

COMP 431 — INTERNET SERVICES & PROTOCOLS

Spring 2009

Homework 1, January 12

Due: January 20, 8:30 AM

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

One of the goals of the course assignments 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. 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 of an FTP server, it 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. This assignment is a simple string parsing problem whose solution we will build on in later assignments.

Specifically, you are to write a program to determine if a command received (a text string) is a valid FTP command. An FTP command is simply a line of text that looks like the following:

```
USER jasleen
```

The FTP command is made up of three elements:

- a 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),
- a command parameter — e.g., the characters “jasleen”, a variable length sequence of characters (not all commands require a parameter, however) , and
- a “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) may be separated by any number of spaces. The “CRLF” immediately follows the command parameter (or command name if no parameter is used for the command).

The USER command is part of the larger 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 are some of the FTP commands:

```
USER<SP+><username><CRLF>
PASS<SP+><password><CRLF>
TYPE<SP+><type-code><CRLF>
SYST<CRLF>
NOOP<CRLF>
QUIT<CRLF>
```

```
<username> ::= <string>
<password> ::= <string>
<type-code> ::= "A" | "I"
<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>
```

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 carriage return-line feed — the default line termination “character” for UNIX). The following strings do not conform to the formal description and would be rejected as invalid or illegal requests.

```
USERjasleen
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 an invalid byte value not one of the 128 ASCII characters).

The Assignment — Processing FTP commands in Java

For this assignment you are to write a Java program to read in lines of characters from standard input (*i.e.*, the keyboard) and determine which lines, if any, are legal FTP commands. For reading input your program should use the Java `BufferedReader` class. To aid in parsing requests you should use the Java `StringTokenizer` class described briefly below.

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).

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.

- For valid FTP commands, list on the next line the string “Command ok”.
- For invalid commands, print out the error message “ERROR -- x ” where x is the name of the token that is missing or ill-formed. You should check for syntax errors as well as ordering errors in the order in which commands are input and emit error messages as appropriate. All acknowledgement and error messages should be formatted *exactly* as shown above.²

For example, if the four sample requests above were read by your program, the output would be:

```
USER jasleen
Command ok
USERjasleen
ERROR -- command
USR jasleen
ERROR -- command
USER jasle*en
ERROR -- username
```

All output should be written to standard output (*i.e.*, to the 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 (when *control-D* is typed from the keyboard under UNIX or *control-Z* on Windows). Your program must not output any user prompts, debugging information, status messages, *etc.*

Your program will be tested using an automated procedure that redirects standard input and output to files so you may want to try using redirection with your program, e.g.,

```
%java your_class_name <inputfile >outputfile
```

Grading

Program(s) should be neatly formatted (*i.e.*, easy to read) and well documented. In general, 75% of your grade for a program will be for correctness, 25% 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). A guide for documenting and structuring programs is available on the course web page.

For this (and most other programming) assignments you will “turn in” your program for grading by placing it in a special AFS directory on a Department of Computer Science machine and sending mail to the TA. To ensure the TA can grade your assignments in an efficient and timely fashion, please follow the following guidelines *precisely*.

- Log on to a CS Department login machine with access to your AFS home directory. (Any CS login machine will work, however the machine *classrom.cs.unc.edu* has been specifically dedicated for classwork.)
- In your AFS home directory, create the directory structure: *comp431/submissions*. (That is, create the directory *comp431* in your home directory and inside this directory, create the directory *submissions*.)
- To allow the instructor and the TA to access the directory while restricting access for all others, execute the following commands:

² This is done for ease of grading. The actual FTP specification only specifies the response numbers and allows arbitrary text to follow the error code/acknowledgement number.

```
fs sa ~/comp431/submissions system:anyuser none
fs sa ~/comp431 jasleen read
fs sa ~/comp431 susu read
```

All subsequent subdirectories you create should inherit these permissions, however, to be safe you should explicitly check that this is the case. If you enter the command `fs la ~/comp431/x` (where *x* is the name of any subdirectory you might create) you should see something like:

```
Access list for /afs/cs.unc.edu/home/<userid>/comp431/x is
Normal rights:
cs-machines l
system:administrators rlidwka
<userid> rlidwk
jasleen rl
susu rl
```

where *<userid>* is your login id. If you see a line for any other user or a line of the form “system:anyuser rl” then something is wrong and you should contact either the TA or one of the instructors before proceeding.

- For each assignment you will create a subdirectory with a name specified in the assignment. You must also name your program as specified in the assignment. For this assignment you should name your final program “FTP1server.java” and store it in a directory named HW1 (in capital letters) inside your `~/comp431/submissions` directory.
- When you have completed your assignment you should put your program and any other necessary parts (subclasses, *.class files, etc). in the specified subdirectory and send mail to the TA (susu@cs.unc.edu) indicating that the program is ready for grading. Send mail to the TA with the subject line “COMP 431 HW1 ready”. If you send mail from an account other than your CS account, please indicate your Computer Science Unix account login (user-id) in the body of the message.
- *Do not change any of your files for this assignment after sending this mail!* The lateness of assignments will be determined by the UNIX timestamps on your program files. If the timestamps on the files change after you submit then (send mail to the TA) you may be penalized for turning in a late assignment.
- All programs will be tested under Linux. You should be able to develop your programs in whatever Java development environment you prefer (e.g., Visual J++ on the PC, etc.) and then upload to Linux. However, it is your responsibility to test and insure the program works properly in Linux (specifically, on the machine `classroom.cs.unc.edu`).
- The program should be neatly formatted (i.e., easy to read) and well documented. In general, 75% of your grade for a program will be for correctness, 25% 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).

Help with the Java StringTokenizer Class

For this assignment you should use the StringTokenizer class to parse input lines. This class contains methods to do a first-level parse and return character strings separated by whitespace. For more information on the StringTokenizer class see the Gosling Java reference or

<http://java.sun.com/products/jdk/1.2/docs/api/java/util/StringTokenizer.html>

The example below takes input line by line from standard input. It prints the line to standard output and then separates each line into tokens (delimited by a space). If the line starts with “TEST” the line is valid

and each subsequent token is also printed, separated by a +. If the line is blank or does not start with "TEST" it is invalid and an appropriate message is printed. The program runs until end of file is reached. For example, the input file containing the strings:

```
TEST a b c d e f word
TEST with blank space      here
TEST (the next line is a blank line)

TST bad line (because of misspelling)
test bad line (because of case)
```

results in the following output:

```
Input = TEST a b c d e f word
Valid input, tokens = TEST + a + b + c + d + e + f + word

Input = TEST with blank space      here
Valid input, tokens = TEST + with + blank + space + here

Input = TEST (the next line is a blank line)
Valid input, tokens = TEST + (the + next + line + is + a + blank + line)

Input =
Blank input line

Input = TST bad line (because of misspelling)
Invalid input

Input = test bad line (because of case)
Invalid input
```

The code for the main parsing routine follows on the next page.

```

public class Parse      // parse a string
{
    static void parse(String inputline) {

        String token;    // The current token

        // declare and instantiate a StringTokenizer for the input line
        StringTokenizer tokenizedLine = new StringTokenizer(inputline);
        // inputline is the string to use

        // If the tokenized line has some tokens
        if ( tokenizedLine.hasMoreTokens() ) {
            token = tokenizedLine.nextToken();    // Get the token

            // See if the string representing the next token matches "TEST"
            // (note that "equals" is case sensitive)
            if ( token.equals("TEST") ) {

                // Indicate the input line was valid...
                System.out.print("Valid input, tokens = " + token);

                // ...and as long as there are more tokens
                while (tokenizedLine.hasMoreTokens()) {
                    // print the tokens with a "+" between each
                    System.out.print(" + " + tokenizedLine.nextToken());
                }
                System.out.println();

                // Otherwise indicate the input was invalid
            } else {
                System.out.println("Invalid input");
            }
        } else {
            System.out.println("Blank input line");
        }
    }
}

public static void fail(Exception e, String msg)    // Exception handling
{
    System.err.println(msg + ": " + e);
    System.exit(1);
}

```

Creating test input for HW 1

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 Java 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 (see example below). In the file, terminate each line with a byte that has the appropriate values (<CR> is the value 13 (decimal) or 0D (Hex), and <LF> is the value 10 (decimal) or 0A (Hex)).

The next question is how to create such a file with these byte values. The easy way is to write a simple Java program that writes your test lines to standard output and redirect the output to a file.

For example:

```
import java.io.*;
class makeLines {
    public static void main(String argv[]) throws Exception
    {
        System.out.print("USER jasleen\r\n");
        System.out.print("USER susu\r\n");
        System.out.print("USER jasleen\n");
        System.out.print("USER jasl");
        System.out.write(200);
        System.out.print("fd\r\n");
    }
}
```

Note that the last line in the file will contain an "invalid" character according to the FTP specification. To create the file with this program use:

```
%java makeLines >testlines
```

To run your Java program (Parse) with this input file:

```
%java Parse <testlines
```

You may also be able to coax your favorite editor into allowing you to create a file with these byte values inserted with some sort of special sequence. For example, the Notepad editor in the Window Accessories programs by default maps the Enter key to <CRLF> (if you create a test file on Windows, be sure to use "binary" mode if you use a FTP program to copy it to Unix).