X-%.2X-%.2X \n", eth->h\_dest[0] , eth->h\_dest[1] , eth->h\_dest[2] , eth->h\_dest[3] , eth->h\_dest[4] , eth->h\_dest[5] );

fprintf(logfile , " |-Source Address : %.2X-%.2X-%.2X-%.2X-%.2X-%.2X \n", eth->h\_source[0] , eth->h\_source[1] , eth->h\_source[2] , eth->h\_source[3] , eth->h\_source[4] , eth->h\_source[5] );

fprintf(logfile , " |-Protocol : %u \n",(unsigned short)eth->h\_proto);

}

void print\_ip\_header(unsigned char\* Buffer, int Size)

{

print\_ethernet\_header(Buffer , Size);

unsigned short iphdrlen;

struct iphdr \*iph = (struct iphdr \*)(Buffer + sizeof(struct ethhdr) );

iphdrlen =iph->ihl\*4;

memset(&source, 0, sizeof(source));

source.sin\_addr.s\_addr = iph->saddr;

memset(&dest, 0, sizeof(dest));

dest.sin\_addr.s\_addr = iph->daddr;

fprintf(logfile , "\n");

fprintf(logfile , "IP Header\n");

fprintf(logfile , " |-IP Version : %d\n",(unsigned int)iph->version);

fprintf(logfile , " |-IP Header Length : %d DWORDS or %d Bytes\n",(unsigned int)iph->ihl,((unsigned int)(iph->ihl))\*4);

fprintf(logfile , " |-Type Of Service : %d\n",(unsigned int)iph->tos);

fprintf(logfile , " |-IP Total Length : %d Bytes(Size of Packet)\n",ntohs(iph->tot\_len));

fprintf(logfile , " |-Identification : %d\n",ntohs(iph->id));

//fprintf(logfile , " |-Reserved ZERO Field : %d\n",(unsigned int)iphdr->ip\_reserved\_zero);

//fprintf(logfile , " |-Dont Fragment Field : %d\n",(unsigned int)iphdr->ip\_dont\_fragment);

//fprintf(logfile , " |-More Fragment Field : %d\n",(unsigned int)iphdr->ip\_more\_fragment);

fprintf(logfile , " |-TTL : %d\n",(unsigned int)iph->ttl);

fprintf(logfile , " |-Protocol : %d\n",(unsigned int)iph->protocol);

fprintf(logfile , " |-Checksum : %d\n",ntohs(iph->check));

fprintf(logfile , " |-Source IP : %s\n",inet\_ntoa(source.sin\_addr));

fprintf(logfile , " |-Destination IP : %s\n",inet\_ntoa(dest.sin\_addr));

}

void print\_tcp\_packet(unsigned char\* Buffer, int Size)

{

unsigned short iphdrlen;

struct iphdr \*iph = (struct iphdr \*)( Buffer + sizeof(struct ethhdr) );

iphdrlen = iph->ihl\*4;

struct tcphdr \*tcph=(struct tcphdr\*)(Buffer + iphdrlen + sizeof(struct ethhdr));

int header\_size = sizeof(struct ethhdr) + iphdrlen + tcph->doff\*4;

fprintf(logfile , "\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*TCP Packet\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

print\_ip\_header(Buffer,Size);

fprintf(logfile , "\n");

fprintf(logfile , "TCP Header\n");

fprintf(logfile , " |-Source Port : %u\n",ntohs(tcph->source));

fprintf(logfile , " |-Destination Port : %u\n",ntohs(tcph->dest));

fprintf(logfile , " |-Sequence Number : %u\n",ntohl(tcph->seq));

fprintf(logfile , " |-Acknowledge Number : %u\n",ntohl(tcph->ack\_seq));

fprintf(logfile , " |-Header Length : %d DWORDS or %d BYTES\n" ,(unsigned int)tcph->doff,(unsigned int)tcph->doff\*4);

//fprintf(logfile , " |-CWR Flag : %d\n",(unsigned int)tcph->cwr);

//fprintf(logfile , " |-ECN Flag : %d\n",(unsigned int)tcph->ece);

fprintf(logfile , " |-Urgent Flag : %d\n",(unsigned int)tcph->urg);

fprintf(logfile , " |-Acknowledgement Flag : %d\n",(unsigned int)tcph->ack);

fprintf(logfile , " |-Push Flag : %d\n",(unsigned int)tcph->psh);

fprintf(logfile , " |-Reset Flag : %d\n",(unsigned int)tcph->rst);

fprintf(logfile , " |-Synchronise Flag : %d\n",(unsigned int)tcph->syn);

fprintf(logfile , " |-Finish Flag : %d\n",(unsigned int)tcph->fin);

fprintf(logfile , " |-Window : %d\n",ntohs(tcph->window));

fprintf(logfile , " |-Checksum : %d\n",ntohs(tcph->check));

fprintf(logfile , " |-Urgent Pointer : %d\n",tcph->urg\_ptr);

fprintf(logfile , "\n");

fprintf(logfile , " DATA Dump ");

fprintf(logfile , "\n");

fprintf(logfile , "IP Header\n");

PrintData(Buffer,iphdrlen);

fprintf(logfile , "TCP Header\n");

PrintData(Buffer+iphdrlen,tcph->doff\*4);

fprintf(logfile , "Data Payload\n");

PrintData(Buffer + header\_size , Size - header\_size );

fprintf(logfile , "\n###########################################################");

}

void print\_udp\_packet(unsigned char \*Buffer , int Size)

{

unsigned short iphdrlen;

struct iphdr \*iph = (struct iphdr \*)(Buffer + sizeof(struct ethhdr));

iphdrlen = iph->ihl\*4;

struct udphdr \*udph = (struct udphdr\*)(Buffer + iphdrlen + sizeof(struct ethhdr));

int header\_size = sizeof(struct ethhdr) + iphdrlen + sizeof udph;

fprintf(logfile , "\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*UDP Packet\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

print\_ip\_header(Buffer,Size);

fprintf(logfile , "\nUDP Header\n");

fprintf(logfile , " |-Source Port : %d\n" , ntohs(udph->source));

fprintf(logfile , " |-Destination Port : %d\n" , ntohs(udph->dest));

fprintf(logfile , " |-UDP Length : %d\n" , ntohs(udph->len));

fprintf(logfile , " |-UDP Checksum : %d\n" , ntohs(udph->check));

fprintf(logfile , "\n");

fprintf(logfile , "IP Header\n");

PrintData(Buffer , iphdrlen);

fprintf(logfile , "UDP Header\n");

PrintData(Buffer+iphdrlen , sizeof udph);

fprintf(logfile , "Data Payload\n");

//Move the pointer ahead and reduce the size of string

PrintData(Buffer + header\_size , Size - header\_size);

fprintf(logfile , "\n###########################################################");

}

void print\_icmp\_packet(unsigned char\* Buffer , int Size)

{

unsigned short iphdrlen;

struct iphdr \*iph = (struct iphdr \*)(Buffer + sizeof(struct ethhdr));

iphdrlen = iph->ihl \* 4;

struct icmphdr \*icmph = (struct icmphdr \*)(Buffer + iphdrlen + sizeof(struct ethhdr));

int header\_size = sizeof(struct ethhdr) + iphdrlen + sizeof icmph;

fprintf(logfile , "\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ICMP Packet\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

print\_ip\_header(Buffer , Size);

fprintf(logfile , "\n");

fprintf(logfile , "ICMP Header\n");

fprintf(logfile , " |-Type : %d",(unsigned int)(icmph->type));

if((unsigned int)(icmph->type) == 11)

{

fprintf(logfile , " (TTL Expired)\n");

}

else if((unsigned int)(icmph->type) == ICMP\_ECHOREPLY)

{

fprintf(logfile , " (ICMP Echo Reply)\n");

}

fprintf(logfile , " |-Code : %d\n",(unsigned int)(icmph->code));

fprintf(logfile , " |-Checksum : %d\n",ntohs(icmph->checksum));

//fprintf(logfile , " |-ID : %d\n",ntohs(icmph->id));

//fprintf(logfile , " |-Sequence : %d\n",ntohs(icmph->sequence));

fprintf(logfile , "\n");

fprintf(logfile , "IP Header\n");

PrintData(Buffer,iphdrlen);

fprintf(logfile , "UDP Header\n");

PrintData(Buffer + iphdrlen , sizeof icmph);

fprintf(logfile , "Data Payload\n");

//Move the pointer ahead and reduce the size of string

PrintData(Buffer + header\_size , (Size - header\_size) );

fprintf(logfile , "\n###########################################################");

}

void PrintData (unsigned char\* data , int Size)

{

int i , j;

for(i=0 ; i < Size ; i++)

{

if( i!=0 && i%16==0) //if one line of hex printing is complete...

{

fprintf(logfile , " ");

for(j=i-16 ; j<i ; j++)

{

if(data[j]>=32 && data[j]<=128)

fprintf(logfile , "%c",(unsigned char)data[j]); //if its a number or alphabet

else fprintf(logfile , "."); //otherwise print a dot

}

fprintf(logfile , "\n");

}

if(i%16==0) fprintf(logfile , " ");

fprintf(logfile , " %02X",(unsigned int)data[i]);

if( i==Size-1) //print the last spaces

{

for(j=0;j<15-i%16;j++)

{

fprintf(logfile , " "); //extra spaces

}

fprintf(logfile , " ");

for(j=i-i%16 ; j<=i ; j++)

{

if(data[j]>=32 && data[j]<=128)

{

fprintf(logfile , "%c",(unsigned char)data[j]);

}

else

{

fprintf(logfile , ".");

}

}

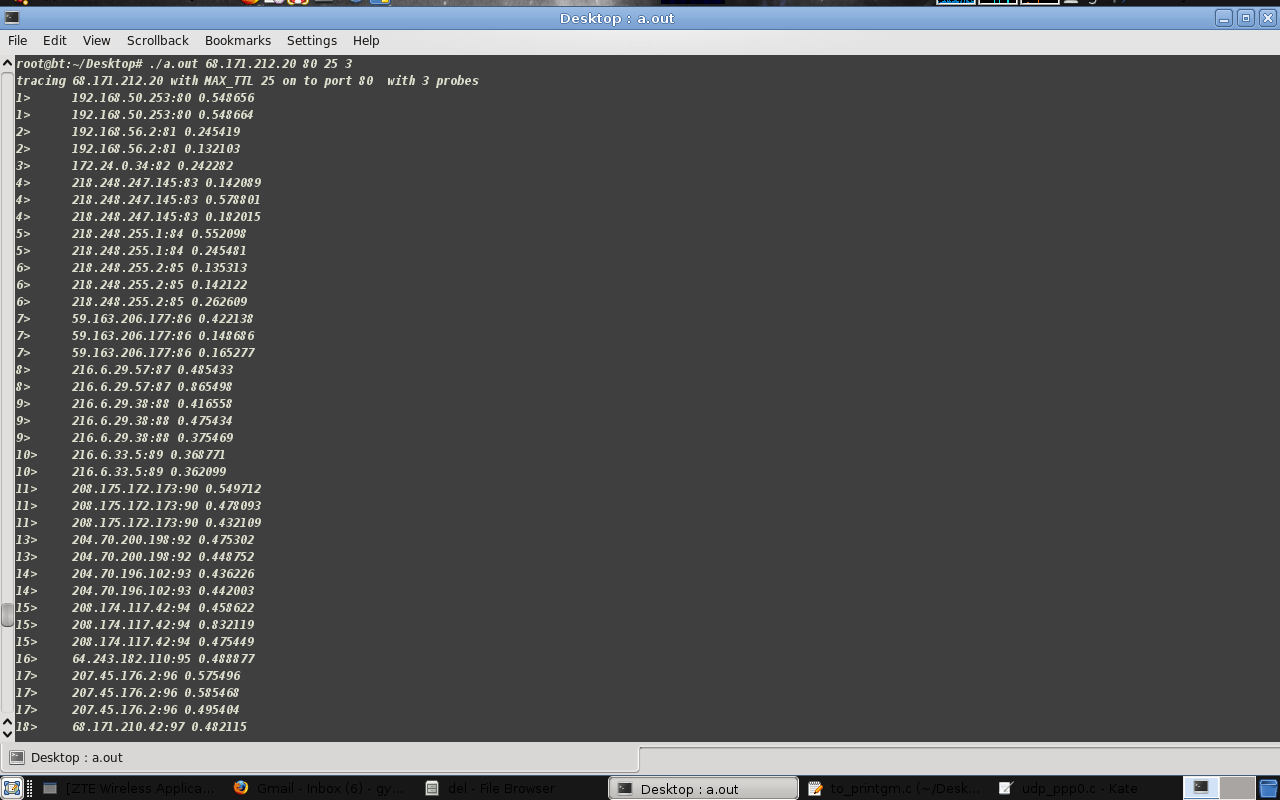
fprintf(logfile , "\n" );

}

}

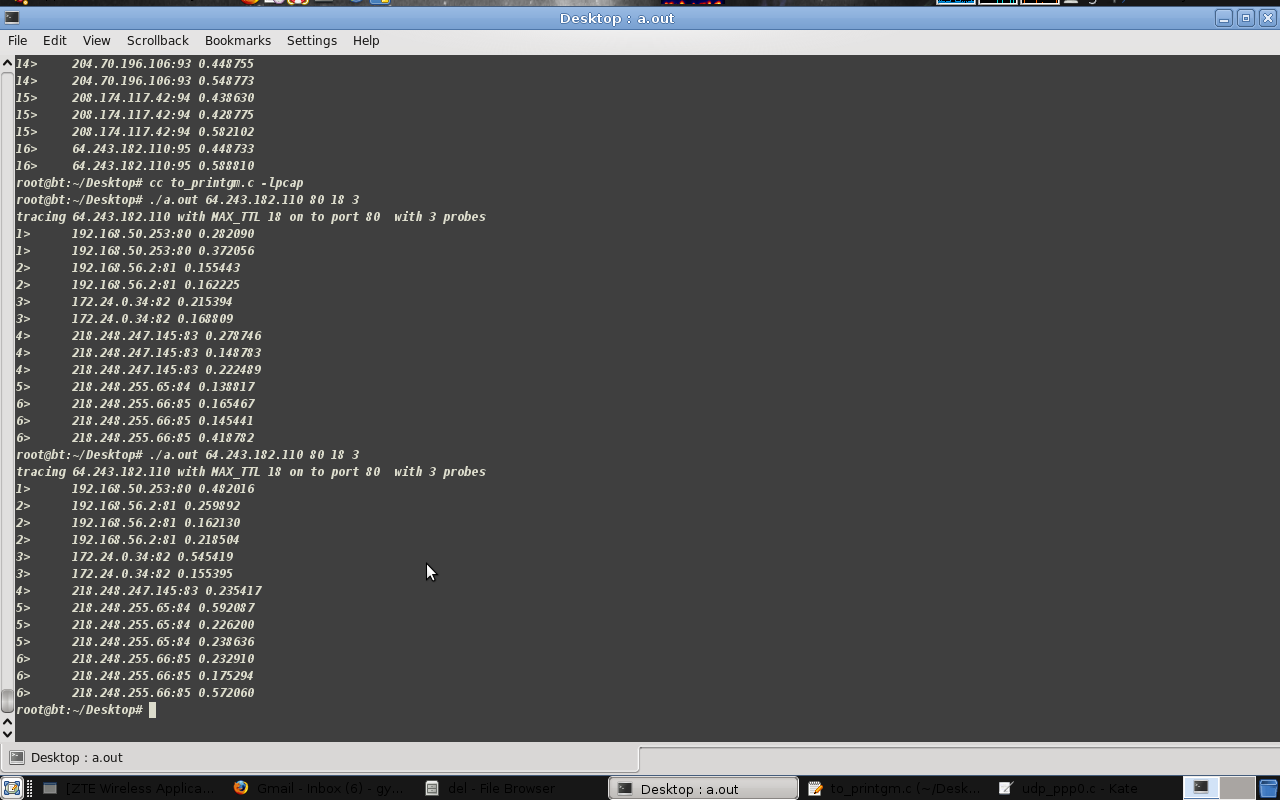
}

**Appendix B – Screenshots**



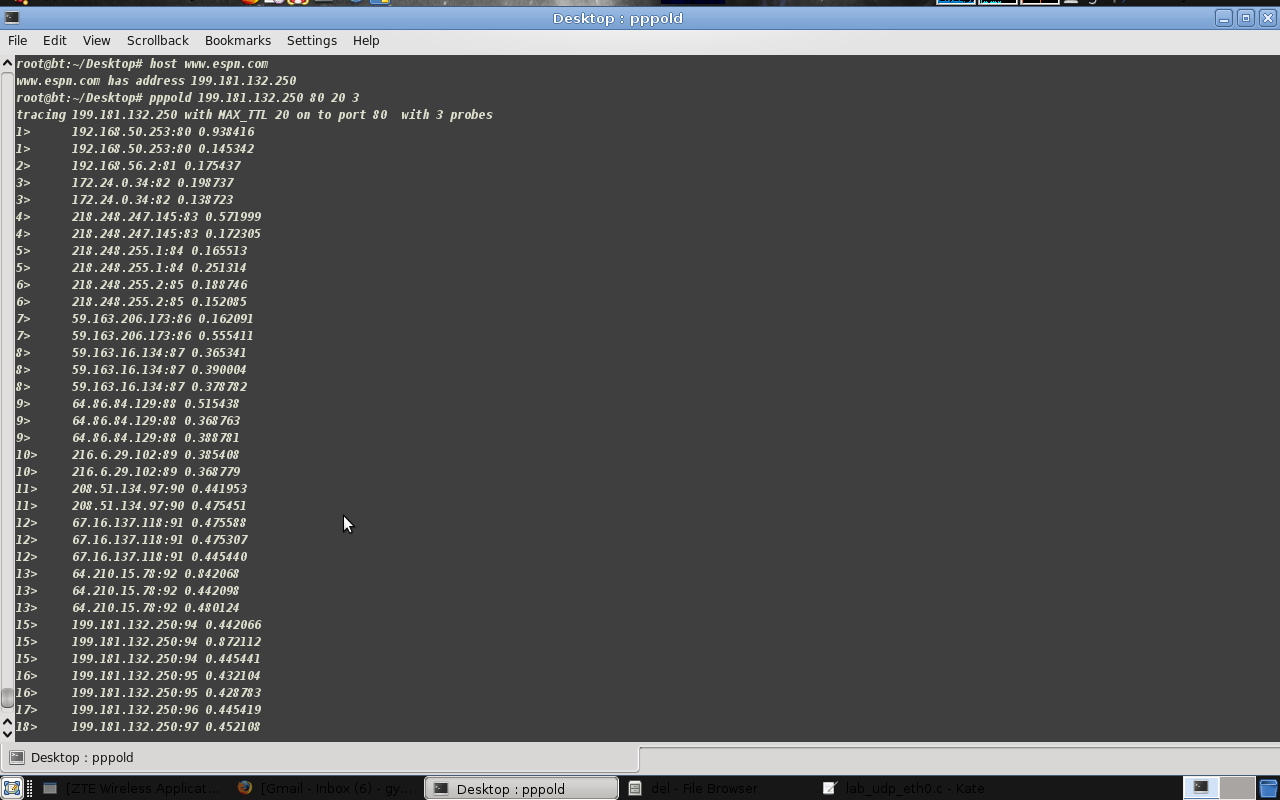
**Figure 8.1 Screenshot 1**

This is an interesting trace to the RVCE website ([www.rvce.edu.in](http://www.rvce.edu.in)), IP address was obtained by using “dig” utility. From the above screenshot it can be observed that we were not successful in tracing to the IP 68.17.210.20. But the 18th hop IP 68.171.210.42 looks like it belongs to the same autonomous system and 68.171.210.42 must be the IP of one of theirs Border routers. And it is configured not to forward the UDP packets into the Autonomous systems inner areas. But on using standard tcp trace route on probing on port 80 it can be seen that IP 68.17.210.20 is the next hop.



**Figure 8.2 Screenshot 2**

This screenshot shows two failed attempts to trace to Google website. It may be because of loss in packets (poor signal), or the configuration on the 6th hop IP, or may be the 6th hop router lost the BGP with the surrounding systems.



**Figure 8.3 Screenshot 3**

This is successful trace to [www.espn.com](http://www.espn.com) , IP address was obtained by using the “host” utility, and obtained as 199.181.132.250. And it can be observed that the ESPN website is located at 15th hop. As we keep probing even after 15 hops, the same IP keeps replying back to us.