



readME

⚙ Status	Done
----------	------

- Name: Erin Liang
- uni: ell2147



The following python program implements a simple file transfer application with at least 3 clients and a server using both TCP and UDP protocols where the overall system offers at least 10 unique files.

Setup

Required Python libraries



these must be installed before running ChatApp. I'll include a requirements.txt file to make your life easier for the next project!

- argparse
- socket
- ipaddress
- os
- threading
- queue
- json
- operator
- prettytable
- sys

Running the program

Server mode

- general usage:

```
python3 ChatApp.py (-s | -c) <port>
```

- example: initiating ChatApp server running on port 1025

```
python3 ChatApp.py -s 1025
```

Client mode usage

- general usage:

```
ChatApp.py (-s | -c) <name> <server-ip> <server-port> \
<client-udp-port> <client-tcp-port>
```

- example: initiating ChatApp client “erin” running locally that communicates on port 1026 with the server that listens on port 1025. “erin” listens to file requests from other clients on port 1027.

```
python3 ChatApp.py -c we 0.0.0.0 1025 1026 1027
```

Help

- a help option is included to make interacting with the program arguments easier:

```
python3 ChatApp.py -h
```

- For both modes, arguments are also printed out upon successful program initiation

```
$ python3 ChatApp.py -c we 0.0.0.0 1025 1026 1027
=====
Printing args:
server False
client True
name we
server-ip 0.0.0.0
server-port 1025
client-udp-port 1026
client-tcp-port 1027
=====
...
```

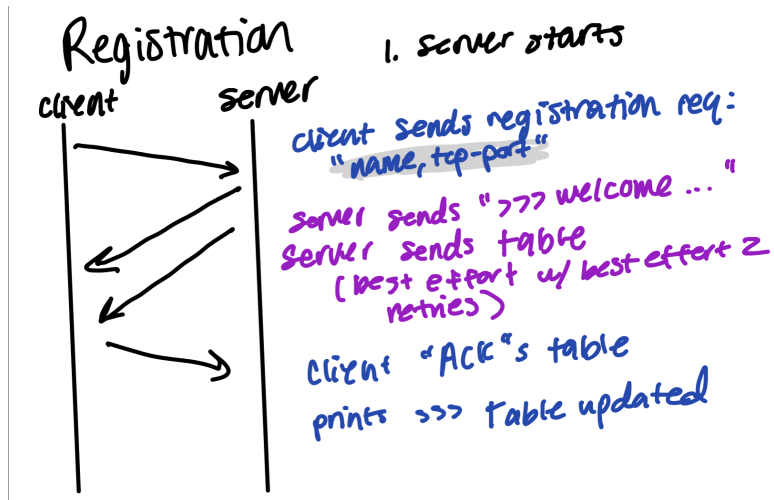
Program Features

- The client can register, offer files to other clients by sending a UDP message to the server, request files from another client, list files, and deregister

- the following diagrams help demonstrate the sequence of messages exchanged for each successful command. See commented code for more details.

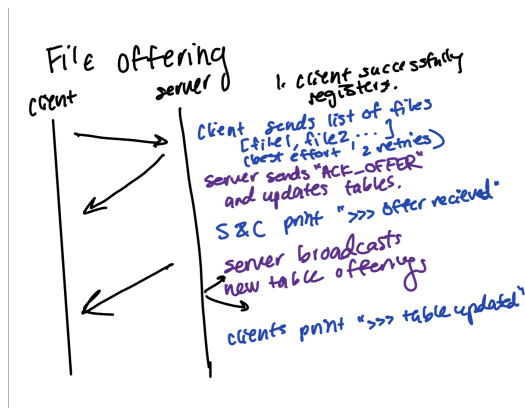
Registration

- successful registration messages exchanged:



File Offering

- successful file offering messages exchanged

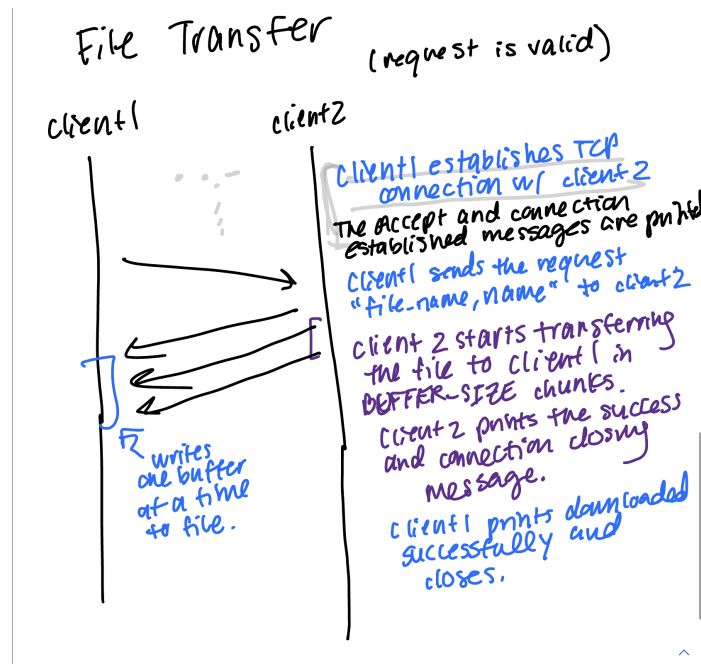


File Listing

- done entirely on client-side and just really using prettytable library

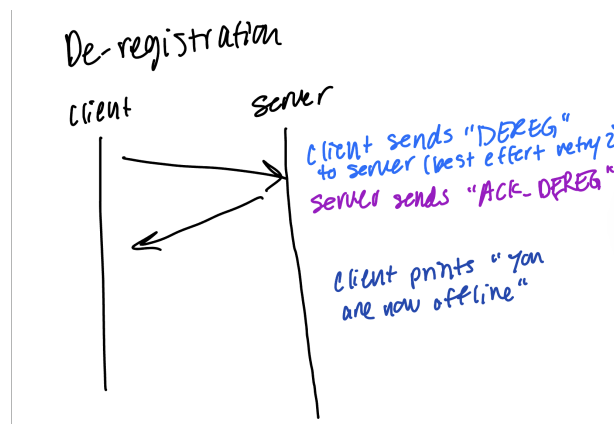
File Transfer

- successful file transfer messages exchanged



De-Registration

- successful de-registration sequence:



Explanation of Algos + Data Structures



example runs corresponding to the grading rubric are included in the [Testing section](#). These can also be found in the tests.txt file included in the submission.




The print messages with "!!!" and "DEBUG" can be uncommented to see all the messages passed between the ChatApp instances. In the future, logging levels would be nice.

General Control Flow + Notes on Threading with shared sockets

- The ACK system employs a best effort retry twice mechanism. In all cases where the program instance is expecting an ACK and retries up to 2 times, we mark the client as offline.

Ed — Digital Learning Platform

Ed is the next generation digital learning platform that redefines collaboration, communication, and computational thinking


 <https://edstem.org/us/courses/36439/discussion/2833815>

Client Control Flow

- After the client successfully registers, the client needs to do three things simultaneously:
 - Listen for incoming TCP connections from other clients.
 - Listen for incoming UDP messages from the server.
 - Listen for user input.
- Each of these tasks is handled by a separate thread. The listening for user input happens in the main thread and the rest are in daemon threads so we don't have to `ctrl+C` multiple times to end the client program.
- After registration, the client's UDP socket is used to receive table updates from the server, receive ACKs that its file offering has been received, and receive ACKs that the deregistration went through.
 - if we `recvfrom` in more than one of these threads, the thread that is waiting for an ACK could receive a table update and not know how to handle it (or vice versa). Instead of trying to pass the received message to another thread, which introduces a good deal of complexity, we just have one thread listening to the UDP socket.
 - This thread, executing `listen_for_server_updates`, will be the only thread that `recvfrom`s the client UDP socket and puts the message on the appropriate ACK queue or updates the table.
 - TODO: might not have to do queues because at a given time, the client is only executing one command at a time. Separate queues are safer though for peace of mind for grading, so I kept them in.
 - reference:

Ed — Digital Learning Platform

Ed is the next generation digital learning platform that redefines collaboration, communication, and computational thinking

 <https://edstem.org/us/courses/36439/discussion/2831859>

Server Control Flow

- The server is always listening to the socket bound at the specified port and funnels requests to the appropriate handlers depending on the type of request. The request types are:
 - registration
 - deregistration

- file offers
- The server does not have any threading, though it would be helpful. See future work section.

Server's Registration Table

- this is a dictionary with the following format:

```
(client_ip, client_udp_port) : {
    "name": name,
    "status": status,
    "client_ip": client_ip_address,
    "client_tcp_port": client_tcp_port,
    "files": set(of, file, names),
}
```

- When new clients register, `files` is initialized to an empty set
- why do we have (clientIP, client_udp_port) as the key?
 - This is basically the clientAddress. There are a number of reasons for using this server-side data structure:
 1. Checking whether a given clientAddress is active (during broadcasts) is efficient
 - `self.table[client_address]["status"] == "active"`
 2. Because the server is always listening to incoming client requests, it needs to frequently differentiate between different types of requests: registrations, offers, and deregistrations.
 - Checking for if the clientAddress exists in the dictionary keys is semantically equivalent to checking for whether the client has already registered. Note that we do not support `rereg`.
 - An alternative approach to distinguishing between different types of requests is to include the type of request in the message itself
 - e.g. This is also implemented in the code. If the client has already registered, the request is either a `dereg` or a file `offer`, from which we can differentiate from the message itself. See the `dereg` section.
 3. The only operation that would be sacrificed would be performing the lookup of whether a name is already taken by the client (but this doesn't happen that often anyway because there aren't that many clients that come and go).
- why is files a set?
 - Per the spec, a client cannot offer two files with the same. name
- this server table is updated when:
 - A client deregisters
 - files becomes empty, status becomes offline
 - a client offers another file
 - file is added to `files`, the client view is updated

Client's table (transformed table)


- this is represented in `FileServer.client_table_view` and `FileClient.local_table`. The format of the dictionary is the following:

```
{"file_name,client_name" : (client_ip, client_tcp_port), ...}
```

- why this key?
 - when requesting a file, the client needs to be able to quickly check whether that file actually belongs to that client.
 - need a str, int, float, bool or None to send over sockets using `json` dumps and loads to serialize and deserialize the dictionary, so we need to either use one field or concatenate multiple

Sending a Dictionary using Sockets in Python?

My problem: Ok, I made a little chat program thing where I am basically using sockets in order to send messages over a network.

 <https://stackoverflow.com/questions/15190362/sending-a-dictionary-using-sockets-in-python>



- we can't use `file_name` to index because multiple clients can offer the same file. Also, one host can have multiple instances of chatapp clients that offer the same file, so we can't index by `file_name,client_ip`. `file_name,client_name` guarantees us a unique key and the leftover values conveniently makes up the address where the client can request the file from the client.
- the udp ports are not included in this view because they are only for server-client communication.
- An alternative approach could be to call some sort of transformation algorithm to transform the server table to the client view every time it needs to be broadcasted. However, because server updates are likely common, we just choose to store extra state and maintain both views as we go.

Known Bugs

- Because of multithreading, the user prompts in client mode can become quite messy and interspersed with the server's broadcasts to the clients. For example, if client2 offers a file while client1 is waiting for the user's input, this is likely to appear on client1's terminal:

```
>>> >>> [Client table updated.]
# First >>> is from client1's waiting
# second >>> is from the table broadcast update.
```

- Not bugs, but there are some `TODOS` in the code to remove code paths where there should be an invariant (i.e. the code should never execute).

Assumptions

- The server should start before the client tries to connect to it. Otherwise, the behavior is undefined

- It is assumed that different clients will provide different `client-tcp` and `client-udp` ports from each other. Clients providing the ports that are already in use will probably fail with `OSError: Address already in use` and cause undefined behavior

Additional Features



most add-ons have to do with better argument handling:


- After the client deregisters, the thread executing the commands will still be going because the behavior is unspecified. All commands except `list` will stop working, returning “invalid command” outputs
- argparsing is more elegant by using the argparsing module
- Usages are printed after unsuccessful client commands
- works on both binary and text file transfer

Future work

- different levels of logging instead of commenting and uncommenting code for debugging.
- Adding multithreading into the server. Otherwise, while you're waiting for a new connection your server will be blocking and won't be able to respond to other messages and not able to send a timely ACK → Ed post

Ed — Digital Learning Platform

Ed is the next generation digital learning platform that redefines collaboration, communication, and computational thinking

 <https://edstem.org/us/courses/36439/discussion/2803329>

- Resetting setdir will lead to undefined behavior if the client instance already offered files.
- Reevaluate whether we need separate queues for dereg ack and offer ack..
- It doesn't make sense that deregister takes the name of a client. It should just be the current client because there are no other options..
- Implement `rereg` as it was alluded to in the spec.
- Restructure code into private helper methods vs public functions
- Fixing prompt ordering with thread race conditions

Testing

Registration Tests

1. Client successfully registers with an available username


```
# Server running on port 1025
$ python3 ChatApp.py -c hey 0.0.0.0 1025 1026 1027
=====
Printing args:
server False
client True
name hey
server-ip 0.0.0.0
server-port 1025
client-udp-port 1026
client-tcp-port 1027
=====
>>> [Welcome, You are registered.]
>>> [Client table updated.]
>>>
```

2. Server rejects registration request with a username already taken by another client.

```
$ python3 ChatApp.py -c hey 0.0.0.0 1025 1029 1030
=====
Printing args:
server False
client True
name hey
server-ip 0.0.0.0
server-port 1025
client-udp-port 1029
client-tcp-port 1030
=====
Client hey already registered. Registration rejected.
```

3. Upon successful registration, client's local table is initialized.

```
$ python3 ChatApp.py -c hey 0.0.0.0 1025 1029 1030
=====
Printing args:
server False
client True
name hey
server-ip 0.0.0.0
server-port 1025
client-udp-port 1029
client-tcp-port 1030
=====
>>> [Welcome, You are registered.]
[DEBUG] LOCAL TABLE: {}
```

4. Server retries sending the table of offered files a maximum of 2 times when the client ack is not received within 500ms. This was tested by commenting out the registration ack to the server.

```
Server view:
=====
Printing args:
```

```

server True
client False
port 1025
=====
[DEBUG] message from ('127.0.0.1', 1029): heyy,1030
[DEBUG] Sending table again...
[DEBUG] Sending table again...
[DEBUG] Sending table again...

```

- Client behavior
 - if only the client's ack is being lost, (i.e. client is receiving the table and its retransmissions), it will print the table three times.
 - Otherwise if it never receives anything from the server, the client program will be stuck on `recv`

File Offering Tests

The following tests assume the directory structure, where `ChatApp` is being run from its own folder

```

|-----dir
|   \_____hello.txt
|   \_____wee.txt
|   \_____jjs.jpg
|   \_____i_love_the_tas.txt
|   \_____1.txt
|   \_____2.txt
|   \_____3.txt
|   \_____4.txt
|   \_____5.txt
|_ ChatApp.py

```

`setdir` tests

- given an invalid directory, the client program doesn't crash and an appropriate error message is printed
- `offer` command should fail with an appropriate error message if no `setdir` command has succeeded

```

Client view:
>>> [Welcome, You are registered.]
>>> setdir fakedir
>>> [setdir failed: fakedir does not exist.]
>>> offer should_fail.txt
>>> [Please set a directory first. Usage: setdir <dir>.]
>>> offer i_love_the_tas.txt
>>> [Offer Message received by Server.]
>>> [Client table updated.]

```

`offer` works for single and multiple (e.g. 3) filename argument(s).

```

>>> offer hello.txt jjs.jpg i_love_the_tas.txt
>>> [Offer Message received by Server.]
>>> [Client table updated.]

```

Server broadcasts on new offered files by client(s) work.

Testing scenario:

- start 3 clients, and server.
- After client 1 successfully offers a file to the server, client 2 & 3 outputs appropriate status messages indicating they have received the new table from Server and updated their local table.

Starting 3 clients and server

```
# server starts
$ python3 ChatApp.py -s 1025
=====
Printing args:
server True
client False
port 1025
=====
...
```

```
# client 1 registered successfully
$ python3 ChatApp.py -c hey 0.0.0.0 1025 1029 1030
=====
Printing args:
server False
client True
name hey
server-ip 0.0.0.0
server-port 1025
client-udp-port 1029
client-tcp-port 1030
=====
>>> [Welcome, You are registered.]
...

# client 2 registered successfully
$ python3 ChatApp.py -c waa 0.0.0.0 1025 1028 1031
=====
Printing args:
server False
client True
name waa
server-ip 0.0.0.0
server-port 1025
client-udp-port 1028
client-tcp-port 1031
=====
>>> [Welcome, You are registered.]
...

# client 3 registered successfully
$ python3 ChatApp.py -c client3 0.0.0.0 1025 1032 1033
=====
Printing args:
server False
client True
name client3
server-ip 0.0.0.0
server-port 1025
client-udp-port 1032
client-tcp-port 1033
```

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

- client 1 (hey) offers a file to the server and the other two clients output appropriate status messages indicating they have received the new table from the server and updated their local table.
 - Note that the client offering the file also gets the new table broadcasted to it.
 - The `DEBUG` prints are commented out for submission.

```
# client 1
>>> offer jjs.jpg wee.txt
>>> [Offer Message received by Server.]
>>> >>> [Client table updated.]
[DEBUG] message: {"jjs.jpg,hey": ["127.0.0.1", 1030], "wee.txt,hey": ["127.0.0.1", 1030]}
```

```
# server
...
>>> [Offer Message Received By Server]
```

```
# client 2 and three have the same output, with
# optional DEBUG messages uncommented
...
>>> >>> [Client table updated.]
[DEBUG] message: {"jjs.jpg,hey": ["127.0.0.1", 1030], "wee.txt,hey": ["127.0.0.1", 1030]}
```

File Listing Tests

- the following tests are a continuation of the server + 3-clients session from the previous test:

Listing the correct file offerings using the table (with proper formatting)

- on any one of the client terminals, after `client3` has also offered `jjs.jpg`

```
...
>>> list
FILENAME  OWNER   IP ADDRESS  TCP PORT
jjs.jpg   hey     127.0.0.1   1030
jjs.jpg   client3 127.0.0.1   1033
wee.txt   hey     127.0.0.1   1030
```

Proper message when no files are being offered

- scenario: 1 client 1 server, client `list`s right after a successful registration

```
>>> [Welcome, You are registered.]
>>> list
>>> [No files available for download at the moment.]
```

File updated when client table updated

- client 1 offers three files and lists them
 - note that the prompt messages get messed up slightly when the client threads are both waiting for user input and receiving updates from the server's broadcasts.

```
$ python3 ChatApp.py -c hey 0.0.0.0 1025 1029 1030
=====
Printing args:
server False
client True
name hey
server-ip 0.0.0.0
server-port 1025
client-udp-port 1029
client-tcp-port 1030
=====
>>> [Welcome, You are registered.]
>>> [Client table updated.]
>>> setdir dir
>>> [Successfully set dir as the directory for searching offered files.]
>>> offer jjs.jpg jjs.jpg 1.txt 2.txt
>>> [Offer Message received by Server.]
>>> >>> [Client table updated.]
list
  FILENAME  OWNER  IP ADDRESS  TCP PORT
1.txt      hey   127.0.0.1   1030
2.txt      hey   127.0.0.1   1030
jjs.jpg    hey   127.0.0.1   1030
>>> >>> [Client table updated.]
```

- client 2 joins and lists the files

```
$ python3 ChatApp.py -c waa 0.0.0.0 1025 1028 1031
=====
Printing args:
server False
client True
name waa
server-ip 0.0.0.0
server-port 1025
client-udp-port 1028
client-tcp-port 1031
=====
>>> [Welcome, You are registered.]
>>> [Client table updated.]
>>> list
  FILENAME  OWNER  IP ADDRESS  TCP PORT
1.txt      hey   127.0.0.1   1030
2.txt      hey   127.0.0.1   1030
jjs.jpg    hey   127.0.0.1   1030
```

- client 2 offers some more files

```
>>> setdir 3.txt
>>> [setdir failed: 3.txt does not exist.]
>>> setdir dir
>>> [Successfully set dir as the directory for searching offered files.]
```

```
>>> offer 3.txt 4.txt 5.txt hello.txt jjs.jpg wee.txt
>>> [Offer Message received by Server.]
>>> >>> [Client table updated.]
```

- both client 1 and 2 list the files

```
# client 2: waa
list
FILENAME  OWNER  IP ADDRESS  TCP PORT
1.txt     hey   127.0.0.1   1030
2.txt     hey   127.0.0.1   1030
3.txt     waa   127.0.0.1   1031
4.txt     waa   127.0.0.1   1031
5.txt     waa   127.0.0.1   1031
hello.txt waa   127.0.0.1   1031
jjs.jpg   hey   127.0.0.1   1030
jjs.jpg   waa   127.0.0.1   1031
wee.txt   waa   127.0.0.1   1031

# client 1: hey
>>> [Client table updated.]
>>> list
FILENAME  OWNER  IP ADDRESS  TCP PORT
1.txt     hey   127.0.0.1   1030
2.txt     hey   127.0.0.1   1030
3.txt     waa   127.0.0.1   1031
4.txt     waa   127.0.0.1   1031
5.txt     waa   127.0.0.1   1031
hello.txt waa   127.0.0.1   1031
i_love_the_tas.txt waa 127.0.0.1 1031
jjs.jpg   hey   127.0.0.1   1030
jjs.jpg   waa   127.0.0.1   1031
wee.txt   waa   127.0.0.1   1031
```

File Transfer Tests

Client can successfully request and receive a file offered by another client. The received file content should be exactly the same as that of the offered file of the host.

- from the previous session with 2 clients and a lot of files:
- client `waa` requests `1.txt` from `hey`

```
# client waa terminal
>>> request 1.txt hey
< Connection with client hey established. >
< Downloading 1.txt... >
< 1.txt downloaded successfully! >
< Connection with client hey closed. >
```

```
# client hey terminal
>>>
< Accepting connection request from 127.0.0.1 >
< Transferring 1.txt... >
< 1.txt transferred successfully! >
```

```
< Connection with client waa closed. >
...
```

- 1.txt now appears in the working directory of ChatApp.py

Appropriate status messages should be printed at critical points of the file transfer, similar to the example provided in the specification.

- See above status messages

An appropriate error message should be printed when the client tries to request a non-existent file or a file from an incorrect client

- client `waa` requesting a non-existent file

```
>>> request this_doesnt_exist hey
< Invalid Request >
```

- from an incorrect client

```
>>> request 1.txt wee
< Invalid Request >
```

De-Registration Tests

dereg command de-registers the client without exiting the client program.

- waa deregisters

```
>>> dereg waa
# No output printed.
```

When a client dereg'ed after offering file(s), the server should broadcast the updated table of offered files to other active clients. Upon receiving the broadcasted table, active clients should output appropriate status messages to indicate that its local table has been updated.

- continued from above test
- client `hey` terminal

```
>>> [Client table updated.]
list
FILENAME  OWNER  IP ADDRESS  TCP PORT
1.txt     hey    127.0.0.1   1030
2.txt     hey    127.0.0.1   1030
jjs.jpg   hey    127.0.0.1   1030
```

 **thanks!**