MÄLARDALEN UNIVERSITY

Bachelor Thesis (C level)

Network Oriented C-Programming

Supervisor: Conny Collander
Author: Jorge Miguel Castellote Navas

1. ABSTRACT

This document describes a possible laboratory practice's red thread which would lead to achieve a basic formation in network programming. This thread consists of four exercises that shall guide the student through the most important and popular network programming concepts and techniques.

Each assignment has been designed to be placed in their correspondent network course, combining the theoretical load with specific tasks that may show the way to implement network applications under a free OS environment, using the C language. Every practice focus on a concrete goal, a specific network related popular program that will have to be studied or implemented and where the student will comprehend the importance of programming when learning a network related degree.

The first practice will be concentrated on familiarizing with the most important network shell commands, as well as analysing the working procedure of a simple protocol analyser that will be implemented by the student in the fourth course.

The second assignment will be concentrated on building a first network application, which will be an instant messaging program. In order to simplify the implementation and avoid the usage of complicated network concepts, a working environment has been designed by providing the students with simple functions that implements those concepts that are, for the moment, out of the student's reach.

The third exercise shall guide the student to complete the implementation of an Echo or Echo Reply program, helping to understand the importance of rigour when dealing with protocols, as well as revealing the working procedure of a connectionless oriented application.

Finally, the fourth practice is based on the implementation of a basic protocol analyser, where the student will have to work with all the OSI levels and deal with the *raw* data that is flowing in the networks.

Table of Contents

List of Figures

2. INTRODUCTION

The following thesis describes a possible path of laboratory practices that would lead the network students to be able to build network applications, making the students gain knowledge of the most important concepts related with network programming, such as the client-server architecture, the P2P architecture, the usage of sockets and protocol handling; always using the C language and working under the Linux environment.

The main goal of the thesis is to combine the usage of the mentioned concepts in order to construct real programs that are useful both for the students and for the network's field, so this way the learner will also feel motivated to consider him capable of creating his/her own instant messaging program or even his/her own protocol analyser (commonly called *sniffer*). The aim is to make the theoretical concepts of programming become practical for the network student.

Each practice has been designed for a specific level, so when combined will result on a real red thread where each practice will add more difficulty and also more interesting results.

On the first practice, the purpose is to get the learner into the network world, showing how to find and configure the computer's network devices by using shell scripts, which is the medium step between the command line and the real C programming. This practice will also contain assignments to be taught in the usage of a basic protocol analyser, which besides; will be implemented by the same student in the last laboratory practice.

For the second practice, the goal will be to generate a P2P instant messaging program with a similar working interface as the used by the famous *Skype* or *Yahoo Messenger*. As the full implementation of these types of application would require a higher programming level than the expected for the learner when coursing the laboratory, most of the code will be provided, inserted in functions, in order to simplify the codifying process, but achieving program that actually works and has a proved usefulness.

It will be in the third practice where the learner shall take his/her telematic's fundamentals to the test, building their first PDU and dealing with a real protocol as UDP in order to build an Echo or Echo Reply program, better known as *Ping*, where the student shall learn the basics of a full network-oriented application, start dealing with sockets and be familiar with the client-server architecture.

And finally, the last practice's aim will be the implementation of a real protocol analyzer, by been guided through the usage of a C library. During the carry out of this practice, the student will manage to figure out how to deal with *raw* data, to see how the network's traffic really is and to interpret this data correctly, separating headers and payload, and comprehending the high value of the information that moves through the networks.

3. BACKGROUND AND PURPOSE

Nowadays software engineering's state shows that the vast majority of the applications have a telematic section. A large number of these programs take advantage of the internet and the networks to communicate the people, the programs their selves or even the information with the users. Some of them only use their network oriented part to be updated or to feedback the developers with status and errors reports. As the software complexity increases, so do the telematic; in example, these days' applications can ensure a money transaction or even a telematic vote.

In order to be able to implement large and complex applications, more specific knowledge is needed in every developer. This people can not be experts in every field (data bases, graphic engines, artificial intelligence…). The problem for these software engineers appears when one of these fields (the networking), needs a huge background study to understand the current situation of networks. Protocol development, advanced forwarding techniques, network security, etc… They are big fields by their selves, even if they belong to the network field. A world apart exists between the workstations, maybe as big as the one that lies in every machine, or maybe bigger, and that is probably the main reason why network engineering is being taught as a unique degree.

Software engineers and network engineers have a lot of common points. The first one and probably the most important is the binary system. The fact of dealing with the same data format has greatly helped to integrate software and network engineering. Network devices have been integrated as basic peripherals in the computer architectures, dealing with the computer core and with the OS in the same language. Actually, a programming language that operates network devices can be the same as the used by software developers to implement an application.

But, if the software developers have to learn more advanced techniques every day, are they going to learn the new network techniques as well? Probably, it would be easier to show a network engineer how to program with C language, rather than explaining how a network works to a software engineer. Nevertheless, the border that separates the computer world of the networking world is not a clear and narrow line; therefore computer engineers may need to know about networks as well as networkers need to learn about computers.

In conclusion, a large sort of network concepts must be applied when creating applications and they should be implemented by network engineers in order to ensure the highest accuracy and efficiency of the programs. But these network oriented application's fragment must be codified by using programming languages, such as C.

For these reasons, a red thread of laboratory practices have been designed and implemented in order to develop and improve those programming concepts learned by the network's students during the programming courses.

4. THEORETICAL BACKGROUND

When trying to achieve the needed programming knowledge for a network degree, there are some basic common concepts that should be acquainted by every networker, regardless of those characteristic concepts of the concrete language.

Here is a brief description of the main concepts that will be used in the practices:

4.1.Client Server Architecture

The client server architecture is a model to develop information systems where the transactions are divided in independent processes that cooperate between themselves to exchange information, services or resources. The process that initiates the dialogue or requests resources is called client, and the process that attends for these requests is called server.

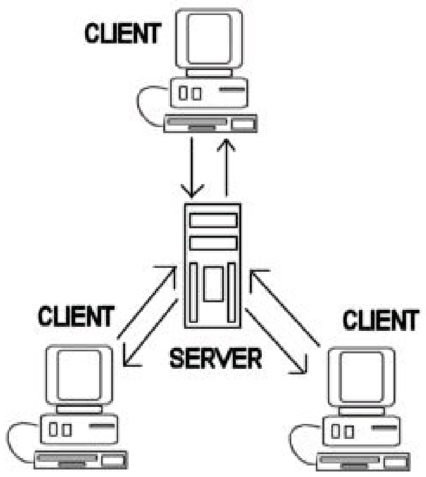


Figure 1: Client-Server Architecture

In this model, the applications are divided in such as a way that the server has the zone of the application that should be shared by several users, and the client encloses the zone that belongs exclusively to one user.

Clients usually make actions such as:

- User's interface management.
- Capturing and validating input data.
- Creation of requests and their answer's reports.

In the other hand, the servers carry out the following tasks:

- Shared peripherals management.
- Concurrent accesses control to shared data.
- Establish communication links with other networks

Whenever a client needs a service, it requests for it to the correspondent server and it answers it by providing the service. Usually, but not necessary, client and server are stored in different processors. Among the main client server architecture, it should be highlighted that:

- The server offers a unique and properly defined interface to all the clients.
- The client does not need to know the server's logic, but its external interface.
- The client does not depend on the physical location of the server or the equipment type or the OS.
- Modifications made in the client shall imply few or none modifications in the client.

4.2. Peer-to-Peer Architecture

Basically, P2P architecture refers a network that has not got either clients or servers, but several nodes that behave as well as servers or as clients of the other network's nodes. This architecture contrasts with the client server model traditionally used in Internet applications. In this kind of design, all nodes have the same behaviour and can carry out the same operations, even though the local configuration, processing speed, bandwidth and storing capability can be different.

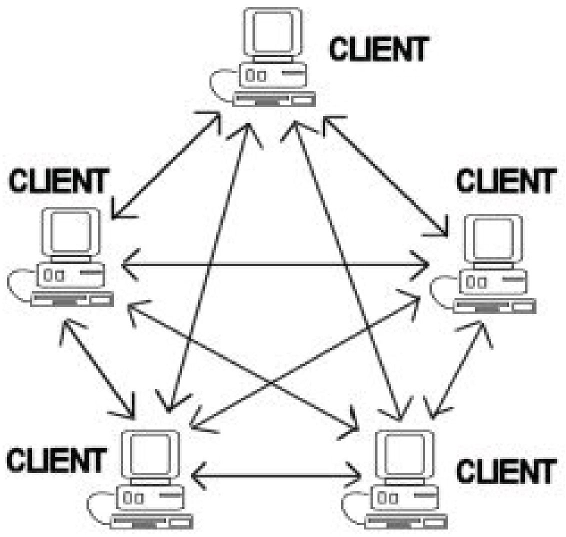


Figure 2: P2P Architecture

4.3. Sockets

Sockets are a communication endpoint. They offer an access interface to the lower levels services. A socket can be defined as a data exchange structure that is also able to configure the associated parameters of the lower level. Is the main tool to use the access points between communication levels such as the OSI levels.

When defining what a socket is in the UNIX environment, a socket is a file descriptor (an integer associated to an open file). When programs make any I/O operation in UNIX, they do it through a file descriptor, and sockets are not an exception.

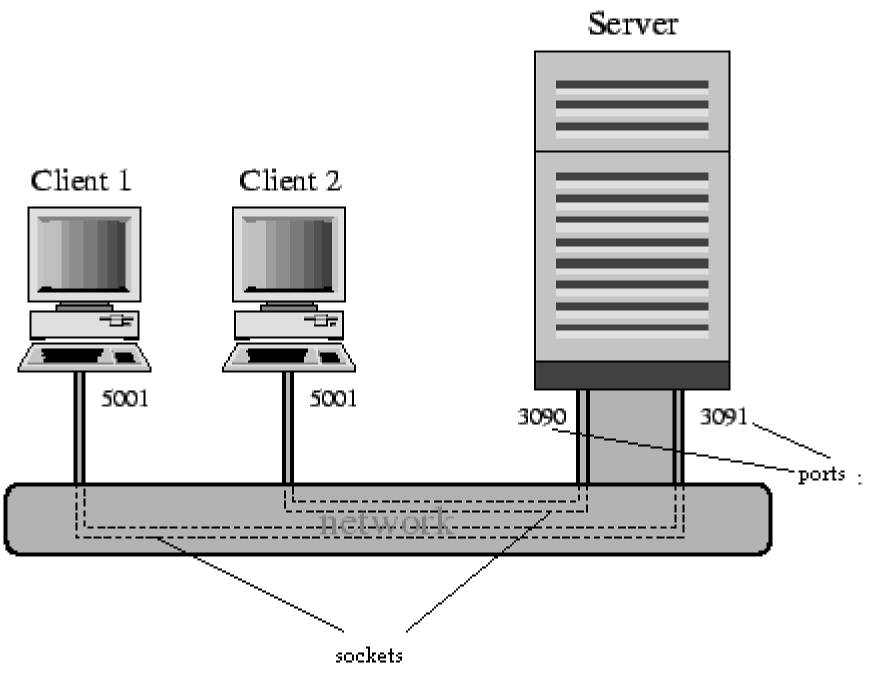


Figure 3: Sockets

Depending on how the communication is oriented, there are different types of sockets:

- Stream sockets (`SOCK_STREAM`): Connection oriented; Full-duplex; reliable; guarantees the correct order of transmitted data.
- Datagram sockets (`SOCK_DGRAM`): Connectionless oriented; no reliable; limited size of the message; does not guarantee the correct order of the transmitted data; each message is independent of the others.
- Low level sockets (`SOCK_RAW`): Allows access to levels that are below the transport level.

A more accurate description of the sockets and concretely datagram sockets can be found in the socket appendix 14.

4.4.Libpcap

Libpcap is an open source C library that offers an interface to the programmer that is used to capture packets in the networks layers. Besides, Libpcap is perfectly portable to a great number of OS.

More information can be found in the appendix D, section13.3 .

5. THE ANALYSIS

How is a practice defined? How is the learner leaded to accomplish an assignment? How can the student build a network program without previous experience?

There are many ways to guide to the implementation of a practice, probably as much as students are, for that reason a unique way of focus each practice may insufficient to make all the students develop their programming ability and understand the importance of programming in the networks field.

This section makes an effort to identify what is needed to define a laboratory practice, which are the requirements, which are the objectives.

The main goal is to apply programming concepts to real applications; therefore the development of a practice should end in a concrete result, not only theoretical aspects. The student should be able to see a final result of the taught concepts, understand that they are useful, that they are necessary.

When trying to develop laboratory practices for students, there are two big fields to focus the attention at:

- **The technical domain:** Even though there must be a concrete result, every network program or application's code has a part of it that is designed exclusively to make the application work in a concrete way, which does not belong to network concepts, but to computer science. This code will not help to understand network concepts. For that reason, the practices should reduce this kind of code by, for example, giving it to the students in already codified functions, or designing the practice in a way that the most part of it contains network related code. The aim of this field is to focus only in what is really needed for the students to learn about network programming, avoiding other programming styles or techniques.

- **The motivation:** As network engineers are not programmers, they are usually uninspired to deal with code, so the practice must be given a practical goal, something that is useful and that is easy to understand its usefulness. Besides, the way to guide the student to the practice's goal is also important; not all the students understand the concepts by following the same procedure. For this reason, each practice should have a different environment, a different point of view to confront it and carry it out.

According to these two fields, here are some global requirements that should be established in order to implement each laboratory practice:

- **Functional**: Each practice's result should be able to be appreciated by the student separately from the course, it should be something that can be used and tried at their own home, that really works and that is useful for a networker. It should be a networker tool.

- **Accurate complexity level**: As the whole thesis develops a whole red thread, the complexity should begin at a very low level for the first practice, and should increase proportionally during the following practices.

- **Link the practices**: In order to give the student a global point of view of the knowledge that should be accomplished, it is important to include common features or even common results en every practice, so each practice is not independent from the others.

- **Different solutions**: When dealing with communication applications, there are many ways to solve a concrete problem, e.g., connection oriented or connectionless oriented. During the implementation of the practices, the student should learn also to decide which solution is better for a specific problem or situation.

- **The importance of libraries**: Thanks to the C network libraries, a huge amount of code has been already implemented, and a larger amount of complexity has been decreased. This is a concept that must be valuated by the student.

The combination of these requirements with a different approach on each practice shall help to design a correct environment for the student where the knowledge and the motivation can easily be achieved. Four laboratory practices have been designed. Despite of the first one, the other three develop an entire application.

The following chapters describe the four laboratory practices, how they have been implemented, their working procedures and the way that will be presented to the student.

6. INTRODUCTION LABORATORY PRACTICE

6.1. Introduction

This practice's goal is to make the student be familiar with the low level network shell commands and the shell scripts. It will also try to show how the traffic is going around through the networks by using a protocol analyser that will be implemented by the same student in following practices.

6.2. Working Procedure

To achieve these objectives, there are five scripts that the learner will have to complete or analyse:

- **Example script**: This script shows the basic operations that can performed in a script.
- **MAC address finder**: This script uses the arp command to get a machine's MAC by using Address Resolution Protocol.
- **Pipeline script**: By combining the *ifconfig*, *grep* and *awk* commands to make a pipeline, this is an script that shows only the IP addresses of a workstation when calling *ifconfig*.
- **Interface configuration script**: Brings an interface down, brings it up and configures it with an IP address, a mask and a default gateway, leaving it ready to work in a network.

When it comes to the protocol analyser, the student will analyze the traffic in two different situations. The first one is the web traffic, where the learner will understand the full process of how a web page arrives to the machine that asked for it. In the second situation, the analysed traffic will be Telnet. The student will see how the remote machine is controlled from the local workstation, and that the Telnet data is not encrypted, understanding the importance of the security in the networks.

7. INSTANT MESSAGING PROGRAM

7.1.Introduction

The following application establishes a connection oriented communication between two workstations, which will carry the data transmitted from one of the machines to the other. The data that is exchanged will consist of character strings, as the goal of the application is to allow the different machines' users trade with text messages.

When the communication has been established, both users will be able to send and receive text messages, which will be displayed in a graphic interface.

How is the practice given to the student?

The approaching of this practice from the student's point of view will be providing him/her with a set of functions that can be easily used to generate this application. The main concept that is expected to be taught is the P2P architecture, making both machines involved in the communication act as client and server at the same time. A briefly approximation to the usage of sockets is also given. All the functions are described in the appendix B, section 11.3.

7.2.Working Procedure

According to the P2P architecture, the application must be the same in every machine, as no difference is made between the workstation that initiates the conversation and the one that receives it; therefore only one module has been implemented.

The program starts by asking the user if a new communication is desired (start a chat conversation), or if (s)he wants to stand by for incoming connections (figure 4). The fact of initiating the program in one way or the other does not really makes a difference in the normal working procedure, but is needed in order to decide which machine will establish the communication and which workstation will accept incoming communications.

After the starting mode selection, the program will be waiting for incoming connections (if initiated in this way), or will solicit the name of the other machine, trying to find it and displaying an error message if the connection failed.

Once the connection has been established, a chatting window will be displayed (figure 4), where sent and received messages will be received. Since the application will be blocked due to the *scanw()* function, waiting for the user to type a message; a thread will take care of the incoming messages, which allows the program to offer a live update, without the needing of typing a message to check if a new message has arrived, which would be the normal procedure without the thread's help. The following graphic describes the working procedure through a states machine.

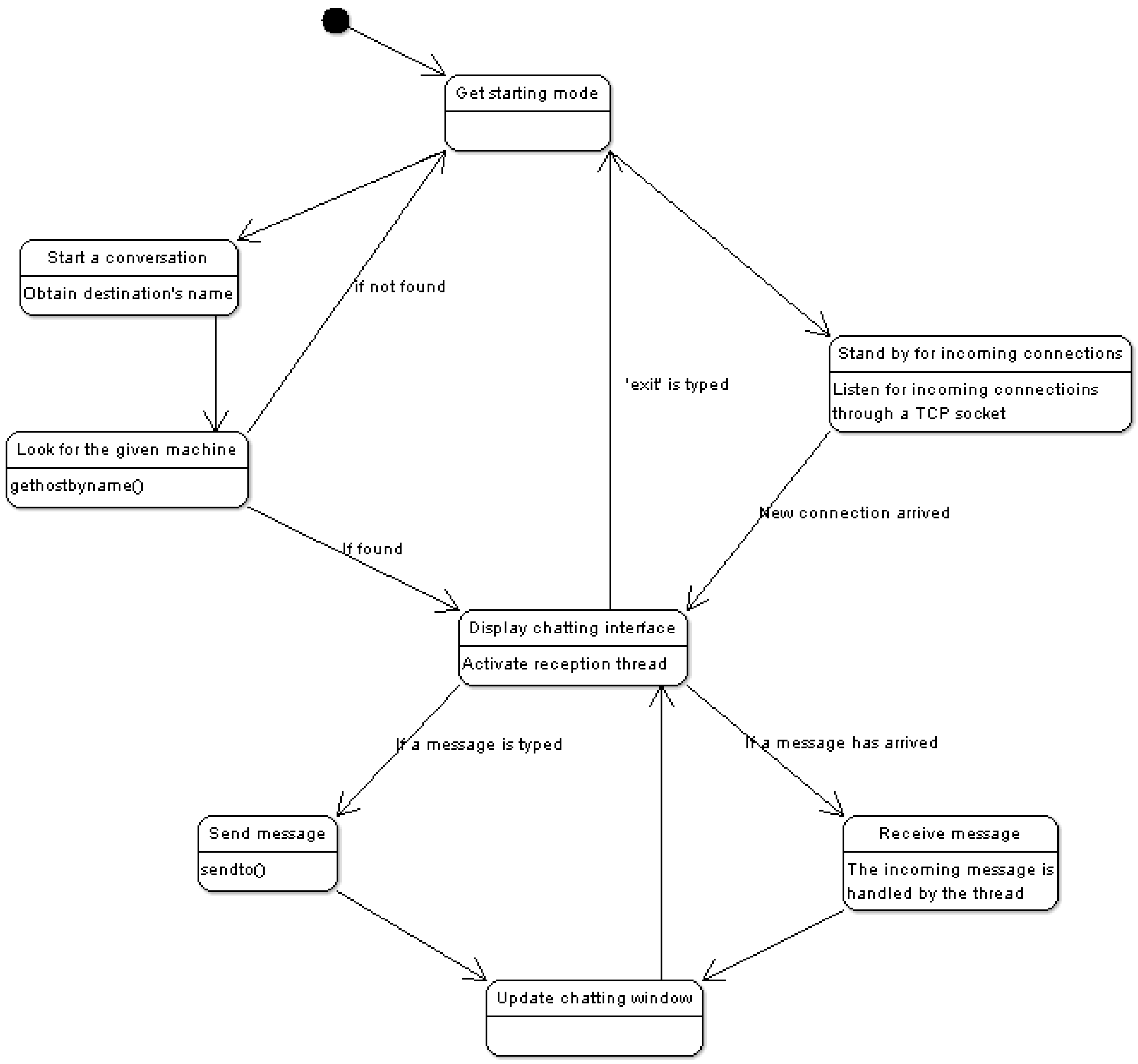

Get starting mode
Start a conversation
Obtain destination's name
if not found
Look for the given machine
gethostbyname()
'exit' is typed
Stand by for incoming connections
Listen for incoming connectioins through a TCP socket
New connection arrived
If found
Display chatting interface
Activate reception thread
If a message is typed
If a message has arrived
Send message
sendto()
Receive message
The incoming message is handled by the thread
Update chatting window

Figure 4: P2P Instant Messaging Working Procedure

Once the chatting window has been displayed, messages are exchanged and displayed on it. The window is erased when no more messages can be displayed. To terminate the program, a concrete word or characters set must typed. For the current version, the word is ‘exit’, it is to say, whenever one of the users type ‘exit’ exclusively, the conversation terminates in both machines, returning to the main menu.

The graphic interface has been created by using the ‘ncurses’ library.

7.3.Data Structures

This section describes all the non local variables that have been used to generate the application:

- **Hostent structure**

This structure is used to store the server's information when looking for it with the gethostbyname routine.

- ***Sockaddr_in* structures**

The structure that stores the socket's information. There will be one to store the local address and another one to store the remote address.

- **Socket**

Socket file descriptors. There are two of them. One of the sockets is used to listen for incoming connections (if the program is started in stand by mode), and another one is used to exchange the text messages.

- **pthread_mutex_t**

A mutual exclusion descriptor used to protect the screen of being printed simultaneously by the application and the thread.

- **pthread_t**

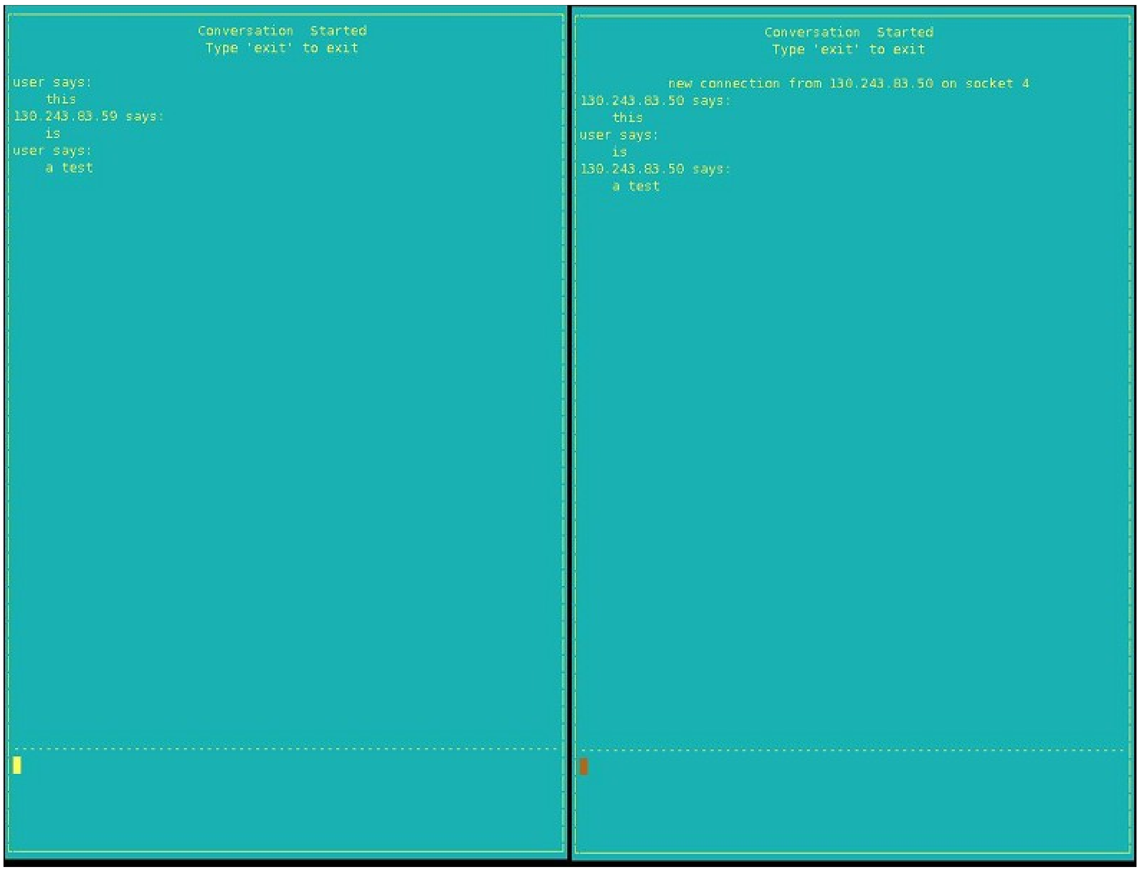
Thread descriptor that will handle the incoming messages by using the *recv()* function and will print the message on the chatting window.

- **Window**

Pointer to a ncurses' window structure where the conversation will take place.

7.4.Results

The following image shows the status of a non finished conversation. More results of the different situations are shown in the appendix B, section 11.3.



```
                  Conversation  Started
                   Type 'exit' to exit

user says:
    this
130.243.83.59 says:
    is
user says:
    a test
```

```
                  Conversation  Started
                   Type 'exit' to exit

            new connection from 130.243.83.50 on socket 4
130.243.83.50 says:
    this
user says:
    is
130.243.83.50 says:
    a test
```

Figure 5: P2P Instant Messaging Test

8. PING EMULATOR

8.1.Introduction

This application emulates the working procedure of the "Echo or Echo Reply Message", defined in the Internet Control Message Protocol (RFC-792). Basically, it implements two different modules that will follow the client-server architecture. The first one is a server module that will be listening through a datagram socket (non-connection oriented) in a fixed port, waiting for echo messages. When an echo message is received, the server will return the same payload in an echo reply message. The other module is a client, which will build an echo message and send it to the machine that is specified by the user. This application does not replace the Ping command.

How is the practice given to the student?

The approaching of this practice is step-by-step. The learner will be taught, step by step, how to build the framework for a client-server oriented application, dealing with connectionless oriented communications. At the beginning, some code will be provided in functions in order to achieve a running application quickly, and after the provided functions will be gradually substituted by the student's code. The full practice is detailed in the appendix c, section .

8.2.Working Procedure

As defined in the client-server architecture, the application is divided in two different programs, each one running in a different machine, designed to interact together. The communication process is based in datagram sockets, one for each module/program, which means that no connection will established between the two machines. Of course, one machine can interact with another as a client, and at the same time communicating with a third different machine as a server, therefore it will be running both modules at the same time.

The two modules implement a checksum routine that follows the same procedure described in the RFC-792, which is the 16-bit one's complement of the one's complement sum of the ICMP message starting with the ICMP Type. This routine will be used to calculate the checksum field when building an echo or echo reply message, and to validate a received message.

The application has been implemented by using the data structures provided by the C libraries, concretely by <sys/types.h>, <sys/socket.h>, <netinet/in.h> and <netdb.h>. The usage for these libraries in the non-connection oriented context can be found in the "Basic Usage of Datagram Sockets" appendix, located at the end of this document.

8.2.1.Server

This module will start by creating a datagram socket and binding it with the needed information, it is to say, the protocol family that the socket needs to work with, the port that will be used to listen for echo messages and the IP address of the interface where the socket will be placed.

After the socket has been created, the program will enter in an infinite loop, waiting for new requests and answering them as they arrive. When an echo message arrives (in a UDP packet), the server will ensure that all the fields are correct and that the checksum field is valid, so neither the packet has been corrupted or incorrectly built.

Once the incoming packet has been validated, the module will build an echo reply message, including exactly the same data in its payload that has been previously received. The following step will be sending the built packet back to the sender. The information about the machine that sent the message will be given with the incoming message, through the *recvfrom* C routine.

The following diagram describes what has been explained above.

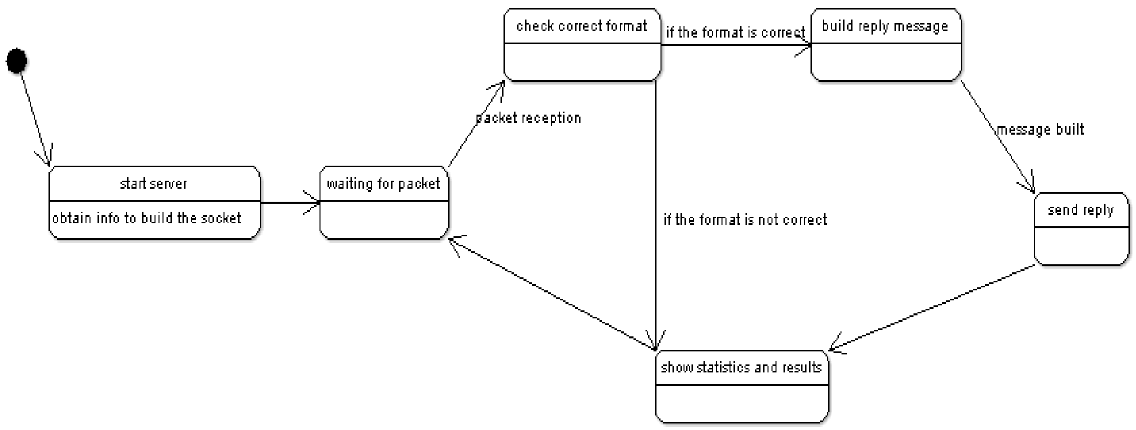


Figure 6: Server Working Procedure

8.2.2.Client

In this case, the program will start by looking for the host that was indicated in the command line when running the program (e.g. prompt>./ping 5 *hostname*). The number of echo messages that will be sent is also indicated in the command line. This searching procedure is possible thanks to the *gethostbyname* routine.

Once the host has been found (otherwise the program will terminate after showing the corresponding error message), the client will create a socket and bind it with the information of the server (protocol family, port and IP address), which has already be stored when calling *gethostbyname*.

The next step will be to build the packets and to send them by using the *sendto* C routine. In order to follow the RFC-792 in a more precise way, the payload of each message is generated randomly. This procedure is based in the *srand* C routine and the local time of the machine when generates the packet.

After the message has been sent, the received answer will be analyzed to ensure its correct format and checksum, and statistics of the procedure will be displayed (fields of the packet, origin address and data content).

The following diagram describes what has been explained above.

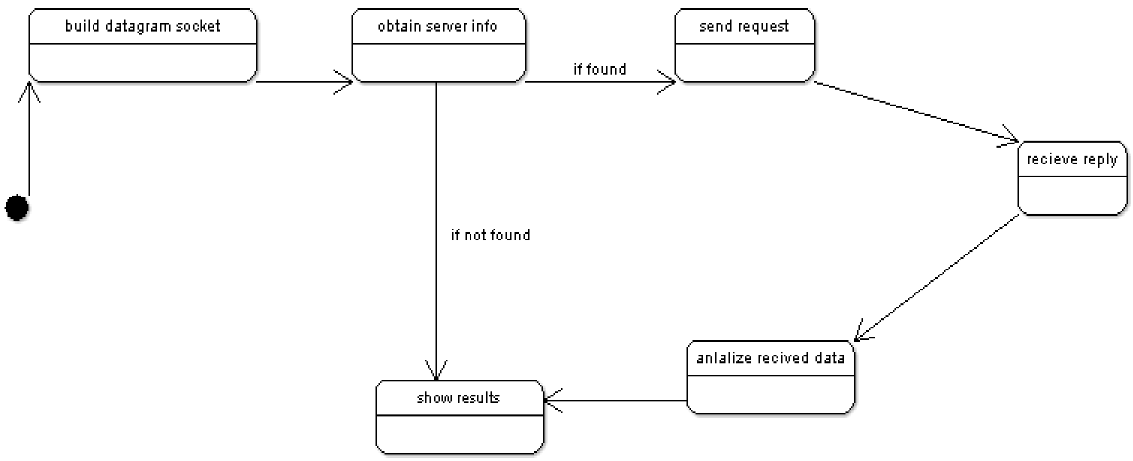


Figure 7: Client Working Procedure

8.3.Data structures

This section describes all the data types that have been used in the application. Most of this data types are used for both modules, but a few of them are specific for the client. Local variables such as counters are not defined.

8.3.1.Common Data Structures

- ICMP Echo Message

This structure contains the same fields with the same size as those described in the RFC-792, which is shown below:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-------------------------------+-------------------------------+
|     Type      |     Code      |           Checksum            |
+---------------+---------------+-------------------------------+
|          Identifier           |        Sequence Number        |
+-------------------------------+-------------------------------+
|     Data ...
+------------
```

Figure 8: Echo PDU

- Socket descriptors

Each module has one socket descriptor, defined as an integer.

- *Sockaddr_in* structures

The structure that stores the socket's information. The client has one to bind with its socket, and the server has two, one for its socket and another one to bind the client information and be able to send information back.

- Buffers

Char pointers used to point to the information that wants to be sent or received, and passed as argument for the sending and receiving routines.

8.3.2.Client Structures

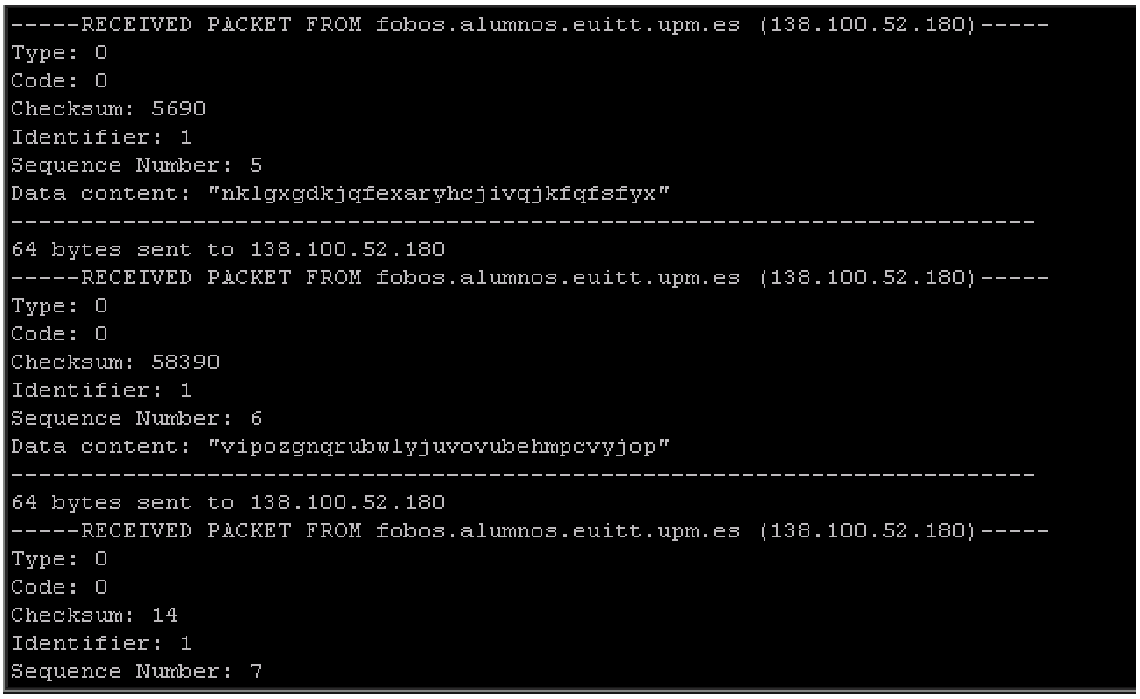
- Hostent structure

This structure is used to store the server's information when looking for it with the *gethostbyname* routine.

8.4.Results

Two files are obtained as a result, one contains the client's code and the other one includes the server's code. Once the server is running in a concrete workstation, an Echo request to this machine (if the used port is open) will result in the client's display of the Echo reply packet, showing each one of its fields.

The following picture shows the normal output after executing "*./ping 8*" (number 8 corresponds to the number of messages sent).



```
-----RECEIVED PACKET FROM fobos.alumnos.euitt.upm.es (138.100.52.180)-----
Type: 0
Code: 0
Checksum: 5690
Identifier: 1
Sequence Number: 5
Data content: "nklgxgdkjqfexaryhcjivqjkfqfsfyx"
--------------------------------------------------------------------------
64 bytes sent to 138.100.52.180
-----RECEIVED PACKET FROM fobos.alumnos.euitt.upm.es (138.100.52.180)-----
Type: 0
Code: 0
Checksum: 58390
Identifier: 1
Sequence Number: 6
Data content: "vipozgnqrubwlyjuvovubehmpcvyjop"
--------------------------------------------------------------------------
64 bytes sent to 138.100.52.180
-----RECEIVED PACKET FROM fobos.alumnos.euitt.upm.es (138.100.52.180)-----
Type: 0
Code: 0
Checksum: 14
Identifier: 1
Sequence Number: 7
```

Figure 9: Ping Emulator Results

9. TCP PROTOCOL ANALYZER

9.1.Introduction

In this case, the application will be able to capture a concrete type of network traffic (TCP traffic) as raw data, interpret it by separating the packet headers and extracting the payload, it is to say, the useful information of the PDU (Packet Data Unit). The program will also display all the information in a “readable” format.

The application is based on the libpcap open source C library, which provides an interface to the programmer that can be used to grab packets from the networks and defines traffic filters.

A full explanation of the usage of libpcap and its working procedure can be found in the appendix., section 13.3.

How is the practice given to the student?

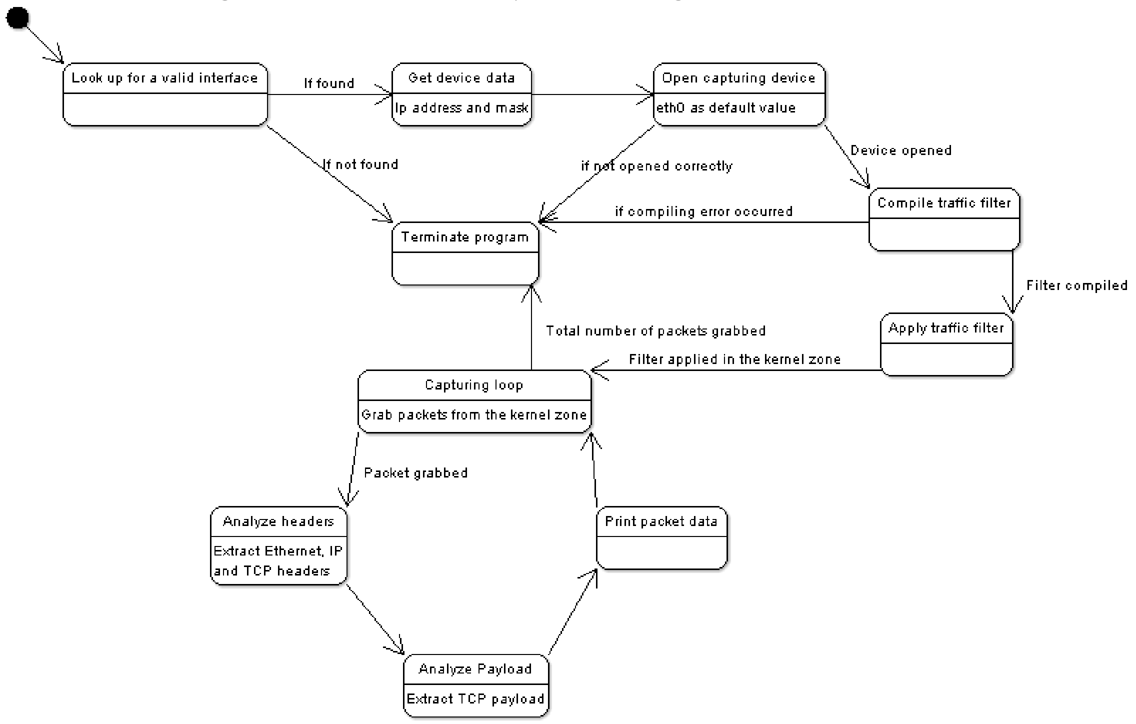
For this practice, the given background information is a tutorial of how to use the libpcap library, showing how to create basic analyzers in order to give the student enough knowledge to build their own TCP analyzer with specific requirements.

9.2.Working Procedure

The program is executed with two parameters. The first one is the number of packets that will be captured before terminating; and the second parameter is the traffic filter, specified between inverted commas.

The following diagram shows the working procedure once the program has been executed.

Figure 10: Protocol Analyzer Working Procedure



As we can see, the program needs to find a capturing device, which requires special permissions when executing it. Once the device has been opened and its addresses have been obtained, the program will try to compile the filter that has been given, which needs to be provided in the libpcap format (see appendix D, section 13.3 for further details).

When the filter has been correctly compiled and applied, a capturing loop will start. Each a time a packet is grabbed, the information of the Ethernet, IP and TCP headers is extracted, as well as the payload. The format of this information is changed from network format, and after printed in the standard output. Each captured packet is stored in a character string as raw data, and is extracted by using accurate data structures with the same size and parameters of the target information, to cast the character string.

In order to be able to control the network device and the packets that are captured from it, the application needs to enter the kernel zone. Libpcap provides safe routines to protect the kernel from any possible failure and its consequent malfunction. This concept is explained in greater detail in the appendix D, section 13.3.

9.3.Data Structures

Data structures are a basic pillar for this application. When a packet is grabbed, all the information is a long string of characters without any delimitations, marks or format, therefore we use data structures pointers that emulates exactly the type of information we want to gather (Ethernet header, for example), and make them point to the beginning of this desired information, with the help of the castings; what means getting all the information automatically without making complicated binary operations. This process as well as the provided data structures (from libpcap) is described carefully in section 13.3.

The following data structures have been created specifically for this application:

- **Struct info**

This structure contains all the data that is going to be printed. The data is copied from the Ethernet, IP and TCP structures or directly from the raw data to the correspondent field. It is also the basic piece to create a linked list that includes all the captured packets.

- **Tnode**

This is the unit of the list. It has a struct info field and a pointer to the next node. The usage of the list has not been defined yet as there is no usage of the captured packets after they have been grabbed. Its utility has been defined for possible further analysis that needs to study a concrete number of the captured packets.

- **Struct my_ip**

This data structure emulates the IP header, naked of options. Contains the same number of fields with the same size, so when casted in the appropriate position of the raw data's character string, will automatically include the IP header in itself. The following image describes the IP header according to the RFC-791:

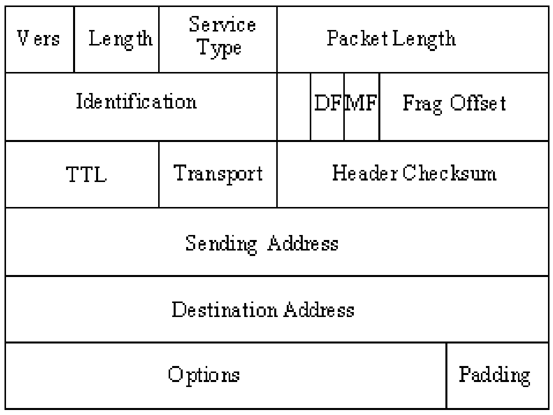


Figure 11: IP Header

9.4.Results

The number of possible results that can be obtained is as big as the different number of packets that go round the network. ARP, RARP, UDP, web traffic, telnet and ssh communications, etc…Can be easily sniffed.

Nevertheless, the application focus its efficiency in TCP traffic, therefore is easier to understand the content of TCP packets when they are printed, rather than other transport layer packet.

The following information is displayed for each grabbed packet:

- **Ethernet Data**

Destination and source address is displayed, as well as the protocol carried and the total length.

- **IP Data**

Destination and source addresses, carried protocol, header and packet length, version, time-to-live and offset.

- **TCP Data**

Source and destination ports, header and payload length, and the payload content.

The payload content is usually encoded, so only estrange characters can be seen during a encoded communication, but traffic such as normal web traffic, telnet or chat communications can be clearly read.

10.APPENDIX A: INTRODUCTION LABORATORY PRACTICE

10.1.Introduction

In this laboratory practice you will learn how to use network shell commands by writing basic scripts and learn how to use a basic protocol analyzer, making you able to analyze the traffic that goes around the networks..

Take a look on the following script; it will give you the necessary knowledge to build the required scripts of the assignments. Test it by changing the permissions of the file to executable with `chmod`.

```
#!/bin/sh
# example.sh
# This is a comment
# Do not change the first line. It must be there.

echo "This system is: `uname -a`" # uses the output of the command
echo "My name is $0" # intrinsic variables
echo "You gave me $# parameters: "$*
echo "The first parameter is: "$1
echo -n "What is your name? " ; read your_name
echo Notice the difference: "Hello, $your_name"
echo Notice the difference: 'Hello, $your_name'

#Using control sentences
FOLDS=0 ; FILES=0
for file in `ls .` ; do
  if [ -d ${file} ] ; then # if the file is in the folder
    FOLDS=`expr $FOLDS + 1`  # FOLDS = FOLDS + 1
  else if [ -f ${file} ] ; then
   FILES=`expr $FILES + 1`
  fi
  case ${file} in
    gif|*jpg) echo "${file}: graphic file" ;;
    *.txt|*.tex) echo "${file}: text file" ;;
    *.c|*.f|*.for) echo "${file}: source code file" ;;
    *) echo "${file}: generic file" ;;
  esac
done

echo "There are ${FOLDS} folders and ${FILES} files"
ls | grep "ZxY--!!!WKW"
if [ $? != 0 ] ; then # Extract the result of last command
  echo "ZxY--!!!WKW not found"
fi
echo "enough... For further information type 'man bash'."
```

10.2.Assignments

10.2.1.Script 1: Finding the MAC address

Imagine that you are working in a concrete network and you need to find a concrete workstation's MAC. The command `arp -n "hostname/IP address"` will be enough. Now let's try to include this command in a shell script. Start your file with `#!/bin/bash` (all the scripts should start with this line), include the command and get the hostname or the IP address from the command line, e.g. (let's assume that find_mac.sh is the name of your script): `./find_mac.sh 192.168.1.1`.

10.2.2.Script 2: Finding the route

When your machine tries to exit your network, does it use always the same gateway? Sometimes, large networks may have different gateways depending on the destination. Maybe one router will act as gateway for all those packets that are leaving the global network, and other router can be the destination of those packets that are heading other networks of the same global network. Copy the following script and find out how many gateways do you have by trying to reach other networks.

```
#!/bin/bash
#get_route.sh
#gets the route to the given host
#usage: ./get_route.sh "destination host name"
#e.g. ./get_route.sh www.google.es

route -nv get $1
```

10.2.3.Script 3: Pipelines

Here you are going to use something called pipeline. A pipeline is a set of processes chained by their standard streams, so that the output of each process ("stdout") feeds directly as input ("stdin") of the next one (see pipe at wikipedia for more information).

Besides, you are going to use the ifconfig and route commands for the first time. Ifconfig is a network command that allows you to configure the network interfaces parameters, as well as shows you the current configuration. Route allows you to manually manipulate the routing tables of your computer, as well as shows you the current configuration.

Copy the following script, execute it, and describe exactly what the script does, and discuss each parameter of the pipeline. Help yourself with `man ifconfig`, `man grep` and `man awk`. HINT: Try to execute this script in a machine with more than two network interfaces.

```
#!/bin/sh
#pipeline.sh

ifconfig -a | grep 'inet.[0-9]' | grep -v '127.0.0.1' | awk '{print
$2}'
```

10.2.4.Script 4: Configuring the Interfaces

When attempting to configure an interface either in Linux or UNIX, the process follows these steps:

1. Bring the interface down
2. Bring the interface up to configure it
3. Name the interface with a valid IP address and its correspondent net mask.
4. Add the gateways to send the packets to depending on the destination. Usually only the default gateway is manually configured.

Use the following script to build yours with the appropriate parameters:

```
#!/bin/bash
#configure_network.sh
#Configures a network interface

INT="eth0"
IP="192.168.2.150"
MASK="255.255.255.0"
GWAY="192.168.2.1"

ifconfig $INT down
sleep 1
ifconfig $INT up
ifconfig $INT $IP netmask $MASK up
route add default gw $GWAY

echo "Initializing Device: $INT"
```

10.2.5.The protocol analyzer

"*A protocol analyzer is a computer software or computer hardware that can intercept and log traffic passing over a digital network or part of a network. As data streams travel back and forth over the network, the sniffer captures each packet and eventually decodes and analyzes its content according to the appropriate RFC or other specifications*".

We are now going to use one of these sniffers. Execute the "sniffer" file by applying the following usage:

```
./ping [number of packets that will be grabbed] "type of traffic to capture"
```

The type of traffic we will capture is TCP, so if you want to capture twenty packets of this type. Write:

```
./ping 20 tcp
```

Once the program is running, open a web browser and make a search in any internet searcher. Now take a look at the packets and explain why they have those Ethernet and IP addresses.

Execute the sniffer again to capture 110 packets, and telnet another machine. What is in the content of the last packets? Why are there several packets that have no payload?

11.APPENDIX B: INSTANT MESSAGING LABORATORY PRACTICE

11.1.Introduction

This practice is designed to understand the P2P architecture and to make the students able to implement a P2P instant messaging program.

11.2.Assignments

A company's call centre needs to build an instant messaging application to communicate their employers while they are on the phone. The desired application needs the following requirements:

- The application will have two starting modes.
- In the first mode, the program will ask for the name of the machine that wants to be communicated. The name will be taken through the standard input (keyboard). After this step, the conversation shall start. It is the same procedure as double clicking in a contact you want to have a conversation with when using a standard instant messaging program.
- In the second mode, the program will wait for incoming connections, blocking the application until a new conversation starts.
- When starting in the first mode, only machines running in the second mode shall be reached.
- Once the conversation has begun, users shall be able to type messages at the bottom of the chatting box. Sent and received messages must be displayed at the top of the chatting box.
- The application will use a simple graphic interface. Routines to create the needed windows and to communicate with the user will be provided.

11.3.Theoretical Background

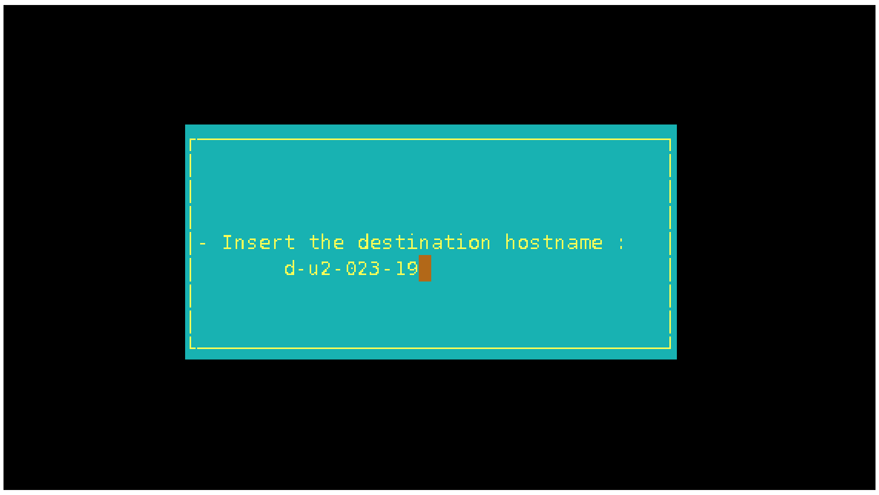
In order to build this application, you will be given 10 routines that you may use to build the program. You may find them in the provided "chat.h" helping file, and they will be implemented in the "chat.c" file. The mentioned routines are described as follows, ordered by alphabetic order:

5. **void close_app(void)**

As there is a graphic engine and a thread running at the same time with your application (don't worry, you don't even need to know what a thread is or how to use the graphic engine), you will need to tell them that the application is about to finish, so call this routine before your program exits.

6. void get_host_window(char *)

In order to get the name of the machine you want to connect with for your application, this routine will generate a new window where the user will be asked for the mentioned host name. The typed name will be copied to the char * parameter. Here is an example of how does it works:



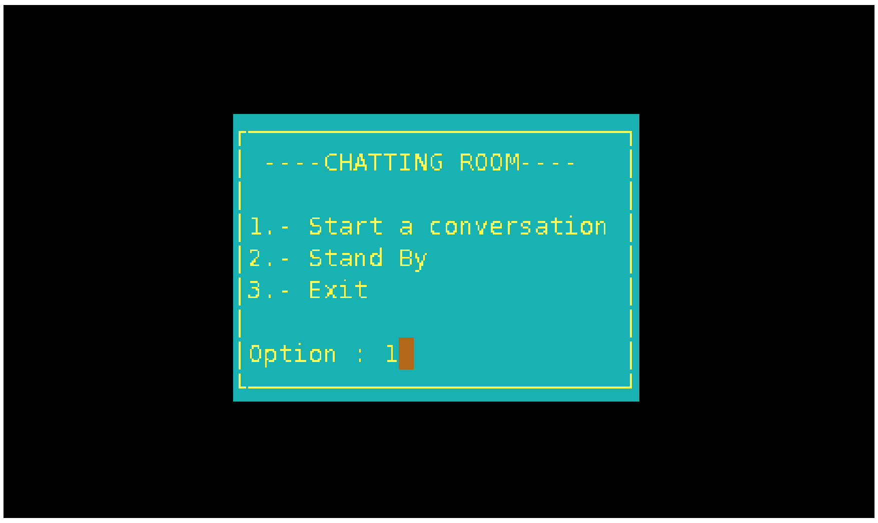
```
- Insert the destination hostname :
       d-u2-023-19
```

Figure 12: Introducing the Hostname

In this example, the user has already typed the destination host name (computer number 19 in the 023 computer room of the second floor), and is about to press Enter.

7. int menu (void)

This routine generates a window where the user can choose either the starting mode, or terminate the program. It returns the selected option. This is how it works:

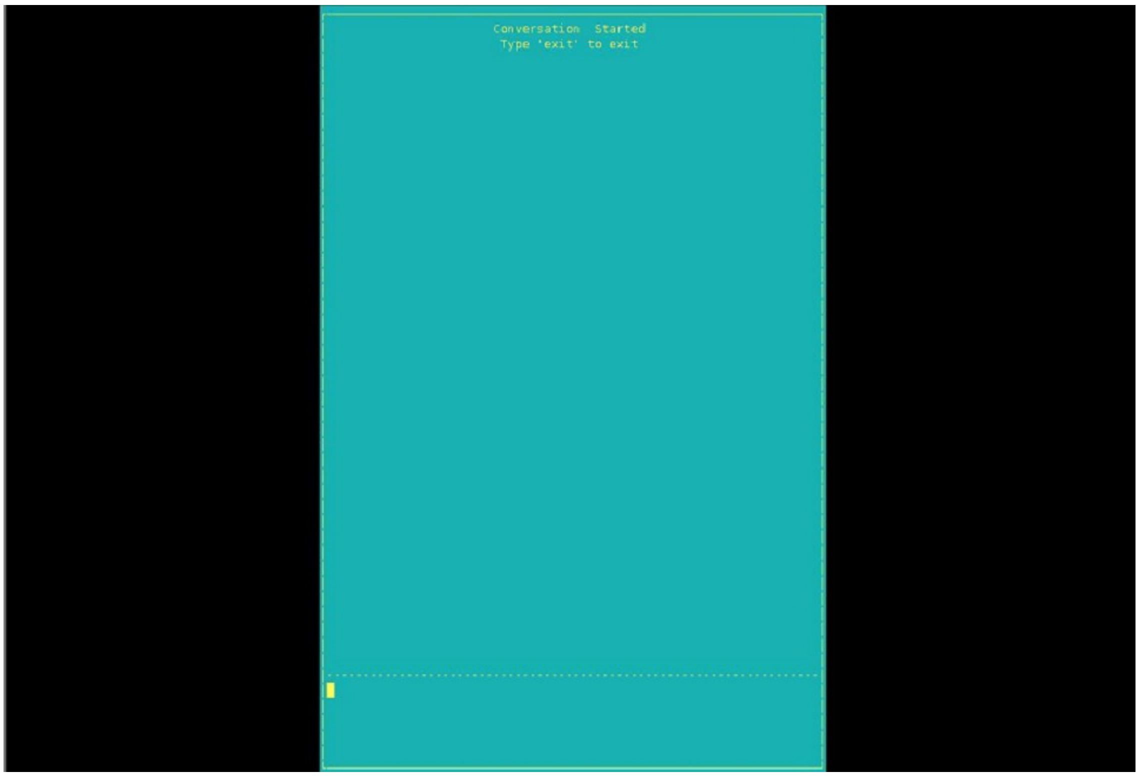


```
 ----CHATTING ROOM----

1.- Start a conversation
2.- Stand By
3.- Exit

Option : 1
```

Figure 13: Menu

8. WINDOW * print_chat_box(void)

Prints the chatting box where the conversation takes place. It returns a pointer to this window. You will need it to tell the following routines where to print a message or where to get a written message. Do not panic with this new "WINDOW *" type. Just declare one of these windows at the beginning of your main routine and use it as a parameter after obtaining a valid value for it with this routine. The following image shows the result.



```
Conversation Started
Type 'exit' to exit
```

Figure 14: Chat Window

9. void receive_message (WINDOW * win)

Once the conversation has started, a thread will take care of the incoming messages. This is needed to print income messages while the application is waiting for the user to send a message. Otherwise the user could not see that a message has arrived until he/she sent a new message, because the write_message routine blocks the program until it gets a new message to send. This "live-update" process is possible because the mentioned thread is running at the same time that your application does. Do not forget to insert your window as a parameter for this routine.

10. void set_up_app (void)

Use this routine at the very beginning of your program to start the graphic engine and the thread.

11. int set_up_client(char*)

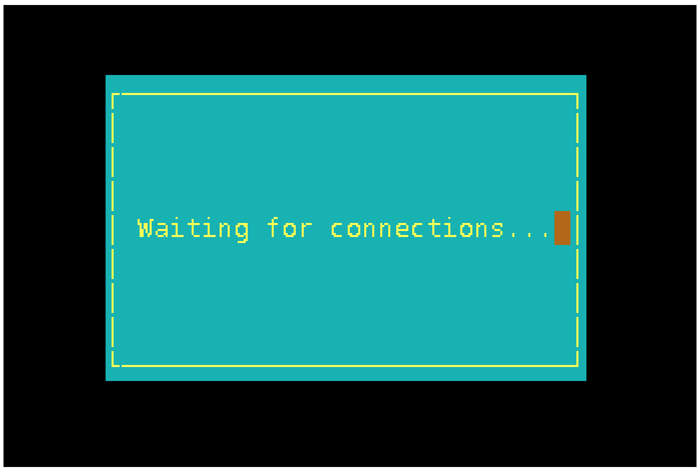
Depending on the starting mode, it could be said that the application will be running as a client (when attempting to contact another machine), or as a server (when waiting for connections). Use this routing to set up all the needed parameters from the two machines you are connecting. Be careful, if something went wrong when trying to connect, the routine will return an integer below zero.

12. void set_up_server(void)

If the user decides to run the application as a server, use this routine to let the other machines find you.

13. void wait_window(void)

When the user decides to stand by for incoming connections (run as server), a "waiting-for-connections window" shall be displayed. This routine will make this for you. Here is the graphic result:



```
Waiting for connections...
```

Figure 15: Waiting for Connections

14. int write_message (WINDOW *, char *)

This routine gets a message from the user and prints it in the chatting box. It copies the message in the given character string. Returns '1' if a normal message was typed, and '0' if only carriage return (Enter) was pressed.

11.3.1.Remarks

11.3.1.1.Three non-used routines.

If you decide to take a look on the “chat.c” and/or the “chat.h” file, you will find out that there are another three routines that are not described above. The reason is that these routines are being used by some of the above detailed.

11.3.1.2.Start variables.

Before starting to implement the program, some variables must be declared before the main routine. They are global variables that need to be used by the provided routines. They are shown as follows:

```
struct hostent *he;  /*structure used by functions to store
information about a given host */
struct sockaddr_in localaddr; /* local address */
struct sockaddr_in remoteaddr;/* server address */
int sock=0;       /* socket descriptor */
int lSock;        /* listener socket descriptor */
int i=0;          /* global variable used to store the 'y' axe position
in the chat box */
pthread_mutex_t mutex; /* thread mutex descriptor used to protect data
when printing int the screen */
```

Regarding the int i variable, please initialize it to ‘3’ before calling any of the routines that print in the box, as two or three lines will be placed at the top of the mentioned box.

It is also necessary to define the port where the communication will take place. Avoid ports below 1000. An example is given.

#define PORT 6069

11.3.1.3.Sending messages

In order to establish contact with the usage of sockets, no routine is provided to send the message that is written by the user (copied in the character string that is used as a parameter for “write_message”). You will need to use one of the routines from the sockets libraries. Here is the synopsis (obtained from the UNIX/Linux manual):

```
ssize_t send(int s, const void *msg, size_t len, int
flags);
```

Where *int s* is the socket used to send the message (*sock* in your application), *const void *msg* is the character string you want to send, and *size_t len* is the length of the message (you can use the result of *sizeof()* to obtain this length). No flags are needed, so a simple ‘0’ will be enough for that parameter. The number of bytes sent is returned, and ‘0’ bytes are sent if the other endpoint of the communication has closed.

11.3.1.4.Libraries

These are all the libraries you may need to build the application:

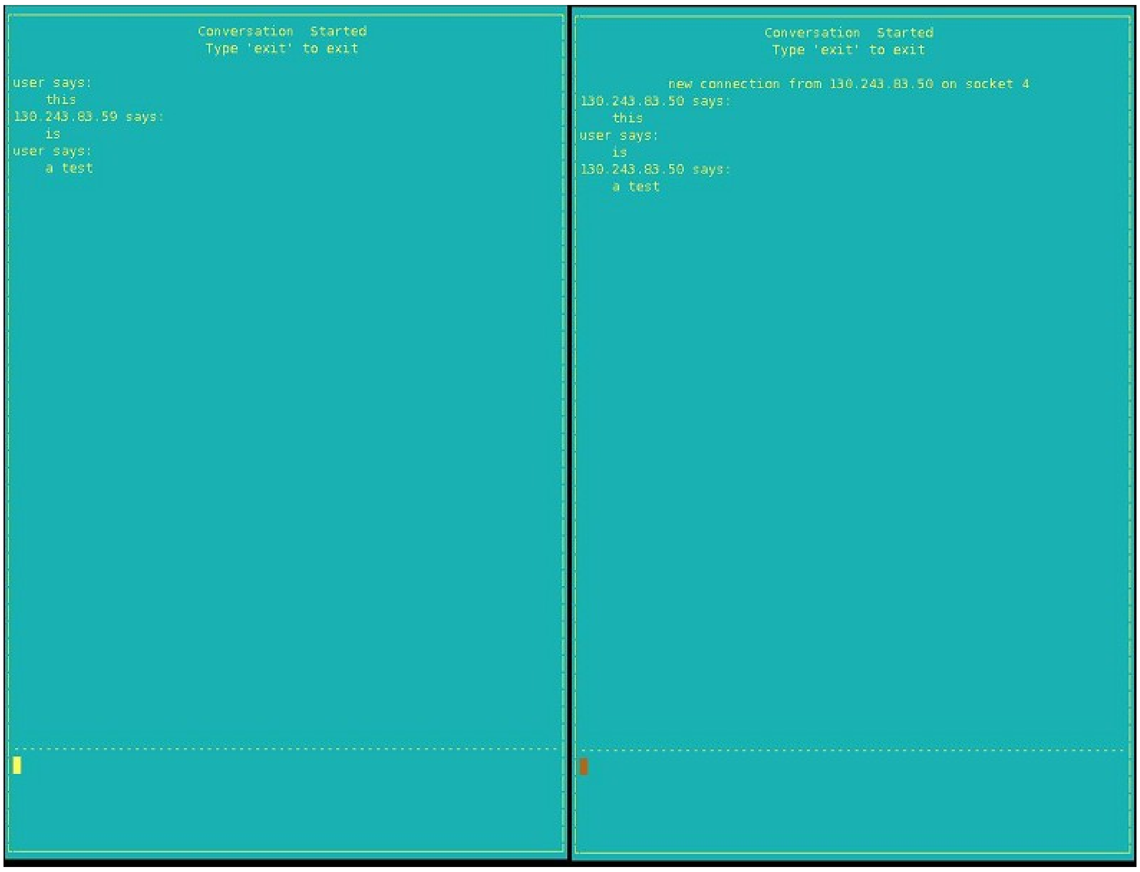
```
#include <stdio.h>
#include <stdlib.h>
#include <ncurses.h> /*Graphic engine*/
#include <string.h>
#include <unistd.h>
#include <sys/types.h>  /*network library*/
#include <sys/socket.h> /*network library*/
#include <netinet/in.h> /*network library*/
#include <arpa/inet.h>  /*network library*/
#include <netdb.h>      /*network library*/
#include <pthread.h>
```

11.3.2.Working Procedure Recommendations

Before codifying, it is important to make an idea of what are we going to do. First of all, keep in mind that there are three *cases* in the main menu. You may need to show this menu at the beginning of the program, but also when a conversation has finished, so consider making a loop until the exit option is pressed.

To finish a conversation, a special character or word should be inserted. For example, if one of the users types the word 'exit', the conversation will finish. The thread will show that the other endpoint have closed his socket. Here you go some graphic examples of the working procedure.

Same conversation watched in the two connected machines:

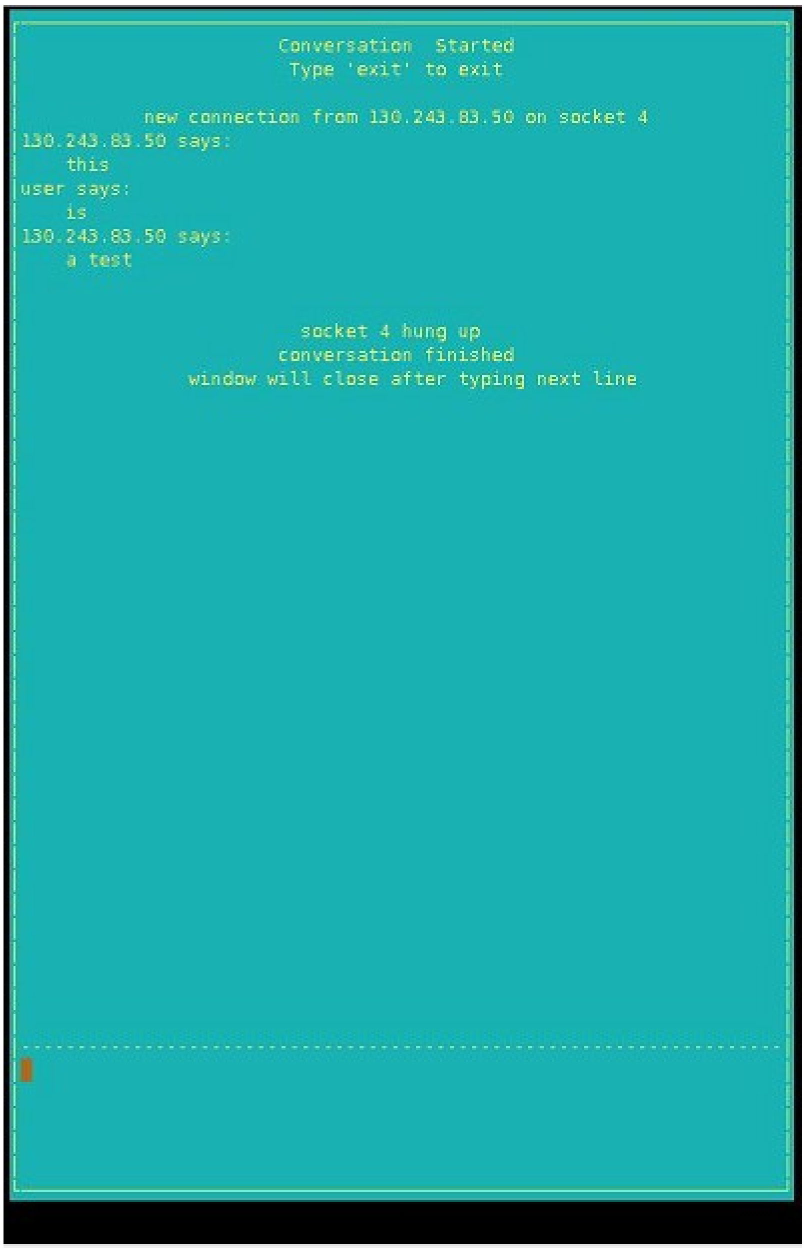


```
                 Conversation  Started
                 Type 'exit' to exit

user says:
    this
130.243.83.59 says:
    is
user says:
    a test
```

```
                 Conversation  Started
                 Type 'exit' to exit

            new connection from 130.243.83.50 on socket 4
130.243.83.50 says:
    this
user says:
    is
130.243.83.50 says:
    a test
```

Figure 16: Test Conversation

The conversation has been terminated because the other endpoint typed ‘exit’.



```
                    Conversation  Started
                     Type 'exit' to exit

            new connection from 130.243.83.50 on socket 4
130.243.83.50 says:
    this
user says:
    is
130.243.83.50 says:
    a test


                      socket 4 hung up
                    conversation finished
             window will close after typing next line
```

Figure 17: Terminated Conversation

12.APPENDIX C: PING EMULATOR LABORATORY PRACTICE

12.1.Introduction

Lots of network applications are being used nowadays, such as traceroute, ethereal (protocol analyser), e-mail applications, voice over IP applications, etc…This kind of applications need a large knowledge about how do the networks work in order to be built; therefore common programmers are not able to create this kind of applications. A person with specific knowledge is needed. This is the main reason why networkers need to know about programming languages, because only they know how do a protocol works, or how to maximize the communication performance.

Objectives

In this lab you will build an application that emulates the ping command by using the C language, the client-server network architecture, and the sockets communication system.

It is recommended to take a look on the RFC 792, section "Echo or Echo Reply Message"; in order to understand how the Ping works.

It may also be useful to read the given appendix about connectionless sockets, so you can get used to the framework and easily understand how the lab is developed.

12.2.Assignments

12.2.1.Connecting Machines

In this first assignment you will build a connectionless oriented application that sends characters between two machines, using provided functions from the ‘udp.h’ file.

According to the client-server network architecture, the machine that sends the echo message will be the client, and the machine that receives the message will be the server. Each machine/computer, will be able to be client and server at the same time. The client operates through one socket, where the data will be sent and received, and it will be linked to one of the computer’s port. On the other hand, the server will establish its communication through a different socket, which will be linked to the same port. The information about the used protocol, the port and other features needs to be stored in a special structure.

Assignment 1

1.1 ***[O]*** *Build a C file for the client, another one for the server and include the* 'udp.h' *header file.*

1.2 ***[O]*** *In the C language, if you want to use sockets, you will need a socket descriptor (it can be an integer), and some information about the machines that the socket is connected with. This information needs to be stored in a special type, a structure called '*`struct sockaddr_in`*'. The client only needs information about the server, but the server needs information about itself and about the client that is connected with. Create one* '`sockaddr_in`' *structure for the client and two for the server. Create also a socket descriptor for each file.*

1.3 ***[O]*** *Find out which fields is the sockaddr_in structure made of and describe briefly its function.*

1.4 ***[O]*** *You are now ready to use the header file. Build one socket for the client and another one for the server by using the given functions. You will need to get the hostname from the command line that executes the application (e.g.: '*`./ping hostname`*'), by using the* argv *vector in the* main *function. Remember that the hostname that is needed to build the client socket has to be an existing host with the server file running on it.*

1.5 ***[O]*** *It is time to exchange the data and run the first phase. By using the send and receive functions of the header file, try to send a character string ('*`char`*' or '*`uint8_t`*' type), and print it in the server when the data arrives. The character string must be obtained from the keyboard input stream. This means that the user will decide which characters are sent. You will also need to close the socket at the end of the source code (check '*`close`*' at the Unix manual). After this section you will have been able to establish a connectionless communication between two machines.*

1.6 ***[W]*** *At this moment, the server is closed after printing what has been received. Modify the application to allow multiple accesses to the server. The server must be able to print multiple accesses from the same machine as well as multiple accesses from different machines during the same session. After this step you will need to close the server by using the Unix '*kill*' command. Why?*

1.7 ***[O]*** *For this section, the message that has been received in the server must be returned to the client. In this case, the client will show what has been sent and what has been received. Print both the sent character string and the received character string.*

1.8 ***[O]*** *In order to improve the consignment and the reception, create two buffers in each file, one for the consignment and another one for the reception. From now on, the data that is transmitted must be taken out from the consignment buffer and stored in the reception buffer. This means that only these buffers can be used as a valid parameter for the 'send' and 'receive' functions. Link the buffer's pointer to the data that will be transmitted.*

12.2.2.Implementing the connection

After building the skeleton of the application, you will now implement those functions that are being used from the header file. In this assignment you will build the sockets. Place the code you will make in the place where you have used the header file's function.

Please consider that most of the functions you will use may return specific values in case an error has occurred, so be sure that you are observing every possible result. The 'perror' function may be useful to show what kind of error occurred.

Assignment 2

2.1 ***[O]*** *Start with the 'server_socket' function. First of all, you will need a socket descriptor. Get it from the '*`socket`*' function. Learn how to use it in the Unix manual. Review how does the Ping application work and which OSI levels is working on in order to fill the in parameters of the '*`socket`*' function (domain, type and protocol). You will also find information in the given appendix.*

2.2 ***[O]*** *Fill the parameters of the sockaddr_in structure. You can use '*`INADDR_ANY`*' to get the first available IP address for the socket. Fill the '*`sin_zero`*' field with zeros. The field is 8 bytes long and can be filled with the '*`memset`*' function. The port needs to be stored in a network format, therefore you may need to use the '*`htons`*' function (host-to-network), in order to transform from the host byte order to the network byte order . Avoid using ports below 1000.*

2.3 ***[O]*** *Now it is time to bind the socket with the server information. Use the '*`bind`*' function (check the usage at the Unix manual) to link the information with the socket. After this section the '*`server_socket`*' function will have been implemented, so check the correct operation of your application.*

2.4 ***[W]*** *The creation of the client socket is quiet different. This time you will have to find the server and get its information. This info will be stored in a pointer to a '*`hostent`*' structure. Find out which are the fields that compose the hostent structure and describe briefly its function.*

2.5 ***[O]*** *Create a hostent structure in your client and get the host information by using the '*`gethostbyname`*' function (more information in the Unix manual).*

2.6 ***[O]*** *Get the client's socket descriptor by using the '*`socket`*' function.*

2.7 ***[W]*** *Fill the information about the server in the created '*`sockaddr_in`*' structure. Be sure you are using the same port that has been used above when filling the information of the server. In order to fill the server IP address field('*`sin_address`*' field of the sockaddr_in structure), use the h_addr field of the hostent structure (e.g.:* `server_addr.sin_addr =*((struct in_addr *)he->h_addr`*). Find out how to make it in the appendix. After this step, both functions will have been implemented. Check the correct working procedure of your application.*

12.2.3.Assignment 3

It is now time to implement the 'send_data' and 'receive_data' functions. Place the code you will make in the place where you have used the header file's function.

3.1 ***[O]*** *Start with the '*`send_data`*' function. This function is based on the '*`sendto`*' function, which can be found at the Unix manual and at the appendix. Use this function to implement '*`send_data`*'. Help yourself with the sizeof operator. Remember to use the defined buffer for the outgoing the data. No specific flags are needed. Pay attention to the '*`sockaddr_in`*' structure you are giving to the function. It is an in/out parameter and you will need to make a casting.*

3.2 ***[O]*** *The following step will be the '*`receive_data`*' function. Is based on the '*`recvfrom`*' function, that can be found also at the Unix manual and at the appendix. Once the* `send_data` *function is working, you should not have problems with this implementation.*

3.3 ***[O]*** *Print in the screen the number of bytes that are sent and the number of bytes that have been received in the client as well as in the server.*

12.2.4.The Ping Command

Congratulations. You have achieved to implement the basic skeleton of every connectionless client-server application. From now on, you will shape up the application in order to make it work as a Ping emulator. Keep in mind the working procedure of the Ping command in order to understand every step you are taking.

Assignment 4

4.1 ***[O]*** *The first step is to define the Packet Data Unit that will be used as the communication unit. Defining PDUs is very useful in order to avoid unexpected quantity of bytes that may overload the reception buffers. Explain which are the fields of the Echo messages, what is their purpose and show the byte size of each field (use the information shown at the RFC 792).*

4.2 ***[O]*** *Create a structure that contains the same fields as the defined in the RFC. Each field, as well as the whole structure, must have the same size of the fields defined in the RFC. Generate two packets (PDUs) in each file. One is intended to be the consignment PDU while the other one will be used as the reception PDU.*

4.3 ***[W]*** *Fill the client PDU with the correct parameters. To fill the checksum field, use the given function in the header file, but do not use it until the rest of the PDU has been filled. To insert the data you can put your own characters or use the given function to generate random data, which would put your application closer to the real Ping.*

4.4 ***[O]*** *Link the PDU's structure pointer to the transmission buffers and send the PDU to the server. Print the received data in the server's screen.*

4.5 ***[W]*** *Once the PDU has been received at the server, make sure that all the parameters are correct, specially the checksum, by checking the content of the incoming PDU in your application.*

4.6 ***[O]*** *Make the needed modifications to the received PDU in order to obtain an Echo reply message and send it back to the client (remember to use the buffers for the transmitted data).*

4.7 ***[O]*** *Print the received data at the client's screen, split by fields, and the emulator will be basically implemented.*

12.2.5.Optional Assignment

5.1 ***[O]*** *When displaying the information received back in the client, show the name of the sender as well as its IP address.*

5.2 ***[O]*** *Implement the '*`do_checksum`*' header's function. The checksum is the 16-bit ones's complement of the one's complement sum of the ICMP message starting with the ICMP Type. When computing the checksum, the checksum field should be zero. HINT: use the '<<' and '>>' operators to solve the binary operations.*

[O] *Implement the '*`random_data`*' header's function.* HINT: *Look up in the Unix manual about the '*`rand`*' and '*`srand`*' functions. Try to avoid the appearance of special characters as carriage return, which may alter the normal output.*

13.APPENDIX D: TCP PROTOCOL ANALYZER LABORATORY PRACTICE

13.1.Introduction

For this laboratory practice, you will build your own protocol analyzer, commonly called *sniffer*. In order to implement this application, you will learn the fundamental concepts of the **libpcap** library, which is a very flexible tool to take a look on what is happening inside the networks. If the *RAW* sockets are our voice in the networks, *libpcap* will be our ears. But before start codifying, let us make a quick review of how do the LAN networks work.

13.1.1.Ethernet LANs introduction

Probably, one of the best ways of reviewing the network elements is to follow the path that one of their multiple packets follows.

Whenever an application needs to send data through the network, the first thing it does is encapsulate this data in a transport protocol (TCP, UDP). From that moment, the data becomes to be called payload. A transport protocol is used to specify how we want our packet to travel. When thinking about this, is not a new concept; when we go on holidays to the beach, we send postcards to our friends, but when we want to send an academic dossier we do it through certified mail. The reason to choose between one method and the other depends on the importance of the information, the cost price, if we are in a rush, etc…This concept can fit perfectly in the network's field, because each transport protocol has some advantages and disadvantages.

We already know that our packet will travel through certified mail (TCP), but now, how do we specify our destination? Every single machine inside the same network has a unique identifier. This identifier is called IP address and is a logical addressing, it is to say, is independent of the architecture on which the network has been built (Ethernet, Token Ring, Token Bus, etc…).

Once the IP header has been added to our packet, the only step left is to add the corresponding headers of the specific network type that our packet will travel through, which in our case is the Ethernet header (see the following image).

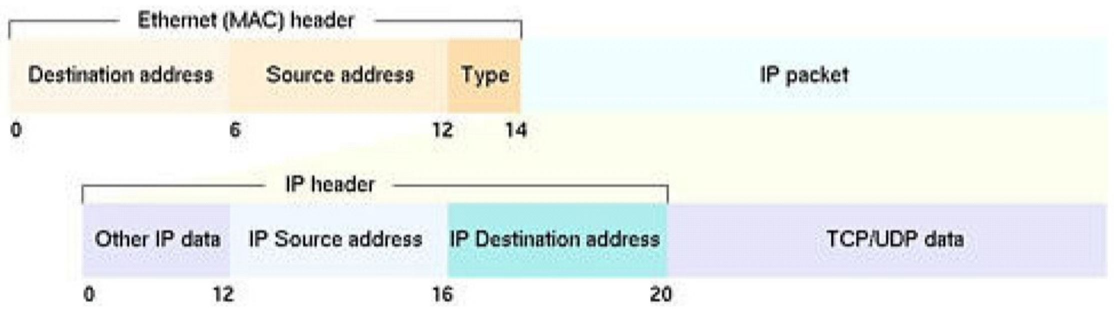


Figure 18: Ethernet and IP Headers

In an Ethernet LAN, every addressable device has an associated Ethernet address (commonly called MAC address or simply MAC) that has been recorded in the device by the manufacturer. One of the fields that the Ethernet header is made with is the MAC address of each communication endpoint.

How does the network know which is the associated MAC of an IP address? The easiest answer would be storing a list of MAC-IP equivalences in every machine. This solution actually exists in real systems, but has got the inconvenience of having a high maintenance cost and little flexibility. Therefore Ethernet networks have an IP to MAC resolution mechanism and viceversa. These protocols are called *ARP* and *RARP*.

Once we have got all this information together (Payload + TCP + IP + Ethernet), which is called Ethernet Frame, we are ready to start the transmission. In a simple model as the one we are working on, the packet is not forwarded directly to its destination, but copied by the hub to every network wire. Every machine will read the first 6 bytes of the frame where the destination MAC is codified, but only the one that has the same MAC that is shown in the packet will read the rest of the packet. As you can see, the network has not got any mechanism to ensure that a sent packet will be read only by the real destination. A small change in the network driver's configuration (promiscuous mode) shall be enough to let our machine read the full content of every single packet that travels through the network.

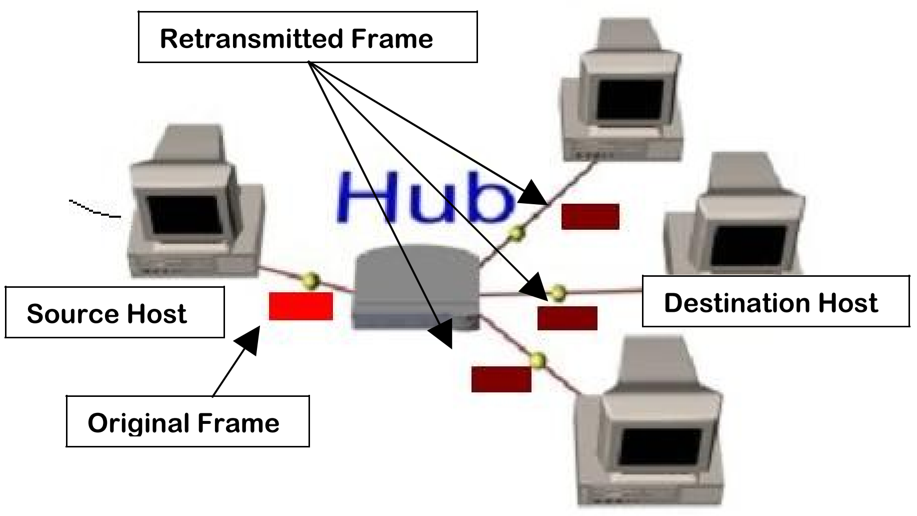


Figure 19: Non-Switched Network

13.2. Assignments

Once you have read all the below information, tried the examples, read the information again and started to understand how the Libpcap works, build an application that captures packets containing the TCP protocol, and display the following information from each packet:

- Ethernet source address
- Ethernet destination address
- Ethernet protocol type
- Ethernet frame size
- IP source address
- IP destination address
- IP payload’s protocol
- IP header length
- IP packet’s size
- TCP’s payload

13.3.Theoretical Background: LIBCAP

13.3.1.What is libpcap?

Libpcap is an open source C library that offers an interface to the programmer that is used to capture packets in the networks layers. Besides, Libpcap is perfectly portable to a great number of OS.

13.3.2.Program Outlining

No matter how complicated is the program we want to build with libpcap, it will always follows this basic outline:

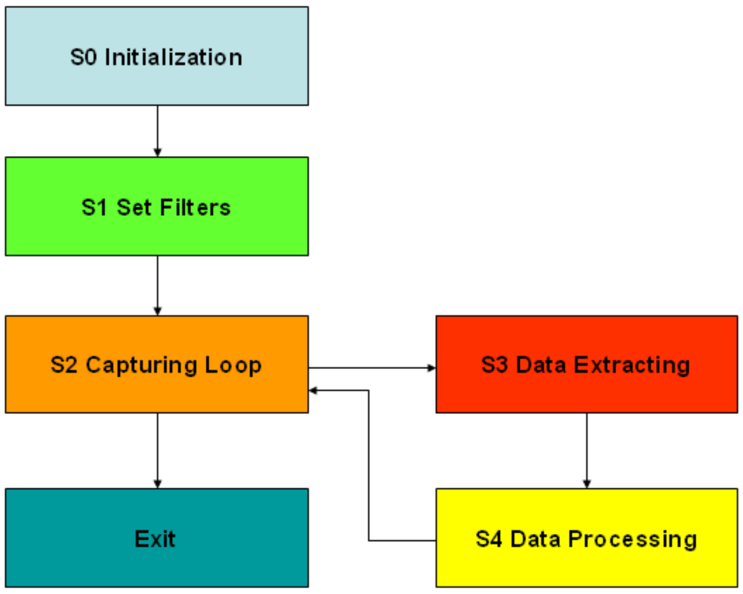


Figure 20: Libpcap Outlining

In the following sections some examples of each phase are developed.

13.4.(S0) Gathering System Information

As you can see in the above graphic, the first phase of our program is the initialization. This phase includes all the routines that are able to get information about our system: Network interfaces, configuration of these networks (net mask, network address), etc…The main routines are described as follows:

13.4.1.Specific Functions

- **char *pcap_lookupdev(char *errbuf)**

Returns a pointer to the first available capturing valid device. Returns NULL and an error description in *errbuf* in case an error occurred.

- **int pcap_lookupnet(char *device, bpf_u_int32 *netp, bpf_u_int32 *maskp, char *errbuf)**

Once the name of the valid interface has been obtained, we can look up for its network address (not the IP address), and its subnet mask. *device* is a pointer to a character array that contains a valid interface name, *netp* and *maskp* are pointers to bpf_u_int32 on which the function will copy the network address and the mask respectively. In case of error, the function reutns '-1' and an error description in errbuf.

- **int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf)**

This function returns all available capturing valid interfaces. There might be more interfaces that for one reason can not be opened to capture (due to permissions lack). These interfaces will not be shown in the list. In order to call this function, a non-initialized pointer of pcap_if_t type is enough. The function will convert this pointer in a linked list that will contain each interface and their associated data. In case of error, '-1' is returned and a description of the error will be allocated in *errbuf.*

- **int pcap_datalink(pcap_t *p)**

This routine returns the associated data link type of a network interface. The return value can take the following values:

DLT_NULL BSD loopback encapsulation
DLT_EN10MB Ethernet (10Mb, 100Mb, 1000Mb, and up)
DLT_ IEEE802 IEEE 802.5 Token Ring
DLT_ ARCNET ARCNET
DLT_SLIP SLIP
DLT_PPP PPP
DLT_FDDI FDDI
DLT_ATM_RFC1483 RFC 1483 LLCSNAP-encapsulated ATM
DLT_RAW raw IP, the packet starts with an IP header
DLT_PPP_SERIAL PPP HDLC-like framing mode (RFC 1662)
DLT_PPP_ETHER PPPoE
DLT_C_HDLC Cisco PPP con HDLC framing, defined in 4.3.1 RFC 1547
DLT_IEEE802_11 IEEE 802.11 wireless LAN
DLT_LOOP OpenBSD loopback encapsulation
DLT_LINUX_SLL Linux cooked capture encapsulation
LT_LTALK Apple LocalTalk

13.4.2.Example 1. Our First Libpcap Program

```
/*******************************************************************
*
* FILE: ldevs.c
* Author: Jorge Castellote
* Original Source Code : Martin Casado
* Original: Martin Casado
* Compiling: gcc ldevs.c -lpcap
*
* Description:
* Looks for the first available network interface and lists its network
* address and its subnet mask
*
*******************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pcap.h>       //include a libpcap

int main(int argc, char **argv)
{
       char *net;  // network address
       char *mask; // subnet mask
       char *dev;  // network device’s name
       int ret;    // returning code
       char errbuf[PCAP_ERRBUF_SIZE]; // error message buffer
       bpf_u_int32 netp;  // network address (raw mode)
       bpf_u_int32 maskp; // subset mask
       struct in_addr addr;

       /*get the first available device*/
       if ((dev = pcap_ookupdev(errbuf))== NULL)
       {printf("ERROR %s\n",errbuf);exit(-1);}
       printf("Device name : %s\n",dev); /*display device’s name*/

       /*ask for the network address and the subset mask*/
       if ((ret = pcap_lookupnet(dev,&netp,&maskp,errbuf))==-1)
       {printf("ERROR %s\n",errbuf);exit(-1);}

       /*get the network address in a human readable form*/
       addr.s_addr = netp;
       if ((net = inet_ntoa(addr))==NULL)
       {perror("inet_ntoa");exit(-1);}
       printf("Network address : %s\n",net);

       /* do the same as above for the device's mask */
       addr.s_addr = maskp;
       mask = inet_ntoa(addr);
       if ((net=inet_ntoa(addr))==NULL)
       {perror("inet_ntoa");exit(-1);}
       printf("Subnet Mask : %s\n",mask);
       return 0;
}
```

13.5.(S2) Grabbing Packets

Once we know how to get a list of the installed interfaces in our system and their configurations, we are ready to start capturing.

There are several functions that can be used to capture packets. The main differences between them are: The number of packets we want to grab, the capturing mode (normal or promiscuous), and the way their *callbacks* functions are defined (a callback function is invoked whenever a packet is grabbed).

13.5.1.Specific Functions

- **pcap_t * pcap_open_live(char *device, int snaplen, int promisc, int to_ms, char *errbuf)**

Before entering the capture loop, a pcap_t type descriptor is needed, and we will use this function for that purpose. The first parameter (*char *device*) is the name of the network device on which we would like to start capturing (ANY and NULL values forces the capture in every available device.

The second parameter (*int snaplen*), specifies the maximum number of bytes that can be captured.

The *promisc* argument indicates the opening mode; a value different of ‘0’ initiates the capture in promiscuous mode, ‘0’ for normal mode.

As it will be detailed later on section 2.5.1, programs based in libpcap are executed in the user zone of the OS, but the capture itself is made in the Kernel zone, therefore an area change is needed, but this changes are very costly and should be avoided as much as possible in order to optimize the performance, and that is why this function allows us to specify how many milliseconds we want the Kernel to keep gathering packets before passing them to the user zone.

If the function returns NULL an error has occurred and a description may be found at *errbuf.*

- **int pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback, u_char *user)**

This function is used to capture and process packets. *cnt* indicates the maximum number of packets to be processed before terminate, ‘-1’ to capture indefinitely. *callback* is a pointer to the function that should be invoked when a packet is processed. To allow a function pointer to be pcap_handler, the routine must receive three parameters:

u_char pointer: The first u_char pointer is yet to have a defined utility.

pcap_pkthdr structure

u_char pointer: Here is where the packet lies, we will see how to interpret it later on.

The function returns the number of grabbed packets or '-1' in case of error, where *pcap_perror()* and *pcap_geterr()* can show a more detailed message of the error.

- **int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)**

Very similar to *pcap_dispatch*, with the difference that does not terminates when a timeout error occurred. Returns a negative integer in case of error and '0' if the *cnt* number of packets was successfully accomplished.

- **u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)**

Reads only one packet and returns an u_char pointer with its content, without the necessity of declaring any callback function.

13.5.2.Example 2. Usage of pcap_next

```
/*******************************************************************
* FILE: pcap_next.c
* Author: Jorge Castellote Navas
*
* Compilation: gcc -lpcap -o pcap_next pcap_next.c
*
* Example of how to use the pcap_next function.
*******************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <pcap.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>

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;

	/*Look up for a device*/
	if ((dev = pcap_lookupdev(errbuf))==NULL)
	{printf(" %s\n",errbuf);exit(1);}
	printf("Opening : %s\n",dev);

	/*Open a descriptor*/
	if ((descr = pcap_open_live(dev,BUFSIZ,0,-1,errbuf)) == NULL)
	{printf("pcap_open_live(): %s\n",errbuf);exit(1);}

	/*Grab our first packet*/
	if ((packet = pcap_next(descr,&hdr))==NULL)
	{printf("Error capturing the packet\n");exit(-1);}
	printf("Grabbed packet of size %d\n",hdr.len);
	printf("Received at %s\n",ctime((const time t*)&hdr.ts.tv sec));

	return 0;
}
```

13.5.3.Example 3. Pcap_loop usage

```
/**********************************************************************
* FILE: pcap_loop.c
* Author: Jorge Castellote
*
* Compilation: gcc -o pcap_loop pcap_loop.c -lpcap
*
* Example of how to use the pcap_loop function, and to define a
* Callback function. Shows a counter displaying the number of packets
* grabbed. 15 packets are grabbed
*
**********************************************************************/
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>

/*callback function. Invoked each time a packet is grabbed*/
void my callback(u char *useless, const struct pcap_pkthdr* pkthdr, const
u_char* packet)
{
       static int count = 1;
       fprintf(stdout," %d, ",count);
       fflush(stdout);
       count++;
}

int main(void)
{
       int pack_num = 15;
       char *dev;
       char errbuf[PCAP_ERRBUF_SIZE];
       pcap_t* descr;
       const u_char *packet;
       struct pcap_pkthdr hdr;
       struct ether_header *eptr; /* Ethernet header*/
       bpf_u_int32 maskp; // subnet mask
       bpf_u_int32 netp;  // network address

       /*Look up for a device*/
       dev = pcap_lookupdev(errbuf);
       if (dev == NULL)
       {fprintf(stderr," %s\n",errbuf); exit(1);}
       else
       {printf("Opening %s in promiscuous mode\n",dev);}

       /* Get the network address and subnet mask*/
       pcap_lookupnet(dev,&netp,&maskp,errbuf);

       /* Start capturing in promiscuous mode */
       descr = pcap_open_live(dev,BUFSIZ,1,-1,errbuf);
       if (descr == NULL)
       {printf("pcap_open_live(): %s\n",errbuf); exit(1); }

       /* Entering the loop (15 times) */
       pcap_loop(descr,pack_num,my callback,NULL);

       return 0;
}
```

13.6.(S1) How to filter only what we want

13.6.1.What is a packet/socket filter?

We have already discussed that Libpcap is a functions library and that the processes which executes these functions do it in the user space. Nevertheless, the real capturing zone takes place in the Kernel area, therefore a mechanism able to trespass this border must exists. Besides, do it in a safe and efficient way, because every single failure in such a deep layer of the system would decrease the whole system's performance.

Let us imagine an application able to sniff the network looking for packets with the 135 TCP source port (sign of some workstation is infected with the Blaster virus). If there was not any filtering system, the Kernel would not know which packets our application is interested on, so it should trespass the Kernel-User Space border for each single packet that travels through the network.

In order to avoid this situation, the solution comes by establishing a filter that only allows packets which their TCP source port is 135 trespasses the border. This is the main duty of Packet/Socket Filter.

There is not a unique filtering system. Actually, almost every OS rewritestheir own solution: NIT for SunOS, BPF for BSD (originally) and more recently LSF (Linux Socket Filter), for Linux.

For this section we will only focus in BPF, for being the most spread out and the reference architecture for LSF.

The working procedure of BPF is based on two great components: The *Network Tap* and the *Packet Filter*. The first one is responsible of grabbing packets from the network device's driver and bring them to those applications that ask for them. The second one is in charge of deciding if the packet should be accepted (Ethernet addresses matching) and, in an affirmative case, how many bytes of the packet should be delivered to the application (it would not make sense to deliver a packet with the Ethernet headers included).

In the following image we can see a general schema of the BPF working procedure inside the system. Whenever a packet arrives to a network interface, the most common option for the driver is to send the packet to the protocol stack, as long as BPF is not active, in which case the packet will be processed by it before being sent to the stack.

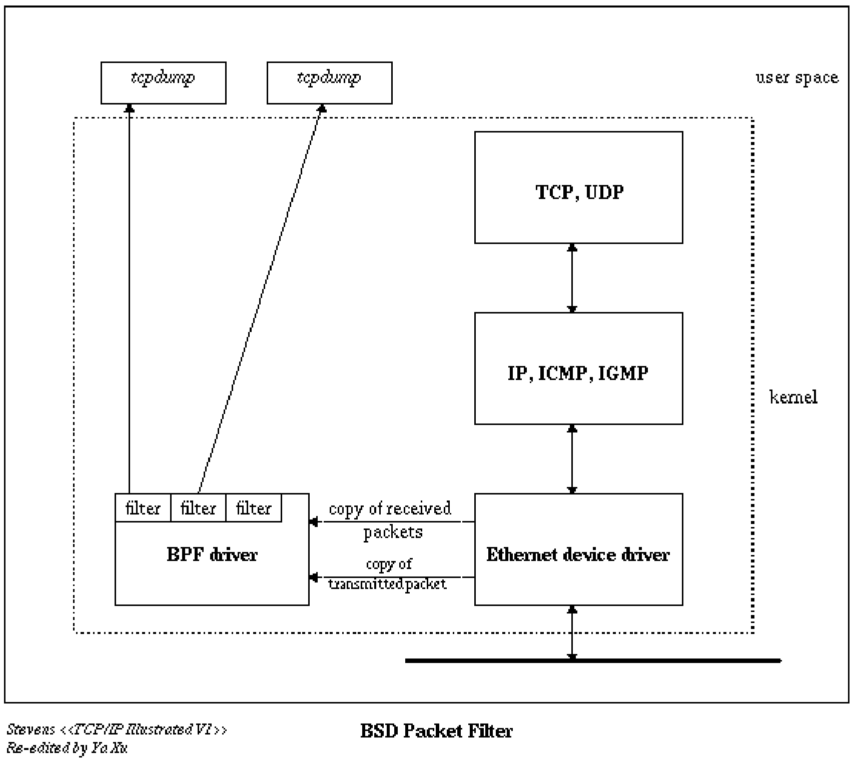


Figure 21: BPF Working Procedure

BPF will be in charge of comparing each packet with the established filters, copying each packet in the designated buffer of the applications which have established a filter that fits with the packet contents. In the case that no coincidence is found, the packet is returned to the driver, which will act normally.

13.6.2.Filtering Primitives

BPF has its own language for filtering programming, a low level language. However, libpcap implements a much easier and friendly language to define the filters. Obviously, this language must be compiled to be understood by BPF before being applied. This language has become very popular to define capturing filters thanks to TCPDUMP.

A very small introduction to this language is needed to accomplish the assignment, which is detailed in the following paragraphs.

The expression or sentence that is used to define the filter has got several primitives and three possible modifiers to these primitives. This expression can be *true*, in which case the packet is passed to the user zone (to our application), or *false*, the case where the packet is ruled out without leaving the Kernel zone, as seen in the previous section.

The three possible modifiers are:

- **Type**

  Can be *host*, *net* or *port*; indicating respectively: machine/workstation (e.g. host. 192.168.1.1), a whole network (e.g. net 192.168) or a concrete port (e.g. port 25). *Host* is assumed as the default type.

- **Dir**

  Specifies from where we are going to listen the data flow. We have *src* or *dst* and we can combine them with *or* and *and*. When it comes to the peer to peer protocols, we can substitute them for *inbound* or *outbound*. E.g. If we want to capture the data flow that has got 10.10.10.2 as the destination address and 192.168.1.2 as the source address, the filter will be "**dst 10.10.10.2 and src 192.168.1.2**". If we want to capture all the packets with 192.168.1.1 as the destination address (it does not matter where is it coming from) or 192.168.1.2 as the source address (without caring about the destination), then the expression will be "**dst 192.168.1.1 or src 192.168.1.2**". The addresses can keep being combined by using brackets as well as *or* and *and*. Of course this can be combined with the type modifiers.

- **Proto**

  In this case is about the protocol we want to capture. It can be *tcp*, *udp*, *ip*, *ether*, *arp*, *rarp* and *fddi*.

These expressions can always be combined with brackets and logical operators, such as negation (!not), chain (&&and), alternative (||or). Some of the primitives that can be used are described as follows. What appears between '[]' is optional, and | (pipe) means XOR.

[dst | src] host *machine*

True if the destination or source address matches with ***machine***, which can be an IP address, or a name that can be solved by the DNS.

Example:

Capture traffic originated by the IP 192.168.1.1:

```
src host 192.168.1.1
```

ether src | dst | host *edir*

This filter is true if the origin address (src), destination (dst), or both (host) match with edir. It is mandatory to specify one of the three modifiers.

Example:

Capture traffic with source or destination 00:15:C6:EE:EC:10

```
ether host 00:15:c6:ee:ec:10
```

ip proto protocol

For this case we will be listening to the indicated protocol. This protocol can be icmp, icmp6, igmp, igrp, pim, ah, esp, udp, or tcp.

Fortunately, we can use the aliases tcp, udp, and icmp to designate ip proto tcp, ip proto udp, etc…

Example:

Capture all the UDP packets:

```
udp
```

13.6.3.Specific Functions

int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize, bpf_u_int32 netmask)

This function is used to compile a filter program in the TCPDUMP format (*char* str*) into its BPF equivalent expression (*bpf_u_int32 netmask*). It is possible that the original program is modified in order to be optimized if required in the *int optimize* parameter. The *netmask* parameter is the local network mask, which can be obtained by calling pcap_lookupnet.

In case an error has occurred, the function returns '-1'. Use pcap_geterr for a greater detail description.

int pcap_setfilter(pcap_t *p, struct bpf_program *fp)

Once the filter has been compilated, only applying it is remaining. Pass the compiled filter (fp) to this function.

In case an error has occurred, the function returns '-1'. Use pcap_geterr for a greater detail description.

13.6.4.Example 4. Applying filters

```
/***********************************************************************
* FILE: pcap_filters.c
* Author: Jorge Castellote
*
* Compilation: gcc -o pcap_filters pcap_filters.c -lpcap
*
* Example of how to filter traffic with Libpcap, the program gets a
* filter as a parameter, compiles it, applies it and loops for fifteen
* times grabbing all the packets in promiscuous mode
*
***********************************************************************/
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>

/*callback function. Invoked each time a packet is grabbed*/
void my_callback(u_char *useless, const struct pcap_pkthdr* pkthdr, const
u_char* packet)
{
       static int count = 1;
       fprintf(stdout," %d, ",count);
       fflush(stdout);
       count++;
}
int main(int argc, char **argv)
{
       int pack_num = 15;
       char *dev;
       char errbuf[PCAP_ERRBUF_SIZE];
       pcap_t* descr;
       const u_char *packet;
       struct pcap_pkthdr hdr;
       struct ether_header *eptr; /* Ethernet header */
       struct bpf_program fp;     /* Compiled filter container */
       bpf_u_int32 maskp;         /* Network mask */
       bpf_u_int32 netp;          /* Network address */

       if (argc != 2)
       {fprintf(stdout,"Usage %s \"filtering expression\"\n",argv[0]);
       return 0;}
       dev = pcap_lookupdev(errbuf);
       if (dev == NULL)
       {fprintf(stderr," %s\n",errbuf); exit(1);}
       else
       {printf("Opening %s in promiscuous mode\n",dev);}

       /*Extract the network address and the subnet mask*/
       pcap_lookupnet(dev,&netp,&maskp,errbuf);
       /*Start capturing in promiscuous mode*/
       descr = pcap_open_live(dev,BUFSIZ,1,-1,errbuf);
       if (descr == NULL)
       {printf("pcap_open_live(): %s\n",errbuf); exit(1); }

       /*Compile the filter*/
       if (pcap_compile(descr,&fp,argv[1],0,netp) == -1)
       {fprintf(stderr,"Error compiling the filter\n"); exit(1);}
       if (pcap_setfilter(descr,&fp) == -1) /*apply the filter*/
       {fprintf(stderr,"Error applying the filter\n"); exit(1);}

       pcap_loop(descr, pack_num, my_callback, NULL); return 0;}
```

13.7.(S3) Interpreting the data

Remark: This section details the Ethernet LANs only.

As we where reviewing in the introduction section, when an application wants to send data through the network, it must add the protocol headers that will be used during the transmission.

In the previous section we reached an *u_char* * with the raw data, but in order to get understandable information, we need to make the duty that the protocol stack would have make for us, it is to say, extracting and interpreting the headers.

Even though the most important data structures we will find in the headers are detailed, it is recommended to take a look to the most important RFCs:

RFC 793 (TCPv4)

RFC 791 (IP)

RFC 768 (UDP)

RFC 826 (ARP)

RFC 792 (ICMPv4)

The first header we will found when analyzing a packet is the Ethernet header, defined in the ethernet.h file at /usr/includes/net/, where we find the following:

```
/* This is a name for 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__));

/* Ethernet protocol ID’s */
#define ETHERTYPE_PUP 0x0200 /* Xerox PUP */
#define ETHERTYPE_IP 0x0800 /* IP */
#define ETHERTYPE_ARP 0x0806 /* Address resolution */
#define ETHERTYPE_REVARP 0x8035 /* Reverse ARP */
#define ETHER_ADDR_LEN ETH_ALEN /* size of ethernet addr */
#define ETHER_TYPE_LEN 2 /* bytes in type field */
#define ETHER_CRC_LEN 4 /* bytes in CRC field */
#define ETHER_HDR_LEN ETH_HLEN /* total octets in header */
#define ETHER_MIN_LEN (ETH_ZLEN + ETHER_CRC_LEN) /* min packet length */
#define ETHER_MAX_LEN (ETH_FRAME_LEN + ETHER_CRC_LEN) /* max packet length */
```

From this code we observe that the type in charge of storing a Ethernet header is called ether_header, which has three main fields: Source address (ether_shost), destination address (ether_dhost), and the packet type that carries, which can be:

ETHERTYPE_PUP Xerox PUP

ETHERTYPE_IP IP type packet

ETHERTYPE_ARP ARP type packet

ETHERTYPE_RARP RARP type packet

In the netinet/ether.h file we can find some useful functions to deal with Ethernet addresses.

```
/*Functions to convert from an Ethernet 48 bytes address to readable text */
extern char *ether_ntoa ( const struct ether_addr * addr) THROW;
extern char *ether_ntoa_r ( const struct ether_addr * addr, char * buf)
THROW;
/*Functions to convert from text format to an Ethernet 48 bytes address */
extern struct ether_addr *ether_aton ( const char * asc) THROW;
extern struct ether_addr *ether_aton_r ( const char * asc, struct ether_addr *
addr) THROW;
/* Mapping from a hostname to an Ethernet 48 bytes address */
extern int ether_hostton ( const char * hostname, struct ether_addr * addr)
```

Once we know how the Ethernet header is organized, we can easily extract it from our *u_char * packet*, as we know that it is placed in the first 14 bytes. Actually, the first 14 bytes always belong to an Ethernet header. Nevertheless, the following header's length will depend on the value of *ether_type*. In this case we will suppose that the packet is IP type (ether_type == ETHERTYPE IP), which would mean that after the first 14 bytes we will find an IP header. The structure that will store the header has also been defined in the ip.h file (/usr/include/netinet/):

```
/*
* Structure of an internet header, naked of options.
*/
struct ip
{
#if BYTE_ORDER == LITTLE_ENDIAN
      unsigned int ip_hl:4; /* header length */
      unsigned int ip_v:4; /* version */
#endif
#if BYTE_ORDER == BIG_ENDIAN
      unsigned int ip_v:4; /* version */
      unsigned int ip_hl:4; /* header length */
#endif
      u_int8_t ip_tos; /* type of service */
      u_short ip_len; /* total length */
      u_short ip_id; /* identification */
      u_short ip_off; /* fragment offset field */
#define IP_RF 0x8000 /* reserved fragment flag */
#define IP_DF 0x4000 /* dont fragment flag */
#define IP_MF 0x2000 /* more fragments flag */ 20
#define IP_OFFMASK 0x1fff /* mask for fragmenting bits */
      u_int8 t_ip_ttl; /* time to live */
      u_int8 t_ip_p; /* protocol */
      u_short ip_sum; /* checksum */
      struct in_addr ip_src, ip_dst; /* source and dest address */
};
```

Even though this section's goal is not analyzing the content of the headers, it would be useful to take a look at the *protocol* field as it will help us to identify the third and last header. There are numerous protocols, but the most important are ICMP (protocol == 1), TCP (protocol == 6), and UDP (protocol == 17).

Whenever we try to extract the payload of a protocol, these are the steps we may follow:

15. Extract the Ethernet header (first 14 bytes).
16. Look up for the *ether_type*, in order to know if it is IP.
17. If it is IP, look up for the protocol field.
18. Go to /usr/include/netinet/ and check how our final protocol's header is.
19. After knowing the size of this header, the payload starts in the following byte.

14.APPENDIX E: BASIC USAGE OF DATAGRAM SOCKETS

14.1.INTRODUCTION TO PROCESSES COMMUNICATION MECHANISMS

Processes communication is essential among the network programmers. There are different mechanisms for these communications:

14.1.1.Low level mechanisms

Solve the data communications between processes. The programmer takes care of the remaining tasks, which are:

- Data representation.
- Communication protocols definition.

These mechanisms can be defined in two different fields:

- Local field: Files, pipes, queues (FIFO), shared memory (between threads, processes).
- Local/Remote field: Sockets, message handling distributed systems (Bea Tuxedo, etc…).

14.1.2.High level mechanisms

The programmer finds the following tasks solved:

- Data representation.
- Protocol

A good example would be RPC (Remote Procedure Call).

14.1.3.Higher level mechanisms

An object oriented approach is added (distributed objects).

- Remote method call.
- Other services, such as object localizing or remote object creation.

Some good examples are RMI, CORBA and SOAP.

14.2.SOCKETS

The sockets appeared for the first time in UNIX BSD 4.2, when adapting the TCP/IP architecture in the UNIX OS. The design of the sockets tries to generalize by being independent from the protocol. The following information about sockets is focused on the datagram sockets.

14.2.1.Description

Sockets are a communication endpoint. They offer an access interface to the lower level's services. A socket can be defined as a data exchange structure that is also able to configure the associated parameters of the lower level. Is the main tool to use the access points between communication levels such as the OSI levels.

When defining what a socket in the UNIX environment is, a socket is a file descriptor (an integer associated to an open file). When programs make any I/O operation in UNIX, they do it through a file descriptor, and sockets are not an exception.

14.2.2.Creation

To create a socket means to request a communication point creation to the OS (buffers, etc…), that allows send/receive/configure through any lower level service.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int domain, int type, int protocol);
```

This function returns a socket descriptor.

14.2.3.Domains

A domain represents a protocol family. The goal is to offer a unique interface for any transport level or lower level. This implies offering a functions group, a naming mechanism to allow every format of the different level's services, and a configuration procedure to specify the lower level's service.

A socket is associated to a domain when created, and least two sockets with the same domain are needed to establish a communication (one in each ending point).

14.2.4.Types

Depending on how the communication is oriented, there are different types of sockets:

- Stream sockets (`SOCK_STREAM`): Connection oriented; Full-duplex; reliable; guarantees the correct order of transmitted data.

- Datagram sockets (`SOCK_DGRAM`): Connectionless oriented; no reliable; limited size of the message; does not guarantee the correct order of the transmitted data; each message is independent of the others.
- Low level sockets (`SOCK_RAW`): Allows access to levels that are below the transport level.

14.2.5.Protocol election

When creating the socket, three parameters must be specified, which are the domain, the type, and the protocol. When the given domain has more than one protocol of the specified type, you will need to choose among the family protocol.

For example, when choosing the AF_INET domain, SOCK_DGRAM type, there is no need of choosing a specific protocol, as only UDP is a datagram protocol in the Internet protocol family.

14.2.6.Naming

Each socket has a unique name. This name will be the file name when belonging to the PF_LOCAL family, and an IP address (32 bits) plus a transport port (16 bits), when the socket belongs to the PF_INET family.

There is a generic structure where this information will be stored, which is the struct sockaddr. When creating a socket (domain + type + protocol), a specific naming is used in the PF_INET family: ‘struct sockaddr_in’. Casting is needed when using the structures.

14.2.6.1.Naming in the PF_INET family

```
<netinet/in.h>

typedef uint32_t in_addr_t;
struct in_addr {
        in_addr_t    s_addr;
};

struct sockaddr_in
{
   short int sin_family;        /* Addresses Family */
   unsigned short int sin_port; /* Port */
   struct in_addr sin_addr;     /* Internet Address */
    unsigned char sin_zero[8];     /* Byte stuffing to
preserve the struct sockaddr original size */
};
```

A ‘0’ value in the port means that the OS will choose a free one automatically.

14.2.7.Name appointing

Once the socket has been created, a name must be given in order to allow processes to reference it and to use it. Remember that functions have generic naming (sockaddr), but the specified protocol naming must be used (sockaddr_in in this case); therefore 'casting' must be used.

```
#include <sys/types.h>
#include <sys/socket.h>
```

int bind (int socket, const struct sockaddr *addr, socklen_t addr_len);

14.2.8.Obtaining name

We can also get the socket name:

```
#include <sys/types.h>
#include <sys/socket.h>

int   getsockname   (int   s,   struct   sockaddr   *addr,
socklen_t *addr_len);
```

14.2.9.Name transformation

Allows transformations from the 'X.X.X.X' format to the socket's interface format (32 bits number).

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char * inet_ntoa(struct in_addr in);
int inet_aton(const char *cp, struct in_addr *pin);
```

14.2.10.Obtaining address

The Domain Name Service (DNS), allows translating from 'machine.my.domain' to an IP address, it is to say, the name of the server or CPU to a common IP address. The obtained info will be stored in 'hostent' structure.

```
#include <netdb.h>

struct hostent * gethostbyname (const char *name);

struct hostent * gethostbyaddr (const char *addr, int
len, int type);

struct  hostent {
         char    * h_name;  /* official name of host */
         char    ** h_aliases; /* A NULL-terminated
array of alternate names for the host */
         short   h_addrtype;   /* host address type,
usually AF_INET */
         short   h_length;  /* length of address */
         char    ** h_addr_list;   /* A zero-terminated
array of network addresses for the host. Host addresses
are in Network Byte Order */
};
#define h_addr  h_addr_list[0]  /* First address in the
addresses list */
```

14.2.11.Byte arrangement

Every integer of the socket's interface is in network format. This format follows the big-endian criteria. The format must be inverted for local use (to be used when programming). There are some functions that may help.

```
#include <sys/param.h>

u_long htonl (u_long hostlong);
u_short htons (u_short hostshort);
u_long ntohl (u_long netlong);
u_short ntohs (u_long netshort);
```

14.2.12.Datagram sockets: Sending

In order to send a message, the following function can be used:

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t sendto (int socket, const void *msg, size_t
len, int flags, const struct sockaddr *to, socklen_t
tolen);
```

- Used protocol: UDP.
- Address: “struct sockaddr_in”.
- Flags: always ‘0’.
- Asynchronous sending: The message is copied to a buffer, if the buffer is full, the socket will block (default option).

14.2.13.Datagram sockets: Reception

In order to receive a message:

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recvfrom (int socket, void *buf, size_t len,
int flags, const struct sockaddr *from, socklen_t
*fromlen);
```

- If the ‘from’ parameter is not NULL, is filled with the origin name. It has to be ‘struct sockaddr_in’ type.
- ‘fromlen’ is an in/out parameter.
- Synchronous reception (default).
- Extracts from the OS buffer to the given buffer.
- If the message does not fit in the given buffer, the remaining message will be ruled out.
- If there are no messages, the function blocks.

15.SUMMARY AND CONCLUSIONS

15.1.Network Programming Needs Specific Programmers

Network programming is a huge field. There are still a lot of parts of the network programming that are important, such as security, WML, etc... that are also part of the network field. And these kind of programming is developing at a very high speed. Generic networks are becoming the standard way of linking devices, substituting the specific networks as the telephone network or even the mobile phone network. And only specific programmers will be able to develop these networks efficiency at its maximum.

15.2.Learn How to Learn

Building this small network applications without the help of open source code and documentation would have been impossible. Even when dealing with a concrete part of the network programming as this thesis does, the concepts are very specific, there are a lot of small ideas and notions that have to be learned in order to create a specific network application. And of course not all of them can be taught in a degree. Therefore the aim of a accomplishing a laboratory practice must also be learning how to learn about specific divisions of the network programming by ourselves.

15.3.Network Programming Requires Basic Programming

The problem that I found when developing interesting practices, or at least practices that ends in whole application, is that a small knowledge on c-programming must be achieved before starting these practices. Basic network programming requires previous concepts in normal programming.

16.REFERENCES

[1] **Jacqueline A. Jones & Keith Harrow.** *Problem Solving with C.* Scott/Jones. 1996.

[2] **S.J. Mullender.** *Distributed Systems*. Addison-Wesley. 1993

[3] Tim Carstens. *Programming with Pcap.* http://www.tcpdump.org/pcap.htm

[4] **Steven McCanne and Van Jacobson**. *The BSD Packet Filter: A New Architecture for User-level Packet Capture.* http://www.tcpdump.org/papers/bpf-usenix93.pdf

[5] **Carles Pina i Estany**. *Ncurses para remolones*. http://bulma.net/body.phtml?nIdNoticia=2004

[6] **W.R. Stevens**. *Advanced programming in the Unix environment.* Addison-Wesley. 1992

[7] **G.A. Andrews.** *Concurrent programming, principles and practice.* The Benjamin/Cumming Publishing Company. 1991

[8] **G.F.Coulouris, J.Dollimore,T.Kindberg.** *Sistemas distribuidos, conceptos y diseño.* Addison-Wesley. 2001

[9] IETF. *IETF RFC page.* http://www.ietf.org/rfc.html