Project 2: FTP Client

Implement a real protocol using two sockets

This project is due at 11:59pm on Monday, February 5, 2024.

Description

The goal of this project is to deepen your ability to write network code. At this point you can open a socket and send and receive simple messages. In this project, you will implement a client for a more complex protocol, that has more features, and uses two sockets rather than one.

Specifically, in this project you will develop a client for the File Transfer Protocol (FTP) protocol. We have setup an FTP server for you to use when developing and debugging your client. The server is available at ftp://ftp.4700.network.

What is FTP

Developed in 1971, the File Transfer Protocol (FTP) is one of the oldest protocols still in common use today. FTP's development predates TCP/IP, yet the protocol still works on the modern internet.

FTP is a client-server oriented protocol for uploading, downloading, and managing files. The FTP server's job is to listen for incoming connections from clients and then respond to their requests. These requests include all common file and directory operations, such as listing the files in a directory, making and deleting directories, uploading files from the client machine to the server, and downloading files from the server to the client machines. FTP clients connect to FTP servers and issue commands to tell the server what to do.

Because the FTP protocol is so old, it has many, many features, some of which are vestigial and no longer make sense on the modern internet (e.g., uploading a file in 36-bit, EBCDIC format directly to a line-feed printer), and others that are so esoteric that they are rarely used and supported. Wikipedia has an extensive article on the FTP protocol, as well as a list of all FTP protocol commands. Fortunately, a FTP client only needs to support a minimum, basic set of commands in order to function. We outline the necessary functionality for the FTP client you will develop below. Our reference implementation is roughly 300 lines of Python3, including self-documentation and extensive error checking.

FTP Server Decorum

We have set up a FTP server for you to help you develop and debug your client. It is available at ftp://ftp.4700.network. To use the FTP server you must login with a username and password. After logging in, the server will give each student access to an isolated folder for uploading and downloading files.

WARNING: DO NOT TEST YOUR FTP CLIENT ON PUBLIC FTP SERVERS

There are many FTP servers out there on the internet. Even though some of these FTP servers are "public" and allow anonymous access (i.e., no password required), **we ask that students confine their testing to our server**. Since your clients will behave weirdly and violate the FTP protocol during development, you don't want to send this kind of anomalous traffic to unsuspecting FTP servers, as it may alarm their administrators.

WARNING: DO NOT UPLOAD GIANT OR ILLEGAL FILES TO OUR SERVER

Our server does not have unlimited capacity. If we observe a student uploading large files to the server (say, larger than 10 megabytes), those files will be deleted and we will cut off the student's access.

Similarly, students are responsible for the files they upload to the server and are expected to exercise good [illegible] upload to the server. Any student that violates this rule will be reported to the university administration, along with copies of the offending files.

Username and Password

Your username for the FTP server is your **Khoury username** (not your Northeastern username). To get your password, you must SSH to the Khoury login machine (*login-students.khoury.northeastern.edu*, or one of its many clones) and run a command provided by Khoury systems. This command is:

```
$ 4700-ftp-password [Your NUID]
```

Input your NUID with leading zeroes. The output of this command is a long string that is your password for the FTP server. Here is a full example of SSHing to the Khoury login machine and generating an FTP password:

```
cbw@clx-ubuntu:~$ ssh cbw@login-students.khoury.northeastern.edu
[cbw@login-students ~] 4700-ftp-password 001234567
Your password is: 9c2bf5c59edb601cc01a52c1f3452ddf4bd87f86f27bc87bc6839bc067057eb4
```

Getting Familiar with FTP

If you've never used an FTP client or server before, it may be helpful to use an existing client to familiarize yourself with how FTP works at a high-level. We recommend that beginners start with FileZilla, which is a popular, feature packed, open-source FTP client that works across all major operating systems. Note that FileZilla is a GUI client; in this project you will be building a command line client.

Unix and BSD-compliant systems like Linux and MacOS have a command line FTP client named *ftp* that is typically pre-installed, or can be installed easily from the command line. This command line ftp client is interactive: running it opens an FTP shell where the user can type FTP commands like "dir" and "cd" to view and change remote filesystem directories, respectively. The "help" command will list available FTP commands, and "help [command name]" provides helpful information about a specific command.

The *curl* command line program, which is also available for most Unix and BSD-based systems, supports downloading files from FTP servers. It has a command line syntax that is most similar (though not identical) to the FTP client that you will be developing in this assignment.

Your Program

Your goal to write a basic FTP client application. This client will run on the command line, and must support the following six operations: directory listing, making directories, file deletion, directory deletion, copying files to and from the FTP server, and moving files to and from the FTP server.

Your FTP client must execute on the command line using the following syntax:

```
$ ./4700ftp [operation] [param1] [param2]
```

operation is a string that specifies what operation the user is attempting to perform. Valid operations are *ls*, *mkdir*, *rm*, *rmdir*, *cp*, and *mv*. Each operation requires either one or two parameters, denoted by *param1* and *param2* on the command line. *param1* and *param2* are strings that either represent a path to a file on the local filesystem, or a URL to a file or directory on an FTP server.

4700ftp is not required to print anything to STDOUT or STDERR. That said, students may add functionality that prints out FTP protocol messages (e.g., to aid debugging), directory listings, and errors (e.g., network or socket errors, FTP protocol errors, etc.).

Here is an example of a full-fledged *4700ftp* implementation. Note that it includes two optional parameters (*–verbose* and *–help*); **your client is not required to have these optional parameters**. The help for this client describes

```
$ ./4700ftp --help
usage: 4700ftp.py [-h] [--verbose] operation params [params ...]

FTP client for listing, copying, moving, and deleting files and directories on remote FTP servers.

positional arguments:
operation      The operation to execute. Valid operations are 'ls', 'rm', 'rmdir',
              'mkdir', 'cp', and 'mv'
params         Parameters for the given operation. Will be one or two paths and/or URLs.

optional arguments:
-h, --help     show this help message and exit
--verbose, -v  Print all messages to and from the FTP server

# Available Operations

This FTP client supports the following operations:

ls <URL>                Print out the directory listing from the FTP server at the given URL
mkdir <URL>             Create a new directory on the FTP server at the given URL
rm <URL>                Delete the file on the FTP server at the given URL
rmdir <URL>             Delete the directory on the FTP server at the given URL
cp <ARG1> <ARG2>        Copy the file given by ARG1 to the file given by
                         ARG2. If ARG1 is a local file, then ARG2 must be a URL, and vice-versa.
mv <ARG1> <ARG2>        Move the file given by ARG1 to the file given by
                         ARG2. If ARG1 is a local file, then ARG2 must be a URL, and vice-versa.

# URL Format and Defaults

Remote files and directories should be specified in the following URL format:

ftp://[USER[:PASSWORD]@]HOST[:PORT]/PATH

Where USER and PASSWORD are the username and password to access the FTP server,
HOST is the domain name or IP address of the FTP server, PORT is the remote port
for the FTP server, and PATH is the path to a file or directory.

HOST and PATH are the minimum required fields in the URL, all other fields are optional.
The default USER is 'anonymous' with no PASSWORD. The default PORT is 21.

# Example Usage

List the files in the FTP server's root directory:

$ ./4700ftp ls ftp://bob:s3cr3t@ftp.example.com/

List the files in a specific directory on the FTP server:

$ ./4700ftp ls ftp://bob:s3cr3t@ftp.example.com/documents/homeworks

Delete a file on the FTP server:

$ ./4700ftp rm ftp://bob:s3cr3t@ftp.example.com/documents/homeworks/homework1.docx

Delete a directory on the FTP server:

$ ./4700ftp rmdir ftp://bob:s3cr3t@ftp.example.com/documents/homeworks

Make a remote directory:

$ ./4700ftp mkdir ftp://bob:s3cr3t@ftp.example.com/documents/homeworks-v2

Copy a file from the local machine to the FTP server:

$ ./4700ftp cp other-hw/essay.pdf ftp://bob:s3cr3t@ftp.example.com/documents/homeworks-v2/essay.pdf

Copy a file from the FTP server to the local machine:
```

(Il)Legal Libraries

Part of the challenge of this assignment is that **all FTP request and response code must be written by the student, from scratch**. In other words, you need to implement the FTP protocol yourself. Students may use any available libraries to create socket connections and parse URLs. However, you may not use **any** libraries/modules/etc. that implement FTP. Obviously, your code is also not allowed to invoke system tools that implement FTP like *ftp* or *curl*.

For example, if you were to write your FTP client in Python3, the following modules would all be allowed: *socket* and *urllib.parse*. However, the following modules would **not** be allowed: *urllib.request*, *ftplib*, and *pycurl*.

If students have any questions about the legality of any libraries please post them to Piazza. It is much safer to ask ahead of time, rather than turn in code that uses a questionable library and receive points off for the assignment after the fact.

Implementation Details

In this assignment, you will develop an FTP client. This client must be able to login to a remote FTP server and perform several operations on the remote server. In this section, we will explain how to connect to an FTP server, the format of FTP protocol requests and responses, and the FTP commands that your client must support.

Modern FTP implementations use TCP/IP as their transport and network protocols. Thus, to connect to an FTP Server, your client will need to open a TCP socket. By default, FTP servers listen on port 21, although users may override the default port by specifying a different one on the command line. Once your client has connected a TCP socket to a remote server, it will begin exchanging text-formatted requests and responses with the FTP server.

> Make sure your FTP client reads the Hello message from the server!
>
> FTP servers always send a Hello message to clients after the control socket connects. Make sure your client reads this message before sending any FTP commands! If your client does not do this, you run the risk of introducing a non-deterministic race condition bug into your client. Historically, this is one of the most common errors students make during this project.

All FTP requests take the form:

```
COMMAND <param> <...>\r\n
```

COMMAND is typically a three or four letter command, in all caps, that tells the FTP server to perform some action. Depending on what command is sent, additional parameters may also be required. Note that parameters should not be surrounded by < and > symbols; we use those to denote things in messages that are optional. All FTP requests end with \r\n.

After each request, the FTP server will reply with at least one response. Some requests will elicit two responses. Additionally, FTP servers send a welcome message after the TCP connection is opened, before the client sends any requests. All FTP responses take the form:

```
CODE <human readable explanation> <param>\r\n
```

CODE is a three digit integer that specifies whether the FTP server was able to complete the request.

- 1XX codes indicate that more action is expected (e.g., waiting for a file to download or upload);
- 2XX codes indicate success;
- 3XX indicates preliminary success, but more action is required (e.g., your client sent a username, but now a valid password is required);

- 4XX, 5XX, and 6XX codes indicate that an error has occurred.

More details on FTP response codes can be found here. Some servers include an optional, human-readable explanation in each response that explain what happened or what is expected of the client. These human-readable messages are useful for debugging purposes. Responses may also include a parameter that is necessary for the client to function (most importantly, for the PASV command, see below). All FTP responses end with \r\n. Some, but not all, include a period before the \r\n.

Your FTP client must be able to send the following FTP commands:

- `USER <username>\r\n`

  Login to the FTP server as the specified *username*. If the user does not specify a username on the command line, then your client may assume that the username is "anonymous". This is the first request that your client must send to the FTP server.

- `PASS <password>\r\n`

  Login to the FTP server using the specified *password*. If the user specified a password on the command line, then this is the second request that your client must send to the FTP server. If the user did not specify a password on the command line then your client may skip this request.

- `TYPE I\r\n`

  Set the connection to 8-bit binary data mode (as opposed to 7-bit ASCII or 36-bit EBCDIC). Your client should set the TYPE before attempting to upload or download any data.

- `MODE S\r\n`

  Set the connection to stream mode (as opposed to block or compressed). Your client should set MODE before attempting to upload or download any data.

- `STRU F\r\n`

  Set the connection to file-oriented mode (as opposed to record- or page-oriented). Your client should set STRU before attempting to upload or download any data.

- `LIST <path-to-directory>\r\n`

  List the contents of the given directory on the FTP server. Equivalent to *ls* on the Unix command line.

- `DELE <path-to-file>\r\n`

  Delete the given file on the FTP server. Equivalent to *rm* on the Unix command line.

- `MKD <path-to-directory>\r\n`

  Make a directory at the given path on the FTP server. Equivalent to *mkdir* on the Unix command line.

- `RMD <path-to-directory>\r\n`

  Delete the directory at the given path on the FTP server. Equivalent to *rm -d* on the Unix command line.

- `STOR <path-to-file>\r\n`

Upload a new file with the given path and name to the FTP server.

- `RETR <path-to-file>\r\n`

  Download a file with the given path and name from the FTP server.

- `QUIT\r\n`

  Ask the FTP server to close the connection.

- `PASV\r\n`

  Ask the FTP server to open a data channel.

Control Channel, Data Channel

The FTP protocol is a bit unusual in that it requires not one, but two socket connections. The first socket that your client will open to the FTP server is known as the control channel. The control channel is typically the connection to port 21 on the FTP server. The control channel is for sending FTP requests and receiving FTP responses. However, **no data is uploaded or downloaded on the control channel**. To download any data (i.e., a file or **a directory listing**) or upload any data (i.e., a file) your client must ask the server to open a data channel on a second port.

The FTP command to open a data channel is PASV. The client sends PASV to the FTP server, and it responds with a message that looks something like this:

```
227 Entering Passive Mode (192,168,150,90,195,149).
```

Code 227 indicates success. The six numbers in parenthesis are the IP address and port that the client should connect a TCP/IP socket to to create the data channel. The first four numbers are the IP address (192.168.150.90 in this example) and the last two numbers are the port. Port numbers are 16-bits, so the two numbers represent the top and bottom 8-bits of the port number, respectively. In this example, the port number is (195 « 8) + 149 = 50069.

The semantics of the data channel are always the same: **once the data transfer is complete the data channel must be closed by the sender**. What changes is *who* closes the channel. If the server is sending data (i.e., a downloaded file or a directory listing) then the server will close the data socket once all data has been sent. This is how the client knows that all data has been received. Alternatively, if the client is sending data (i.e., uploading a file), then the client must close the data socket once all data has been sent. This is how the server knows that all data has been received. If the client wishes to upload or download additional data, e.g., perform multiple operations during a single control channel session, then one PASV data channel must be opened per operation.

Note that the control channel (i.e., the first socket) must stay open while the data channel is open. Once the data channel is closed, the client is free to end the FTP session by sending QUIT to the FTP server on the control channel and closing the control socket.

Language

You can write your code in whatever language you choose, as long as your code compiles and runs on Gradescope. Do not use libraries that are disallowed for this project, Similarly, your code must compile and run on the command line. You may use IDEs (e.g., Eclipse) during development, but do not turn in your project without a Makefile. Make sure you code has **no dependencies** on your IDE.

Suggested Implementation Approach

When starting work on this project, we recommend implementing the required functionality in the following order.

- **Connection Establishment.** Add support for connecting and logging in to an FTP server. This includes establishing a TCP control channel, correctly sending USER, PASS, TYPE, MODE, STRU, and QUIT commands. Print out responses from the server to confirm that each command is being received and interpreted correctly.
- **MKD and RMD.** Implement support for making and deleting remote directories. These commands are simpler because they do not require a data channel. Verify that your client is working by using a standard FTP client to double check the results.
- **PASV and LIST.** Implement support for creating a data channel, then implement the LIST command to test it.
- **STORE, RETR, and DELE.** Complete your client by adding support for file upload, download, and deletion. Double check your implementation by comparing it to the results from a standard FTP client.
- Double check that your client works successfully on a CCIS Linux machine, e.g., login.ccs.neu.edu

Submitting Your Project

To turn-in your project, you should submit the following three things:

1. Your thoroughly documented source code that implements your FTP program.
2. A `Makefile` that compiles your code.
3. A plain-text (no Word or PDF) `README.md` file. In this file, you should briefly describe your high-level approach, any challenges you faced, and an overview of how you tested your code.

Your `README.md`, `Makefile`, source code, etc. should all be placed in the root of a compressed archive (e.g., a .zip) and then uploaded to Gradescope. Do not use the compress or archive option on a Mac – Gradescope does not parse these archives. Instead, use the `zip` program on the command line. Alternatively, you can check **all** these items into Github, download a zip file from Github, and submit that to Gradescope.

Grading

The grading in this project will be broken down as follows:

Item	Percentage of Grade
Program correctness	85%
Style and documentation	15%

You will receive full credit if (1) your code compiles, runs, and produces the expected output, (2) you have not used any illegal libraries, and (3) your `README,md` is clear, your source code is readable, well-documented and robustly implemented, and your submission does not contain any prohibited files, e.g., binaries or object files. All student code will be scanned by plagiarism detection software to ensure that students are not copying code from the Internet or each other.

To test your *4700ftp* client, we will ask it to perform a number of tasks on the command line, including uploading and downloading files, creating and deleting directories, and deleting files. In other words, we will exercise its expected functionality. If your client does not obey the command line specification then we will not be able to exercise its functionality, and you will lose points. Similarly, points will be lost for each expected feature that is not implemented correctly.

← Project 1: Socket Basics

Project 3: BGP Router →