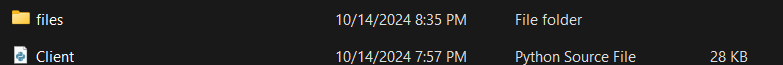
CMPSC 473 Lab – P2P Distribution System – Kevin O’Connor

<https://github.com/KevinConanOConnor/PeerToPeerFileSharing>

The files ‘Client.py’ and ‘Server.py’ implement a peer-to-peer file distribution system. Each client uses a files subdirectory present in the same as it as a source of files to share and download to.



The client module should only be run using ‘Python client.py’ after the server/tracker module has first been run/started using ‘Python server.py’. Upon running the client, if the connection to the server is established successfully, the process will print the sharable files of the client (these would be the files present in files folder). This will be followed by a list possible user commands.A computer screen with white text

Description automatically generated

From here I will show a sample interaction demonstrating a client sharing their file with another directly over a peer-to-peer connection:

A screenshot of a computer

Description automatically generated

First off, we will start with two clients. As we can see client1 has 2 files in its files folder while client 2 has an empty files folder. Client 1 will connect and register a file for sharing:A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

Here I will point out one of the issues that occurred, handling socket connections and user input and output were handled on separate threads. As a result, their prints are oftentimes intermixed. Anyway, now allegedly our file should be successfully registered on the server. Looking at the server’s log…

A screen shot of a computer code

Description automatically generated

We can see that the file registration was successfully received. Now opening up a second client and requesting the server’s file list…

A black background with white text

Description automatically generated

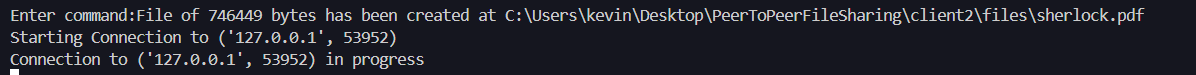
A screen shot of a computer

Description automatically generated

We can see that sherlock.pdf is in the list of files that we should be able to download. Seeing this, the user on client2 decides to initiate the download of the file.

A black background with white text

Description automatically generated



Looking at the log, we are able to see we received the listening address of an owner of that file from the server, and that we have initiated a connection with them.

A screen shot of a computer

Description automatically generated

Typing progress after the initiation of this message, we can see that the download of sherlock.pdf has been completed!

A black background with white text

Description automatically generated

As can be seen, client2’s files folder is no longer empty…

A screenshot of a computer screen

Description automatically generated

And from the server’s logs, we now have 2 peers who are currently sharing sherlock.pdf! . The steps for a third peer to initiate the download would be the same, yet now they would have 2 peers to receive files from.

Now let’s take a look at the core details of actual exchange of messages that occurs behind the scenes to facilitate this outcome.

**Key Protocol Details:**

**FILEREG -** (Client -> Server) – Client registers possession of all chunks of a file with the server. Done in response to a user input

Passed Data: File name, file size, chunk count, and client's listening socket address.

**FILEREGREPLY -** (Server -> Client) – Server informs client where registration was successful or not

Passed Data: File name, success status

**FILELISTREQ -** (Client -> Server) – Client asks server for a list of files it can download

Passed Data: N/A

**FILELISTREPLY -** (Server -> Client) – Server returns a list of filenames that peers are sharing

Passed Data: list of filenames

**FILELOCREQ -** (Client -> Server) – This is initiated when a client downloads a file. The client will ask the server for a list of peers and what chunks they have as well as their listening sockets so that the client can open connections with them and handle exchanging files with them directly.

Passed Data: filename

**FILELOCREPLY -** (Server -> Client) – Server returns a list of users, what chunks of the file they hold, and their listening sockets.

Passed Data: users, chunks, listening sockets.

**CHUNKREQ -** (Peer -> Peer) – After connecting with a peer and comparing the list of chunks he needs against the list of chunks his peer has. A peer looking to download from the other will ask for a random chunk from the intersections of these sets.

Passed Data: filename, chunks

**CHUNKSEND -** (Peer -> Peer) – After receiving a request for a chunk, the peer will package it to bytes and send it. Making sure to include a hash result that the peer on the other side can evaluate the bits they have received to determine the chunks are the same.

Passed Data: filename, chunk index, data, hash

**CHUNKREG -** (Client -> Server) – After receiving a chunk, the client lets the server know he is in possession of that chunk and will be able to share that chunk with others

Passed Data: filename, chunk index

**CHUNKREGACK -** (Server -> Client) – Server Acknowledges that the client was registered as being in possession of the chunk

Passed Data: filename, chunk index

Top-to-bottom Code Structure

While I felt my code was unfortunately pretty disorganized, successive categories do generally build upon and utilize features implemented before them in a top-to-bottom manner.

Server Code: Client Code:

Global Info Global Info

Sending/Receiving Messages Hashing Code

Message Protocol Handling Local File Operations (read/write/etc.)

Connection Handling Connection Management

Main Event Loop Sending/Receiving Messages

Input Handling

Message Protocol Handling

Connection Handling

Main Event Loop

**Implementation Specifics:**

As you will have noticed, in this scenario the server never has any possession of the chunks or files themselves. It merely has a list of socket connections with clients associated with a selector and some associated state. For each file it stores at minimum a list of users, what chunks they have, and their associated listening socket. On the other hand, the client’s state mostly consists of a list of files they are currently downloading or have downloaded during this session, keeping note of what chunks they have received and which ones they still need. The client also keeps a list of socket connections attached to a selector along with some associated state.

The socket connections themselves are handled on a separate thread and their own event loops from the user input. Both the client and server operate in the same way, looping through connections and essentially taking care of business based on the state of the connection. For example, if a socket’s output buffer has stuff to send out, it will send it. If a socket can receive it will receive. Additionally when peer to peer connections are not waiting on a chunk from another, they will handle requesting chunks from as part of this loop.

**Implementation Issues:**

Firstly, the most obvious issue that is difficult to handle for any distributed system is that there are so many possible edge cases and sequences of events that must be considered as possibilities. For example, what if a client is downloading something while the last holder of the file disconnects? While the system tries to answer some edge cases, it is mostly only capable of providing basic peer-to-peer data transfer capabilities.

Another issue is with the sharing of directories. While the system provides this capability, sometimes the registering of multiple files has issues with blocking. It is handled on the same thread as input and output, and it takes waiting for the next user input for the following files to be registered with the server.

Speaking of threads, the input and output runs on a separate thread from the code that handles socket interactions. While this decision was made to avoid user input blocking listening and handling socket interactions, this means that outputs of the two threads oftentimes are intermixed.

Another acknowledgement I must make is that my code is disorganized and terribly standardized. Ultimately, I encountered many obstacles and additional details that I hadn’t thought of when I first thought about how I would design my project. As a result, I was oftentimes hopping around changing details of the implementation and the code lacks somewhat lack a coherent structure as a result.

Demo:

Run\_demo.bat launches the server as well as client1, client2, and client3. Client1 should be the only one with any files in it’s shareable file folder. Since the commands are run through Python’s REPL, this sets the system up for a demo.

Clean\_2\_and\_3.bat simply clears client2 and client3’s folders to restore the demo to the start state for ease of use.

I chose to use a windows batch file here because I used windows and makefile was acting up.