COMPENG 4DN4
Laboratory 3
Online File Sharing Application

The objective of this lab is to develop client and server network applications that implement file sharing. The code will be written in Python 3 using the Berkeley/POSIX socket API discussed in class. The server software is run on a file sharing server and manages a directory that contains files to be shared. The client software communicates with the server in order to upload, list and retrieve the shared files. The Python code must implement packet broadcasting for service discovery and use execution threading so that the server can interact with multiple concurrent client connections. The working system has to be demonstrated by each team using either of the formats: live demo at MS Teams or screenshot-based report.

1 Description

The software to be developed consists of separate server and client applications written using Python 3. The required functionality is discussed as follows. Note that in the Marking Scheme section below, there is a description of output that the software must generate when doing the (live or screenshot) demonstration.

Note that the client-server communications will not work between different hosts of the team members unless you are using the same home network router. For this reason, most students will have to use different shell windows of the same computer for running the server and the clients.

Server: The server code is run on a server host (or a server window) and listens for UDP broadcasts and TCP connections from clients. The details are as follows.

1. The server is started from the command line in a shell window. It is configured with a file sharing directory location that will contain files that are shared among the clients.
2. After startup, the server continually listens for UDP packets on the file sharing Service Discovery Port (SDP), e.g., 30000.

   When a broadcast packet is received on the SDP that contains the message:

   `SERVICE DISCOVERY`

   the server responds to the client (using the received client source address and source port). The response contains the name of the file sharing service e.g., "Mel's File Sharing Service" (Choose a name that includes one or more of the team member's names).
3. Once the server is started, it also listens for incoming TCP client connections on the File Sharing Port (FSP), e.g., 30001. When a client connection occurs, the server accepts

commands from the client that involve listing, uploading and downloading shared files. The server must be able to handle a client connection at the same time while continuing to listen for service discovery broadcasts.

The client-to-server commands are sent from the client to the server. They are given as follows:

- `list` : The client wants the server to return a directory listing of the file sharing directory.
- `put <filename>` : The client wants to upload the file indicated. The uploaded file will be stored by the server in the file sharing directory using the file name `<filename>`.
- `get <filename>` : The client wants the server to download the indicated file.

The above commands must be communicated over the TCP connection to the server. A simple file sharing protocol must be defined for this purpose. It is convenient to encode each command transmitted by the client into the first byte(s) of the transmitted data. A good way to do this is to define a command dictionary that maps commands to the fixed length command values that are transmitted over the TCP connection, e.g.,

```
CMD = {
  "get" : 1,
  "put" : 2,
  ...
}
```

When the server receives an integer 1 in the first byte of a transmission from the client, it interprets it as the start of a "get" command, e.g., when a `get <filename>` is typed at the client prompt, the following client/server interaction would occur. First, the following command is sent to the server over the TCP connection:

```
-------------------------------------------
| get command (1 byte)  | ... file name ... |
-------------------------------------------
```

Here, we have assumed that commands are encoded into a single byte. The server is continuously listening for 1-byte commands. When this is received by the server, it recognizes the get command, i.e., `CMD["get"]` in the command byte, reads the file name, then replies with the following response:

```
-----------------------------------------
| file size (8 bytes) | ... file data ... |
-----------------------------------------
```

The client can then `recv` the 8-byte file size and do socket `recv` until all the file data has been downloaded. Download `file_transfer_protocol.py`. It provides a simple implementation for file downloading using the basic protocol outlined above.

Client: The client code is run on a client host (or client window) to access the server. The details are as follows.

The client is started from the command line in a shell window. It is configured with a local file sharing directory that may contain files for sharing. When the client starts, it presents a prompt to the user, and awaits commands. The commands are as follows:

- `scan`: The client transmits one or more `SERVICE DISCOVERY` broadcasts (sent to IP address `255.255.255.255`) and the SDP, and listens for file sharing server responses. When a service response is received, the client outputs this information on the command line. e.g., "Mel's File Sharing Service found at IP address/port IP address, port". If no responses are heard within a timeout period, it returns with a "No service found." message.
- `Connect <IP address> <port>` : Connect to the file sharing service at `<IP address>` `<port>`

  (You may chose to change the prompt when a connection has been established.)
- `llist` ("local list") : The client outputs a directory listing of its local file sharing directory.
- `rlist` ("remote list") : The client sends a `list` command to the server to obtain a file sharing directory listing. The remote listing is output to the user.
- `put <filename>` : Upload the file `<filename>` by issuing a `put` command to the server.
- `get <filename>` : Get `<filename>` by issuing a `get` command to the server, who will then respond with the file. The file will be saved locally.
- `bye` : Close the current the server connection.

2 Requirements

Teams: You can work in teams of up to 4 students.

Demonstration: Each team can choose ONE of the following two formats.

[1] Live demo. Each team should reserve a 20 minute time slot in one of the designated lab sessions. Time slots will be assigned on a first-come-first-served basis using the Doodle scheduler. All team members have to show up. Make sure you are certain of the team's schedule, since once a time slot is booked, it cannot be changed. Further details of the lab demo sign-up are on the course web site page for Laboratory 2.

The team member who will represent the team for the demo will share his/her computer screen with the TA (and other team members), and follow the "Marking Scheme" below to demo your work. The team may have different members to demo different parts of the work. However, only one person can present at a time because you have to demo the client(s) and server window at the same time.

[2] Screenshots. There is no need to reserve a live-demo time slot if you choose this option. Instead, you should submit a report that includes screenshots for each step as required by the "Marking Scheme" below and a brief description for each screenshot.

Marking Scheme: The assigned mark consists of two parts, i.e., 90% for demonstration (live or screenshot-based report) and a 10% for implementation (your Python code and related description, check items 1 and 3 in the Writeup below).

The demonstration consists of the following, where each step below is weighted equally in the demonstration component of the mark.

1. Start the server in a shell window. The server should output the shared directory files that are initially available for sharing.
2. The server should print output indicating that it is listening on the host SDP for incoming service discovery messages on the SDP, e.g., "Listening for service discovery messages on SDP port `<port number>`."
3. The server should print output indicating that it is listening on the host FSP for incoming connections on a particular TCP port, e.g., "Listening for file sharing connections on port `<port number>`."
4. Start the client in a new shell window of the same host (if you run the client and server on the same computer) or a shell window on another host (if you have another computer at home). The client will prompt the user for commands. The user issues a `scan` command and the client should find and report the server's availability.
5. The user issues an `llist` command. The client outputs a listing of its local file sharing directory.
6. The user issues a `connect` command and connects to the server. The server should output a status line indicating that a TCP connection has been established, e.g., "Connection received from `<IP address>` on port `<port>`."
7. The user issues an `rlist` command. The client outputs a listing of the remove file sharing directory obtained from the server.
8. The user issues a `put` command, to upload a file to the server. This is followed by an `rlist` command to show that the file was uploaded. The system should be able to transfer any file type, e.g., text, music, video, image, etc.
9. The user issues a `get` command, to download a file from the server. This is followed by an `llist` command to show that the file was downloaded.
10. The group should show that the server can interact with multiple concurrent client connections.
11. The user issues a `bye` command. The server should output that the connection had been closed.
12. The group must show that if the server exits during a file upload, there will be no partial file remnant remaining in the shared directory.

Writeup: You must upload the following documents to your Avenue To Learn Lab 3 dropbox:

1. A PDF report that briefly describes how you implemented the client and server applications. Make sure that the names and student ID numbers are on the front page of the report.
2. Screenshots and related descriptions based on the lab requirements for teams doing screenshot-based demo. This can be in the same file as item 1 of the Writeup or a separate file. For teams doing live demo, this is not needed.
3. A single Python source code (py) file that includes both the client and server classes that you created.