FTP Client Program

The role of the FTP client program is to provide an interface that allows a human user to enter high-level requests, which get validated before generating the appropriate FTP protocol commands that will be sent to the server to fulfill the user's request. Parts of the FTP client program are provided to you for this assignment. Specifically, the two programs FTP_Client.py and FTP_ReplyParser.py are intended to provide the base functionality for what the final client will ultimately support. At a high level, FTP_Client.py takes in user commands from stdin, parses these commands, and generates the appropriate responses that will eventually be sent to the server. On the other hand, FTP_ReplyParser.py validates server responses and outputs the correct information back to the user. Since each of these programs currently read from stdin, they have been provided in two separate files for simplicity. Your job in this assignment is to merge the two programs into FTP_Client.py, and extend the functionality to include socket programming that will facilitate communication with an FTP server.

The FTP client program takes one command line argument: an initial port number for a listening socket that the client uses to allow the server to make an *FTP-data* connection. Only three types of requests will be accepted from the user, a connect request (CONNECT), a get request (GET), and a quit request (QUIT). The specification of the input format for these requests, in our usual BNF-like notation is:

```
CONNECT<SP>+<server-host><SP>+<server-port><EOL>
GET<SP>+<pathname><EOL>
QUIT<EOL>

    <server-host> ::= <domain>
         <domain> ::= <element> | <element>"."<domain>
        <element> ::= <a><let-dig-hyp-str>
<let-dig-hyp-str> ::= <let-dig-hyp> | <let-dig-hyp><let-dig-hyp-str>
   <let-dig-hype> ::= <a> | <d> | "-"
              <a> ::= One of the 52 alphabetic characters "A" through
                      "Z" in uppercase and "a" through "z" in lowercase
              <d> ::= One of the characters representing the ten
                      digits 0 through 9
    <server-port> ::= character representation of a decimal integer in
                      the range 0-65535 (09678 is not ok; 9678 is ok)
       <pathname> ::= <string>
         <string> ::= <char> | <char><string>
            <SP>+ ::= one or more space characters
           <char> ::= any one of the 128 ASCII characters
            <EOL> ::= the normal line termination character(s) for the
                      system environment where your program is running.
```

The client echos each user input line to standard output along with the corresponding commands specified in the section titled **Processing user requests.** The generated commands then get sent to the server, which processes the commands and fulfills the user's request.

A CONNECT request must be the first user input accepted by the client — this is without exception, not even a valid QUIT request can appear before a valid CONNECT request. The user may also input a new CONNECT request at any time. When a valid CONNECT request is accepted from the user, the client

attempts to establish a TCP socket connection to the server program which it expects to be running on the host and port specified in the parameters of the CONNECT command. This connection is the ***FTP-control*** connection, and remains open until the user enters another valid CONNECT request (which will initiate a new FTP-control connection). When the client successfully connects to the server, it is prepared to receive the server's greeting reply ("220 COMP 431 FTP server ready.") on the FTP-control connection. *When the client program receives any reply from the server on the FTP-control connection, it displays the output specified in the section titled* ***FTP Reply Parsing***.

After the FTP-control connection has been established and the initial server reply received, the client sends the four-command sequence specified in **Processing user requests** (USER, PASS, SYST, and TYPE) to the server using the FTP-control connection. Before the client program sends a command to the server on the FTP-control connection, it will first echo that command to standard output. After each command in this sequence is sent to the server, the client will read and process the corresponding reply received from the server on the FTP-control connection before sending the next one. If the client is unable to connect to the server, it will write "CONNECT failed" to standard output and prepare to read the next input from the user.

If the user enters an additional valid CONNECT request, the existing FTP-control connection is closed and a new one established using the host and port specified in the CONNECT command.

When a valid GET request is input by the user, the client will send the two-command PORT-RETR sequence to the server on the FTP-control connection and process the server's reply to each command before proceeding to the next. The PORT command's parameter should specify the IP address of the host where the client program is running and the port number specified as a command line argument to the client. Each time a new PORT command is sent to the server, the port number should be incremented by 1 (NOTE: the client will be using a different listening socket and associated port for each FTP-data connection). Before sending the PORT command to the server, the client creates a *listening* socket specifying the port number used in the PORT command to be sure that port is ready for the server's FTP-data connection. If the listening socket cannot be created, the client writes "GET failed, FTP-data port not allocated." to standard output and reads the next user input line.

After the RETR command has been sent to the server, the client will accept the server's FTP-data connection on the listening socket and then read the bytes for the requested file from the new socket created for that connection. The client continues to read data bytes from the server until the End-of-File (EOF) is indicated when the server closes the FTP-data connection (the client will also close the FTP-data connection at EOF). Note that the server replies to the RETR command are received on the FTP-control connection. Only file data is received on the FTP-data connection.

The file data read from the FTP-data connection is then written to a new file in a directory named retr_files in the client's current working directory (be sure to create this directory using the UNIX "mkdir" command before you try to execute the client). The file's name in the retr_files directory has the form "filexxx" where xxx is the number of valid RETR commands the client has recognized so far during this execution of the program. For example, the first valid RETR command will result in writing the data to the file named file1 in the retr_files directory; the second valid RETR command will result in a file named file2,

etc. The file transferred with a RETR command may have arbitrary content and is thus written as a stream of bytes (using file I/O).

If the client receives an unexpected reply from the server or receives a reply that indicates an error condition (4xx or 5xx reply-code), the current sequence of commands to the server is ended and a new user input line read from standard input. A user's QUIT request is handled by sending the FTP QUIT command to the server, receiving the server reply shown below, closing the FTP-control connection and then terminating execution. The server reply to QUIT is

```
221 Goodbye
```

Processing user requests

For each valid CONNECT request, the client will reset any internal state (other than the number of copied files) to the initial program state and create the appropriate FTP protocol commands necessary for interactions with an FTP server program. The first line written to standard output following a CONNECT request simply provides a response to the user's request line. If the CONNECT request is valid, the response output is:

```
CONNECT accepted for FTP server at host <server-host> and port <server-port><EOL>
```

where `<server-host>` and `<server-port>` represent strings extracted from the user request. Following the response line, the program will then generate the following sequence of valid FTP commands and send them to the server program one-by-one:

```
USER anonymous<CRLF>
PASS guest@<CRLF>
SYST<CRLF>
TYPE I<CRLF>
```

Note that the `<username>` and `<password>` tokens generated by the program are restricted to the constant values "anonymous" and "guest@". The only form of login to be supported by the simple FTP client will be anonymous FTP for which no registered user name is required for access to files using the FTP server.

Once a valid CONNECT request is processed, the user may enter any number of GET requests, each of which indicates a file to be retrieved from the FTP server named in the most recent CONNECT request. If the GET request is valid, the response written to standard output is:

```
GET accepted for <pathname><EOL>
```

where `<pathname>` represents the string extracted from the user request. The client will then generate the following sequence of valid FTP commands and send them to the server program following the above response line:

```
PORT <host-port><CRLF>
RETR <pathname><CRLF>
```

In the generated commands above, the token <host-port> is defined by the BNF-like specification given in Homework 1. Those specifications are repeated here:

```
  <host-port>  ::= <host-address>","<port-number>
<host-address> ::= <number>","<number>","<number>","<number>
 <port-number> ::= <number>","<number>
     <number>  ::= character representation of a decimal integer
                   in the range 0-255
```

The <host-address> value that gets generated is to be the Internet address assigned to the host machine where the client is running. It can be obtained through the methods of "socket" module as illustrated in the following code fragment:

```
my_ip = socket.gethostbyname(socket.gethostname())
```

The string referenced by my_ip is a host address in the "dotted decimal" format described in Homework 1 (e.g., 152.2.129.144) and must be translated to the form specified above for <host-address>. The initial <port-number> value is the command-line argument passed to the client, and gets incremented by one after each PORT command is generated. The value is converted to the format specified above for <port-number> by doing the *inverse* computation corresponding to the conversion of port values specified in HW1, expressing the result as a character representation of decimal integers.

When the input line is a valid QUIT request, the client writes the following line to standard output:

```
QUIT accepted, terminating FTP client<EOL>
```

The program will then generate the following FTP command to send to the server following the above response line. The program will then terminate.

```
QUIT<CRLF>
```

NOTE: All commands that get sent to the server are also output on stdout on the client as well.

FTP Reply Parsing

The general format of FTP reply lines generated by the FTP server is provided below:

```
<reply-line>   ::= <reply-code><SP><reply-text><CRLF>
<reply-code>   ::= <reply-number>
<reply-number> ::= character representation of a decimal integer
                    in the range 100-599
<reply-text>   ::= <string>
```

The <reply-text> can be any text message that provides useful information concerning the outcome of processing an FTP command – for this assignment, the client assumes the reply text is consistent with the requirements of HW 1. Additionally, the definition of the <string> token here is to be taken from HW 1.

For each server reply that gets received, the client will:

- Echo the line of input (*i.e.*, print the line of input unchanged to standard output).
- For valid FTP replies, output on standard output the following line: "FTP reply `<reply-code>` accepted. Text is: `<reply-text><EOL>`" where `<reply-code>` and `<reply-text>` are extracted from the server reply
- For invalid replies, print out the error message "ERROR -- `<error-token>`" where `<error-token>` is the name of the token that is missing or ill-formed according to the specification for FTP replies.

  `<error-token> ::= "reply-code" | "reply-text" | "<CRLF>"`

Client Example

Here is an example showing how the client program's output should look with a correct sequence of valid user requests. Since the high-level client commands (CONNECT, GET, and QUIT) do not require CRLF, FTP_Client.py can be tested directly on the terminal/command prompt. **NOTE:** in this example the user's input lines that have been echoed are marked with a "-" to make the example clear. The "-" will not appear in the program output. The client was run by executing the command: "`python3 -u FTP_Client.py 8080`".

```
-CONNECT comp431-1sp21.cs.unc.edu 9000
CONNECT accepted for FTP server at host comp431-1sp21.cs.unc.edu and port 9000
FTP reply 220 accepted. Text is: COMP 431 FTP server ready.
USER anonymous
FTP reply 331 accepted. Text is: Guest access OK, send password.
PASS guest@
FTP reply 230 accepted. Text is: Guest login OK.
SYST
FTP reply 215 accepted. Text is: UNIX Type: L8.
TYPE I
FTP reply 200 accepted. Text is: Type set to I.
-GET pictures/jasleen.jpg
GET accepted for pictures/jasleen.jpg
PORT 152,2,131,205,31,144
FTP reply 200 accepted. Text is: Port command successful (152.2.131.205,8080).
RETR pictures/jasleen.jpg
FTP reply 150 accepted. Text is: File status okay.
FTP reply 250 accepted. Text is: Requested file action completed.
-QUIT
QUIT accepted, terminating FTP client
QUIT
FTP reply 221 accepted. Text is: Goodbye.
```

In addition to what is written on standard output, the client should also have copied the content of `pictures/jasleen.jpg` from the server to the file `retr_files/file1` in the working directory where the client is running.