Computer Engineering 4DN4
Lab #3
Online File Sharing Network Application

The objective of this lab is to develop a client/server network application that implements file sharing. The code will be written in Python 3 using the Berkeley 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 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 in one of the laboratory sign-up sessions.

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 demonstration.

Server: The server code is run on a server host and awaits 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 broadcast packets, i.e., IP address
   `255.255.255.255`
   on the file sharing Service Discovery Port (SDP), 30000 (all groups should use this same port so that you can discover other group's services).
   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). Note that the broadcasts will likely only work when the hosts are on the same IP subnet.
3. Once the server is started, it also listens for incoming TCP client connections on the File Sharing Port (FSP), 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 multiple client connections at the same time while continuing to listen for service discovery broadcasts.

The client-to-server commands are as follows:

- `list` : The server returns a directory listing of the file sharing directory.
- `put <filename>` : The server responds with an `ok` message. The client that issued the `put` then uploads the file. This file is stored in the file sharing directory using the file name `<filename>`.
- `get <filename>` : The server responds by sending the file.
- `bye` : The server closes the client connection.

Client: The client code is run on a client host 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, 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 had been established.)
- `llist` ("local list") : The client gives 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. When the server responds with an `ok` message, the client sends the file over the connection.
- `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 3 students.

Demonstration: The working system must be demonstrated by the team, with all members in attendance. If the team size is 2 or more, the system will be demonstrated on two of the team's laptops, with the server running on one, and multiple clients running on the other. A broadband router will be used in the lab demo sessions through which the laptops can communicate. If the team size is 1 student, the system can be demonstrated using a single laptop.

Each team needs to immediately 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. 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 3.

Marking Scheme: The assigned mark consists of two parts, i.e., 90% for demonstrating the system and a 10% discretionary component for questions answered, based on your Python code, during the demonstration. 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 laptop 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 laptop 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 shell window on the client laptop. 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 also upload a report on your application to Avenue To Learn. The report should provide listings of your code and briefly describe how you implemented everything.