PART A: Learn to use datagram sockets by example.

Read and understand programs **Recv_udp.c** and **Send_udp.c**. Compile each program (**C9**) and execute them both in your local host using the following instructions:

1. Run **Recv_udp** (note that this program contains an infinite loop, so you will want to add '**&**' to the execution command so the shell will return to the prompt, or just open another terminal window).

2. Run **Send_udp** several times, passing **localhost** as command line parameter, and observe what happens.

3. Go back to the source code and do your best to comment **each line** (in the files **Recv_udp.c** and **Send_udp.c**), specifically state where a **system call** is issued and explain what the line accomplishes. Your comments should be short and straight to the point.

4. Write the code for a new function called **printsin()** according to the specifications below. uncomment the calls to **printsin()** in **Recv_udp.c** and build the executable.

   The function prototype must be:
   void printsin(struct sockaddr_in *sin, char *pname, char* msg)

   The function will print the string indicated by *pname* ended with a line break, then it will print the string indicated by *msg*, then the string "ip= " follow by the IP address of the host associated with this socket address structure in dotted-decimal notation, a comma, and finally the string "port= " followed by the port number associated with this socket address structure. When this function is called, it should produce output as in the examples below:

   function call: **printsin(&s_in, "UDP-SERVER:", "Local socket is:");**
   output:
   UDP-SERVER:
   Local socket is: ip= 0.0.0.0, port= 9000

   function call: **printsin((struct sockaddr_in*)&from, "UDP-SERVER: ", "Packet from:");**
   UDP-SERVER:
   Packet from: ip= 127.0.0.1, port= 33080

5. Change the server and client code as following: The client will send your name to the server and then will receive the server response. On the server side, the server

will receive the client name (instead of the current msg structure) and will send back its name. (make sure to given a distinct name to both)

6. Copy the server and client code to new files named **Server.c , Client.c, Router.c** and change them as following:
   Motivation: Node A and Node C communicate via node B (the router).

   Requirements:
   When Node B receives a message from Node A:
   a. B chooses a random number uniformly from range [0,1].
   b. If the number is greater than X it B will forward the message, otherwise, it will drop the message (delete it).

   On the vice versa, B always forwards any message from C.

   -where X is an argument received from the command prompt.

   Important Note:

   Make sure Node A and Node C can message each other from the command prompt freely, and when one side wants to disconnect he can do it by typing **exit**

Part B: IP addresses, hostnames and .... HTTP

I. Server-Client

2 Review the files **net_server.c** and **net_client.c** and compile them. Be sure you understand them.

3 In one terminal window, execute net_server. In another window, execute net_client. Briefly explain what you see.

4 Type the command **nslookup <hostname>**, where hostname is the name of the computer you are working on, to learn the IP address of that computer (you can use another way to check it as well).

5 In net_client.c, change the definition of IP_ADDRESS to the address of the computer you are working on. Recompile net_client.c.

6 Now, run net_server and net_client again. Explain what you see.

7 Suppose you run net_client when net_server is not running? Try this and explain what you see (use TCPDUMP or WIRESHARK to check your assumption ,provide screen shot).

8 Wouldn't it be great if we could run net_server and net_client on any computer, without having to change the IP address and recompile the client? The way we will do this is by using the getaddrinfo system call. This function lets us supply a hostname as a string, and it will resolve that hostname to a result of type struct sockaddr. The program nslookup.c illustrates the use of getaddrinfo. Use this program, compile it, run it a few times supplying different hostnames as arguments, and review the code to understand how it works (add relevant comments to the code).

9 Now, modify net_client.c so that it takes the hostname as a command-line argument. Use code from nslookup.c to resolve this hostname to a result of type struct sockaddr, and then use the result you obtain to connect to this address rather than the hard-coded IP_ADDRESS.

II. WGET

The wget program allows you to fetch the contents of a URL and save it to a new file. If you've never used wget, try using it to fetch the file named by http://www.yahoo.com. In this part of the lab, we will build a simple analog to wget. Our program will take a URL as a command-line argument and write the response from the web server to STDOUT.

1. One problem we will face in building our simple web client is that of parsing URLs. Luckily, we can build a very simple web client while only parsing a limited class of URLs: those of the form <protocol>://<hostname>/<path> or <protocol>://<hostname>:<port>/<path>. The method illustrates a simple parser for URLs of this form.
2. To put together your simple web client, make an appropriately named copy of your program and then make the following changes.
    - The command-line argument should be a URL rather than a hostname.
    - Parse the URL and connect to the hostname and port specified.
    - The HTTP protocol, in its simplest form, is very, very simple. After connecting the socket, write a request of the following form (where ↲ indicates a newline character):

      GET *url* HTTP/1.0↲

      HOST: *hostname*↲

      ↲

      ** when *url* is indicated, it's correlated to *path* from the URL re-structuring

3. At this point, data from the web server should start arriving. Repeatedly read chunks of data from the socket into a character array and then write the data to STDOUT. You will know all of the data has been read when the return value from the read(...) syscall is zero (indicating 0 bytes read). When all of the data has been read, close the socket. (Note that the data read from the socket will *not* end with a null (\0) character. You'll need to account for this somehow when you are writing the contents of the buffer to STDOUT.)

4. Examine the result of running your program from previous step for the URL http://www.yahoo.com The *HTTP header* is separated from the contents of the file by a blank line. What information do you see in the header?
5. Examine the result of running your program from step previous for the URL http://www.yahoo.com/does-not-exist. How is the HTTP header different from what you saw?

6. List at least different 5 response code you have gotten from 5 different URLs, and their meaning.

Submission instructions:

1. Make sure that all of your source files compile using the following command:
   gcc -Werror -Wall -Wvla -g inputFile.c -o outFile.out
   Source files that will not cimpule using this line will not be graded at all.
2. Make sure you submit the following file in a ZIP file named in your ID:
   a. All of your **final source files** (with comments as required above).
   b. pcap file with detail explanation about the messages in the file regarding your code
   c. A text/doc file with your answers to the open questions and a doc file explaining what your program doing including notes stating whether your programs work to specifications or not. If you know that they don't behave correctly, make sure to explain what is right and what is wrong.