File Transfer Protocol (FTP), Client and Server – Step 3

In this assignment you will shift your attention from the server to constructing portions of a simple FTP client program. The role of the FTP client program is to provide a user interface that allows a human user to enter high-level requests and generate the appropriate FTP protocol commands to accomplish the user’s request. It also provides feedback to the user on the validity of user inputs and the success or failure of the request. In HW1 and HW2 you have developed parts of a server program that can process FTP protocol commands without considering how the client program generates those commands. In HW3 you will construct parts of the client program to create commands and parse the FTP replies that would be returned by a server program. For this assignment you will develop TWO small programs. Most of the code from these programs will be substantially reused in your final FTP client program.

Program One – The user interface and FTP command generation.

This program will read standard input to accept input lines that a human user can use to request simple FTP operations. 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-str>
<let-dig-str> ::= <let-dig> | <let-dig><let-dig-str>
<let-dig> ::= <a> | <d>
  <a> ::= any one of the 52 alphabetic characters "A" through "z"
in upper case and "a" through "z" in lower case
  <d> ::= any one of the characters representing the ten digits 0 through 9
<server-port> ::= character representation of a decimal integer in the range 0-65535
<pathname> ::= <string>
<string> ::= <char> | <char><string>
<SP>+ ::= one or more space characters
```
Program one will read standard input to accept input lines that represent user requests. For each line of input your program should:

- Echo the line of input using System.println (i.e., duplicate the line of input on standard output, but use \n as the line termination character).
- For valid user requests, print on standard output the lines specified in the section “Processing for valid user requests” (below) for each of the three requests (CONNECT, GET, QUIT)
- For any valid user request that appears before the first CONNECT request, print out the error message “ERROR -- expecting CONNECT”.
- For invalid user requests (even if they appear before the first CONNECT), 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 above specification for user requests.

Processing for valid user requests

A CONNECT request must be the first user input accepted by the program (the user may also input a new CONNECT request at any time)---this is without exception (not even a valid QUIT request can appear before a valid CONNECT request). For each valid CONNECT request, the program will reset any internal state 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><CRLF>

where <server-host> and <server-port> represent strings extracted from the user request.

Your program will then generate the following sequence of valid FTP commands and write them to the standard output following the above response line:

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 your simple FTP client and server 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.

Your program will then generate the following sequence of valid FTP commands and write them to the standard output 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 2. Those specifications are repeated here:

\[
<\text{host-port}> ::= <\text{host-address}>,<\text{port-number}>
<\text{host-address}> ::= <\text{number}>,<\text{number}>,<\text{number}>,<\text{number}>
<\text{port-number}> ::= <\text{number}>,<\text{number}>
<\text{number}> ::= \text{character representation of a decimal integer in the range 0–255}
\]

The <host-address> value you generate is to be the Internet address assigned to the host machine where your program is running. It can be obtained through methods of the InetAddress class in java.net.* as illustrated in the following code fragment:

```java
String myIP;
InetAddress myInet;
myInet = InetAddress.getLocalHost();
myIP = myInet.getHostAddress();
```

The string referenced by myIP is a host address in the “dotted decimal” format described in Homework 2 (e.g., 152.2.129.21) and must be translated to the form specified above for <host-address>.

The <port-number> value is to be created in your program by initializing a variable to the value 8000 and incrementing it by one after each PORT command is generated. The value must be converted to the format specified above for <port-number> by doing the inverse computation corresponding to the conversion of port values specified in HW2, expressing the result as a character representation of decimal integers.

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

QUIT accepted, terminating FTP client<EOL>

Your program will then generate the following FTP command and write it to the standard output following the above response line. The program will then terminate.

QUIT<CRLF>

Here is an example showing how your output should look with a sequence of valid commands assuming your program is running on eagle.cs.unc.edu (NOTE: in this example the input lines that have been
echoed are marked with a “-” to make the example clear. Do NOT include the “-” in your program output).

-CONNECT swift.cs.unc.edu 9000
CONNECT accepted for FTP server at host swift.cs.unc.edu and port 9000
USER anonymous
PASS guest@
SYST
TYPE I
-GET pictures/jasleen.jpg
GET accepted for pictures/jasleen.jpg
PORT 152,2,128,10,31,64
RETR pictures/jasleen.jpg
-CONNECT capefear.cs.unc.edu 21
CONNECT accepted for FTP server at host capefear.cs.unc.edu and port 21
USER anonymous
PASS guest@
SYST
TYPE I
-GET index.html
GET accepted for index.html
PORT 152,2,128,10,31,65
RETR index.html
-QUIT
QUIT accepted, terminating FTP client
QUIT

Program Two: The FTP Reply Parser

Recall that in HW2 the format of FTP reply lines generated by the FTP server was introduced. Program two is to be a simple parser for the FTP reply. The format of FTP reply lines is:

```plaintext
<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. Remember that the definition of the `<string>` token here is to be taken from HW 2 (characters other than `<CR>` and `<LF>`).

Program two will read standard input to accept input lines that represent FTP replies. For each line of input your program should:

- 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 input line
• 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 above specification for FTP replies.

\[<error-token> ::= \text{"reply-code"} \mid \text{"reply-text"} \mid \text{"<CRLF>"}\]

Here is an example of how your output from program two should look. (NOTE: in this example the input lines that have been echoed are marked with a “-” to make the example clear. Do NOT include the “-” in your program output).

-220 COMP 431 FTP server ready.
FTP reply 220 accepted. Text is : COMP 431 FTP server ready.
-331 Guest access OK, send password.
FTP reply 331 accepted. Text is : Guest access OK, send password.
-230 Guest login OK.
ERROR -- reply-code
-Port command successful (152.2.131.205,8080).
ERROR -- reply-code
-650 File status okay.
ERROR – reply-code

Assignment Notes

As in Homework 1 and 2, both of these programs should read from standard input, echo all input lines and write the corresponding output to standard output. As before, your program must not output any additional user prompts, debugging information, status messages, etc.

In both programs you should continue processing lines of input until end-of-file is reached on the input stream. If errors are encountered on input lines you should simply emit the appropriate output and begin the parse of the next input.

Testing

To aid in testing, sample input and output files are provided (MakeInputClient1.java, MakeOutputClient1.java, MakeInputClient2.java, and MakeOutputClient2.java). 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 to test your code on classroom.cs.unc.edu using the following steps:

\[
<\text{unix-promp}\text{t}> \text{javac MakeInputClient1.java} \\
<\text{unix-promp}\text{t}> \text{java MakeInputClient1 > input1} \\
<\text{unix-promp}\text{t}> \text{javac MakeOutputClient1.java} \\
<\text{unix-promp}\text{t}> \text{java MakeOutputClient1 > output1} \\
<\text{unix-promp}\text{t}> \text{java FTPclient1 > myOutput1} \\
<\text{unix-promp}\text{t}> \text{diff myOutput1 output1} \\
<\text{unix-promp}\text{t}> \text{javac MakeInputClient2.java} \\
<\text{unix-promp}\text{t}> \text{java MakeInputClient2 > input2} \\
<\text{unix-promp}\text{t}> \text{javac MakeOutputClient2.java} \\
<\text{unix-promp}\text{t}> \text{java MakeOutputClient2 > output2}
\]
<unix-prompt> java FTPclient2 > myOutput2
<unix-prompt> diff myOutput2 output2

If your program works correctly, the diff commands should produce no output.

**Grading**

As per the instructions distributed with Homework 1, create a subdirectory named HW3 within the directory ~/comp431/submissions, and place the final version of your program in this directory before the due date. (Be sure access permissions are set correctly for the HW3 directory.) Please give your final programs the names “FTPclient1.java” and “FTPclient2.java”

When your program is complete, notify the TAs according to the piazza post by the TAs (dated Feb 3).

Please remember that your programs will be run and tested in the UNIX environment on classroom.cs.unc.edu. Be sure that your program has been compiled for that environment and the resulting class file is present in your HW3 submission directory.

As before, your program(s) should be neatly formatted (i.e., easy to read) and well documented. In general, 80% of your grade for a program will be for correctness, 20% for “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).