

VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY  
UNIVERSITY OF TECHNOLOGY  
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



---

COMPUTER NETWORK - Assignment 1

# DEVELOP A NETWORK APPLICATION

Semester: 232

---

<b>Advisors:</b>	Dr. Nguyen Le Duy Lai	
<b>Student:</b>	Ta Ngoc Nam	2152788
	Nguyen Thinh Dat	2152507
	Pham Huy Thien Phuc	2053346

HO CHI MINH CITY, APRIL 2024



## Members list & Workload

No.	Full name	Student ID	Duty in the assignment	Percentage of work
1	Ta Ngoc Nam	2152788		100%
2	Nguyen Thinh Dat	2152507		100%
3	Pham Huy Thien Phuc	2053346		100%

# Contents

<b>1</b>	<b>Assignment specifications</b>	<b>3</b>
1.1	Objective . . . . .	3
1.2	Application description . . . . .	3
<b>2</b>	<b>Implementation</b>	<b>5</b>
2.1	Brief overview of the system . . . . .	5
2.1.1	Core functionalities . . . . .	5
2.1.2	Additional functionalities . . . . .	5
2.2	Diagrams . . . . .	5
2.3	Phase 1 . . . . .	7
2.3.1	Functions of the file-sharing application . . . . .	7
2.3.2	Communication protocols used for each function . . . . .	9
2.3.3	Extension functions . . . . .	15
2.3.4	Layered architecture . . . . .	16
2.4	Phase 2 . . . . .	17
2.4.1	Manual document . . . . .	17
2.4.2	Basic sanity test . . . . .	18
2.4.3	Advanced sanity test . . . . .	23
2.4.4	Evaluation . . . . .	26
2.5	Conclusion . . . . .	27

## Chapter 1

# Assignment specifications

### 1.1 Objective

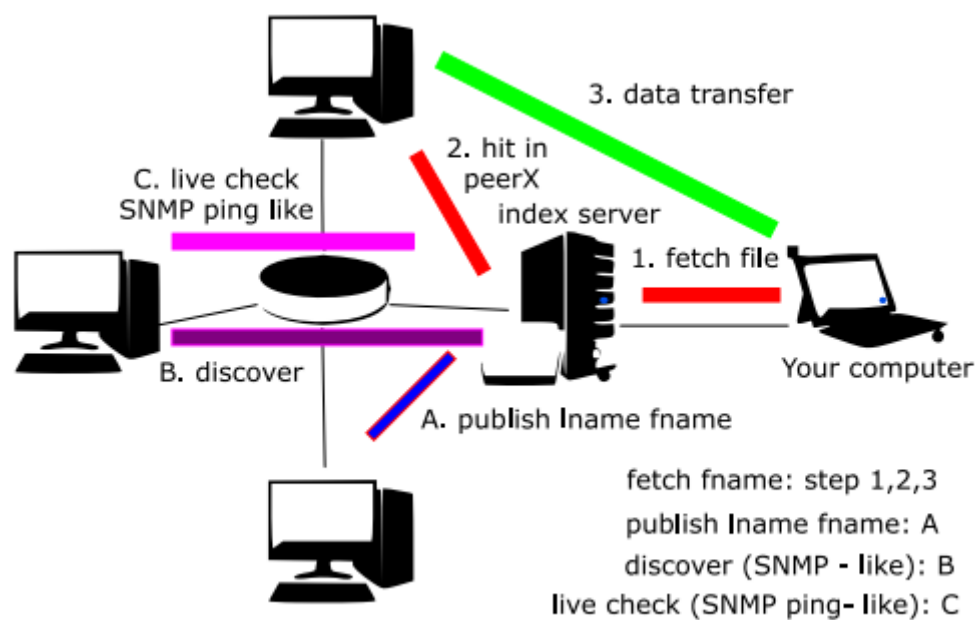
Build a simple file-sharing application with application protocols defined by each group, using the TCP/IP protocol stack.

### 1.2 Application description

- A centralized server keeps track of which clients are connected and storing what files.
- A client informs the server as to what files are contained in its local repository but does not actually transmit file data to the server.
- When a client requires a file that does not belong to its repository, a request is sent to the server. The server identifies some other clients who store the requested file and sends their identities to the requesting client.
- The client will select an appropriate source node and the file is then directly fetched by the requesting client from the node that has a copy of the file without requiring any server intervention.
- Multiple clients could be downloading different files from a target client at a given point in time. This requires the client code to be multithreaded.
- The client has a simple command-shell interpreter that is used to accept two kinds of commands.
  - `publish lname fname`: a local file (which is stored in the client's file system at `lname`) is added to the client's repository as a file named `fname` and this information is conveyed to the server.
  - `fetch fname`: fetch some copy of the target file and add it to the local repository.
- The server has a simple command-shell interpreter
  - `discover hostname`: discover the list of local files of the host named `hostname`
  - `ping hostname`: live check the host named `hostname`

The supported commands are illustrated in Figure 1.1.

- It's important to note that the connecting infrastructure is not implied by the representation in Figure 1.1. All devices are interconnected through the Internet. Separating them is the logical point of view regarding the protocol's activities.



**Figure 1.1:** Illustration of file-sharing system activities

## Chapter 2

# Implementation

### 2.1 Brief overview of the system

In the P2P file-sharing system we implemented, the overall architecture is as follows:

#### 2.1.1 Core functionalities

- **Server:** For each incoming connection from each client, the server create an object of information related to that particular client and stores it in a data structure inside the server's main memory. Server will also create a thread called "Listen Thread" used to listen to the requests (**fetch**, **publish**) from that client. The information object and threads related to a client exists as long as that client is still connecting. The thread that the user (or administrator) enters the command are also used to send the requests (**ping**, **discover**) to the client.
- **Client:** When the client starts the P2P application, using the pre-configured server's address, the client will connect to the server, sending to the server important information to successfully establish a connection to the server. Then the client will create 2 threads: one is used to listen to requests from server (**ping**, **discover**), one is used to listen to downloading requests from other peers. The thread that the user enters command is also used to send requests (**fetch**, **publish**) to the server.

#### 2.1.2 Additional functionalities

- **Graceful disconnect:** When a client close the application, that information will be sent to the server so that the server can quickly remove that client's related information from the memory and freeing any resources belonging to that client.
- **Keep-alive:** In the case of losing the connection due to any reason and the sockets aren't closed, either sides of the connection will very likely assume that the connection is still there and their sockets will be still running to receive messages unless there is a TCP or Operating system timeout trigger an exception. That is a waste of resources, especially on the server side as it has to serve many clients at the same time. Because of that, we implemented a "keep-alive" protocol to ensure that both side of the connection will actively close the connection after a pre-determined period of time following a lost connection.
- **Resume downloading:** When a file is being download but for some unpredictable reasons that might prevent the client from downloading the file (network issues, signal strength fluctuation, unstable network, etc.), the downloading process can be temporarily pause within a pre-configured amount of time until the downloading connection is closed. If that connection is closed, the next connection used to download the same file will only download the not-yet-downloaded portion of the file. For example, if the client request a 10MB file A from Peer 1 but Peer 1 has just transmitted 4MB of content before closing, the next time we download file A from Peer 1 or any other peer, we only download the remaining 6MB, given that the client hasn't closed the program.

### 2.2 Diagrams

Overall architecture of the system

## • Class diagram

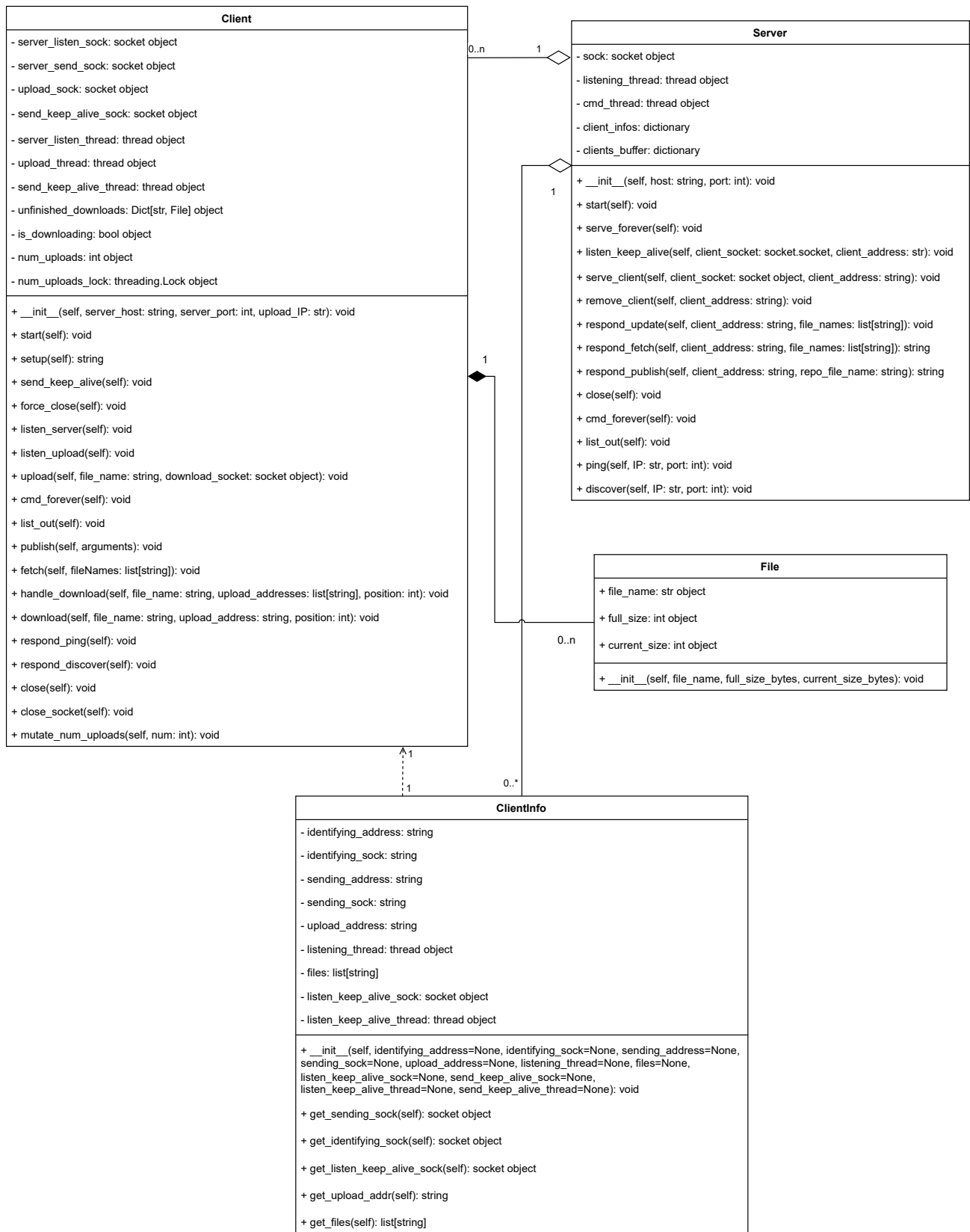


Figure 2.1: Class diagram

- Sequence diagram: <https://drive.google.com/file/d/1ti7tEuAiNyPc-p259T5HYxMBdl6-wjt-/view?usp=sharing>

## 2.3 Phase 1

### 2.3.1 Functions of the file-sharing application

#### Client side

In the client side, we have the following functions:

- `__init__(self, server_host='192.168.1.8', server_port=50004, upload_IP='192.168.1.13')`: This is the constructor of the Client class. It initializes various attributes, including sockets for communication with the server, the upload socket for receiving file upload requests, and server host and port information. It also starts threads for listening to the server and handling file upload requests.
- `start(self)`: This method starts the client by launching the server listening thread, upload thread, and the command input loop.
- `setup(self)`: This method sets up the initial connection to the server. It connects to the server, sends information about the connection, and prints details about the server and upload addresses.
- `send_keep_alive(self)`: Playing role of checking connection between client and server
- `force_close(self)`: If server is down, then we force client to break the connection
- `listen_server(self)`: This method listens continuously for messages from the server. It responds to specific server commands like `ping` and `discover`.
- `listen_upload(self)`: This method listens continuously for incoming download requests from other peers. It creates multiple threads to handle the upload requests in parallel.
- `upload(self, file_name, download_socket)`: This method uploads a file to other peers. It sends the file's size and the file's content to the requesting peer.
- `cmd_forever(self)`: This method listens continuously for user input and processes valid commands. It handles commands such as `publish` and `fetch`.
- `publish(self, arguments)`: This method executes the `publish` command, which involves copying a file from the local directory to the client's repository and notifying the server.
- `fetch(self, filenames)`: This method executes the `fetch` command. It sends a fetch request to the server, which responds with addresses of peers that have the requested files. If multiple files are requested, this method creates separate threads to download them in parallel.
- `download(self, file_name, upload_address, position=0)`: This method downloads a file from another peer using a provided file name and upload address. It includes handling progress and displaying a progress bar.
- `respond_ping(self)`: This method responds to the `ping` message from the server by sending an acknowledgment message `I'm online`.
- `respond_discover(self)`: This method responds to the `discover` message from the server by sending a list of files in the client's repository.
- `close(self)`: This method closes all sockets and notifies the server about the client's intention to close.

Besides, we also have 2 support function (not really related to client, that's why we don't put it in the `client` class)

- `recv_timeout(socket: socket.socket, recv_size_byte, timeout=2)`: Set a threshold for the server to send message to client indicating that server is still alive. After 3 second, if no respond is received, then client automatically break the connection
- `send_timeout(socket: socket.socket, data: bytearray, timeout=2)`: Set a threshold for the client to send message to server indicating that client is still alive. After 3 second, if socket of client doesn't send a confirming message, then connection is lost



## Server side

- `__init__(self, host='192.168.1.8', port=50004)`: This is the constructor of the Server class. It initializes a server socket and sets it up for listening on the provided host and port. It also starts two threads for handling client connections and processing commands.
- `start(self)`: This method starts the server by launching the listening thread and enters a command processing loop.
- `serve_forever(self)`: This method listens for incoming client connections. It accepts clients, extracts necessary information, and spawns threads to serve these clients.
- `serve_client(self, client_socket, client_address)`: This method listens for client commands and responds accordingly. It processes commands like `fetch`, `publish`, `update`, and `close`.
- `remove_client(self, client_address)`: This method is called to remove a client from the server. It acknowledges the client's request and removes the client's information.
- `respond_update(self, client_address, file_names)`: This method is called to update the list of files available for a client. It updates the list of files for a specific client.
- `respond_fetch(self, client_address, file_names)`: This method responds to a `fetch` command by providing a list of available peers to download files from. The response includes the IP and port information of available peers for each requested file.
- `respond_publish(self, client_address, repo_file_name)`: This method processes the client's `publish` command and sends a response to acknowledge the client. It updates the list of files for the corresponding client.
- `close(self)`: This method closes the server socket.
- `cmd_forever(self)`: This method listens for user input and processes valid commands, including 'ping' and discover
- `ping(self, IP, port)`: This method pings a client to check its availability and response time. It sends a `ping` command to the client and measures the response time.
- `discover(self, IP, port)`: This method sends a `discover` command to a client to retrieve a list of files in the client's repository.

Besides, we also have 2 support function (not really related to server, that's why we don't put it in the `server` class) **recv: means receive**

- `recv_timeout(socket: socket.socket, recv_size_byte, timeout=2)`: Set a threshold for the client to send message to server indicating that client is still alive. After 3 second, if no respond is received, then server automatically pop client information (meaning the `ClientInfo`) out of the client list.
- `send_timeout(socket: socket.socket, data: bytearray, timeout=2)`: Set a threshold for the server to send message to client indicating that server is still alive. After 3 second, if socket of server doesn't send a confirming message, then connection is lost

**NOTE:** To make it easier for tracking clients connecting to the server, we created `ClientInfo` class - a supporting class for storing client-related information, belonging to the server side. It includes attributes and methods related to a specific client:

- `__init__(self, identifying_address=None, identifying_sock=None, sending_address=None, sending_sock=None, upload_address=None, listening_thread=None, files=None)`: The constructor for `ClientInfo` initializes attributes to store information about a client, such as IP addresses, sockets, and file information.
- `set_info(self, identifying_address, identifying_sock, sending_address, sending_sock, upload_address, listening_thread, files)`: This method allows you to set or update the information associated with a `ClientInfo` object.
- `get_sending_sock(self) -> socket.socket`: This method returns the sending socket of the client.
- `get_identifying_sock(self) -> socket.socket`: This method returns the identifying socket of the client.
- `get_listen_keep_alive_sock(self)`: Return the socket that is used to check if client is still alive
- `get_upload_addr(self) -> str`: This method returns the upload address of the client.
- `get_files(self) -> list[str]`: This method returns the list of files available in the client's repository.

### 2.3.2 Communication protocols used for each function

For the whole system, we use `socket` library of Python programming language for implementation of each functions.

#### Client side

To begin connecting with server, client will create 4 socket connections:

- `server_listen_sock`: Socket to listen to server messages
- `server_send_sock`: Socket to send messages to the server
- `upload_sock`: Upload address (listen forever for upload requests)
- `send_keep_alive_sock`: Socket that sends message that the client is still alive

We define the protocol for client to establish a socket connection for sending messages to server as follow:

1. Client send connect request to server

```
1 self.server_send_sock.connect((self.host, self.port))
```

2. Then, client send a first message to server, indicating first-time-connect.

```
1 self.server_send_sock.send('first'.encode())
```

For client to receive request from server:

```
1 # second connect to server
2 self.server_listen_sock.connect((self.host, self.port))
```

Now, in order for server to actually get all the information of client, we have to send a series of information including address of client used for sending messages to server, address of client used for uploading data to other peers, and then a list of files from client repository (server will be able to keep all information of client repository files).

```
1 dir_list = os.listdir("repo")
2 send_data = self.server_send_sock.getsockname()[0] + ' ' + str(self.
    server_send_sock.getsockname()[1]) + \
3         ' ' + self.upload_sock.getsockname()[0] + ' ' + str
    (self.upload_sock.getsockname()[1]) + \
4         ' ' + ' '.join(dir_list)
5 self.server_listen_sock.send(send_data.encode())
```

After that, the client will also use the "keep-alive" socket to connect to the server, then sends the information indicating which first-time-connect this keep alive this socket belongs to.

```
1 self.send_keep_alive_sock.connect((self.server_host, self.server_port))
2 self.send_keep_alive_sock.send(('keepalive ' + self.
    server_send_sock.getsockname()[0] + ' ' + str(self.
    server_send_sock.getsockname()[1])).encode())
```

- For publish command (or function), we send to server the name of the repository (after renamed from local folder)

```
1 self.server_send_sock.send(('publish ' + repo_file_name).encode())
```

Then, it waits for server to respond "Success".

```
1 data = ''
2 while not data:
3     data = self.server_send_sock.recv(1024).decode()
```

- For `fetch` command, it's more complicated. The overall sequence diagram of fetching a file can be represented as follow:

[https://drive.google.com/file/d/1HDrwn06\\_-JTrEuLv0-XUFq22g2rUQ-jm/view?usp=sharing](https://drive.google.com/file/d/1HDrwn06_-JTrEuLv0-XUFq22g2rUQ-jm/view?usp=sharing)

Client will send a `fetch` command along with file name(s) it wants to fetch from other peers. Server will search for peers containing requested files from other clients. In the case that there is already a file having the same name with the file user tends to fetch, we will bypass it (filtering)

```
1 # filter
2 filenames = [filename for filename in filenames if filename not in
3               os.listdir("repo")]
4 if len(filenames) == 0:
5     print('All files are already in your repository!')
6     return
7 file_to_addrs: Dict[str, List[str]] = {}
8 for filename in filenames:
9     fetch_cmd = 'fetch ' + filename
10    self.server_send_sock.send(fetch_cmd.encode())
11    data = ''
12    data = recv_timeout(self.server_send_sock, 8000, 20)
13    if len(data) == 0 or data == None:
14        print('Server is offline, no response!')
15        return
16    data = data.decode()
17
18    if data != 'null null':
19        addresses = data.split()
20        addresses = [addresses[n:n + 2] for n in range(0, len(
21                    addresses), 2)]
22        file_to_addrs[filename] = addresses
```

Server will send back a list of available peers for each file to client. Client will now looping through the addresses one at a time, to see if it can successfully download the file with that address. If yes, then it just fetch the file from that client, otherwise it keeps finding other address based on the return address list.

```
1 download_threads = []
2 for i, (filename, addrs) in enumerate(file_to_addrs.items()):
3     download_threads.append(threading.Thread(target=self.
4         handle_download, args=[filename, addrs, i], daemon=True))
5     download_threads[i].start()
6 for thread in download_threads:
7     thread.join()
```

- With `listen_server` function, this acts as a responding function for replying server commands (`ping` and `discover`). If client receives `ping`, it responds with a message `I'm online`

```
1 self.server_listen_sock.sendall('I\'m online'.encode())
```

indicating that client is in ready state with server. If the received message is `discover`, client will send a list of files in repository folder to server.

- With `listen_upload` function, this will first make connection with the peer which is requesting the file. Then, it receive the file's name and passed arguments to the `upload` function to send the requested file to the other peer.

```
1 while True:
2     try:
3         download_socket, _ = self.upload_sock.accept()
4     except Exception as e:
5         break
6     request_file_and_offset = ''
7     while not request_file_and_offset:
8         request_file_and_offset = download_socket.recv(1024).decode()
9     request_file_and_offset = request_file_and_offset.split()
10    request_file = request_file_and_offset[0]
11    request_offset = int(request_file_and_offset[1])
12    thread = threading.Thread(target=self.upload, args=(
13        request_file, request_offset, download_socket), daemon=True)
14    thread.start()
```

- For upload and download functions, these two functions support downloading and uploading files for communicating between peers. When 2 clients are connected, the client containing the requested file will first send the file size to the other peer. After that, it starts sending the main file.

– Upload function:

1. Open the file from **repo/**, read the file size
2. Send the file size to the peer
3. Wait for peer's "ready" message
4. Read the file from offset (for resume downloading), and send the data to the peer

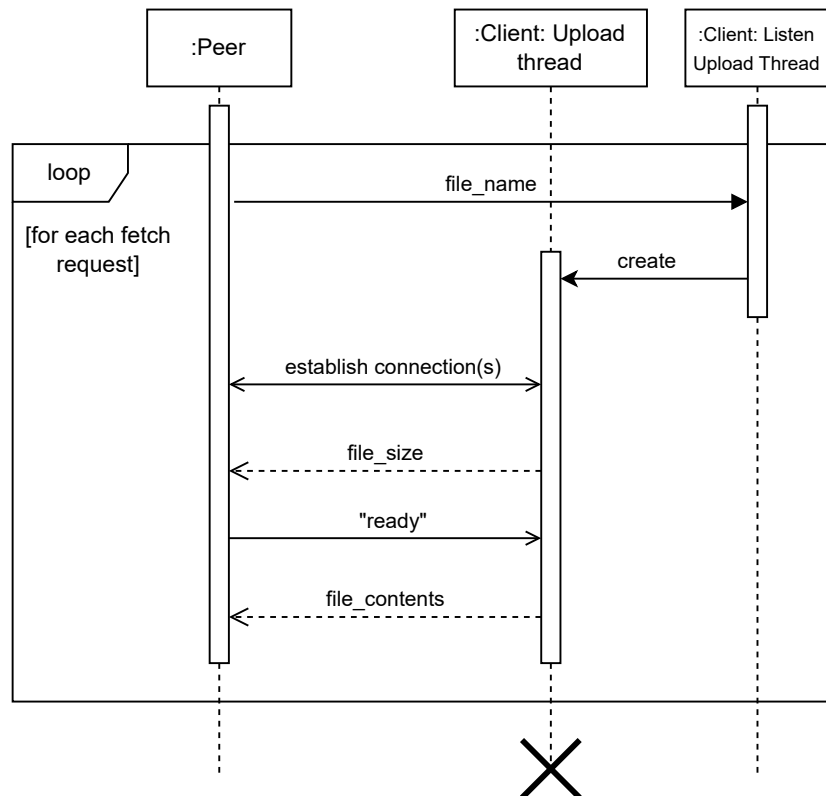
If any error occurs in the above process, **upload** ends

– Download function:

1. Establish connection to the peer holding the file
2. Send the file name and file offset to the peer (file offset is 0 for complete download, any positive number for resuming download)
3. Receive the file size from peer
4. Send "ready" to peer
5. Read the file content from peer and write it to the **temp/** folder
6. If download is fully successful, copy the file from **temp/** to **repo/**. Otherwise leave it in **temp/** and record the partially downloaded information (for resuming download)

If any error occurs in the above process, **download** ends

Overall process can be presented as follow:



**Figure 2.2:** Upload/Download protocols

- For receiving discover command from server, the response will be a list of files in the repository folder of client:

```

1 dir_list = os.listdir("repo")
2 self.server_listen_sock.sendall(' '.join(dir_list).encode())

```

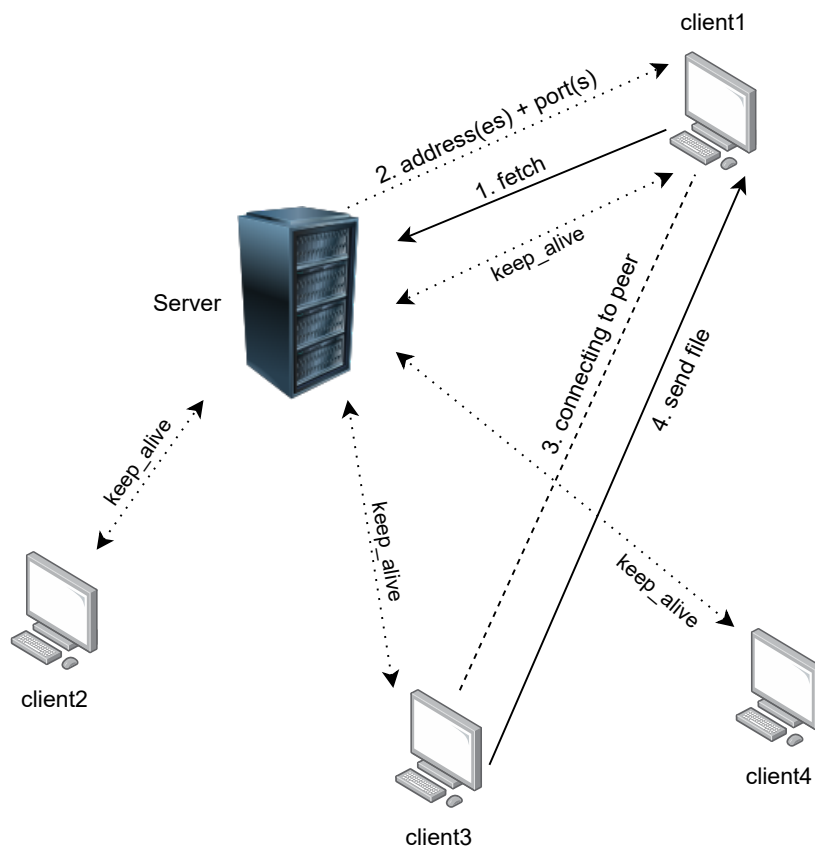
- When client want to close connection, it will send a `close` message to the server, indicating that it wants to terminate current connection (meaning socket object). Also, it will terminate all other socket connections:

```

1 self.server_send_sock.settimeout(5)
2 try:
3     self.server_send_sock.send('close'.encode())
4 except Exception as e:
5     print('Server is offline at shutdown!')
6     self.close_sockets()
7     return
8
9 response = recv_timeout(self.server_send_sock, 1024, 10)
10 if len(response) == 0 or response == None:
11     print('Server is offline at shutdown!')
12 elif response.decode() == 'done':
13     print('Server response: ' + response.decode())
14 self.close_sockets()

```

Overall:



**Figure 2.3:** Client-Server activity and Client-Client activity through fetch function

### Server side

We only initialize 1 socket for connecting with peer as follow:

```
1 client_socket, client_address = self.sock.accept()
2 data = ''
3 while True:
4     data = client_socket.recv(1024).decode()
5     if data != '':
6         break
```

The first time client connects to server (this will be indicated by a **first** message), server will record this client's information into the data structure. Just like the client, server will wait for client command.

- For server to know which files the connecting clients are keeping, the **respond\_time** is created to appending a new file to current repository files' information
- To ping clients, server will first get the **sending socket of client**. Then, it uses this socket to contact to client.

```
1 address = (IP, port)
2 if address in self.client_infos.keys():
3     # retrieve the client info
4     client_info: ClientInfo = self.client_infos[address]
5     # send ping to sending_socket
6     try:
7         client_info.get_sending_sock().send('ping'.encode())
8     except Exception as e:
9         print('Client is offline')
10        return
11    start = time.time()
12    data = recv_timeout(client_info.get_sending_sock(), 1024, 5)
13    end = time.time()
14
15    if len(data) == 0 or data == None:
16        print('Request timed out')
17        self.client_infos.pop(address)
18        return
19
20    latency = end - start
21    data = data.decode()
22
23    print(f"Response latency: {latency*1000:0.3f} ms")
24    print(data)
25
26 else:
27     print('Client is offline')
```

- To discover file information that a client is keeping in its own repository folder, server send **discover** message to client and wait for its response. The response will be a list of files

```
1 address = (IP, port)
2 if address in self.client_infos.keys():
3     # retrieve the client info
4     client_info: ClientInfo = self.client_infos[address]
5     # send ping to sending_socket
6     try:
7         client_info.get_sending_sock().send('discover'.encode())
8     except Exception as e:
9         print('Client is offline')
10        self.client_infos.pop(address)
11        return
12    # wait for response
13    response = recv_timeout(client_info.get_sending_sock(), 8000,
14                            10)
15    if len(response) == 0 or response == None:
16        print('Request timed out')
17        self.client_infos.pop(address)
18        return
19    response = response.decode().split()
20    print('-----')
21    for file in response:
22        print(file)
23    print('-----')
24 else:
25     print('Client is offline')
```

## Both side

To make sure that the connection is active, we also created a keep-alive method, which works like a ping pong game as follow:

- **Client:** For every time interval, **send** a "keepalive" message to the server with a timeout. If timeout exceeded without being able to **send** the message, connection is deemed lost and close the application. If **sent** the message successfully, wait for the server response. If timeout exceeded without being able to receive the message, connection is deemed lost and **close the application**
- **Server:** For every time interval, **receive** a "keepalive" message with a timeout. If timeout exceeded without being able to **receive** the message, connection is deemed lost and remove the client's info and **free any resources associated with the client**. If **receive** the message successfully, send the response to the client with a timeout. If timeout exceeded without being able to receive the message, connection is deemed lost and remove the client's info and **free any resources associated with the client**

But on client's side, closing the application after losing connection to the server is sometime undesirable, especially when uploading/downloading files. **Thus the client can only close when it is not doing any file transferring activity.**

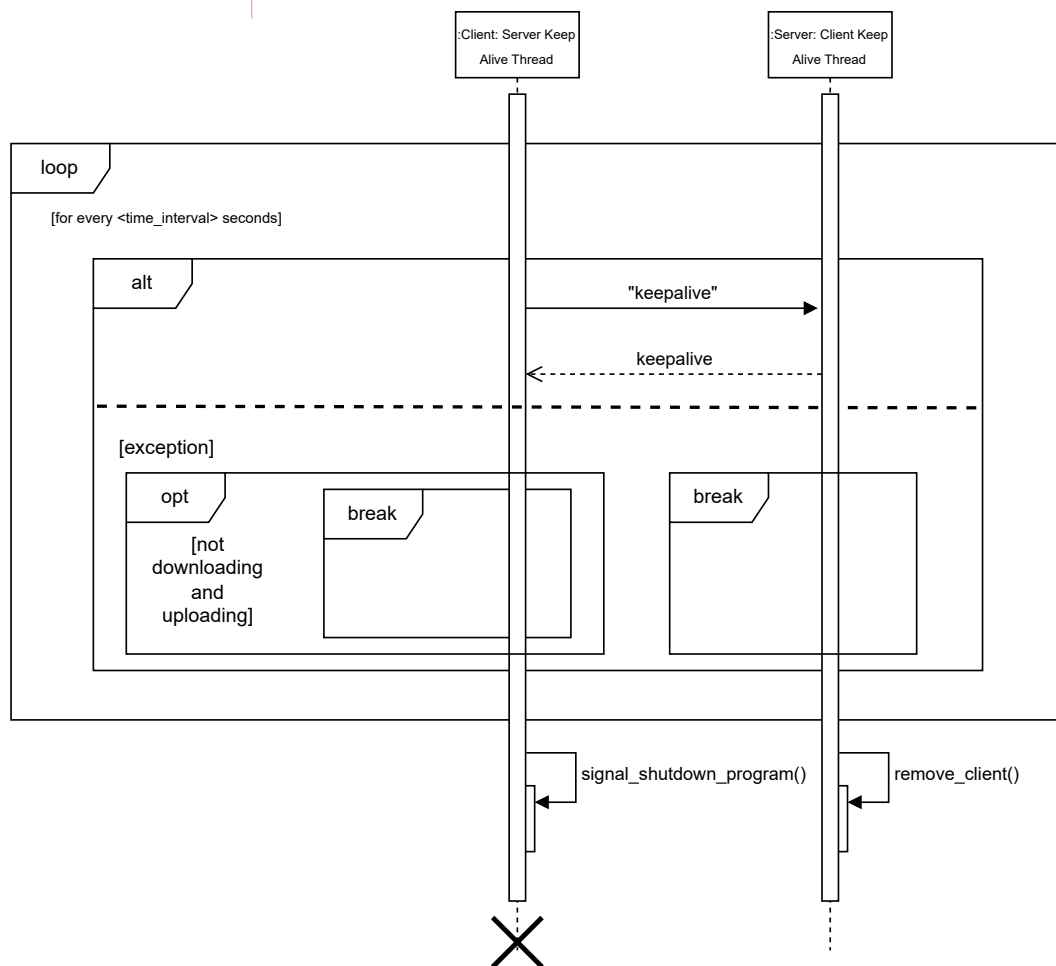


Figure 2.4: Keep-alive protocol

### 2.3.3 Extension functions

#### Client side

- **close:** Close connection with server if user wanted to **and close the client side**



```

1 self.server_send_sock.settimeout(5)
2 try:
3     self.server_send_sock.send('close'.encode())
4 except Exception as e:
5     print('Server is offline at shutdown!')
6     self.close_sockets()
7     return
8
9 response = recv_timeout(self.server_send_sock, 1024, 10)
10 if len(response) == 0 or response == None:
11     print('Server is offline at shutdown!')
12 elif response.decode() == 'done':
13     print('Server response: ' + response.decode())
14 self.close_sockets()

```

- list: List out current files in repository folder of client

```

1 dir_list = os.listdir("repo")
2 print('-----')
3 for file in dir_list:
4     print(file)
5
6 print('-----')
7 print()

```

## Server side

For server side, we add a command list to list out all of clients connecting to server

```

1 for client_address in self.client_infos.keys():
2     print(str(client_address))

```

### 2.3.4 Layered architecture

#### Client side

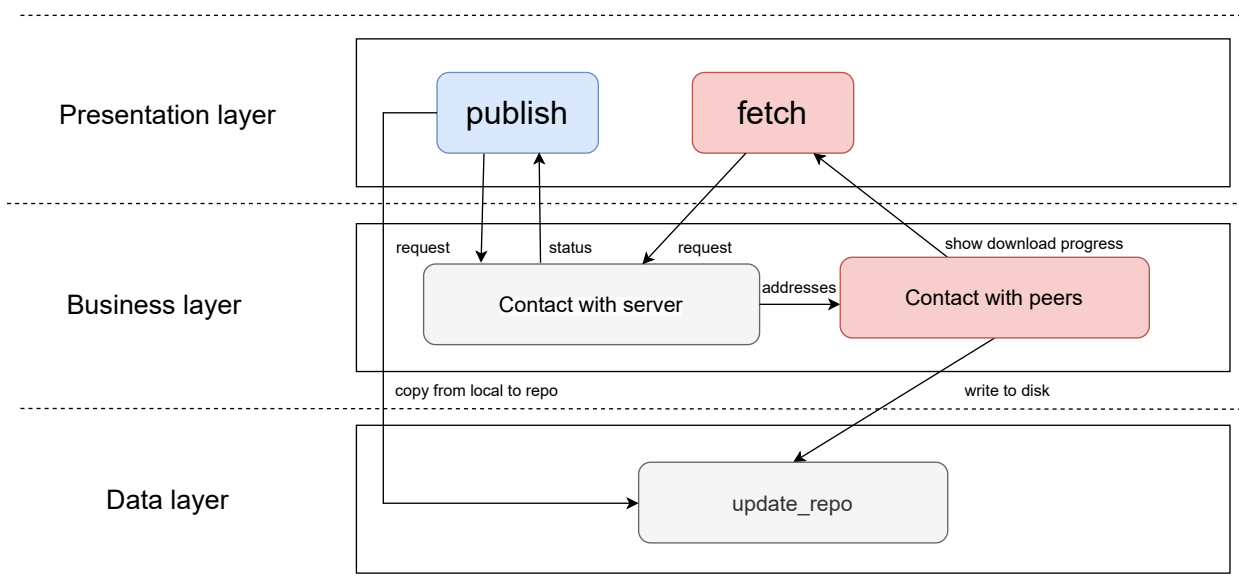


Figure 2.5: Layered architecture of client

## Server side

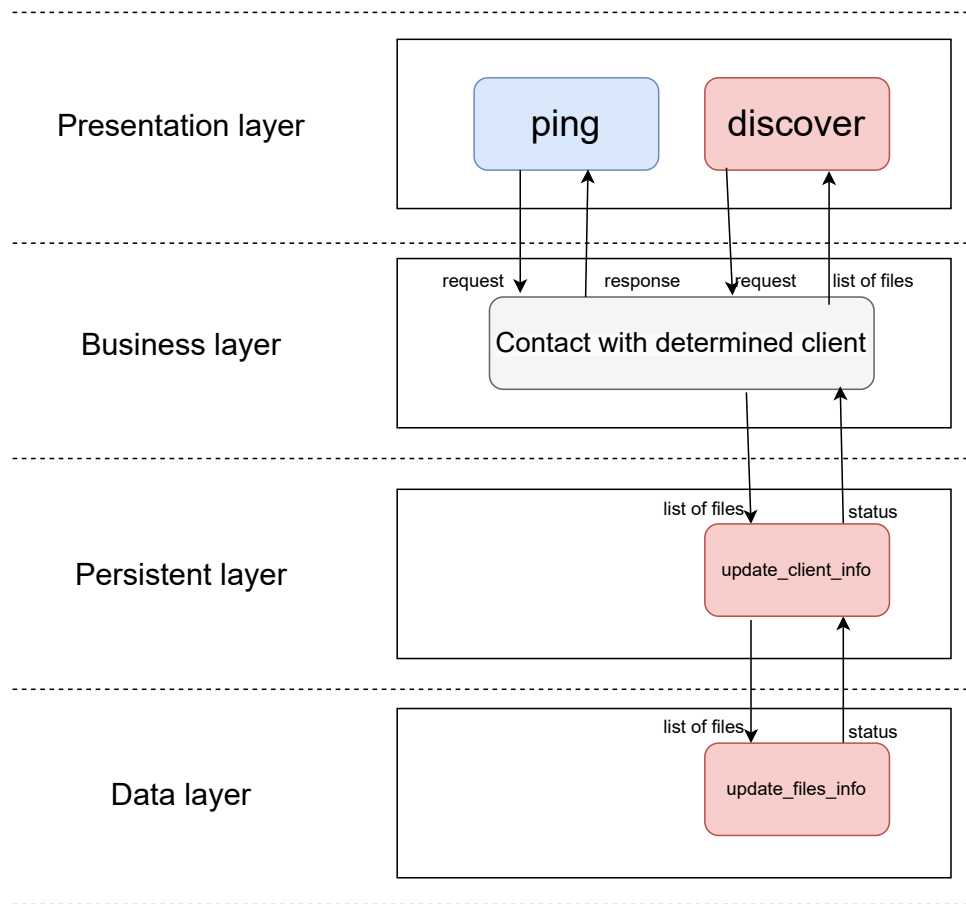


Figure 2.6: Layered architecture of server

## 2.4 Phase 2

Our code has been published here: <https://github.com/nhatkhangcs/231-computer-network-assignment1.git>

### 2.4.1 Manual document

Our document has been published here for your convenience: <https://github.com/nhatkhangcs/231-computer-network-assignment1#readme>

### Prerequisites

Make sure you have Python installed on your system.

### Running the application

- Clone the repository.

```
1 git clone https://github.com/nhatkhangcs/231-computer-network-assignment1.git
```

- Navigate to the project directory.

```
1 cd 231-computer-network-assignment1
```

- Configure the IP and port that the server bind to and ensure that the IP and port that the client connect to matches that server IP and port, configure also the IP that the client upload socket bind to so that they can be reached from other clients for fetching files

- Run the server in folder server.

```
1 python server.py
```

- Run the server in folder client.

```
1 python client.py
```

**NOTE:** If you want to test for multiple clients, navigate to folder `test_client`. Here you can see list of `client<x>` folders with same structure as the client folder. The operation will be the same as above.

## 2.4.2 Basic sanity test

Please note that some of the test below are performed in `localhost` environment due to the chance of our group meeting in person is too scarce to plan an extensive test and document them in a detailed manner. When we had a chance to meet, those tests are performed successfully but not yet documented. But basically the logic are all the same, and we will demonstrate this in real life environment on Demo Day.

### Test server commands

Suppose that the original state of server and client is:

```
(khangEnv) PS C:\Users\Admin\Desktop\Khang\231-computer-network-assignment1\server>
python server.py
Listening on localhost 50004
>> █
```

Figure 2.7: Server fresh state

```
(khangEnv) PS C:\Users\Admin\Desktop\Khang\231-computer-network-assignment1\client>
python client.py
Sending address: 127.0.0.1 51183
Listening address: 127.0.0.1 51184
Upload address: 127.0.0.1 51182
>> █
```

Figure 2.8: Client fresh state

- Test ping command:

Test

Input (to server terminal): `ping 127.0.0.1 51183`

Result: PASSED

```
(khangEnv) PS C:\Users\Admin\Desktop\Khang\231-computer-network-assignment1\server>
python server.py
Listening on localhost 50004
>> ping 127.0.0.1 51183
Response latency: 0
I'm online
>> █
```

Figure 2.9: Output

- Test **discover** command:

Test

Current files in client's repo folder:

```
▼ repo
  ≡ 1MB.txt
  ≡ 32KB.txt
  ≡ send.txt
  ≡ sendFile.txt
  ≡ text.txt
  ≡ text1.txt
```

Figure 2.10: Current files

**Input** (to server terminal): `discover 127.0.0.1 51183`

Result: PASSED

```
>> discover 127.0.0.1 51183
1MB.txt
32KB.txt
send.txt
sendFile.txt
text.txt
text1.txt
>> █
```

Figure 2.11: Output

### Test client commands

Suppose that the original state of server and client is:

```
(khangEnv) PS C:\Users\Admin\Desktop\Khang\231-computer-network-assignment1\server>python server.py
Listening on localhost 50004
>> █
```

Figure 2.12: Server fresh state

```
(khangEnv) PS C:\Users\Admin\Desktop\Khang\231-computer-network-assignment1\client>python client.py
Sending address: 127.0.0.1 52321
Listening address: 127.0.0.1 52322
Upload address: 127.0.0.1 52320
>> 
```

**Figure 2.13:** Client fresh state

- Test **publish** command:

Test

Current state of local folder and repo folder:

▼ local

≡ send.txt

▼ repo

≡ 1MB.txt

≡ 32KB.txt

≡ send.txt

≡ sendFile.txt

≡ text.txt

≡ text1.txt

**Input** (to client terminal): `publish 127.0.0.1 52321`

**Figure 2.14:** Current state of folders

Result: PASSED

```
>> publish send.txt testSend.txt  
Server response: success  
  
>> █
```

Figure 2.15: Output

```
▼ local  
  ≡ send.txt  
▼ repo  
  ≡ 1MB.txt  
  ≡ 32KB.txt  
  ≡ send.txt  
  ≡ sendFile.txt  
  ≡ testSend.txt U  
  ≡ text.txt  
  ≡ text1.txt
```

Figure 2.16: Later state of folders

- Test **fetch** command:

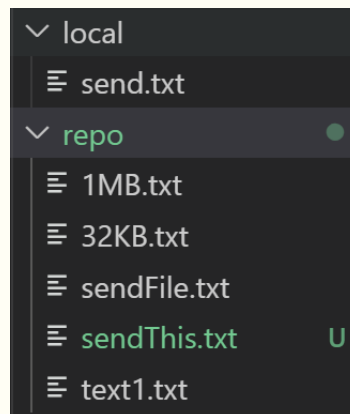
## Test

Suppose we have another client as follow:

```
(khangEnv) PS C:\Users\Admin\Desktop\Khang\231-computer-network-assignment1\test_client\client1> python client.py
Sending address: 127.0.0.1 52915
Listening address: 127.0.0.1 52916
Upload address: 127.0.0.1 52914
>> 
```

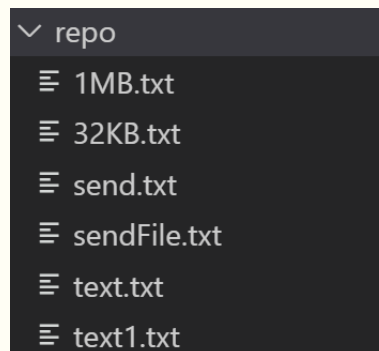
**Figure 2.17:** Another client appears

Its current folder state:



**Figure 2.18:** Current folder files

Current files in client's repo folder:



**Figure 2.19:** Current files

**Input** (to client terminal): `fetch sendThis.txt`

Result: PASSED

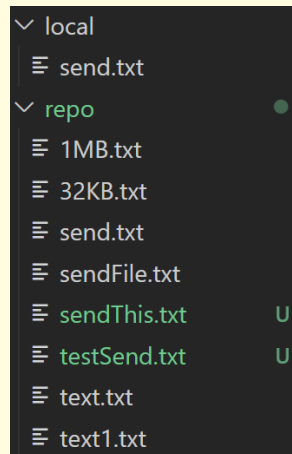


Figure 2.20: Later folder files

### 2.4.3 Advanced sanity test

#### Connection interruption from server

Test

Suppose the client is as follow:

```
(khangEnv) VO HOANG NHAT KHANG > client > server > ?1 ~2 > 3.11.6 >
1ms python client.py
Sending address: 127.0.0.1 60135
Listening address: 127.0.0.1 60136
Upload address: 127.0.0.1 60134
>> 
```

Figure 2.21: Client information

We also have server as follow:

```
(khangEnv) VO HOANG NHAT KHANG > server > server > ?1 ~2 > 3.11.6 >
3ms python server.py
Listening on localhost 50004
>> 
```

Figure 2.22: Server information

Now, during their connection, we forcefully kill server terminal and see if client acknowledge this



## Result: PASSED

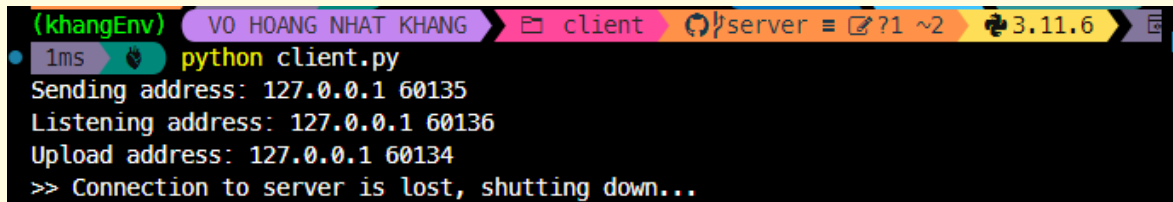
We corrupt the server side:



```
(khangEnv) VO HOANG NHAT KHANG > server > python server.py
Listening on localhost 50004
>> Program interrupted
(khangEnv) VO HOANG NHAT KHANG >
```

Figure 2.23: Server terminal killed

The client can acknowledge this without doing anything:



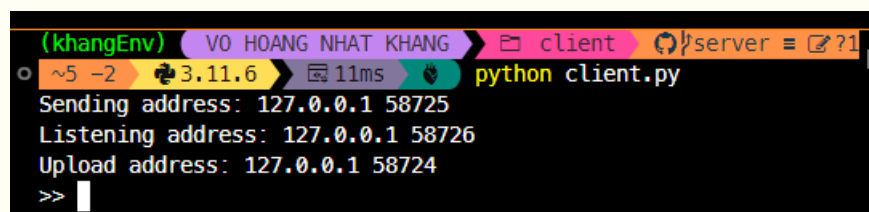
```
(khangEnv) VO HOANG NHAT KHANG > client > python client.py
Sending address: 127.0.0.1 60135
Listening address: 127.0.0.1 60136
Upload address: 127.0.0.1 60134
>> Connection to server is lost, shutting down..
```

Figure 2.24: Client acknowledges and automatically shut down

## Connection interruption from client

### Test

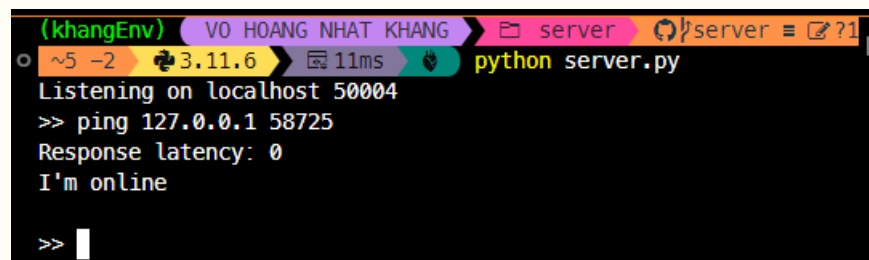
Suppose the client is as follow:



```
(khangEnv) VO HOANG NHAT KHANG > client > python client.py
Sending address: 127.0.0.1 58725
Listening address: 127.0.0.1 58726
Upload address: 127.0.0.1 58724
>>
```

Figure 2.25: Client information

We also have server as follow:



```
(khangEnv) VO HOANG NHAT KHANG > server > python server.py
Listening on localhost 50004
>> ping 127.0.0.1 58725
Response latency: 0
I'm online
>>
```

Figure 2.26: Server information

Now, during their connection, we forcefully kill client terminal and see if server acknowledge this:

## Result: PASSED

We corrupt the client side:

```
(khangEnv) VO HOANG NHAT KHANG > client /server = ?1
~5 -2 3.11.6 11ms python client.py
Sending address: 127.0.0.1 58725
Listening address: 127.0.0.1 58726
Upload address: 127.0.0.1 58724
>> Program interrupted!
Server response: done
```

Figure 2.27: Client terminal killed

The client can acknowledge this without doing anything:

```
(khangEnv) VO HOANG NHAT KHANG > server /server = ?1 ~5 -2 3.11.6
11ms python server.py
Listening on localhost 50004
>> ping 127.0.0.1 58725
Response latency: 0
I'm online

>> ping 127.0.0.1 58725
Client is offline

>> 
```

Figure 2.28: Server acknowledges

## Download progress corruption

S

uppose we want to fetch file 10MB.bin:

```
VO HOANG NHAT KHANG > client /server = ?1 ~6 -2 3.11.6 2m 38.699s python client.py
Sending address: 192.168.43.250 54156
Listening address: 192.168.43.250 54157
Upload address: 127.0.0.1 54155
>> fetch 10MB.bin
```

Figure 2.29: Trying to fetch file 10MB.bin

During the fetching process, turn off the Internet.

S

uppose we want to fetch file 10MB.bin:

```
VO HOANG NHAT KHANG > client /server = ?1 ~6 -2 3.11.6 2m 38.699s python client.py
Sending address: 192.168.43.250 54156
Listening address: 192.168.43.250 54157
Upload address: 127.0.0.1 54155
>> fetch 10MB.bin
Available peers for file 10MB.bin:
192.168.43.191 43047
10MB.bin from (192.168.43.191, 43047): 58% | 5.82M/10.0M [00:03<00:02, 1.65MB/s]
```

Figure 2.30: The download stop

The fetching progress stopped. After turning on the Internet (about 10s), the download continues and eventually finished.

#### 2.4.4 Evaluation

##### Connection establishment:

- Through out severe testing, we can be sure that client and server can establish connection within feasible connective environment. P2P connections (client-client) can also be established if required.
- Protocols in connections are executed correctly and as our expectation.

##### File publication and retrieval:

- Clients can publish files from local folder to local repository folder and server can acknowledge this.
- When establishing connection for file transferring, data is sent correctly. No data corruption detected.

##### Server functionality:

- Server is able to keep track of all clients connecting to it, and also can detect clients that are disconnected. Client information is stored correctly without any issues
- Server can discover clients for file transferring with protocols discussed.

##### Multithreading:

Clients can handle multiple sending files/receiving files task (by diving into threads) without any issues

##### Command-shell interpreter:

- Command-shell interpreter works without issues. All commands from user are received correctly.
- Syntax is clear enough for user to use and master.
- Downloading progress is showed correctly without any major issues

##### Error handling:

- Clients and servers act correctly when receiving invalid commands from user
- Exception and messages thrown are clear enough to understand the error

##### Performance metrics

- The time it takes to upload file to repository or to fetch a file is reasonable
- The time it takes to execute a command is less than 3 seconds (on average)
- Overall responsiveness: Good
- Overall efficiency: Good

##### Scalability:

- Server can handle multiple client connections and still correctly identifies connecting client and disconnected clients
- Clients can connect to multiple other peers and resolve sending/receiving files

##### Network latency:

- Network latency depends heavily on the connection speed of current system
- Under varying network conditions, the protocols executed correctly and exceptions are correctly handled

## Documentation:

Full documents have been published for the convenience of users

## Not yet implemented features:

- Store not-yet-downloaded information even after the client application is shutdown
- Client can search for available files to fetch
- While running, program acknowledge the changes when the user manually mutate the state of `local/`, `repo/`, and `temp/`
- Version control of files (to ensure the file with latest version is fetched)

## 2.5 Conclusion

The development of the file-sharing application with the defined application protocols has been successfully accomplished, utilizing the TCP/IP protocol stack. The application's architecture revolves around a centralized server that efficiently manages client connections and file information. Clients interact with the server by conveying details about their local repositories without transmitting the actual file data.

The key functionality of the application includes the ability for clients to request files not present in their repositories. In such cases, the server identifies potential sources, and the requesting client directly fetches the file from the selected source without server intervention. This decentralized approach enhances efficiency and minimizes server load.

To facilitate concurrent file transfers, the client code has been implemented as multithreaded, allowing multiple clients to download different files from a target client simultaneously. This design choice promotes optimal utilization of resources and improves the overall responsiveness of the application.

The client-side functionality includes a simple command-shell interpreter supporting two commands: `publish` to add local files to the repository and inform the server, and `fetch` to request and retrieve files from other clients. On the server side, the command-shell interpreter enables discovery of local files and live checking of client hosts.

The successful implementation of this file-sharing application showcases effective communication between clients and the server, demonstrating the robustness of the defined application protocols. The use of TCP/IP protocol stack ensures reliable and secure data transfer across the network. This project has provided valuable insights into the design and implementation of a distributed file-sharing system, addressing challenges such as decentralized file retrieval, multithreading, and command interpretation. The collaborative efforts in developing this application contribute to a comprehensive understanding of networked systems and application protocols.