CSCI 4273 Network Systems

Programming Assignment 1

UDP Socket Programming

Due: Thursday, Feb 11th by 11:00 pm

Note: This assignment can be done in pairs, but is recommended individually

The purpose of this assignment:

1. An appropriate understanding of C language.
2. An understanding of tools (compilers in Linux, shell, etc.).
3. Understanding the socket programming interface in C program.
4. Transferring file between a client and a server using socket.

In this assignment, you will build two programs in C, one for the client which will simply send a command and the other for a server which will send a suitable response back to the client.

Backgrounds:

1. UDP (Use Datagram Protocol) is a connectionless transmission model with a minimum of protocol mechanism. It has no handshaking dialogues and no mechanism of acknowledgements (ACKs). Therefore, every packet delivered by using UDP protocol is not guaranteed to be received, but UDP avoids unnecessary delay, thanks to its simplicity.

Program Requirements

The client:

1. The client must take two command line arguments: an IP address of the machine on which the server application is running, and the port the server application is using. (The IP address can be obtained using host command. Type "man host" at the shell prompt for more information on how to use the hostname command. See graphic below for sample).

```
erro0301@elra-01:~$ host elra-02.cs.colorado.edu
elra-02.cs.colorado.edu has address 198.59.7.12
```

Here is an example with the client taking two command line parameters:

$ gcc uftp_client.c -o client # Compile your c-program

$ ./client 192.168.1.101 5001 # Running your client with given server IP and port number

2. It should prompt the user to type any of the following commands:
    - get [file_name]
    - put [file_name]
    - delete [file_name]
    - ls
    - exit
3. It must then send the command to the server.
4. Then it must wait for the server's response. Once the server responds, it should print appropriate messages, if any, on the standard output.

The server:

1. The server must take one command line argument: a port number for the server to use. You should select port #'s > 5000. For example:

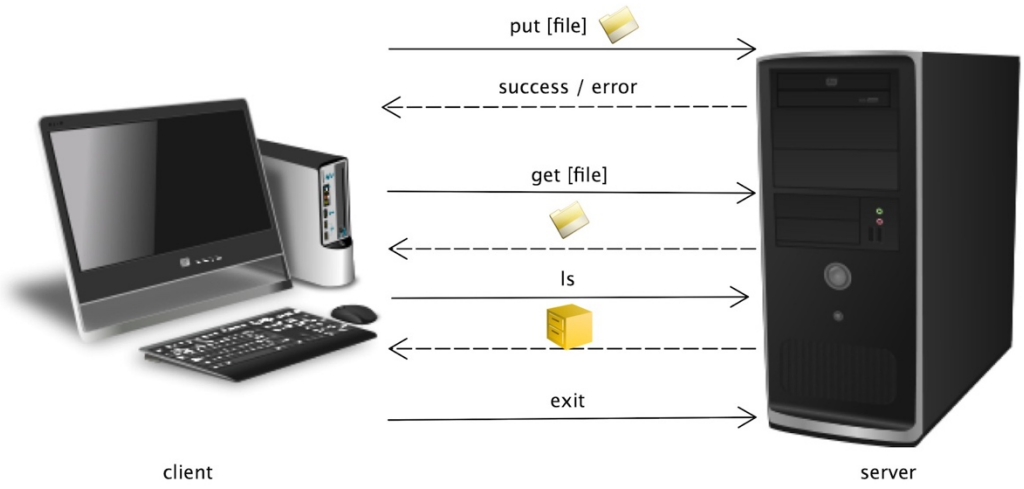
    $ gcc uftp_server.c -o server # Compile your c-program

    $ ./server 5001 # Running your server with port number 5001

2. It should wait for a UDP connection after binding to port that given as above (it is 5001 in the example above).
3. Depending on the commands received, the server responds to the client's request in the following manner:
    - “get [file_name]”: The server transmits the requested file to the client (use files of small size in order of 2 to 5 KB for transfer like any jpeg file).
    - “put [file_name]”: The server receives the transmitted file by the client and stores it locally (use files of small size in order of 2 to 5 KB for transfer like any jpeg file).
    - “delete [file_name]”: The server delete file if it exists. Otherwise do nothing.
    - “ls”: The server should search all the files it has in its local directory and send a list of all these files to the client.
    - “exit”: The server should exit gracefully.
    - For any other commands, the server should simply repeat the command back to the client with no modification, indicating that the given command was not understood.

Notes:

- Both the client and the server should be in a loop. For the purpose of this assignment, the client can be terminated by typing Ctrl-C.
- The client side/directory should contain at least three files with their names hard-coded as foo1, foo2, and foo3.



What To Turn In:

You should test your programs thoroughly so that they do not crash or hang. Your code must be organized and clear with comments/explanations for your functions and arguments. Please turn in the two .c source files, one readme.txt file, and two makefiles. The makefile must compile both uftp_client.c and uftp_server.c into their respective executables. The readme.txt file must explain what you have done and how to run it. The documentation does not have to be long, but does have to be very clear. The code that you turn in for this programming assignment must be your own original work and must compile and run on the CSEL machines. Upload the following files via GitHub Education (see instructions at end of document):

README.txt

client/uftp_client.c

client/makefile

server/uftp_server.c

server/makefile

We recommend you to put client and server in separate folder to better maintain your local files. The makefile in client/server folder should be used to compile client/server source code, respectively.

Getting Started:

(Please read as this may answer some of your questions)

An udp_example.tar file should be located from the initial GitHub repo. To extract files from udp_example.tar on a Linux machine, type 'tar -xvf udp_example.tar'. This should create a directory named “udp” which contains several files including “udp_client.c” and “udp_server.c” and testing files. These files contain a pseudo-code and not the actual implementation. You will have to modify both programs accordingly. After modifying the programs, following commands should be used to compile the programs.

To compile the files type:

- gcc udp_client.c -o client
- gcc udp_server.c -o server

To run the programs after compiling:

- Run the server by typing : ./server [port_number]
- Then run the client by typing : ./client [ip_address] [port_number]

The IP address is the server's IP address. The port number is the server's port number. You are informing the client where the server is located. You can test your application locally by running both the client and the server on the same machine. In this case, you can type “localhost” in place of the IP address. Feel free to use these files as a baseline for your design. You will have to modify parts of the code, especially for the server where parsing of the message is required.

Header Files:

Examine the code in the provided programs. The list of header files that need to be introduced in the socket programming are listed below.

- sys/socket.h : The header file socket.h includes a number of definitions of structures needed for sockets.
- netinet/in.h : The header file in.h contains constants and structures needed for internet domain addresses.
- arpa/inet.h : The header file contains definitions for internet operations.
- netdb.h : This header file contains definitions for network database operations.

Functions:

A brief explanation of some of the functions used in the code is provided here. However, for in depth understanding of the functions, please read the manpages of the functions.

- socket() : The input parameters of the function lets you determine which type of socket you want in your application. It may be a TCP or UDP socket. The function returns a socket descriptor which can prove helpful later system calls or -1 on error. A quick look at the function:
  - sockfd = socket(PF_INET,SOCK_DGRAM,0);
  - sockfd is a UDP socket.
- bind() : Once we have our socket ready, we have to associate it with a port number on the local machine. It will return -1 on error. When calling bind function, it should be noted that ports below 1024 are reserved. Any port number above 1024 and less than 65535 can be used. A quick reference:
  - bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr);
- sendto(): This function is used to send the data. Since it is used in unconnected datagrams, the input paramater of this function includes the destination address.It returns the number of bytes sent on success and -1 on error.
  - ssize_t sendto(int sockfd, void *buff, size_t nbytes, int flags, const struct sockaddr* to, socklen_t addrlen);
  - “buff” is the address of the data (nbytes long).
  - “to” is the address of a sockaddr containing the destination address.
- recvfrom() : This function is used to receive the data from an unconnected datagram socket. The input paramters contain the address of the originating machine. It returns the number of bytes received on success and -1 on error
  - ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags, struct sockaddr* from, socklen_t *fromaddrlen);

Note: It will be beneficial if you read the Beej’s Socket Programming guide (provided as a reference book for the course) before starting the actual assignment.

How do we test:

- **We run your client application and server code on one of CSEL machines (mentioned in the last two pages of this document).**
- All basic requests work well (put, get, delete, ls, exit) when the server is remotely running. Each request gives 20 credits when it works correctly. If you fail running server remotely but successful running in locally, you will get 15 credits for each successful request. For **reliable transfer** (see below, bonus 10 credits), you must transfer a large file (>5 MB) into a **remote** server.

Non-blocking client (not required)

Also, on the client side, it may be beneficial for you to configure a non-blocking socket, so the client does not lock up waiting for a message from the server in recvfrom(). If the socket receive buffer is empty for a blocking UDP socket, the calling application is put to sleep until a UDP datagram arrives. If a non-blocking UDP socket cannot return a complete datagram, then it returns immediately with an error status of EWOULDBLOCK, which you should check for. To set a socket as non-blocking, use fcntl(). A typical line might look like:

fcntl(sockfd, F_SETFL, O_NONBLOCK). See the manage for fcntl for more information. It will be easier to use a default blocking socket on the server side.

Reliable Transfer (10 Extra Credits)

Implement a reliable transfer mechanism. Due to the best-effort nature of the packet switching, the file transfer from your client to the server may experience a few packet losses. Come up with your own mechanism to overcome this issue. You may try to send a relatively large file (>100MB) between the client and the server and check the file received has no errors. You can compare the MD5 hash of the file received with the MD5 hash of the file sent.

Hints: Implementing redundancy would be an easy solution. Implementing a reliable UDP transfer in the protocol would be challenging but would be a good learning experience.

#>md5sum foo3

```
684db93c969206cdc71cb0bf56b8c395  foo
```

#>md5sum foo3.received

```
684db93c969206cdc71cb0bf56b8c395  foo.received
```

The two files (the file sent and the file received) are identical if the hash values of the two files match.

HOW TO:

Using Man Pages

If you wish to read the man pages of any of the above functions, type man [function_name]. For example, in order to check the man page of sendto, we will have to type man sendto. Press "q" in order to quit out of the man page.

Other useful C library methods to look up may be: atoi(), htons(), bzero(), bcopy(), strncmp(), strncpy(), fopen()

CSEL Machines

Your code must compile and execute on the machines in the CSEL Linux cluster. Everyone should already have an account (just use your CU IdentiKey username and password). Details about the CSEL Linux cluster can be found here: https://csel.cs.colorado.edu/helpatcs_elra.html#remote-access. You will be able to access your lab account via SSH from anywhere in the world (practically). If you have not previously used SSH, there are many references online. You can also post questions to the class discussion board. SSH allows you to remotely connect to another server or virtual machine. From a terminal window, you can type: "ssh [username]@elra-01.cs.colorado.edu" to log into the server elra-01.cs.colorado.edu. There are other machines available (see below).

For the Host Name (aka server) use one of the following (elra == educational labs remote access):

- elra-01.cs.colorado.edu
- elra-02.cs.colorado.edu
- elra-03.cs.colorado.edu
- elra-04.cs.colorado.edu

Turn-in instructions:

We are using GitHub classroom to turn in and track assignments. There exists some suggest Git literature on the course homepage. Please commit code early and often. Make sure both team member names are in the README. Here is the link to join the assignment:

- https://classroom.github.com/g/raYUDd8i
- When clicking the link, you will be asked to create a team. Please name your team:
    - LASTNAME1-FIRSTNAME1-LASTNAME2-FIRSTNAME2
    - Where LASTNAME1-FIRSTNAME1 is the first team member's name and LASTNAME2-FIRSTNAME2 is the second team member's name.

- Only one GitHub repo is needed per team. You can do this assignment individually if you choose. Students can form their own teams, with at most 2 students on a team.

External References:

Link to useful UNIX tutorial

- http://www.tutorialspoint.com/unix/index.htm

Link to widely used Unix shell commands tutorial :

- http://infohost.nmt.edu/tcc/help/unix/unix_cmd.html

Following is the link to a simple UDP client/server system in C/Unix environment

- https://www.cs.rutgers.edu/~pxk/417/notes/sockets/udp.html

Following link is the highly rated Socket Programming tutorial in the Web. It explains all system calls used for socket programming in a clear way and also about Networking concepts in general.

- http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html

Following link is to useful shell debugging tools and commands

- http://www.tutorialspoint.com/gnu_debugger/index.htm

The following link is to the online tar man page

- http://manpages.ubuntu.com/manpages/intrepid/man1/tar.1.html

Quick makefile how to

- http://mrbook.org/tutorials/make/