



UNIVERSITY of NEW ORLEANS

DEPARTMENT OF COMPUTER SCIENCE

CSCI 4311: Computer Networks & Telecommunications

Instructor: *Vassil Roussev*

Programming Project (Part 1)

File Transfer Protocol (FTP) Client Implementation

Assigned: Sep 22, 2009

Goal

The goal of the programming project, which is broken into three parts, is to build and test a simple FTP client. The ultimate success of your assignment will be based on the ability of your client to interoperate with FTP server currently in use on the Internet.

Due Dates

- Program #1: Tue, Sep 29
- Program #2: Tue, Oct 6
- Program #3: Tue, Oct 13

Introduction

At a high-level, an FTP client/server interaction is an exchange of textual messages, which represent the various commands submitted by the client and the responses given by the server. Thus, correct message generation and parsing is a necessary precondition for the success of your work. More generally, this is true of any public application-layer protocol on the Internet so your project is representative of traditional network programming. Recall that, there are three distinct components to an operational FTP system: server, client, and a user interface (UI) agent.

Goal #1: The UI agent and FTP command generation

The role of the UI agent 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.

Program #1: csci4311.ftp.UIAgent

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: 0-9
<server-port> ::= character representation of a decimal integer: 0-65535
<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. For Java <EOL>
        is typically <CR> followed by <LF> but <LF> alone is also
        allowed. For these programs you need not check for a
        specific line termination on input lines and, therefore,
        you can use the readLine() method of the BufferedReader
        class.
```

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 (i.e., duplicate the line of input on standard output).
- 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 invalid user requests, print out the error message "ERROR -- <error-token>" where <errortoken> is the name of the token that is missing or ill-formed according to the above specification for user requests.

```
<error-token> ::= "request" | "server-host" | "server-port" | "pathname"
```

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). 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><EOL>
```

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:

```
<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
```

NOTE: The quirky syntax of the PORT command will be explained in class.

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:

```
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 above (e.g., 152.2.129.21) and must be translated to the form specified above for <host-address>. The <portnumber> 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.

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 a cs.uno.edu machine.

CONNECT cook.cs.uno.edu 9000

```
CONNECT accepted for FTP server at host cook.cs.uno.edu and port 9000
USER anonymous
PASS guest@
SYST
TYPE I
```

GET photos/white-elephant.jpg

```
PORT 152,2,128,10,31,64
RETR photos/white-elephant.jpg
```

CONNECT cook.cs.uno.edu 21

```
CONNECT accepted for FTP server at host cook.cs.uno.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
```

Here is an example showing how your output should look with a sequence of valid commands assuming your program is running on a cs.uno.edu machine.

NOTE #1: The port number in the protocol is represented by two decimal numbers (call them p1 and p2) that are computed in a bit of a quirky way: $p1 = \text{port}/256$, $p2 = \text{port}\%256$. Thus, in the example above, port number represented by 31 and 64 is $31*256+64 = 8000$.

NOTE #2: If you are behind a firewall (e.g. at home) you will not be able to connect outside it because your client must implement passive mode. That will be required in the second part, however, you may start implementing now, if you have the extra time.

Goal #2: The FTP reply parser**Program #2: csci4311.ftp.ReplyParser**

Program two is to be a simple parser for the FTP reply. 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 an FTP command. 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 <errortoken> is the name of the token that is missing or ill-formed according to the above specification for 1FTP replies.

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

Here is an example of how your output from program two should look.

```
220 CSCI 4311 FTP server ready.
FTP reply 220 accepted. Text is: CSCI 4311 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
```

Goal #3: FTP Client Program

You can now put the pieces together and create a working FTP client. In this assignment, you will extend your FTP "client" programs from the previous assignments to interoperate over a network by using TCP sockets. To do this you will need to implement all elements of the FTP protocol and replace some stdin/stdout I/O and parts of your file I/O with socket I/O. Further, the file I/O to read and write a file that was entirely contained in the P1 program will be split between the client and server – the server will read all the bytes in a requested file, write those bytes to the client over an FTP-data connection, and the client will read these bytes from the connection and write them to a local file.

Program #3: `csci4311.ftp.FTPClient`

Your FTP client program should take one command line argument: an initial port number for a "welcoming" socket that the client will use to allow the server to make an FTP-data connection. Your client should accept input requests from a human user using standard input. The client should echo each user input line to standard output along with the corresponding response line as specified before. When a valid CONNECT request is accepted from the user, the client should attempt to establish a TCP socket connection to the server program which it expects to be running on the host and port specified in the user's input. This connection is the FTP-control connection and it should not be closed until the user enters another valid CONNECT request (which will initiate a new FTP-control connection). When your client successfully connects to the server it must be prepared to receive the server's greeting reply (e.g., "220 CSCI 4311 FTP server ready.") on the FTP-control connection. When your client program receives any reply from the server on the FTP-control connection, it should display the output specified for the FTP reply parser program. Thus, if the user input line is :

```
CONNECT cook.cs.uno.edu 9000
```

The following lines would be displayed on standard output.

```
CONNECT cook.cs.uno.edu 9000
CONNECT accepted for FTP server at host cook.cs.uno.edu and port 9000
FTP reply 220 accepted. Text is: CSCI 4311 FTP server ready.
```

If the client is unable to connect to the server, it should write "CONNECT failed" to standard output and prepare to read the next input from the user. After the FTP-control connection has been established and the initial server reply received, the client should send the four-command sequence specified above (USER, PASS, SYST, and TYPE) to the server using the FTP-control connection. Before your client program sends a command to the server on the FTP-control connection, it should first echo that command to standard output. After each command in this sequence is sent to the server, the client should read and process the corresponding reply (as specified in Programs 1 and 2) received from the server on the FTP control connection before sending the next one.

If the user enters additional valid CONNECT requests the existing FTP-control connection is closed and a new one established using the host and port specified in the CONNECT. When a valid GET request is accepted from the user, the client should send the two-command sequence specified in Program 2 (PORT, RETR) 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 "welcoming" socket and associated port for each FTP-data connection). Before sending the PORT command to the server, the client should create a "welcoming" 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 "welcoming" socket cannot be created, the client should write "GET failed, FTP-data port not allocated." to standard output and read the next user input line.

After the RETR command has been sent to the server, the client should accept the server's FTP-data connection on the "welcoming" socket and then read the bytes for the requested file from the new socket created for that connection. The client should continue 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 should 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 should be written to a new file in a directory named `retr_files` in the client's current working directory (be sure to create this directory first). The file's name in the `retr_files` directory should have the form "filexxx.yyy" where xxx is the number of valid RETR commands your client has recognized so far during this execution of the program and yyy is the file extension of the original file (e.g., jpg). 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 should be written as a stream of bytes (Java I/O using the `FileOutputStream` classes). 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 should be ended and a new user input line read from standard input. A user's QUIT request should be handled as specified in P2 with the additional requirements that the client should send the FTP QUIT command to the server, receive the reply shown below, close the FTP-control connection and then exit. The server reply to QUIT is:

```
221 Goodbye.
```

Submission

All programs should read from standard input, echo all input lines and write the corresponding output to standard output. Your submitted program must not output any additional user prompts, debugging information, status messages, etc. 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.

All of your classes must be members of the `csci4311.ftp` package. `jar` your entire submission and send it to the submission mail account (`csci4311@roussev.net`) as an attachment. Use Program N Submission as your subject line, where N is the program number. All submissions must be done from a UNO account for authentication purposes.