RYERSON UNIVERSITY

Faculty of Engineering & Architectural Science

Department of Electrical and Computer Engineering

Course Number	COE 768
Course Title	Computer Networks
Semester/Year	Summer 2020

Instructor	Dr. Baha Uddin Kazi

Assignment/Lab Number:	**Group Project**
Assignment Title	P2P Application

Submission Date	August 9, 2020
Due Date	August 9, 2020

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Ho	Brandon	500727531	01	BH
Austin	Ryan	013249610	01	RA

**By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: www.ryerson.ca/senate/current/pol60.pdf.*

Contents

Introduction

What the Project is About

The objective of this project is to develop and implement the code for a basic peer-to-peer (P2P) file sharing application. The basic idea of a P2P file-sharing network is that many computers come together and pool their resources to form a content distribution system. The computers are called peers because each one can alternatively act as a client to another peer, fetching its content, and as a server, providing content to other peers [1]. P2P technology was used by popular services such as Napster and LimeWire. The most popular protocol for P2P sharing is BitTorrent [2].

For this project, a client host (i.e. representing a user) will query a server that provides an index of file names/addresses. If the file information is stored with the server, the server will send the information to the client. The client will then request the file from the content server that has the actual file. If the file download is successful, the client host will register its own address with the index server as another content server that could provide the file. Any content server is potentially a client and any client could potentially become a content server. The non-index server hosts are thus network peers.

The program will be implemented in Python because of its relative ease of implementation for network programming. It is simple with an easily readable syntax. It is very efficient, requiring fewer lines of code to complete more work. It has multiple libraries and frameworks. It has community and corporate support [3]. IEEE recently ranked Python the number one programming language overall for 2019, citing strengths like “vast number of specialized libraries available for it” [4].

Background Information on Socket Programming

The concept of the socket is fundamental to network programming. Processes that want to communicate with one another need an interface. Transmission is governed by factors such as the address domain (e.g. IPv4), whether the data is streamed and requires a formal connection or segmented and connectionless, and the transport protocol. The specific IP address and port number of the receiving host must be indicated. All that information is accounted for by the socket application programming interface (API).

Analogous to its electrical hardware namesake (i.e. female connector), a network socket is a communication endpoint. It controls and organizes the communication traffic that a node could receive. Structurally it is a file descriptor and returns an integer, but it has instructions for interfacing with a node rather than working with a file. It stores information with respect to the protocol family (e.g. IPv4), the form of the sent data (continuous or segmented), and the governing transport protocol.

Socket programming in Python is particularly convenient because it defaults the most common values for network parameters. Function arguments to create a socket do not need to be explicitly entered when using IPv4 for address family, connected/stream for connection type, or TCP for transport protocol, respectively. With other languages, such as C, that information would have to be explicitly entered whenever a socket object is created.

Like other languages, Python provides access to the BSD socket interface through a *socket module*. The *socket()* function call returns a socket object whose functions implement the various socket system calls. One could bind a socket to an IP address and port number. Through the socket, a node could listen for connection requests, accept them, and make the connection. A node could send and receive data

through the socket, but sockets only pass binary data so data in the form of an object would have to be serialized for transmission and then de-serialized upon reaching its destination. A node can signal the end of a connection by calling the *close()* function on the socket.

Description of the Client and Server Programs

Basic Approach to Implement the Protocol

To implement the protocol, the index server should be active first, to be available when a client wants to interact with it. Next, a peer with a file that other peers may want to download should be activated and have its file registered with the index server. After the file is registered it becomes accessible for download. Client peers access the file by querying the index server for the list of online registered files. The client user is then prompted for which file to download. If the file is registered with the index server, the index server arranges for the content server to send the file to the client, provided the file is available. The client peer then registers its possession of the file with the index server and thus becomes a content server itself. Future client peers searching for the file would be directed to the latest peer to register the file. When a content server wants to de-register a file, it would do so by contacting the index server.

Essentially, packets of information, protocol data units (PDUs), are being passed back and forth between hosts. Each PDU consists of a 100-byte payload portion, where data is stored, and a one-byte label portion, where the type of data being transmitted is identified by a letter. The payload portion may in turn have multiple subcomponents for multiple chunks of information. In this project, the PDU will be represented by a *namedtuple* object. In a language like C, the ideal data structure would be a *struct*.

To transmit and receive PDUs, socket objects must be created and bound to the relevant host address. For secure, reliable transmission (i.e. TCP protocol), a formal connection must be made between the hosts via their sockets. For insecure/less reliable transmission (i.e. UDP protocol), a formal connection is not made. Objects like namedtuples cannot be transmitted through a socket as a socket will only accept binary data. The PDUs therefore must be serialized into byte form for transmission, then de-serialized back into object form by the receiver for interpretation. The PDUs were sent by invoking *s.send()* and received with *s.recv()* (The “s” prefix represents a connected/bound socket in this project. See “Detailed Description” in next section). They were serialized with *pickle.dumps()* and de-serialized by with *pickle.loads().*

When a PDU is received, the receiving host must make decisions based largely on they label of the PDU and/or the presence or not of data. A table of instructions based on the PDU label is given below.

Table 1: Host instructions based on PDU label.

PDU Type	Directive/Message	Responsibility (Receiving Host)
R	Register the content identified in payload	Index Server
D	Download the requested content	Content Server
S	Search content server	Index Server
T	De-register the content	Index Server
C	Write the content data	Client
O	List on-line registration content	Index Server
N/A	Send acknowledgement	Index Server
N/A	Send error message	Index Server
A	Acknowledgement (message)	Client/Content Server
E	Error	Client/Content Server

Received bytes are read or written (as applicable) continuously in a while-loop. Once there is no more data, or in some cases when a timeout exception is triggered, the reading/writing operation stops. If no data is received at all when it is expected, an exception is triggered, and an error message is returned to the sender. Communication between a peer and the index server is via the UDP protocol. It is connectionless and provides no guarantee of successful or error-free transmission, but it is faster. The trade-off is considered acceptable given that data between the server and peers consist mainly of messages, and not files. Transmission between peers involves file transfer, so the more reliable TCP protocol, which is connection-oriented and checks for errors, was used for peer-to-peer traffic.

Various specialized functions were called to implement the protocol. To access those functions, several external modules/libraries had to be imported. The *socket* module was discussed above. The *namedtuple* module from the *collections* module allows one to create a multi-field data structure with meaning assigned to each field. The *os.path* module allows one to implement some useful functions on pathnames (such as finding/accessing them). The *pickle* module implements an algorithm for serializing and de-serializing a Python object structure. The use of *pickle.dumps()* converts a Python object to a byte stream (for transmission through a socket, which only passes byte streams). The use of *pickle.loads()* reverses the process, returning the byte stream into a Python object. The *time* module provides various time-related functions, such as *time.sleep().* The *select* module gives access to *select()* and *poll()* functions, such as *select.select(socket_list, outputs, exp)*. The *threading.Thread* and *_thread* modules provide low-level primitives for working with multiple threads (also known as “light-weight processes”, “tasks”) [5].

Detailed Description of the Index Server Program

The index server is responsible for maintaining the list of online registration content, managing the registering and de-registering of files, and directing search requests to the appropriate content server. Besides the *socket* module, the server needs the *time*, *pickle*, *namedtuple* (from *collections*), and the *_thread* modules.

The index server code starts with preliminary definitions such as defining the PDU and the files list as types of namedtuple object, declaring an empty (initially) array for the server file registration list, setting the port number and host ID, and creating a UDP socket object. The program also introduces a thread

count variable, setting it to zero. A key functionality of the server in this project is its ability to handle multiple client requests/processes simultaneously.

The index server's operation begins by binding the socket object to the host's IP address and port number. If an error occurs, an exception is triggered. After binding the socket, the server indicates that it is waiting for a connection and starts listening for potential clients (up to five in the buffer). If a connection request is accepted and a connection is made, the server will print out the IP address and port number of the host that it is connected to. It will call the "start_new_thread" function, which takes the "client_thread" function and the accepted connection "object" as arguments. The thread count will increase by one and there will be a printout stating: "Connected Peer Number: [thread count number]". When the "client_thread" module completes its process, the connection will close, then the socket will close. Each client will go through the same process with the index server.

The client_thread module defines how a packet is processed once connection is established. The packet is received in binary form then de-serialized back into a PDU namedtuple object. A message is printed acknowledging receipt of the specific PDU from a peer. The label of the PDU is checked for PDU type. The type could be "R", "T", "S", or "O".

If the type is "R", then the peer name, filename, and peer address are extracted and stored in variables. A "file_exists" variable is declared and set to "false" (to assume that the file to be registered does not already exist in the registry). Every index in the server file registration list is matched against the connected peer name–file name combination. If there is a match, then the "file_exists" variable is set to "true" and a PDU with an error message is sent back to the client telling them to pick another username to register that file. If the file does not exist, then the peer name, filename, and peer address are added to the server file registration list and an acknowledgement PDU is sent back to the client.

If the type is "T", then the peer name, filename, and peer address are also extracted and stored in variables. A "loop_count" variable is declared and set to zero. Every index in the server file registration list is matched against the connected peer name–file name combination. If there is a match, then the matching entry is removed from the server file registration list, and acknowledgement PDU is sent to the client, and a message stating that the file has been removed is printed out. When there are no more entries in the server file registration list (i.e. the length of the list equals loop count), then an error PDU is sent to the client stating that the file does not exist.

If the type is "S", then the file name is extracted and stored in a variable. A "found_content" variable is declared and defaulted to "false". A "target_address" variable is declared and set to null. Every index in the server file registration is matched *in reverse order* against the file name. If there is a match, then the "found_content" variable is set to "true" and the address of the matched file name is stored in the "target_address" variable. An S-type acknowledgement PDU is sent to the client and a message stating that the PDU is being sent is printed. If the content is not found, then a PDU with an error message is sent to the client.

If the type is "O", then the server first checks to see if there are any entries in the server file registration list. If there are no entries, then an O-type PDU is returned to the client with message: "No User, List Empty". A message stating that such a PDU is being sent to the peer is printed. If there are entries in the list, then the peer name, filename, and peer address of every entry is sent to the client in

an O-type PDU. The messages are sent 0.1 seconds apart to prevent the pickling errors that could occur when multiple messages sent in quick succession and then received in a single *.recv()* function.

Detailed Description of the Client Program

The P2P client sends connection requests to the index server. It is responsible for sending requests to the index server to register or de-register a file. It is responsible for requesting the list of online registered content files and choosing which file it wants to download. It is responsible for requesting the list of local registered content files. Finally, it is responsible for quitting an online session (after deregistering its files). Besides the *socket* module, the client program needs the *time*, *os.path*, *pickle*, *select*, and the *namedtuple* (from *collections*) and *Thread* (from *threading)* modules. The P2P client may also function as a content server. In order to accommodate concurrent operation as client and server, the client and server functions were implemented as separate threads.

Like the server code, the P2P client code also starts with preliminary definitions and declarations. A UDP socket is created. The port number and host ID are set. A socket timeout is set at one second. The socket is connected to the index server host ID and port number. Finally, a PDU is created as a namedtuple object (with the same structure as in the code for the index server).

The code for the P2P client is structured with function definitions at the top: *select_name()*, *de_register()*, *register()*, *download_file()*. The actual operation begins with a prompt for the user to input a username and to choose a listening port number for the P2P server. The client then starts both client and server function threads and executes its *client_function()* function.

The *client_function()* function takes no arguments. It introduces a file index list as an empty array and declares a termination flag variable set to “zero”. The function will execute its processes if the termination flag remains “zero”. When the user decides to quit (by selecting “Q” when prompted), the *de_register()* function will be automatically called . The user’s files will be de-registered, the local file registration list will be cleared, the socket will close, and the termination flag will be set to “one”, terminating the session. The function begins by prompting the user to choose a client function from the list: “O” (get online list); “L” (list local files); “R” (register a file); “T” (de-register a file); or “Q” (quit the program).

If the user selects “O”, then an “is_list_empty_flag” is declared and set to “zero” and an O-type PDU is sent to the index server. The responding PDU from the index server is downloaded and de-serialized until a timeout exception is triggered, signalling download completion. If the received PDU is Type-O, then its payload is checked. If the payload is a message saying the that the registration list is empty, then the “is_list_empty_flag” is set to “one” and the user is again prompted to indicate which client function it wants executed. If there are entries in the list, then the peer name and file name of each entry is displayed, and the user is prompted for which file to download. An S-type PDU with the filename request as the payload is then created and sent to the index server. As before, the server’s response is downloaded and serialized until a timeout exception is triggered. If the responding PDU is also Type-S, then the address of the peer where the file is stored is extracted from the payload and the *download_file()* module is called, with the peer address, filename request, and file index list as arguments. If the response from the index server was a Type-E PDU, then the error message from the payload is displayed.

If the user selects “L”, then the *os.listdir()* function is called with the folder containing the files as an argument, and the files are printed out.

If the user selects “R”, then the user is prompted for the name of the file to register and then the *register()* function is called with the username, filename index request, peer port number, and file index list as arguments.

If the user selects “T”, then the user is prompted for the name of the file to de-register. If the file exists, the *de_register()* function is called with the same arguments as the *register()* function, except the filename index request parameter is replaced with a filename index delete request parameter. If the file does not exist or is not associated with the user, a message stating such is printed out.

A P2P client can also function as a content server if it has registered a file with the index server. Like the *client_function()* function, the *server_function()* function takes no arguments. Preliminary declarations include the creation of a TCP socket for communication between peers, a socket timeout set to one second, and empty arrays for outputs, exceptions, and the sockets of peers requesting a connection.

The *server_function()* begins by binding its socket to the local host and peer port and by listening for potential clients (up to five can queue). Ready sockets are added to the list of sockets. If there was a TCP binding error, an exception is triggered, a “Failed to connect” message is printed, and the flagged socket is added to the exceptions list.

The content server will listen for available sockets which are ready through the *select()* function. If any have a file descriptor of “-1”, they are invalid—considered terminated—and will be removed from the list. The content server will check the socket list for sockets to establish a connection. When a new connection is made with a peer, the peer gets added to the socket list. If there are no waiting sockets, the content server will receive a packet from a peer, de-serialize it, and verify that it is Type-D. If there is no data, a timeout exception will be triggered.

If it is Type-D, the content server will check the directory for the file. If the file is there, it will be opened, read, and sent to the client peer that requested it in a PDU of Type-C. The file would then be closed and a notification stating that transmission is complete, and the connection will be closed, will be printed. As with the client function, messages would be sent with time delays of 0.1 seconds to prevent pickling errors.

If the content server cannot find the file, an error PDU (i.e. E-type) with a “File does not exist” message would be sent back to the client peer and the connection would be closed.

If the item in the socket list is not incoming data, it would be removed from the socket list.

The P2P client implements three functions: file registration (*register()*), file download (*file_download()*), and file de-registration (*de_register()*). The file registration function takes four arguments: username, filename, peer port, and file index. When the function is called, an R-type PDU is created and sent to the index server. The response packet from the index server is downloaded and de-serialized until there is no more data. The PDU type is then checked. If it is Type-A (an acknowledgement PDU), the filename is added to the file index and the payload’s contents is printed out. If the response

was a Type-E PDU, the error message is printed out and the user is prompted for a different username to send with a new file registration request. The new R-type PDU is prepared and sent.

The function to download a file takes the filename, the address (as an array) of the peer that has it registered, and the file index has arguments. When called, the IP portion of the address gets stored in the first cell and the port number portion gets stored in the second cell. A new TCP connection is established between the involved peers. A new socket is created at the destination, a socket timeout is set to two seconds, and the connection is made using the IP and port number. A D-type PDU with the filename as payload is created, serialized, and sent to the content server.

The response received from the content server is written to a newly opened file and de-serialized. If the PDU type is "C", the *register()* function is called to register the file. A message stating that the file was received is printed. If the response is Type-E, then the error message in the payload is printed and the fact that the connection will close is printed. The file is then closed, and the new socket is closed.

The function to de-register a file takes the same arguments as the one to register a file: username, filename, peer port, and file index. A T-type PDU is prepared with username, filename, and peer address as the payload. It is serialized and sent to the index server. The response from the server is de-serialized and its label is checked. If it is Type-A (an acknowledgement PDU), then the file is removed with the *file_index.remove()* function and the message from the Type-A PDU is printed. If the response is a Type-E PDU, then the message from its payload is printed and the user is prompted for the filename to be de-registered. Another Type-T PDU is prepared and sent to the index server. The process will continue until a filename is successfully de-registered.

Observations and Analysis

The following are tests of basic functionality.

Test 1: List Local Files Versus Display Online Registry

This is a reasonable preliminary test to show the difference between a file existing and a file being registered. The first command is expected to succeed. The second command is expected to fail.

Figure 1: List of local files versus online registry when no files have registered.

```
Please enter preferred username:Leonardo
Please enter listening port number for the P2P server:6010

Please choose from the list below:
[O] Get online list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program

L
file1.txt
file2.txt
file3.txt
file4.txt

Please choose from the list below:
[O] Get online list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program

O
List is Empty---Nothing to download
```

The test succeeded. The local files exist and can be displayed, but there are no files registered so the online registry cannot be displayed.

Test 2: Have All Files Registered and Display Registry

With entries in the registry, a call to view the registry or download files should be successful. The test is expected to succeed.

Figure 2: Online registry with all files registered.

```
R
What file do you want to register with the Index Server?
file4.txt
Successfully Registered

Please choose from the list below:
[O] Get online list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program

O
User: Leonardo file: file1.txt
User: Donatello file: file2.txt
User: Raphael file: file3.txt
User: Michaelangelo file: file4.txt

List Download Complete
What file do you want to download?
```

The test was successful.

Test 3: Try to Register a File Already Registered to Same Username

The program is designed to only allow one username to register a file. The attempt should fail, and an error message should be triggered.

Figure 3: Registering a file already registered to the same username.

```
Please enter preferred username:Leonardo
Please enter listening port number for the P2P server:6050

Please choose from the list below:
[O] Get online list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program

R
What file do you want to register with the Index Server?
file1.txt
File associated with another User --> Pick another user name
Please enter preferred username:
```

The test was successful.

Test 4: Attempt to Download Files with Duplicated and Unique Username

The code was designed to block peers with the same username from downloading the same file but allow peers with a different username to download and register the same file. The test should be successful.

Figure 4 Downloading of file with duplicated username and unique username.

```
Please choose from the list below:
[O] Get online list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program

O
User: Leonardo file: file1.txt
User: Donatello file: file2.txt
User: Raphael file: file3.txt
User: Michaelangelo file: file4.txt
User: Dante file: file1.txt
User: Leonardo file: file2.txt
User: Leonardo file: file3.txt
User: Leonardo file: file4.txt
```

The test was successful. A second user named Leonardo had to change his name (to Dante) to download file1.txt, which was already registered to the first Leonardo. After registering file1.txt under the changed name, the other files were registered under the *original name*, with *no user-input reversion to the original name*.

Test 5: De-Register Files with Original Username

The second Leonardo should be able to de-register his files. However, his registration of file1.txt was under the name “Dante”. This test will see whether the user at port 6050 (Leonardo/Dante) can still access his registration under the temporary name “Dante”.

Figure 5a: Deregistration of files under original username.

```
0
User: Leonardo file: file1.txt
User: Donatello file: file2.txt
User: Raphael file: file3.txt
User: Michaelangelo file: file4.txt
User: Dante file: file1.txt
```

De-registration of the files under the name “Leonardo” was successful.

Figure 5b: Deregistration of file registered under temporary (and inaccessible) username.

```
T
What file do you want to de-register with the Index Server?
file1.txt
Timeout-No more data
Exception in thread Thread-1:
Traceback (most recent call last):
  File "C:\Users\User\AppData\Local\Programs\Python\Python38-32\lib\threading.py", line 932, in _bootstrap_inner
    self.run()
  File "C:\Users\User\AppData\Local\Programs\Python\Python38-32\lib\threading.py", line 870, in run
    self._target(*self._args, **self._kwargs)
  File "C:/Users/User/PycharmProjects/P2PApplication/P2Ppeer5.py", line 306, in client_function
    de_register(the_user_name, filename_index_delete_request, the_peer_port, file_index_list)
  File "C:/Users/User/PycharmProjects/P2PApplication/P2Ppeer5.py", line 53, in de_register
    if deregister_response_packet_pdu.data_type == "A":                                    # if response data_type is 'A', done
AttributeError: 'NoneType' object has no attribute 'data_type'
```

An exception was triggered when attempting to de-register file1.txt from the second Leonardo account. The name is registered under “Dante”, who was a temporary fiction. Dante remains in the registry for file1.txt, apparently unreachable with the current code.

Test 6: Quitting the Program

Quitting the program automatically executes the de-register file(s) function beforehand.

Figure 6a: Demonstration of the *quit()* functionality.

```
Please choose from the list below:
[O] Get online list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program

Q
Successfully De-Registered
Client Session Terminated
```

In this scenario, all users except the second Leonardo username have quit. The files of other users are no longer registered after they quit the program. The second Leonardo has been frozen because he tried to de-register a file listed under a temporary name. Trying to de-register a file under a username that is not the one the file is registered would normally trigger an error stating that only usernames registered to a file could de-register that file. The error in this case is different.

Figure 6b: Demonstration that files registered under a temp name are inaccessible by the present code.

```
Please choose from the list below:
[O] Get online list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program

O
User: Dante file: file1.txt

List Download Complete
What file do you want to download?
```

Conclusions

The specs were largely met, but not without challenges or room for improvement. The objective of this project was to develop and implement a peer-to-peer file-sharing application. The basic functionalities of client-to-index server interaction and peer-to-peer interaction were achieved. A client can register and de-register files with the server. A client can request to see an online registry and download registered files from a peer ("content server"). clients can request to see the local file directory. Peers can quit the application and have their registered files automatically downloaded. A peer can run client and server processes simultaneously ("multithreading"), a functionality that was particularly challenging to incorporate.

Despite the generally positive results, the tests also revealed a bug. According to specs, if a client tried to register or download a file that was already registered to a peer with the same username, the client was supposed to be blocked and prompted to change their username. That functionality worked. However, a username change was not permanent. Files without a name conflict that were subsequently registered by the peer were done so under the *original* username. The system no longer recognized the substitute username as belonging to the peer, so the file registered under that substitute username could not be accessed to be deregistered. By contrast, the files registered under the original username could be deregistered. In summary, the system has implementation problems because it was designed to manage files based on username, but it was not designed to keep consistent track of mandatory username changes. The file registry kept a record of the substitute username, but the "host registry" did not. One might alter the code to have the server register the file according to host address, but that would change the original specs, which were based on username matching. It is a problem whose solution is arguably beyond the scope of this project.

References

1. Tanenbaum, A. S., and D. J. Wetherall. *Computer Networks, 5th ed.* Prentice Hall: Toronto, © 2010, p. 748.
2. Wikipedia. "Peer-to-Peer". Retrieved August 4 2020, from https://en.wikipedia.org/wiki/Peer-to-peer.
3. GeeksforGeeks. "Why is Python the Best-Suited Programming Language for Machine Learning?" 27-08-2019. Retrieved Aug. 7 2020, from https://www.geeksforgeeks.org/why-is-python-the-best-suited-programming-language-for-machine-learning/.
4. Cass, Steven. "The Top Programming Languages 2019". IEEE Spectrum. Sept. 6, 2019. Retrieved August 6 2020, from https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019.
5. Python. Python Module Index. Retrieved August 9, 2020, from https://docs.python.org/2/py-modindex.html.

APPENDICES

P2Pserver Source Code

```
# P2Pserver.py
# Brandon Ho 500727531
# Ryan Austin 013249610

import socket
import time
import pickle
from collections import namedtuple
from _thread import*

ThreadCount = 0
PDU = namedtuple('PDU', ['data_type', 'data'])                    # create namedtuple() object PDU with four fields data_type and data
Files_List = namedtuple('Files_List', ['peer_name', 'file_name', 'address'])
Server_File_Registration_List = []                                # server list---> fill with files to be register
```

```
port = 60000                                    # Server Port for index server
host = "localhost"                              # Get local machine name
s = socket.socket(socket.SOCK_DGRAM)            # Create UDP socket object


def client_thread(connection):      # <-------- client operation

   while True:
      packet = connection.recv(200)                 # wait to receive data from client (Blocking socket)
      if not packet:
         break
      peer_pdu = pickle.loads(packet)               # decode serialized packet using pickle() function to PDU namedtuple() object
      print('Received PDU<---' + str(peer_pdu) + '<--- from Peer')

      if peer_pdu.data_type == 'R':                 # check peer_PDU for "R" --> Registration Type and set peer_name, file_name
and address to temp variables

         my_peer_name = peer_pdu.data.get('peer_name')
         my_file_name = peer_pdu.data.get('file_name')
         my_peer_address = peer_pdu.data.get('address')
```

```
        file_exists = False                                     # check if peer's file exist on server--> flag set to default False before check

        for i in Server_File_Registration_List:                 # check server for existence of peer's file

            if my_peer_name == i.peer_name and my_file_name == i.file_name:          # check for duplicate's of peer's name and file_name
                file_exists = True                              # peer's file exist on server set flag to true
                error_pdu = PDU('E', 'File associated with another User --> Pick another user name')       # if peer name and file already exist create
error PDU object
                connection.send(pickle.dumps(error_pdu))                                # serialize error PDU using pickle() function and send to Peer
                break

        if not file_exists:                                     # if file does not exist register it with the server in Sever_File_Registration_List
            Server_File_Registration_List.append(Files_List(my_peer_name, my_file_name, my_peer_address))
            ack_pdu = PDU('A', 'Successfully Registered')                   # create  acknowledgement PDU
            print("Successfully Registered User :" + my_peer_name + " and File :" + my_file_name)
            connection.send(pickle.dumps(ack_pdu))                  # serialize ack PDU using pickle() function and send to peer

    if peer_pdu.data_type == 'T':                               # check peer_PDU for "T" --> De-registration Type and set peer_name, file_name
and address to temp variables
        loop_count = 0
        my_peer_name = peer_pdu.data.get('peer_name')
```

```
        my_file_name = peer_pdu.data.get('file_name')
        my_peer_address = peer_pdu.data.get('address')

        for i in Server_File_Registration_List:                    # check Sever_File_Registration_List for existence for file to de-register with
index server
            if my_peer_name == i.peer_name and my_file_name == i.file_name and my_peer_address == i.address:
                Server_File_Registration_List.remove(i)            # if found remove file from check Sever_File_Registration_List
                ack_pdu = PDU('A', 'Successfully De-Registered')   # create acknowledgement PDU
                print("De-registered:" + " " + i.peer_name + " " + i.file_name)
                connection.send(pickle.dumps(ack_pdu))             # serialize ack PDU using pickle() function and send to peer
                break

            if loop_count == len(Server_File_Registration_List):
                error_pdu = PDU('E', 'File does not exist')        # if file does not exist create error PDU
                connection.send(pickle.dumps(error_pdu))           # serialize error PDU using pickle() function and send to peer

    if peer_pdu.data_type == 'S':                                  # check peer_PDU for "S" --> Search Type --- set peer_name and file_name temp
variables

        my_file_name = peer_pdu.data.get('file_name')              # the file the peer requested
```

```
        target_address = " "
        found_content = False

        for i in reversed(Server_File_Registration_List):                    # if peer and file found ---> set found flag to true and set target ip address to that of Peer with file
                                                                             # NOTE Server_File_Registration_List is searched in reversed to take latest file & address association
            if my_file_name == i.file_name:                                  # my_peer_name == i.peer_name and
                found_content = True
                target_address = i.address
                break

        if found_content:                                                    # if found flag is true---> send ip address associated with file to peer
            search_ack_pdu = PDU('S', target_address)                        # create search_ack PDU
            connection.send(pickle.dumps(search_ack_pdu))                    # serialize search_ack PDU using pickle() function and send to peer
            print("Send--->" + str(search_ack_pdu))

        if not found_content:
            error_pdu = PDU('E', 'File Request does not exist')              # if file not found on index server send error---> create error PDU
            connection.send(pickle.dumps(error_pdu))                         # serialize error PDU using pickle() function and send to peer
```

```
    if peer_pdu.data_type == 'O':                                   # check peer_PDU for "O" --> List Index Type

        if len(Server_File_Registration_List) == 0:

            print("Send to Peer--->" + str(PDU(data_type="O", data={'peer_name': "No User ",
                                                'file_name': "List Empty",
                                                'address': " "})))
            registration_pdu = PDU(data_type="O", data={'peer_name': "No User ",        # create "O" type registration PDU to with name, file and address information
                                        'file_name': "List Empty",
                                        'address': " "})

            connection.send(pickle.dumps(registration_pdu))

        else:
            for i in Server_File_Registration_List:

                print("Send to Peer--->" + str(PDU(data_type="O", data={'peer_name': i.peer_name,
                                                'file_name': i.file_name,
                                                'address': i.address})))
```

```
            registration_pdu = PDU(data_type="O", data={'peer_name': i.peer_name,          # create "O" type registration PDU to with name, file and address information
                                                        'file_name': i.file_name,
                                                        'address': i.address})
            connection.send(pickle.dumps(registration_pdu))                    # serialize registration PDU using pickle() function and send to peer
            time.sleep(0.1)                                                    # delay required---> multiple messages in quick succession might be received in a single .recv() causing pickling errors


    connection.close()



try:
    s.bind((host, port))                    # reserve IP address and port for script
except socket.error as e:
    print(str(e))

print("Waiting for connection....")
s.listen(5)                             # Now wait for client connection.

while True:
    conn, address = s.accept()                                                 # Establish connection with client.
```

```
    print('Connected to: ' + str(address[0]) + ':' + str(address[1]))
    start_new_thread(client_thread, (conn,))                    # start new client thread for each Peer that connect--> with args "conn" to
forward socket access
    ThreadCount += 1
    print("Connected Peer Number:" + str(ThreadCount))


s.close()
```

P2Ppeer Source Code

```
# P2Ppeer.py
# Brandon Ho 500727531
# Ryan Austin 013249610
'''
A P2P client
It provides the following functions:
- Register the content file to the index server (R)
- Contact the index server to search for a content file (D)
    - Contact the peer to download the file
    - Register the content file to the index server
```

```
- De-register a content file (T)
- List the local registered content files (L)
- List the on-line registered content files (O)
'''

import time
import os.path
import socket
import pickle
from collections import namedtuple
import select
from threading import Thread


s = socket.socket(socket.SOCK_DGRAM)              # Create a socket object "UDP" for peer to index-server communication
host = "localhost"                                # 127.0.0.1 == "localhost" == " " --> default machine address
port = 60000                                      # Reserve port for peer to index server communication
s.settimeout(1)                                   # timeout -->1 second
s.connect((host, port))                           # establish connection to index server
PDU = namedtuple('PDU', ['data_type', 'data'])    # define the PDU structure using namedtuple
```

```
def select_name():
    return input('Please enter preferred username:')



def de_register(user_name, filename, peer_port, file_index):
    deregister_response_packet_pdu = None

    delete_request_pdu = PDU(data_type="T", data={"peer_name": user_name,                    # create new 'T' PDU using username, filename, IP address and port number
                                                  "file_name": filename,
                                                  "address": ("127.0.0.1", peer_port)})
    s.send(pickle.dumps(delete_request_pdu))                                  # send serialized PDU using pickle() function to  index server

    while True:

        try:
            deregister_response_packet = s.recv(200)                          # receive response "download_response_packet" from server
            deregister_response_packet_pdu = pickle.loads(deregister_response_packet)          # receive response PDU from index server
        except socket.timeout:
            print("Timeout-No more data")
```

```
    if deregister_response_packet_pdu.data_type == "A":                    # if response data_type is 'A', done
        file_index.remove(filename)                                        # remove de-registered files from local file_list_index--->so don't double remove file when quitting
        print(deregister_response_packet_pdu.data)
        break

    elif deregister_response_packet_pdu.data_type == "E":                  # a user may need to retry multiple times to deregister a file on server
        print(deregister_response_packet_pdu.data)
        filename_re_request = input(" What file do you want to de-register with the Index Server? ")      # ask user for another file name to deregister

        delete_request_pdu = PDU(data_type="T", data={"peer_name": user_name,          # create new 'T' PDU using username, filename, IP address and port number
                                                      "file_name": filename_re_request,
                                                      "address": ("127.0.0.1", peer_port)})
        s.send(pickle.dumps(delete_request_pdu))                           # send 'T' PDU to index server


def register(user_name, filename, peer_port, file_index):
```

```
    file_index_request_pdu = PDU(data_type="R", data={"peer_name": user_name,                    # create 'R' pdu using username, filename, IP address and port number
                                                      "file_name": filename,
                                                      "address": ("127.0.0.1", peer_port)})
    s.send(pickle.dumps(file_index_request_pdu))                                                 # send 'R' PDU to index server

    while True:

        try:
            download_response_packet = s.recv(200)                                               # receive response "download_response_packet" from server
            download_response_packet_pdu = pickle.loads(download_response_packet)                # receive response PDU from index server
        except socket.timeout:
            print("Timeout-No more data")
            break

        if download_response_packet_pdu.data_type == "A":                # if response data_type is 'A', done
            file_index.append(filename)                                  # keep track locally what files are being registered with the index server
            print(download_response_packet_pdu.data)
            break

        elif download_response_packet_pdu.data_type == "E":              # Error --> user already associated with file
```

```
        print(download_response_packet_pdu.data)                    # a user may need to retry multiple times to register a username on server!
        username = select_name()                                    # ask user to change username
        file_index_request_pdu = PDU(data_type="R", data={"peer_name": username,            # create new 'R' PDU using username, filename, IP address and port number
                                                "file_name": filename,
                                                "address": ("127.0.0.1", peer_port)})
        s.send(pickle.dumps(file_index_request_pdu))



def download_file(file_name, address, file_index):

    ip = address[0]                             # copy ip portion of address array for destination
    port_number = address[1]                    # copy port number of address array for destination

    # establish new TCP connection
    destination_peer_socket = socket.socket()           # create socket TCP
    destination_peer_socket.settimeout(1)               # set timeout 2 second ---> make sure all data is received
    destination_peer_socket.connect((ip, port_number))      # connect to peer socket using destination IP and port#

    # create a 'D' type pdu asking for the file
    download_pdu = PDU('D', file_name)                  # create PDU requesting file
```

```
    serialized_my_pdu = pickle.dumps(download_pdu)          # serialize PDU using pickle() (binary data req. for TCP transmission)
    destination_peer_socket.send(serialized_my_pdu)         # send serialized PDU to peer
    print("send", download_pdu)
    with open(ip + " " + str(port_number) + " " + file_name, 'w') as f:                    # open file to write information received from server


        while True:


            try:
                destination_packet = destination_peer_socket.recv(200)                  #  receive packet from peer----buffer must be larger that 100bytes for named tuple object overhead
                received_pdu = pickle.loads(destination_packet)                         # decode serialized packets using pickle() function


            except socket.timeout:
                print("No more data to download")
                break


            if received_pdu.data_type == "C":                                   # if received PDU data_type = C write data to file
                if received_pdu.data == "#?#?#?#":
                    register(the_user_name, file_name, the_peer_port, file_index)       # NOTE-->if content successfully transferred to Peer, register file with this Peer identity on the index server
                    print('Received<---', received_pdu)
```

```
                break

            f.write(received_pdu.data)                          # write data to file
            print('Received<---', received_pdu)

        elif received_pdu.data_type == "E":                     # if received PDU data_type = E error--> close connection

            print(received_pdu.data)
            print('Connection closed')
            break

    f.close()                                           # close content file
    destination_peer_socket.close()                             # close connection


def server_function():

    server_socket = socket.socket()  # this is a TCP connection
    server_socket.settimeout(1)      # set timeout 1 second
    socket_list = []  # holds socket of peer requesting a connection
    outputs = []  # empty
```

```
    exp = []  # exceptions

    try:
        server_socket.bind(('localhost', the_peer_port))
        server_socket.listen(5)

    except socket.error as msg:

        print(" Socket binding error failed to connect" + str(msg))

    socket_list.append(server_socket)  # add ready sockets to list
    exp.append(server_socket)

    while True:

        try:
            readable, writable, exceptional = select.select(socket_list, outputs, exp)       # listen for available sockets which are ready through select()
function
        except ValueError:
            for i in socket_list:                                    # remove any terminated sockets ---> file descriptor -1 if invalid
                if i.fileno() == -1:
```

```
            print(socket_list.remove(i))

        for sock in readable:

            if sock == server_socket:                                   # check Socket_List for sockets to establish connection

                file_request_conn, addr = server_socket.accept()            # new connection made with peer

                socket_list.append(file_request_conn)                   # add peer to Socket_List ----> peers to share with

            else:

                try:

                    server_packet = sock.recv(200)                      # receive packet from Peer --->Buffer must be greater than 100bytes for
namedtuple overhead

                    if server_packet:

                        my_pdu = pickle.loads(server_packet)            # decode serialized packet back into PDU

                        if my_pdu.data_type == "D":                     # check PDU the data_type (it should be 'D' type)

                                                                    # NOTE----> my_PDU.data should contain file name
```

```
                if os.path.isfile("./Peer_Files/" + my_pdu.data):                    # if file exist open it and send to peer who requested file---otherwise send error

                    read_data = open("./Peer_Files/" + my_pdu.data, 'r')             # open requested file and read it

                    while True:                                    # loop to send contents of buffer 'data_to_send' until complete
                        data_to_send = read_data.read(99)                  # read with Buffer 99 bytes
                        if data_to_send == "":                         # if EOF---> break
                            data_pdu = PDU("C", "#?#?#?#")                 # send the file using 'C' with marker to signify end of content
                            sock.send(pickle.dumps(data_pdu))              # send contents of buffer after serializing data_PDU using pickle()
                            print('Sending--->' + str(data_pdu))
                            break

                        data_pdu = PDU("C", data_to_send)                      # send the file using 'C' data_type PDU
                        sock.send(pickle.dumps(data_pdu))                      # send contents of buffer after serializing data_PDU using pickle()
                        time.sleep(0.1)                                # delay to avoid pickling errors with consecutive file being transfered
                        print('Sending--->' + str(data_pdu))

                    read_data.close()                                  # close opened file

                    print("Done sending---Connection Closed")
```

```
                else:

                    print("File does not exist server function")
                    error_pdu = PDU("E", "File does not exist")                    # if file doest not exist send 'E' pdu for error to peer
                    sock.send(pickle.dumps(error_pdu))                             # send PDU after serializing using pickle() function

                    print("Connection Closed")
            else:

                if sock in socket_list:                                            # if not incoming data remove socket from socket_list
                    socket_list.remove(sock)
        except socket.timeout:
            print("Timeout-No data available")


def client_function():
    file_index_list = []                                                           # list of local files registered on index server
    termination_flag = 0                                                           # terminate client operation when flag is set
    while True:                                                                    # command menu for client
        if termination_flag == 0:
```

```
        command = str(input('\nPlease choose from the list below:\n'
                            '[O] Get online list\n'
                            '[L] List local files\n'
                            '[R] Register a file\n'
                            '[T] De-register a file\n'
                            '[Q] Quit the program\n\n'))


        if command == 'O':
            is_list_empty_flag = 0                          # flag to skip download input request if no files listed on index server---> 1 = no files listed, 0
= files listed
            registration_list_pdu = PDU('O', " ")                                   # create "O" type PDU----NO data required
            s.send(pickle.dumps(registration_list_pdu))                                 # send PDU to index server


            while True:


                try:
                    registration_packet = s.recv(200)                               # receive_registration_list_packet from server
                    client_packet_pdu = pickle.loads(registration_packet)                   # client_packet using pickle() function


                except socket.timeout:
                    print("\nList Download Complete")                                 # registration_packet timed out---List must be complete
```

```
                break

            if client_packet_pdu.data_type == "O":

                if client_packet_pdu.data.get('file_name') == "List Empty":
                    print("List is Empty---Nothing to download")
                    is_list_empty_flag += 1
                    break
                print("User: " + str(client_packet_pdu.data.get('peer_name')) + " " +          # print index server registration list
                      "file: " + str(client_packet_pdu.data.get('file_name')))

        if is_list_empty_flag == 0:                                  # if list provided by index server isn't empty ask for user to download file

            filename_request = input("What file do you want to download?\n")                  # request file from user

            file_request_pdu = PDU(data_type="S", data={"file_name": filename_request})          # create 'S' type PDU to query server for peer
with file

            s.send(pickle.dumps(file_request_pdu))                                  # send PDU to the index server after serializing with pickle()
function
```

```
            while True:

                try:
                    download_info_packet = s.recv(200)                          # download_info_packet from server----buffer must be larger that
100bytes for named tuple object overhead
                    download_info_pdu = pickle.loads(download_info_packet)              # receive 'S' or 'E' PDU in response
                except socket.timeout:
                    print("Timeout-No more data")
                    break

                if download_info_pdu.data_type == "S":
                    peer_address = download_info_pdu.data                           # location of the file -----> Peer's address
                    download_file(filename_request, peer_address, file_index_list)        # establish P2P connection to the peer
                    break

                elif download_info_pdu.data_type == "E":
                    print(download_info_pdu.data)                                   # print error message sent from server
                    break

        if command == 'L':
```

```
            directory = os.listdir('Peer_Files')                    # list local files---> make sure to run shell from "Final Project Folder
            for files in directory:
                print(files)                                        # the folder path is relative to where the Python shell has been launched

        if command == 'R':

            filename_index_request = input("What file do you want to register with the Index Server?\n")       # get the file name to register from
user

            register(the_user_name, filename_index_request, the_peer_port, file_index_list)                 # register file with server

        if command == 'T':

            filename_index_delete_request = input("What file do you want to de-register "
                                                  "with the Index Server?\n")                   # get the file name from user

            if filename_index_delete_request in file_index_list:                    # check to see if user registered the file--> prevents user from
de-registering another peer's files
                de_register(the_user_name, filename_index_delete_request, the_peer_port, file_index_list)
            else:
                print("File doesn't exist or is not associated with this User\n"
```

```
                "--->NOTE only the Peer who registered the file can remove it from the Server")

        if command == 'Q':

            for file in file_index_list:
                de_register(the_user_name, file, the_peer_port, file_index_list)

            file_index_list.clear()                                 # clear local file registration index list
            s.close()                                               # close connection with socket
            termination_flag += 1                                   # set termination flag to stop client operation
            break
    print("Client Session Terminated")


the_user_name = select_name()                                       # request username

the_peer_port = int(input('Please enter listening port number for the P2P server:'))          # request port for peer

# create two threads to run client and server operation concurrently
p1 = Thread(target=client_function)                         #args=(the_user_name, the_peer_port))
p = Thread(target=server_function)                          #args=(the_peer_port,))
```

```
p1.start()
p.start()
```