תרגיל מסכם - הגשה ביחידים
תאריך הגשה: 12.7.2022 בחצות לתיבת ההגשה
התרגיל ניתן מראש כדי שתוכלו לתכנן את זמנכם בהתאם
לא יאושרו איחורים
שימו לב שלתרגיל זה יש מספר חלקים
בנוסף אל תשכחו כיצד מגישים קוד

UDP programming

Warning

1. Make sure to have no more than one process **`recv_udp`** running at the same time. The same recommendation applies to any process that you put in the background for this lab. Failure to observe this warning will give you undesirable results.

2. Make sure to kill all your processes at the end of a work session. The last thing you want is to leave a lot of stuff running in the background for the next user to take your machine. Remember that to list all your processes, you can use:

   ```
   ps -ef | grep your-username
   ```

3. The **`ps`** command gives a list of processes with their associated process identifications (or PID; see the second column in the results). To terminate a process, you can issue the following command:

   ```
   kill -9 pid
   ```

Problem 1: Learn to use datagram sockets by example.

Read and understand programs **`send_udp.c`** and **`recv_udp.c`**. Compile the programs and execute them both in your local host.
First, run **`recv_udp`** (note that this program contains an infinite loop, so you will want to put in the background with **`&`** so you can have your shell prompt back, or just open another terminal window).
Then, run **`send_udp`** several times, passing **`localhost`** as command line parameter, and observe what happens. (If you want to experiment a bit more, you can run each program on a different machine in the lab - of course in that case when you run **`send_udp`** you should instead pass in the command line the symbolic name of the host where you started **`recv_udp`**.)

Now, go back to the source code and do you best to comment each line where a system call is issued explaining what the line accomplishes. Your comments should be short and straight to the point. (You may have to browse man pages to find the information you need.)

Finally, 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, "RECV_UDP:", "Local socket is:");`**
output:

```
RECV_UDP:
Local socket is: ip= 0.0.0.0, port= 9000
```

function call: **`printsin((struct sockaddr_in*)&from, "RECV_UDP: ", "Packet from:");`**

```
RECV_UDP:
Packet from: ip= 127.0.0.1, port= 33080
```

Problem 2: Create a gateway process that simulates datagram loss.

Taking inspiration from the programs you were given in Problem 1, write three little programs according to the specifications below:

- **`source.c`** : Takes the name of a host on the command line, creates a datagram socket to that host (each student will be assigned their own port number P to use), then enters an infinite loop in each iteration of which it sends a datagram onto the socket carrying in its body an integer number, increments the integer, then sleeps for one second only to repeat the cycle upon waking up. Note that this program is nearly identical to **`send_udp.c`**.

- **`gateway.c`** : Takes the name of a host on the command line and creates a datagram socket to that host (using port number P+1), it also creates another datagram socket where it can receive datagrams from any host on port number P; next, it enters an infinite loop in each iteration of which it receives a datagram from port P, then samples a random number using **`((float)random())/((float)RAND_MAX)`** - if the number obtained is greater than 0.5, the datagram received is forwarded onto the outgoing socket to port P+1, otherwise the datagram is discarded and the process goes back to waiting for another incoming datagram. Note that this gateway will simulate an unreliable network that loses datagrams with 50% probability. Note: in order to be able to witness always the same behavior in the gateway, you should seed the random number generator before you make any calls to **`random()`**, that is, call **`srandom(seed)`** first for some hardcoded value of seed.

- **`sink.c`** : Is almost identical to **`recv_udp.c`**, that means, it creates a socket to receive datagrams from any host on port P+1, then enters an infinite loop where it receives a datagram and prints to the screen the information of where the datagram came from (you can print the IP address in dotted-decimal notation) and what message it contains.

Hand in:

- **Problem 1:** One printed copy of **`send_udp.c`** and **`recv_udp.c`**.
  And the soft copy and compiled binary.
- **Problem 2:** One printed copy of **`source.c`**, **`gateway.c`**, and **`sink.c`**. Again, And the soft copy and compiled binary.
- For each of the problems above, write a note stating whether your programs work to specifications or not. If you know that they don't behave correctly, make sure to explain what they do right and what they don't do right.

TCP programming

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

- Use the files net_server.c and net_client.c and compile them.
  Review the programs to be sure you understand them.
- Compile both programs. In one terminal window, run `net_server`. In another, run `net_client`. Briefly explain what you see (by writing relevant comments in the code).
- 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).
- In `net_client.c`, change the definition of IP_ADDRESS so that it is the address of the computer you are working on. Recompile `net_client.c`.
- Now, run `net_server` and `net_client` again. Explain what you see.
- Suppose you run `net_client` when `net_server` is not running? Try this and explain what you see (use TCPDUMP to check your assumption).
- 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 in the code).
- 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`.

Part B: A simple web client

9. 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`.
10. 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.
11. To put together your simple web client, make an appropriately named copy of your program from step 8 and then make the following changes.

a. The command-line argument should be a URL rather than a hostname.
b. Parse the URL and connect to the hostname and port specified.
c. 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

4. 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.)

- Examine the result of running your program from step 11 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?
- Examine the result of running your program from step 11 for the URL `http://www.yahoo.com/does-not-exist`. How is the HTTP header different from what you saw in step 12?