CSEE 4119 Computer Networks Programming Assignment 1 - File Transfer

Professor Vishal Misra

due: 03/24/23 @ 11:59 PM

*Adapted from Professor Gil Zussman's Simple Chat Application assignment

1 Introduction

The objective of this programming assignment is to implement a simple file transfer application with at least 3 clients and a server using both the TCP and UDP protocol where the overall system offers at least 10 unique files. You are asked to create one program in either Python (strongly encouraged) or C. The program should have two modes of operation, one is the server, and the other is the client. The server instance is used to keep track of all the clients in the network along with their IP addresses and the files they are sharing. This information is pushed to clients and the client instances use these to communicate directly with each other to initiate file transfers. All server-client communication is done over UDP, whereas clients communicate with each other over TCP. The functionalities and specification of each program are described in detail below. It is important that you read the entire assignment when you receive it so that you can start thinking about how to tackle each part. And make sure to start early!

2 Functionalities

The complete file transfer application can be broadly classified into five functions outlined below. Each function involves either the client part or the server part or a combination of the two. The five functions and their respective parts in both the server and the client are explained in the following sections.

2.1 Registration

For the registration function, the server takes in registration requests from clients using the UDP protocol which means that the server needs to be started before the clients can start coming online. The purpose of the registration is that only registered clients should be able to offer files and will receive the updated list of files shared by other registered clients. Not all clients that register must offer files. The server has to maintain a table with the nick-names of all the clients, their status, the files they are sharing along with their IP addresses and port numbers for other clients to request files. This functionality has server and client components.

Client mode:

- The client has to communicate with the server using the IPaddress and the port number of the server [assume all clients by default know the server information].
  - `$ FileApp <mode> <command-line arguments>` : Start the program for server and client (for example: `FileApp -c` for client and `FileApp -s` for server). The server mode takes one argument: its listening port. The client mode should take five arguments.
  - `$ FileApp -s <port>` : Initiates the server process
  - `$ FileApp -c <name> <server-ip> <server-port> \ <client-udp-port> <client-tcp-port>`

    Initiates client communication to the server. Client name is like a username for this client in this network. Server IP address should be given in decimal format and the port number should be an integer value in the range `1024-65535`. The client listens on 2 ports: `client-udp-port` for communication with the server over UDP, and `client-tcp-port` for listening to TCP connections requests from other clients

for file transfers. **HINT:** multi-threading in Python or C will be very helpful for listening for server updates and listening for TCP connection requests from other clients. For example, if the server IP is `198.123.75.45`, the server port is `1024`, the client's port numbers for listening is `2000` and `2001`, then the command will be: `$ FileApp -c client-name 198.123.75.45 1024 2000 2001`. If arguments are taken in a proper format, a prompt like >>> should be displayed. The application should also be able to perform basic error checking where the IP addresses are valid numbers, and assigned ports are within the range. Otherwise, an appropriate error message should be displayed.

- Successful registration of the client on the server should also display the status message to the client:

  `$ >>> [Welcome, You are registered.]`

  Otherwise, an appropriate error message should be displayed and the client program should exit.
- Every client should also maintain a local table with information about all the other clients (name of offered files, client name, IP, TCP port number). Every client should update (overwrite) its local table when the server sends information about all the other clients. Once the table is received, the client should send an *ack* to the server.
- When the table has been successfully updated, the client should display the message :

  `$ >>> [Client table updated.]`
- There should be two ways to 'disconnect/close' as a client:
  - Silent leave: Once a client disconnects/closes, the server will not be notified. You can expect that the client will not register again using the same information after it exits via Silent leave. To exit or close, a client uses `$ >>> CTRL + C` or simply closes SSH window that the client is running on (both actions need to be implemented, and the system should not crash).
  - Notified leave: De-registers the client, and the de-registration action will be notified to the server. The client status in the server table should be changed to offline. More detailed information is covered in Section 2.5.

Server mode:

- The server process should maintain a table to hold the names, online-status, IPaddresses, TCP and UDP port numbers, and filenames offered of all the clients.
- When a client sends a registration request, it should add the client information (name, IPaddress, tcp and udp port numbers, online-status) to the table. If `name` has already been registered by another client, the server should respond to the client with an error message, rejecting the registration.
- When a client registers successfully, the server should send to the client a transformed version of its table: consisting of all filenames offered by clients along with the client name of the file owner, the IPaddress and TCP port number at which each file could be requested. This allows the registered client to initialize its local table. This transformed table of information about shared files is also broadcasted whenever a client offers a new file to be shared. More detailed information is covered in Section 2.2.
- If the server does not receive an *ack* from the client within 500 msecs, it should adopt a best effort approach by retrying 2 times.

2.2 File Offering

Once the clients are set up and registered with the server, the next step is to implement the functionality for clients to advertise file offering to other registered clients through the server. Whenever the server receives a file offering message, the server adds an entry to its table of filenames offered by clients which is then broadcasted to all active clients. This functionality has server and client functionality.

Client side:

- The client should support the following command to set the directory containing the files it's going to offer.

  `$ >>> setdir <dir>`

  The client should check for the existence of the directory in the filesystem, if it exists, it should print a success message:

```
$ >>> [Successfully set <dir> as the directory for searching offered files.]
```

Otherwise, an error message should be printed:

```
$ >>> [setdir failed: <dir> does not exist.]
```

- To offer a file, the client should communicate to the server over UDP using the UDP port. The client should support the `offer` command to offer one or more files:
  `$ >>> offer <filename1> ...` where `...` means zero or more filenames (delimited by space) could be provided as additional arguments. This command should make the client send to the server a UDP message to notify the server this provided list of filenames it's now able to share with other clients.
- Attempting to offer a file before issuing the setdir command should fail with an appropriate error message.
- The client which sends the message of file offering has to wait for an *ack* from the server and likewise, the server has to send an *ack* once it receives the message.
- If *ack* times out (500 msecs) for a message sent to the server, the client should adopt a best effort approach by retrying two times. If it still fails, it means that the server might be too busy at the moment or there might be a network partition that prevents the client message from reaching the server.
  The appropriate status messages also need to be displayed for each scenario:
  `$ >>> [Offer Message received by Server.]`
  `$ >>> [No ACK from Server, please try again later.]`

Server mode:

- Upon receiving an offer message, the server should send an *ack* back to the client and add zero or more entries to its table. As mentioned in Section 2.1, the server needs to keep track of the nick-names of all the clients, their online-status, the files they are sharing along with their IP addresses and port numbers for other clients to request files. However many tables and and whatever data structures used to keep track of these information is up to you, but you should take care of not adding duplicate entries of file offerings by the same client.
- If the server table(s) is updated, the server should also broadcast to all active clients the most updated list of file offerings. For each file offering, it should contain the filename, the name of client that offered the file, the IPAddress, and TCP port at which the file could be requested. To save network bandwidth, a file offering by an offline client should not be included.

For example:

- 2 clients (A and B)
- Client A offers foo.txt and bar.out to be requested on port 10000
- Client B offers foo.txt and baz.out to be requested on port 11000

Terminal of client A:

```
$ >>> offer foo.txt bar.out
$ >>> [Offer Message Received By Server]
```

Terminal of client B:

```
$ >>> offer foo.txt
$ >>> [Offer Message Received By Server]
$ >>> offer baz.out
$ >>> [Offer Message Received By Server]
```

Upon receiving two offers, the server will have a table with 4 file offerings along with the addresses these could be requested, which could look like the table below:

Filename	Owner	Client IP Address	Port
foo.txt	A	192.168.0.10	10000
bar.out	A	192.168.0.10	10000
foo.txt	B	192.168.0.22	11000
baz.out	B	192.168.0.22	11000

2.3 File Listing

Next, we implement the functionality to allow each client to view the list of files available to be downloaded. When the server pushes the most up-to-date list of file offerings, the client stores and updates its own local table. Upon a file list command, the client formats its local table and prints it to the terminal on the client side. This functionality involves only the client.

Client side:

- The client should support the `list` command to view the list of available file offerings by other clients.

  ```
  $ >>> list
  ```

  Note that the client should only use its local table to print out the information. If no file offerings are available, the following message should be printed:

  ```
  $ >>> [No files available for download at the moment.]
  ```

  Otherwise, a table of the file offerings should be printed (see example below for details). The list should be ordered alphabetically by filename and ties should be broken by using the client name.

For example:
Using the example from Section 2.2 as an running example, now

- Client C registers at the server
- After receiving the list of file offering from the server for the first time, client C issues the list command

Terminal of client C:

```
$ FileApp -c C 128.20.143.10 1000 12000
$ >>> [Welcome, You are registered.]
$ >>> [Client table updated.]
$ >>> list
$ FILENAME   OWNER   IP ADRESS       TCP PORT
$ bar.out    A       192.168.0.10    10000
$ baz.out    B       192.168.0.22    11000
$ foo.txt    A       192.168.0.10    10000
$ foo.txt    B       192.168.0.22    11000
$ >>>
```

Terminal of client A:

```
$ >>> [Client table updated.]
$ >>> list
$ FILENAME   OWNER   IP ADRESS       TCP PORT
$ bar.out    A       192.168.0.10    10000
$ baz.out    B       192.168.0.22    11000
$ foo.txt    A       192.168.0.10    10000
$ foo.txt    B       192.168.0.22    11000
$ >>>
```

2.4 File Transfer

To request a file, the client will first use its table to figure out who the file owner is. Then, the client will establish a TCP connection with the file owner. File transfers are done directly between clients. Since it doesn't involve the server, there is just the client part for this feature.

Client side:

- `$ >>> request <filename> <client>`: if `client` is not the same as the requesting client and `filename` is indeed offered by `client` according to the requesting client's local table, the requesting client should proceed to establish a TCP connection with the `client` that owns the file to request `filename`. Otherwise, an error message `Invalid Request` should be printed to the terminal and the client should not attempt to establish a TCP connection.

- The client should have a separate socket to request files by establishing a TCP connection with another client.
- For simplicity, the client should store the requested file under its starting directory (the directory under which `FileApp -c ...` is run to start the client).
- Appropriate status message should be printed to the terminal at critical points of the file transfer on both clients: TCP connection establishment, upon receiving/transmitting the first data packet, completion of file data transfer, after closing TCP connection (see example below). To distinguish status messages of file transfers from other status messages, these are enclosed within a pair of angle brackets `< >` with no preceding input prompt `>>>`.

For example:
Continuing the running example,

- Client C requests `foo.txt` from client B
- Client B and C establishes a TCP connection, file `foo.txt` transferred successfully, and finally the connection is closed
- Client C requests baz.out from client A
- The request is denied because in Client C's local table, client A doesn't own baz.out, a connection request to client A is not sent

Terminal of client C:

```
$ >>> request foo.txt B
$ < Connection with client B established. >
$ < Downloading foo.txt... >
$ < foo.txt downloaded successfully! >
$ < Connection with client B closed. >
$ >>> request baz.txt A
$ < Invalid Request >
```

Terminal of client B:

```
$ < Accepting connection request from 128.20.143.10. >
$ < Transferring foo.txt... >
$ < foo.txt transferred successfully! >
$ < Connection with client B closed. >
```

2.5 De-registration

This is a book-keeping function to keep track of active clients. This functionality involves both client and server parts.

Client side:

- When a client is about to go offline, it should immediately stop listening or ignore incoming requests on the TCP port for incoming file requests. Then, it has to send a de-registration request to the server to announce that it is going offline: `$ >>> dereg <nick-name>`.
- The client has to wait for an ack from the server within 500 msecs. If it does not receive an ack, the client should retry for 2 times. If it fails all three times the client should display the message:

  `$>>> [Server not responding]`

  `$>>> [Exiting]`

  and terminate the program.
- Please note: you should not terminate the client program after *successful* de-registration. You will be expected to register the client back later to receive future file requests from other clients (if there is any). You do not need to consider a case in which another client uses the same information to register while the client is de-registered.
- Successful de-registration from the server should display the following status message in the client:
  `$ >>> [You are Offline. Bye.]`

Server side:

- When the server receives a de-registration request from a client, it has to change the respective client's status to offline in the table.
- If the client has previously made file offerings, these will be marked as temporarily unavailable until the client logs back in again. The server then has to *broadcast* the updated table of file offerings to all the active (online) clients.
- The server then has to send an *ack* to the client which requested de-registration.

3 Testing

Before submitting your work, please do **test your programs thoroughly**. Your file transfer application should at least work with:

- *One* instance of the program in server mode.
- *Three* instances of the program in client mode.
- *Ten* files being offered with at least *five* being unique.

You must handle business logic errors such as a user trying to log in with an already connected nickname.

4 Submission Instructions

You may use either Python or C for developing the chat application. Detailed submission instructions will be released later but your submission package should include the following deliverables:

1. `README`: Name and UNI should be at the top. The next thing in your README should be explicit command line instructions for compiling and running your program. The file should also contain basic project documentation, program features, brief explanation of algorithms or data structures used, a list of known bugs, and the description of any additional features/functions you may have implemented (the last part is fully optional but the Name/UNI and compilation/running instructions are required).
2. `Makefile`: This is only required if you are using C. The output name of the program should be `ChatApp`.
3. Source code with some comments and clear variable names. If it is a C program, the source code file should be called ChatApp.c and if its a Python program, the source code (and executable in this case) should be called ChatApp.py. All code MUST be in that file.
4. `text.txt`: This file should contain some output samples from the command line on several test cases. This will help others to understand how your programs work in each test scenario. It is optional to include this as a section of your README document.

All of these will be zipped into a file with the following format:
`<last-name>_<UNI>_PA1.zip`

For example, it would be `Asif_tma2153_PA1.zip` for Tameem.

Further instructions on submission (including where to submit) will be released soon.

4.1 Development Environment Requirements

Please do not utilize Windows programming environments including .NET, Visual Studio, VC++, etc. Programs written in C have to be compiled using gcc, not clang or another compiler. All submissions will be compiled, run, and evaluated on Ubuntu 20.04 LTS. If you have any issues with your environment, please let the TA know early on.

For Python development, you must use Python 3.9 or a later version. We will not be grading with versions below that.

Finally, good luck and let us know on EdStem if you have any questions!