ECE361 COMPUTER NETWORKS

File Transfer[1]

Fall 2022

Objective

The following practical labs provide you with some hands-on experience with socket programming. You will use UNIX sockets to implement simple client and server programs which interact with each other to accomplish a file transfer in a connectionless manner.

Lab Assignment

In this assignment, you need to implement a server that opens a socket and listens for incoming data transfer at a specific port number. You also need to implement a client that reads a binary file from the file system and transfers the file to the server. When the server receives the client's data, it writes the data to a file.
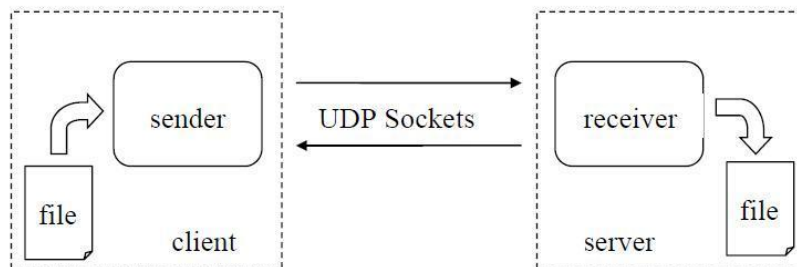
References

- Section 2.4 and related sections on Berkeley API from Chapter 2 of the *Communication Networks* by Alberto Leon-Garcia and Indra Widjaja, McGraw Hill, 2nd Edition, 2004.

- The network socket programming "Beej's Guide to Network Programming" available on the course website under the course section.

Section 1

In this section, you will implement simple client/server programs. The client and server will each use a UDP socket for sending and receiving. Also, the client and server must be run on different UG machines. You can assume reliable delivery of messages for this part of the lab.

Server Program (server.c)

You should implement a server program, called "`server.c`" in C on a UNIX system. Its execution command should have the following structure:

```
server <UDP listen port>
```

Upon execution, the server should:

1. Open a UDP socket and listen at the specified port number
2. Receive a message from the client

    a. if the message is "ftp", reply with a message "yes" to the client.
    b. else, reply with a message "no" to the client.

Client Program (deliver.c)

You should implement a client program, called "`deliver.c`", in C on a UNIX system. The client program will send a message to the server. Its execution command should have the following structure:

```
deliver <server address> <server port number>
```

After executing the server, the client should:

1. Ask the user to input a message as follows:

   ```
   ftp <file name>
   ```

2. Check the existence of the file:
       a. if exist, send a message "ftp" to the server
       b. else, exit

3. Receive a message from the server:

       a. if the message is "yes", print out "A file transfer can start."
       b. else, exit

Question: Can we use string functions on messages?

Section 2.1

Based on the client and server in section 1, you need to measure the round-trip time from the client to the server. Chapter 21 of the glibc manual may be a useful reference for time functions and concepts.

Question: How long is the measured round-trip time?

Section 2.2

In this section, you will implement a client and a server to transfer a file. Unlike simply receiving a message and sending it back, you are required to have a specific **packet format** and implement **acknowledge** for the simple file transfer using UDP sockets. You can assume reliable delivery of messages for this part of the lab. Please see the execution example on page for instructions on how to test your programs.

Packet Format: all packets sent from the client to the server must have the following structure:

```
struct packet {
     unsigned int total_frag;

     unsigned int frag_no;
     unsigned int size;
     char* filename;
     char filedata[1000];
}
```

The `total_frag` field indicates the total number of *fragments* of the file. Each packet contains one fragment. The `frag_no` field indicates the sequence number of the fragment, starting from 1. The `size` field should be set to the size of the data. Therefore, it should be in the range of 0 to 1000. All members of the packet should be sent as a **single message**, with each field separated by a colon. For instance:

```
total_frag = 3
frag_no = 2
size = 10
filename = "foobar.txt"
filedata = "lo World!\n"
```

Your packet should look like this:

```
 packet = "3:2:10:foobar.txt:lo World!\n"
```

Please remember that while the beginning of the packet is in fact just plain text, the data portion of the packet may in fact contain *binary* data. **This means that you should not use string manipulation functions available in C for the data field or for the whole packet.** Only the first part of the packet before data is really a string.

The reason you cannot use string functions is because string functions assume that the data ends with the null character. This character however, may appear *within* the data of the packet. If you were to use `strcpy` on a packet with binary data, some of your data may get lost and your program will not function correctly. You should test your program on both binary data (an image file for instance) as well as a text file. In general, if your program works for binary data, it will work for a text file.

Acknowledgement: **You should implement some sort of acknowledgement to guarantee correct receipt of the file.** For this assignment, you may use a simple stop-and-wait style acknowledgement. The packet format for acknowledgements is up to you.

The server may use ACK and NACK packets to control data flow from the sender. The client should use its UDP socket to listen for acknowledgements from the server. You will have to carefully coordinate between the client and the server to guarantee correct file transfer.

Client Program (deliver.c):

The execution command should have the following structure:

```
   deliver <server address> <server port number>
```

Upon execution, the client program should read data from a file specified by the user, construct `struct packets`, and send the packets to the server using a UDP socket. If a file is larger than 1000 bytes, it needs to be fragmented into multiple packets with maximum size 1000 before transmission.

You should test your code with large files (you can try putting them in the `tmp/` directory on the UG machines; we have heard this does not affect your disk quota). You can use the following command to create a large file (change the "5M" at the end to suit your needs):

```
dd if=/dev/zero of=./test.file bs=4k iflag=fullblock,count_bytes count=5M
```

Server Program (server.c):

```
server <UDP listen port>
```

Upon receiving the first packet in a sequence (i.e. `frag_no = 1`), the program should create a `struct packet` containing the received data, read the file name from it, and create a corresponding file stream on the local file system (the received file *must* have the same name as the sent file). File data read from packets should then be written to this file stream. When the last packet is received, the file stream should be closed.

Section 3

One file is segmented into packets for transfer, and acknowledgement guarantees correct receipt of the file. If one packet from the client is lost, what will happen? If an ACK/NACK packet is lost, what will happen?

Timeout: You should implement a timer for ACK/NACK packets at the client which is based on measured RTT values. After sending a packet, the client should wait for an ACK in a time period of t1. If the ACK packet doesn't come within t1, the client should assume a packet loss and resend it. Your client should print a message whenever it retransmits.

<u>**Question**: How have you seen retransmission timeouts be calculated in class?</u>

Packet drops: You should simulate packet drops at the server to test your retransmission code at the client. Upon receiving a packet, the server should draw a random number and **only send back an ACK and process the packet if the number exceeds some threshold**. For example, given a function `uniform_rand()` that generates a uniformly-distributed random number between 0 and 1, the following code will drop 1/100 packets on average:

```
if (uniform_rand() > 1e-2) {
    Send ack to client
    Process packet
}
```
Your server should print a message whenever a packet drop occurs.

Makefile

You should also prepare a `makefile` that generates the executable file `deliver` from `deliver.c` and the executable file `server` from `server.c`.

Execution Example

Assuming you have a file named `source.jpg` on `ug201` which you wish to send to `ug202` (in this server, port 5000 is used):

On the host `ug202`:

```
  server 5000
```

On the host `ug201`:

```
  deliver ug202.eecg.utoronto.ca 5000
  ftp source.jpg
```

Remember that your two programs need to be in separate folders (e.g., `deliver/` and `server/`) as the file cannot be copied onto itself. You can verify correct operation of your code by performing a binary `diff` on the source and destination files. For example, if running the command `diff deliver/source.jpg server/source.jpg` results in nothing being printed, then `deliver/source.jpg` and `server/source.jpg` are **binary identical** (i.e., every bit is the same); otherwise, `diff` will show you the lines that are different.

Deliverables:

The following should be available for the lab evaluation:

- The client program (deliver.c)

- The server program (server.c)

- Makefile to compile your program

- Any extra header files or source code necessary for correct operation of your code

Your code must be able to run on the UG machines.

Submission Procedure:

For electronic submission, one submission per group is required. Please put all files requested above in a folder named **UTORID1_UTORID2** in which **UTORID1** is the UTOR ID of the first member of the group and **UTORID2** is the UTOR ID of the other member. Then compress the folder and create a UTORID1_UTORID2.tar.gz file.

The following command can be used to tar your folder:

```
 tar -czvf UTORID1_UTORID2.tar.gz UTORID1_UTORID2
```

where the folder UTORID1_UTORID2 contains all the files requested above.

Use the following command on the UG system to submit your code, where X is the number of the lab (ranging from 1 to 5 for 5 programming labs):

```
submitece361f X UTORID1_UTORID2.tar.gz
```

You can perform the electronic submission any number of times before the actual deadline. A resubmission of a file with the same name simply overwrites the old version. To see a list of what you have submitted, use the command:

```
submitece361f -l X
```