COMP 431 — INTERNET SERVICES & PROTOCOLS

Spring 2025

Homework 1, January 9

Due: January 23, 12:15 PM

File Transfer Protocol (FTP), Client and Server – Step 1

The goal of the programming assignments (spanning more than the first half of the course) is to build a simple FTP server and client that will also interoperate with selected implementations of the standard Internet FTP service commonly used in this department. This assignment is the first piece of an FTP server – string-parsing and a basic engine to serve files. In a future assignment, you will extend this program to communicate with a remote client.

At a high-level, an FTP server is simply a program that receives commands from client programs, processes the commands, and sends the results of the processing back to the client as a response to the command. In this abstract view, an FTP server is a program that executes a logically infinite loop in which it receives and processes commands. In this assignment you will develop a portion of the code that will be used by the FTP server to process the commands it receives. A big focus of this assignment is to write code that can conform precisely to protocol specifications – often students (frustratingly) lose points for missing simple syntax and/or corner cases. This assignment needs careful attention to details! We *highly recommend* skimming through this writeup at least once before asking for clarification. This will also help you get a high level view of what you will implement.

You are to write a program to determine if a command received (a text string) is a valid FTP command. For some commands, additional actions also need to be performed. For example, the following is a valid FTP command:

```
USER jasleen
```

An FTP command is made up of three elements:

1. *command name* — e.g., the characters "USER". FTP command names are case insensitive, so any of "User", "user", "uSer", "UsEr", etc., will also work for the command name.

2. *command parameter* — e.g., the characters "jasleen", a variable-length sequence of characters. Not all commands require a parameter.

3. *"CRLF" terminator* — the command must be terminated by the carriage return-line feed character sequence "\r\n" (which is not visible above since these are non-printable characters).

All of these components must appear in the order listed. The command name and command parameter (if present) must be separated by one or more spaces. The "CRLF" immediately follows the command parameter (or command name if no parameter is used for the command).

The USER command is defined by the FTP protocol and serves to identify the user making a file request. Protocols such as FTP are typically specified more formally than the English description above by using a specification notation. These notations are, in essence, a textual form of the syntax diagrams — sometimes called "railroad diagrams" — that are used to specify the formal syntax of a programming language.

For example, the formal description of an FTP command is:[1]

```
<ftp-cmd> ::= <command>[<SP>+<parameter>]<CRLF>
```

The following is a subset of FTP commands that you will implement:

```
      USER<SP+><username><CRLF>
      PASS<SP+><password><CRLF>
      TYPE<SP+><type-code><CRLF>
      SYST<CRLF>
      NOOP<CRLF>
      QUIT<CRLF>
      PORT<SP+><host-port><CRLF>
      RETR<SP+><pathname><CRLF>

     <username> ::= <string>
     <password> ::= <string>
     <type-code> ::= "A" | "I"
     <pathname> ::= <string>
       <string> ::= <char> | <char><string>
           <CR> ::= the carriage return character
           <LF> ::= the line feed character
         <CRLF> ::= <CR> followed by <LF>
          <SP>+ ::= one or more space characters
         <char> ::= any one of the 128 ASCII characters except <CR> or <LF>
    <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
```

In this notation:

- Items appearing (in angle brackets) on the left-hand side of an expression are called *tokens*,
- Anything in quotes is interpreted as a string or character that must appear exactly as written,
- Anything in square brackets ("[," "]") is optional and is not required to be present,
- The vertical bar "|" is interpreted as a logical OR operator and indicates a choice between components that are mutually exclusive.

For example, the USER command above conforms to the formal description and hence is a valid FTP command (assuming it is terminated with a CRLF terminator). The following strings are examples that do *not* conform to the formal description and would be rejected as invalid or illegal requests.

```
USERjasleen
```

[1] As an aside, this form of notation is a variation of a commonly used notation called Backus-Naur Form (BNF). You will often see the syntax of protocols expressed using BNF and variations on BNF.

```
USR jasleen
USER jasle*en
```

The first and second requests contain an invalid command token and the third request contains an invalid user name (where the * represents *any* invalid byte value not one of the 128 ASCII characters – i.e., any byte value that is larger than 127).

Note: the username can have spaces within it (since space is part of 128 ASCII set) – but a username cannot start with the space character (since the <SP>+ captures all spaces till the first non-space character). Hence, "jasleen kaur" is a valid username.

The PORT and RETR Commands

Implementing the first six commands are only parsing exercises. To implement the PORT and RETR commands, you will need to perform a few more tasks in addition to parsing.

The PORT command is used to specify Internet addressing information consisting of a host IP address and port number (We will study the semantics of IP addresses and ports in Chapter 2. For this homework, we will introduce only the conventional syntax for representing them).

For example, the following is a valid PORT command:

```
PORT 152,2,131,205,31,144
```

The FTP syntax for host IP addresses and port numbers, unfortunately, differs from the conventional syntax and must be translated before it can be used in programs. Specifically, while a valid `<host-address>` in an FTP command must have commas (",") used to separate numbers, the form must be translated to have periods (".") as separators before it is used. Thus, the host address `152,2,131,205` must be translated to `152.2.131.205` for use in programs. The port address is even more idiosyncratic in FTP – the two numbers separated by a comma as in `31,144` are to be interpreted as decimal values contained in the high- and low-order bytes of a 16-bit binary representation of an integer in 2's complement. It must be converted into the corresponding decimal value. A simple algorithm for doing so is to multiply the value of the leftmost number by 256 and add the value of the rightmost number. For example, `31,144` is converted by the expression `(31 * 256) + 144 = 8080`.

The RETR command is used to specify the name of a file on the FTP server that is to be copied to the client. For your program, we will assume it specifies the name of a file located in the current working directory from which your program was executed or it specifies the complete pathname for a file located within the space of subdirectories below (rooted in) the current working directory. For example, the following is a valid RETR command:

```
RETR pictures/jasleen.jpg
```

It specifies the file `jasleen.jpg` located in the subdirectory `pictures` of the working directory used by your program. The syntax of valid RETR commands allows any of the 128 ASCII characters (other than <CR> and <LF>) to be specified in the `<pathname>` token. Because we will be treating this name as relative to the program's working directory, if "/" or "\" appears as the first (leftmost) character it should be deleted before the file name is used in your program. (NOTE: for testing your program you will need to have some test files and subdirectories present in the working directory used by your program.)

Once your program has determined that a RETR command is valid, it should "process" the command by copying the named file to the directory named `retr_files` in the program's current working directory.

The file's name in the `retr_files` directory should have the form "filexxx" where xxx is the number of valid RETR commands your program has recognized so far. For example, the first valid RETR command will result in copying the file named in the RETR command 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 named in a RETR command may have arbitrary content and should be treated simply as a stream of bytes (You can use the Python module `shutil` for copying files, or implement a file copy method using regular binary file reading and writing techniques).

This completes the minimum subset of FTP commands necessary to have the basic framework for a working FTP server. In addition, the FTP protocol defines certain constraints on allowable sequences of command/reply exchanges. Briefly stated, they are:

- A valid USER command must precede *any* other command.
- After a valid USER command is handled, a valid PASS command must precede *any other* (non-USER) command. And a valid USER command must precede a valid PASS command – in case multiple valid USER commands are input before a valid PASS command, the PASS command will be matched to the most recent USER command.
- A new valid PORT command must precede ***each*** RETR command (i.e., PORT and RETR commands must appear in pairs, possibly with other valid commands between them).
- Once a valid QUIT command is handled, no subsequent commands are allowed, and the program should terminate. A valid QUIT command is accepted at any point in time (even before a valid USER command).

Reply Lines

Your program should also produce valid FTP reply lines in response to commands. The format of FTP reply lines is:

```
      <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 the preceding command. For your program, **use the following replies formatted exactly as shown below to respond appropriately to input commands.** The blue comment following each reply specifies when it is to be used to reply to a command (do **NOT** include the comment in your output).

`150 File status okay.` // after RETR when the file can be successfully accessed
`200 Command OK.` // after any valid command that does not have a more specific reply (e.g., NOOP, QUIT)
`200 Type set to I.` // after a valid TYPE command with parameter I
`200 Type set to A.` // after a valid TYPE command with parameter A
`200 Port command successful (IP address,port number).` // after a valid PORT command.
NOTE: the IP address and port number in the reply output must be correctly translated as specified above. For example, the valid command:
`PORT 152,2,131,205,31,144`
Should result in the reply:

`200 Port command successful (152.2.131.205,8080).`

`215 UNIX Type: L8.` // after a valid SYST command

`220 COMP 431 FTP server ready.` // written to standard output as the first output from your program

`230 Guest login OK.` // after a valid PASS command

`250 Requested file action completed.` // after the file copy operation for a valid RETR command

NOTE: unlike the other commands, a valid and successfully completed RETR command results in two reply messages, for example:

`RETR pictures/jasleen.jpg`

Should result in the two replies (one when the file is accessed and another when the file copy is done):

```
150 File status okay.
250 Requested file action completed.
```

`331 Guest access OK, send password.` // after a valid USER command

`500 Syntax error, command unrecognized.` // after an input line with an invalid `<command>` token

`501 Syntax error in parameter.` // after an input line with an invalid `<parameter>` token or invalid `<CRLF>` termination

`503 Bad sequence of commands.` // after a valid command that is not in an allowed sequence (exception: see the 530 reply below).

`530 Not logged in.` // after any valid command that precedes a valid USER and PASS sequence. Takes precedence over 503.

`550 File not found or access denied.` // after RETR when the file can NOT be successfully accessed

The Assignment — Processing FTP commands in Python

For this assignment you are to write a Python program on Linux to read in lines of characters from standard input (*i.e.*, the keyboard) and determine which lines, if any, are legal FTP commands. We will provide starter code that implements the USER command. Your task is to implement the rest of the commands specified above based on the provided grammar. Go to the following URL to create your GitHub repository: https://classroom.github.com/a/ezEMUQB0. GitHub should ask you to join the GitHub Classroom, and will then create your repo with the starter code.

For each line of input your program should:

- Echo the line of input (*i.e.*, print the line of input exactly as it was input to standard output).
- Print the appropriate reply line(s). When an error is encountered, skip all input until the next closest <CRLF> or <CR> or <LF> (remember that both <CR> and <LF> produce a newline character).

All output should be written to standard output (*i.e.*, to the Linux window in which you entered the command(s) to execute your program). Your program should format its output *exactly* as shown above.

Your program should terminate when it reaches the end of the input file, or the end-of-file on stdin (when *Control-D* is typed from the keyboard under Linux), or when a valid QUIT command is received. Your program must not output any user prompts, debugging information, status messages, *etc.*

If a valid RETR command is received, then the requested file should be copied to the retr_files directory.

Example 1

Below is the output of the four sample commands from above. (in the example below, assume that all input lines end in a <CRLF> sequence):[2]

```
220 COMP 431 FTP server ready.
USER jasleen
331 Guest access OK, send password.
USERjasleen
500 Syntax error, command unrecognized.
USR jasleen
500 Syntax error, command unrecognized.
USER jasle*en
501 Syntax error in parameter.
USER
501 Syntax error in parameter.
```

Example 2

Below is another example. Again, assume that each line ends with <CRLF>.

```
USER anonymous
PASS foobar
SYST
TYPE I
PORT 152,2,131,205,31,144
RETR pictures/jasleen.jpg
QUIT
```

The corresponding output is:

```
220 COMP 431 FTP server ready.
USER anonymous
331 Guest access OK, send password.
PASS foobar
230 Guest login OK.
SYST
215 UNIX Type: L8.
TYPE I
200 Type set to I.
PORT 152,2,131,205,31,144
200 Port command successful (152.2.131.205,8080).
RETR pictures/jasleen.jpg
150 File status okay.
250 Requested file action completed.
```

[2] Note that if you type in a USER command to your program from the keyboard, depending on how your program is organized you will likely see each line of input twice of your screen: once because Linux is echoing what you type to the window of your terminal/ssh session, and a second time because your program is required to echo the input lines to standard output (the window of your terminal/ssh session).

```
QUIT
200 Command OK.
```

In addition to this output on standard output, your program should also copy the content of pictures/jasleen.jpg to retr_files/file1.

More test cases (both input and output) are defined in the starter code.

Testing

Creating test input for your HW1 program is not so simple as just typing a line of text into your favorite shell program. The issue is that different user interfaces use different mappings of key presses on the keyboard into a resulting character or character sequence. For HW1, the difficulty is that many (most?) shell interfaces do not map the "Enter" key to the <CRLF> sequence. You may get <LF> alone or <CR> alone or <CRLF>. Further, trying to type something that *looks* like the character literals (escape sequences) \r\n will not work either.

The most straightforward way to generate test input that has the <CR> and <LF> included is to create a file of test lines and redirect your standard input to the file. In the input generator script, terminate each line with a <CRLF> sequence, ie. "\r\n". Despite using this generated input, it's possible that your implementation does not work on your computer, in particular on Windows. Therefore, you should use the *classroom.cs.unc.edu* department Linux server to test your program.

To aid in testing, sample input and output files are provided in the starter code. Please note that these sample tests are *not* comprehensive (i.e., you should test your program much more thoroughly than these test files) – and grading will certainly rely on many additional tests. These sample files are provided simply to aid you in initial testing, as well as catching if your program is making basic formatting/syntax mistakes.

Use the provided programs (MakeInput1.py and MakeOutput1.py) to create your own test cases like so:

```
% python3 MakeInput1.py > Input1
% python3 MakeOutput1.py > Output1
```

Now, run your program with the generated input, and redirect the output to another file. Finally, use diff to compare the output of your program to the expected output.

```
% python3 FTPserver.py < Input1 > myOutput1
% diff myOutput1 Output1
```

If your program works correctly, the diff command above will produce *no* output. If not, your program is not following specification correctly.

Rubric

Program(s) should be neatly formatted (*i.e.*, easy to read) and well documented. A guide for documenting and structuring programs is available on the course web page. You should follow these programming style (appropriate use of language features, including variable/procedure/class names), and documentation (descriptions of functions, general comments, use of invariants, pre- and post-conditions where appropriate) conventions.

All programs will be tested on Linux. You should be able to develop your programs in whatever Python development environment you prefer (*e.g.*, VSCode on your PC). However, it is your responsibility to test and ensure the program works properly in Linux (specifically, on the machine *classroom.cs.unc.edu*). In particular, if your program performs differently on your PC than it does on the Departmental server, your grade will be based on your program's performance on the Departmental server.

Here is a breakdown of how your submission will be evaluated.

1. Basic Requirements (20%)

a. parse case-insensitive commands, and distinguish valid commands from invalid ones

```
           user jasleen
331 Guest access OK, send password.
USERCS jasleen
500 Syntax error, command unrecognized.
```

b. parse number of parameters required by each command and report ERROR with correct error-token if a valid command is followed by wrong number of parameter(s)

```
USER
501 Syntax error in parameter.
NOOP jasleen
501 Syntax error in parameter.
```

c. check whether input ends with CRLF

```
"USER jasleen\r\n"
331 Guest access OK, send password.
"USER jasleen\n"
501 Syntax error in parameter.
```

d. recognize simple tests on the other commands

```
USER jasleen
331 Guest access OK, send password.
PASS foobar
230 Guest login OK.
TYPE I
200 Type set to I.
SYST
215 UNIX Type: L8.
```

2. Advanced requirements (20%)

a. recognize valid input in more complexed forms (eg. containing white spaces)

```
USER jasle  en
331 Guest access OK, send password.
```

b. recognize invalid parameters with correct error-tokens

```
PASS 123***
501 Syntax error in parameter.
TYPE C
501 Syntax error in parameter.
```

3. PORT and RETR commands (30%)

a. the PORT command
   i. detect formatting error
   ii. detect whether each `<number>` in `<host-port>` is valid
   iii. convert `<host-port>` to IP-port addresses correctly
   iv. Correct reply line for PORT

b. the RETR command
   i. detect formatting error
   ii. detect whether the file is accessible
   iii. copy the file into retr_files
   iv. can only be called once per valid PORT command

4. Command Ordering (30%): Your program should be able to make sure the ordering of commands are correct, and return correct reply messages.

a. detect the beginning and ending of each FTP session (15%)
   i. Correct 1st line
   ii. QUIT can be anywhere

b. return correct reply if commands are out of order (15%)
   i. USER and PASS must appear in pairs
   ii. PORT must precede RETR
   iii. Nothing can precede USER and PASS (Except QUIT)
   iv. No more commands get processed after QUIT (program should terminate)

Submission

We will use Gradescope for submitting assignments. You will be able to start submitting to Gradescope starting on Friday, January 17 at 3pm.

Important: When you submit to Gradescope, an autograder will run multiple tests against your submission. If you fail any tests, Gradescope will show you the input of the first three failed tests. You may submit 3 times to the autograder without penalty. *After that, you will lose 10% of your grade on this assignment for each additional resubmission (10% grade loss for the 4th resubmission, 20% for the 5th resubmission, and so on).*

Follow these steps to submit your assignment:

1. Use the following entry code to enroll in this course on Gradescope: **R7P3Z4**.
2. Commit *and* push your code to GitHub. Verify on github.com that your code was pushed.
3. Go to "Homework 1" on Gradescope and submit your assignment. If this is your first time using Gradescope to submit coding assignments, Gradecsope will ask you for permission to access GitHub. Grant this permission. You should now see a dropdown menu of your repositories. Select your repository and submit your assignment.