**FTPy!**

**An FTP Client written in Python!**

**Tutors:**

**T Dahl**

**L Stewart**

**SOFT050 Project**

**By Ben Moore**

**Introduction**

I decided for my project to create a command line peer-to-peer FTP client with the ability to send and receive multiple types of file all in one program from the terminal. I had hoped to add RSA encryption to the client so it would be truly encrypted peer-to-peer file sharing it but I didn’t manage to get the encryption working in time.

I chose to create a peer-to-peer FTP Client because at the start of the year I became interested in network programming and wanted to know more about how computers share files over a network. Using TCP/IP sockets.

**Technology Review**

Language

For my project I decided to use Python 2.7 because I started learning it via Codeacademy in September and it is the language I have the most experience using and have the most knowledge about. It also comes pre installed on my Macbook and runs very well on UNIX based systems.

IDE

For my IDE I decided to use Github’s Atom text editor because it has Github integration, which I used to be the version control system for my software. It also includes python syntax highlighting and allows you to edit multiple files at once while viewing the whole project directory on the side of the screen which proved a very useful feature. As it was a command line based program I decided to test and run it from the terminal.

**Design**

As I progressed through the design of my software I realised that the implementations would require a few re-designs from my initial idea. After doing constructing the basic functionality I rebuilt my project again from the ground up after reading about how threading works and realising I would need to implement that into my project in order to achieve the desired functionality I had initially envisioned, below you will find a flow chart of the final design of my project as a whole.

**User returns to**

**Connection failed**

**Send**

**User returns to**

**File receiving failed**

**File received**

Thread

Main program

Key:

**Receive**

**User Interface**

The user interface is relatively simple I used a piece of ASCII art upon start-up to welcome the user to the program and then a command line based main menu which asks the user for a number input. This number input is then used to decide what the program should do.

Once the user enters a number the program will prompt the user to enter the local IP of the machine they’re trying to connect to (It currently only works over local networks). If the user is receiving a file you can leave this blank which opens up all addresses. But for a more secure option you can also type in the address you’re planning to receive a file from in order to only receive a connection from a specific machine. If the address is left blank you also get the option to open a multiple amount of connections if you’re expecting multiple files from multiple clients. If you’re sending you need to enter the address or hostname you’re sending to (hostnames will be converted to IP addresses before being passed to the program) and then you will be prompted to enter the name of the file you want to send in a similar way. Once the file has been chosen and provided it’s in the same directory as the program It will be sent while the user will be sent back to the main menu.

**Code Design**

In order to optimise my code I’ve broken my code up into three files these three files are named as follows:

FTPClient.py:

This file contains the main function of the program and is the file that is initially run to execute the program.

Config.py:

This file contains two lists and two Boolean variables, which need to be globally available to the whole program.

FTPClasses.py:

This file contains all the class declarations, which are used in FTPClient.py, and also contains all the functions of the classes including one stand-alone function.

FTPClient.py:

This file contains the menu() function which is used to print the main menu to the screen and prompts the user for their numerical input. This then gets passed to an if-else statement, which declares instances of classes needed to execute the users intended function.

There is also a section that allows the user to view what the program is currently doing by printing then number of active threads as well as printing to the console an array of the currently executing threads.

Config.py:

This file holds both the variables and arrays, which are required to be globally available to the whole program. These two arrays hold the currently active sending and receiving threads that allow the user to see what the program is currently doing.

FTPClasses.py:

This file is a lot like my programs toolbox, I designed it so all the class declarations and the functions of the classes. This is so if I ever wanted to create a project with similar functionality I can use my own libraries of functions and classes I’ve already written. It also contains some stand-alone functions that are globally available to all classes. I designed my classes so that I can have one main file transfer class which creates it’s own socket as well as holding all the functions for getting information about the machine the user is connecting to. An instance of this class is not created when the program executes instead I created two other classes, sendClass and recvClass which both inherit from the file transfer class. This was done in order to minimalize the amount of code needed so my program adheres to the DRY principle of software development. There is also a loading\_screen class inside this file is defined as a thread object. This class was designed to initialize inside other classes functions it’s use is to display a spinning ‘/’ while the program completes another task in order to let the user know the program hasn’t crashed.



This class as a relatively late addition to my project as it wasn’t critical to demonstrate the core functionally of my program but is a vital part of the user interface design. This class is unique in my project because it inherits from the pythons own thread class. I found out how to use this and the advantages of using a thread class quite late into my project as this was my first foray in to using threading, in retrospect I would if I had more time to write and test the code have re-written some of the other classes in my program so that they too inherit from the thread class as well as their current class using multi-inheritance which would have a more efficient way than the way I’ve currently implemented my threading system.

Below you will find a summary of my programs class relationships.

sendClass():

Methods:

getSendInfo()

sendFile()

threadSend()

FileTrans():

Initialize Socket

Methods:

getConnectionInfo()

connectSocket()

Loading\_screen():

Methods:

Run()

recvClass():

Methods:

recvFile()

threadRecv()

Any methods in red run in separate threads.

It should be noted that the threadSend() and threadRecv() functions open threads which run the sendFile() and recvFile() functions respectively.

**Implementation**

In order to send files over a network my first challenge was to devise a way to save a file as a variable so I could send it using a socket. After some research I found out about the Base64 (Python Software Foundation) Python module which allowed me to open a file as a variable then encode that file into a string of ACSII characters which I could then save as a variable and send over a socket and decode on the other end and save the file with the same name and extension. First move in testing this was creating a small script that opens a file encodes it, decodes it then save it under a different name to make sure that I could see that the encoding works and get some practice using the encoding functions before writing it into my main program.

The next challenge in creating my program was receiving a file. I discovered that I wouldn’t know the size of the file the client would be receiving. When receiving a data packet I learned, you need to set a buffer. Instead of setting a giant buffer that could always be too few bytes to receive whatever file the client is receiving I had to set about finding a more efficient and elegant solution.

The solution I devised was to use a small buffer on the recv() (Foundation)socket command but run it in a loop appending each small chunk of date too the previous piece until the socket stop receiving data. Then once the socket has stopped receiving data I will have the complete encoded string.

In the above screenshot you can also see the solution to the next challenge I found while writing my program sending the filename with the program. In the first version of my program I had a set filename written into the program to test it, the problem being that it saved every file under the same name with the same file extension that limits the file type you can send to whatever extension is written in the program so I had to devise a way to send the file name and extension with the encoded data. In the previous you can see the following line of code:

**file\_name,received\_file = received\_package.split(‘:’)**

This line splits the name from the encoded data because in when sending the file I put a character that would neither be in the file name or the encoded data, which is the colon. Below you can see how I implemented adding the file name to the string of encoded data when sending the file.

Once the file name variable has been appended to the encoded data then it’s saved as a variable named package, which is then sent across the network.

After I’d solve this problem my next challenge became apparent, when running my program it could do one thing at once. This was going to be an issue because while my program was waiting for a connection or sending a file it would just appear to hang and this wouldn’t be good for an end user who may assume that the program was hanging and therefore not working. I also wanted to implement functionality where a user could listen for a file while being sent back to the main menu so they could exit the program safely or send a file simultaneously. To achieve this I had to do some more research until I found out about the threading(Foundation, Python Theading Module Doccumentation) Python module after doing a some research I decided to implement threading into my program so I could run the code that is sending or receiving a file. This would give the user added feedback in know the program has not crashed. In the case of receiving a file the user would be returned to the main menu to they could still use the program while a thread is listening for a connection. Or in the case of sending a file it allows me to print a loading screen to the terminal with a spinning ‘/’ character which again lets the end user know the program hasn’t crashed. After doing my research I decided to implement my solution into my already existing class structure.

I did this by adding a function to both my sendClass and my recvClass these classes are named threadSend and threadRecv respectively.

Both of these classes do the same thing, they create a thread in which the sendFile and recvFile functions are executed and they’re also added to either the sendClientHandlers or recvClientHandlers array which then means that once the thread has been declared and appended to an array it can be started then the variable will be set back to a NULL variable. This was done so that the function could be executed again and a new thread created, this was done so I don’t need to have a new variable every time I want to create a new thread. This was done because I wouldn’t know how many times the user would press the button which would open a new thread and I didn’t want my program to be limited because the whole idea of the multithreading was that the client could handle multiple connections in the true nature of an ftp client. It also allowed me to condense my program into one universal client that can do both sending and receiving instead of having two separate programs one for sending and one for receiving. This was the way my first implementation of the program worked this in turns demonstrates the way that my program has progressed over the course of my project as you can see in the screenshot below.

After the implementation of my threading system I could move both programs I had at that point into one program. Once I had both the sending and the receiving parts running together with my threading implementation so by this point I had all the basic functionality of my program in place.

**Bug Fixing**

Once I’d devised a way to create the basic functionality of my program I set about fixing all the minor bugs in my program. Through my testing I’ve managed to fix a lot of the bugs that cause the program to crash or stop or hinder the core functionality of the program.

The last bug I didn’t get time to fix is a problem with the threading system; I’ve done some research into what’s causing the bug but didn’t get time to fix and test the fix fully. This bug doesn’t cause the program to crash It just presents an error when the user tells the program to listen for connections, while it is already listening connections. I’ve tried to add a ‘listening’ Boolean that changes to true when the program is already listening but sometimes it still presents an error which is a problems I haven’t solved as of yet.