ECE361 Computer Networks

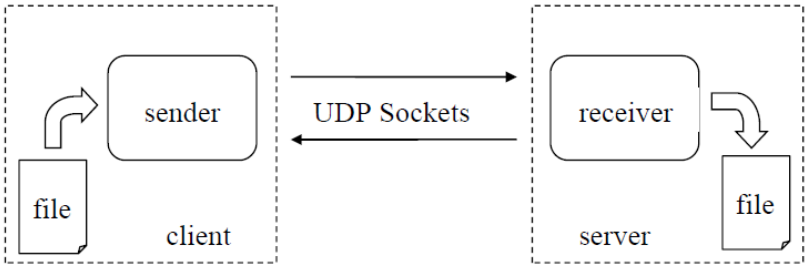
File Transfer Lab

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 particular 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

- Related sections in Chapter 2 of the course textbook.
- The network socket programming "Beej's Guide to Network Programming" posted on the course website.
- Section 2.4 of the "Communications Networks", by Leon-Garcia and Widjaja, McGraw Hill, 2nd Ed., 2004.

Section 1

In this section, you will implement simple client/server programs. The client and server will use a UDP socket for sending and receiving.

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 a message “yes” to the client.
    b. else, reply a message “no” to the client.

Client Program (deliver.c)

You should implement a send program, called deliver.c, in C on a UNIX system. 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 follows the format:

```
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

Based on the above client and server, you need to measure the round trip time from the client to the server.

Question: How long is the measured round trip time?

Section 3

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 socket.

Packet Format: all packets sent between the client and 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 string**, each field separated by a colon. For instance:

```
total_frag = 3
frag_no = 2
size = 10
filename = "foobar.txt"
filddata = "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 in order to guarantee correct receipt of the file.** For this assignment, you may use a simple stop-and-wait style acknowledgement.

The server may use ACK and NACK packets to control data flow from the sender. The client should open a UDP socket to listen for acknowledgements from the server. You will have to carefully coordinate between the client and the server in order 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 user and send it to the server using a UDP socket. If a file is larger than 1000 bytes, the file needs to be fragmented into smaller packets with maximum size 1000 before transmission.

Server Program (server.c):

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

Upon receiving the first packet in a sequence (i.e. `frag_no = 1`), the program should read the file name from the packet and create a corresponding file stream on the local file system. Data read from packets should then be written to this file stream. If the EOF packet is received, the file stream should be closed.

Section 4

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 packet at the client. After sending a packet, the client should wait for an ACK in a time period t1. If the ACK packet didn't come within t1, the client assumed a packet loss happened and resend it.

Question: For a timeout, how do we select the value of t1?

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 <filename>
```

Remember that your two programs need to be in separate folders 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 file.

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.

Submission Procedure:

For electronic submission, one submission per group is required. You have to create a tar ball (a1.tar.gz) with all the files needed to compile and run your programs.

The following command can be used to tar your files:

```
tar –czvf a1.tar.gz <project directory>
```

where the project directory contains your source code, headers, and Makefiles.

Use the following command on the eecg UNIX system to submit your code:

```
submitece361s 1 a1.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:

```
submitece361s –l 1
```