COMP3234B Computer and Communication Networks

Assignment 1: A Simple P2P File Sharing Application (12%)

[Learning Outcome 4]

Due by: 23:59 Friday February 28, 2025

Total mark is 100.

I. OVERVIEW OF THE ASSIGNMENT

In this assignment, you will implement a simple peer-to-peer (P2P) file sharing application using Python socket programming. This application is a much simplified version of real-world P2P systems, like Gnutella and BitTorrent.

In this application, each peer acts as both a client and a server (i.e., runs both a client process and a server process). The client programs upload and download files to and from other peers, while the server programs collaboratively serve file downloading.

II. DETAILED DESIGN OF THE APPLICATION

1. P2P File Sharing Protocol

The P2P system that we are considering uses a fully connected topology where all peers are interconnected, i.e., every peer is given the complete peer list upon joining the system. Each peer runs a client program (process) and a server program (process). A client should be able to upload and download files to and from the servers, while a server should be able to receive, maintain and serve files to clients. The communication between clients and servers uses TCP.

The client program reads input commands from the user through keyboard/terminal, composes protocol request messages, and sends the requests to the specified servers.

The server program creates a server socket listening on a publicly known port to wait for connection requests from the clients. Upon receiving a TCP connection request, the server creates a new connection socket and launches a new thread to handle the requested connection. After the TCP connection is established, the server receives, parses and processes the request messages, and then sends reply messages through the same TCP connection.

In a P2P file sharing system, each file is divided into chunks so that peers may collaborate to serve file download to speed it up. The size of each chunk in our system is 100 bytes, except that the last chunk has the size of the remaining part of the file. For example, a file of 876 bytes is truncated into 9 chunks, the first 8 chunks are 100 bytes long each, and the last chunk contains 76 bytes. From beginning to end of the file, the chunks are numbered 0, 1, 2, 3, ... In this assignment, all the files contain only ASCII characters.

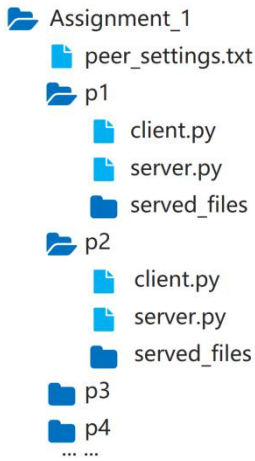

Figure 1. The directory structure of the P2P file sharing system (on a single computer)

To run multiple peers to test the P2P application on one computer, the directory structure of this assignment should be structured as in Figure 1. Suppose the peer IDs for the peers are p1, p2, p3, p4, ... We have a directory for each peer named after its ID, which is called the *peer directory*. Each peer directory contains client.py (to be run as its client process), server.py (to be run as its server process), and one subdirectory "served_files". "served_files" contains the files this peer is serving, i.e., the peer can upload chunks of the files in "served_files" upon client download requests.

File "peer_settings.txt" provides the address information of the server in each peer in the P2P application, one line per peer in the format of "peer_ID ip_addr server_port". "ip_addr" is the IP address of the machine the peer is running on and "server_port" is the publicly known port the peer's server keeps listening on for incoming TCP connection requests. When all the peers are running on the same machine, you may use "localhost" (without quotes) for "ip_addr" for all peers, and specify different ports for different servers to listen on. We also assume that any client and server programs only have access to file "peer_settings.txt" and their own peer directory, and should not access any other peer directory.

The P2P file sharing protocol consists of three types of requests:

#FILELIST

#UPLOAD

#DOWNLOAD

The status codes to be used in a reply message include:

200 — indicating requested action OK

250 — indicating errors

330 — indicating ready to receive/send

#FILELIST. A user can discover files served by specified peers by entering "#FILELIST p1 p2 p3 ..." (without quotes) on the terminal to its client program, where "p1 p2 p3 ..." are peer IDs indicating the peers from which the user wants to retrieve their list of served files. Three dots "..." is an indicator of the varied-length peer ID list; the actual message should not contain them. The

following are the steps of the #FILELIST protocol, to be implemented in the client program and the server program, respectively.

client	server
1. Send "#FILELIST" to each of the servers in "p1 p2 p3 ..."	
	2. Reply "200 Files served: f1.txt f2.txt f3.txt ..." ("f1.txt f2.txt f3.txt ..." are all the files stored in subdirectory "peer_id/served_files" of the server).

#UPLOAD. A user can share its files to a selected set of peers by entering "#UPLOAD filename p1 p2 p3 ..." on the terminal. The file is to be found in subdirectory "served_files" under the peer directory of this user. If the file does not exist, the client program should print "Peer peer_id does not serve file filename" on its terminal, where peer_id is the peer ID of the current peer; otherwise, the client prints "Uploading file filename", and carries out the following #UPLOAD protocol with the respective server programs.

client	server
1. Send "#UPLOAD filename bytes xxx" to each of the servers in "p1 p2 p3 ...", where "xxx" is the size in bytes of the file.	
	2. If the server is receiving the same file in handling another #UPLOAD request, reply "250 Currently receiving file filename". If the server is already serving the file, i.e., the file exists in subdirectory "served_files" of its peer directory, reply "250 Already serving file filename". Otherwise, reply "330 Ready to receive file filename".
3. If the reply code is "330", proceed to step 4; otherwise quit.	
4. Read the ith chunk from the file (starting from the first chunk), and send it to the server using the request message "#UPLOAD filename chunk i xxxxxx", where xxxxxx is the actual content of the ith chunk.	
	5. Reply "200 File filename chunk i received".
6. Repeat steps 4-5 until all chunks in the file are sent.	
	7. If all chunks are received, save the file in subdirectory "peer_dir/served_files" of the server, and reply "200 File filename received".
8. If the reply is "200 File filename received", print "File filename upload success" on the terminal. Otherwise, print "File filename upload failed".	

Upon successful completion of #UPLOAD, the file should appear in the subdirectory "peer_dir/served_files" of the respective servers.

When #UPLOAD is not yet completed, any further #UPLOAD or #DOWNLOAD requests for *the same file* will be turned down by a respective server. However, it is possible for a server to be receiving different files simultaneously from multiple clients.

#DOWNLOAD. When a user wants to download a file in the P2P file sharing system, the peers serving the wanted file will collaborate to speed up the file serving. The user types "#DOWNLOAD filename p1 p2 p3 ..." on the terminal of its client to download the file specified by filename from a group of peers "p1 p2 p3 ...". If the file already exists in "served_files" under the peer directory of this peer, the client prints on its terminal "File filename already exists"; otherwise, it prints "Downloading file filename" and carries out file downloading steps.

The download steps include 2 phases. First, the client sends "#DOWNLOAD filename" requests to the peers in "p1 p2 p3 ...", and then knows from their replies which of them are serving the file. We refer to the set of peers in "p1 p2 p3 ..." that are serving the file as the *serving peers set*.

Next, the client requests chunks from each peer in the serving peers set using "#DOWNLOAD filename chunk i". The chunks to be downloaded from each of these servers are allocated in a round-robin manner: suppose the list of peers in serving peers set are numbered 0, 1 ,2, ..., and the set size is n; chunk i is downloaded from peer r = i % n.

The steps of the complete #DOWNLOAD protocol are as follows.

client	server
1. Send "#DOWNLOAD filename" to each of the servers in "p1 p2 p3 ..."	
	2. If the server serves the file, i.e., the file exists in subdirectory "served_files" of its peer directory, reply "330 Ready to send file filename bytes xxx". Otherwise, reply "250 Not serving file filename".
3. Add all of the peer IDs of the servers whose reply code is "330" into the serving peers set.	
4. If the serving peers set is not empty, proceed to step 5. Otherwise, print "File filename download failed, peers p1 p2 p3 ... are not serving the file", and quit.	
5. Send "#DOWNLOAD filename chunk i" to the respective server in serving peers set, where i is decided using the round-robin method described above.	
	6. Reply "200 File filename chunk i xxxxxx", where xxxxxx is the actual content of chunk i.
7. Repeat steps 5-6 until all chunks are downloaded.	
8. If all chunks of the file are received, save the file into subdirectory "peer_dir/served_files" of the peer, and print "File filename download success" on the terminal. Otherwise, print "File filename	

download failed".	

Upon completion of the download, the file should appear in the subdirectory "peer_dir/served_files", where "peer_dir" is the peer directory of this peer.

2. Client and Server Implementation

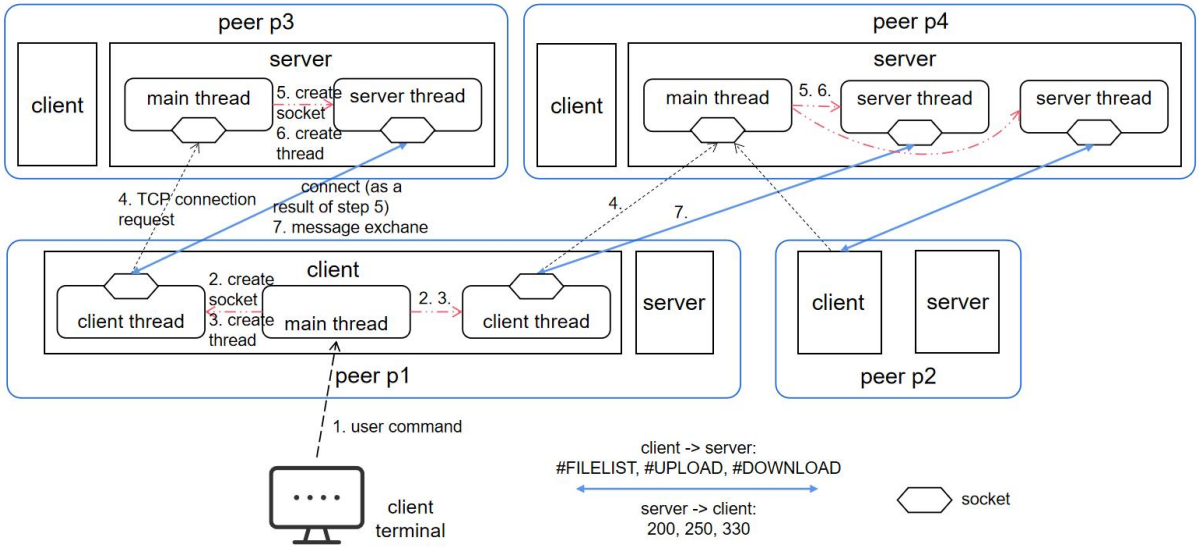

Figure 2. Multi-threading implementation for P2P file sharing.

When the client program is launched, it should read from the file "peer_settings.txt", parse the peer list, and prompt "Input your command: " on the terminal. It should use multi-threading to handle each request (#FILELIST, #UPLOAD and #DOWNLOAD), by creating a separate thread for a TCP socket connection to each server (except the server running on the same peer) and simultaneously handling connections to multiple servers involved in this request with multiple threads.

The server program should keep listening on the designated port as specified in "peer_settings.txt". Upon receipt of a TCP connection request from a client, the server creates a new connection socket delegated to this client, and a new thread to handle the socket. Figure 2 illustrates the multi-threading design of the system.

The main method of the server program takes peer ID as its parameter. For example, you run "python3 server.py p2" to start the server program for peer p2. The client program can be launched using "python3 client.py". Note that the client.py and server.py programs should be the same among the peers, and you just launch the programs under different peer directories to emulate different peers.

III. IMPLEMENTATION NOTES

1) All files to serve in this assignment are text files with suffix ".txt". Filenames and directory names do not contain spaces.

2) A peer either can serve an entire file, or does not serve the file at all. When a peer owns only some chunks of a file, e.g., when #UPLOAD is still in progress, it is not considered to be able to serve the file, i.e., it will turn down #DOWNLOAD requests for this file.

3) In a practical environment, a peer may leave the system at any time, thus breaking the TCP connection. Your program should be resilient to these exceptions without failing. You can prolong the upload/download time by putting the program to sleep for 0.5 seconds after uploading/downloading a chunk, so that you may have enough time to kill some peers to mimic peer leaving during the process.

4) When you run commands "#FILELIST p1 p2 p3 ...", "#UPLOAD filename p1 p2 p3 ..." or "#DOWNLOAD filename p1 p2 p3 ...", some servers in "p1 p2 p3 ..." may not be running, and your client's connection requests to these servers may fail. Your client program should be resilient to this exception without failing, print on the terminal "TCP connection to server peer_id failed" for each offline server, and stop trying to connect to these servers.

5) If #UPLOAD or #DOWNLOAD fails, the partially uploaded/downloaded file should be removed from the respective peer.

6) In addition to the terminal inputs and outputs as described above, you should also print all of the sent requests and received replies in the client terminal, using the following format: "Client (peer_id): request_message" and "Server (peer_id): reply_message", where peer_id is the ID of the peer that the request is bound to or the reply comes from. There are two exceptions: instead of printing "Client (peer_id): #UPLOAD filename chunk i xxxxxx" and "Server (peer_id): 200 File filename chunk i xxxxxx" (#DOWNLOAD reply) directly on the terminal, where xxxxxx could be very long, you should remove xxxxxx before printing.

A sample interaction between a user and peers is given as follows. This is exactly what you will see on the client terminal of the user. The texts are colored for easier reading: the red texts are terminal prompt followed by user commands, the blue texts are client terminal messages as printed out in the protocol, and the remaining in black are the messages exchanged between the client and servers. Your client terminal output should look like this, except order differences of exchanged messages and terminal outputs due to the difference in thread timing.

Input your command: #FILELIST p2 p3 p4
Client (p4): #FILELIST
Client (p3): #FILELIST
Client (p2): #FILELIST
Server (p3): 200 Files served: f1.txt f5.txt
Server (p4): 200 Files served: f1.txt
Server (p2): 200 Files served: f1.txt

Input your command: #UPLOAD f2.txt p2 p3
Uploading file f2.txt
Client (p2): #UPLOAD f2.txt bytes 131
Client (p3): #UPLOAD f2.txt bytes 131
Server (p3): 330 Ready to receive file f2.txt
Client (p3): #UPLOAD f2.txt chunk 0
Server (p2): 330 Ready to receive file f2.txt
Client (p2): #UPLOAD f2.txt chunk 0
Server (p3): 200 File f2.txt chunk 0 received
Server (p2): 200 File f2.txt chunk 0 received
Client (p2): #UPLOAD f2.txt chunk 1
Server (p2): 200 File f2.txt chunk 1 received
Client (p3): #UPLOAD f2.txt chunk 1
Server (p3): 200 File f2.txt chunk 1 received
Server (p2): 200 File f2.txt received

Server (p3): 200 File f2.txt received
File f2.txt upload success
File f2.txt upload success

Input your command: #DOWNLOAD f1.txt p2 p3 p4
Downloading file f1.txt
Client (p2): #DOWNLOAD f1.txt
Client (p3): #DOWNLOAD f1.txt
Client (p4): #DOWNLOAD f1.txt
Server (p2): 330 Ready to send file f1.txt bytes 950
Client (p2): #DOWNLOAD f1.txt chunk 0
Server (p3): 330 Ready to send file f1.txt bytes 950
Server (p4): 330 Ready to send file f1.txt bytes 950
Client (p3): #DOWNLOAD f1.txt chunk 1
Client (p4): #DOWNLOAD f1.txt chunk 2
Server (p2): 200 File f1.txt chunk 0
Client (p2): #DOWNLOAD f1.txt chunk 3
Server (p3): 200 File f1.txt chunk 1
Server (p4): 200 File f1.txt chunk 2
Client (p3): #DOWNLOAD f1.txt chunk 4
Client (p4): #DOWNLOAD f1.txt chunk 5
Server (p4): 200 File f1.txt chunk 5
Client (p4): #DOWNLOAD f1.txt chunk 8
Server (p2): 200 File f1.txt chunk 3
Client (p2): #DOWNLOAD f1.txt chunk 6
Server (p3): 200 File f1.txt chunk 4
Client (p3): #DOWNLOAD f1.txt chunk 7
Server (p4): 200 File f1.txt chunk 8
Server (p2): 200 File f1.txt chunk 6
Client (p2): #DOWNLOAD f1.txt chunk 9
Server (p3): 200 File f1.txt chunk 7
Server (p2): 200 File f1.txt chunk 9
File f1.txt download success

7) You should make sure your message formats, the formats of inputs and outputs, strictly comply to the stated design.

8) You are free to decide the outputs of your server terminal, since they are not evaluated in the assignment marking. We will check the client terminal output, your python code and the optional readme.txt file, in the assignment marking.

9) You can assume that the input commands in the terminal are in correct format.

10) There are some python libraries and functions that you may find useful for this assignment.

System specific functions and parameters:

sys.argv	The list of command line arguments passed to a Python script. argv[0] is the script name. argv[1], argv[2], ... are the arguments passed to the python script. See more at https://docs.python.org/3/library/sys.html

Time related function, which is useful for prolonging program running so you have time to shut down peers to mimic peer dynamics:

time.sleep(secs)	Suspend execution of the calling thread for the given number of seconds. The argument may be a floating-point number to indicate a more precise sleep time. See more at https://docs.python.org/3/library/time.html#time.sleep

OS specific functions:

os.listdir	Return a list containing the names of the entries in the directory given by path. You may use it to iterate the files in "served_files". See more at https://docs.python.org/3/library/os.html
os.getcwd()	Return a string representing the current working directory. See more at https://docs.python.org/3/library/os.html#os.getcwd
os.path.basename(path)	Return the base name of pathname path. You can get the directory name of the current directory using "os.path.basename(os.getcwd())". See more at https://docs.python.org/3/library/os.path.html#os.path.basename

Files reading and writing functions:

with open(filename, mode) as f:	mode: a string describing the way in which the file will be used. 'r' for reading and 'w' for writing. See more at https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files
f.read()	Read the whole file contents and return them as a string. See more at https://docs.python.org/3/tutorial/inputoutput.html
f.write(content)	Write content into a file. See more at https://docs.python.org/3/tutorial/inputoutput.html

String split and join:

str.split(sep=None, maxsplit=-1)	Return a list of the words in the string, using sep as the delimiter string. You may want to use str.split(" ") to split the request and reply strings before further processing. See more at https://docs.python.org/3.6/library/stdtypes.html
str.join(iterable)	Return a string which is the concatenation of the strings in iterable. You may want to use it to construct a request or reply string. See more at https://docs.python.org/3/library/stdtypes.html#str.join

IV. TESTING SCHEME

#FILELIST

1) Retrieve the file lists from multiple servers.

#UPLOAD

2) Upload a file to multiple servers.

3) Multiple clients upload different files to the same server.

4) When a server is receiving a file during #UPLOAD, it receives another #UPLOAD request for the same file (server should reject) or a different file that the server is not serving (server should accept).

5) During the process of uploading, the client or a server is shut down.

6) Upload a file that does not exist in the client's "served_files" subdirectory.

#DOWNLOAD

7) Download a file from multiple servers.

8) Multiple clients download files from the same server.

9) When a server is receiving a file during #UPLOAD, it receives a #DOWNLOAD request for the same file (server should reject) or a different file that the server is serving (server should accept).

10) During the process of downloading, the client or a server is shut down.

11) Download a file that already exists in the client's "served_files" subdirectory.

V. GROUP OR INDIVIDUAL

You should work on this assignment individually, or in a group with at most two students.

You will get 10% bonus mark if you work on the assignment individually, i.e., your mark will be your original mark * 1.1, upper bounded by the full mark of 100.

VI. SUBMISSION

You should submit the following files in a zip file, named as a1-yourUID.zip (individual work) or a1-UID1-UID2.zip (group work; only one submission by one of the group members is needed for each group):

(1) client.py

(2) server.py

(3) A readme file to provide any additional information on the implementation and execution of your programs that you deem necessary for us to know.

Please submit the .zip file on Moodle:

(1) Login Moodle.

(2) Find "Assignments" in the left column and click "Assignment 1".

(3) Click "Add submission", browse your .zip file and save it. Done.

(4) You will receive an automatic confirmation email, if the submission was successful.

(5) You can "Edit submission" to your already submitted file, but ONLY before the deadline.