`

Hotspot (Proxy Server)

Lydia Noureldin

Design Challenges & Model Assumptions

In order to create a server client application a few a design requirements had to be met for the project. I assumed that if a URL was in the proper format, meaning the communication protocol, followed by the website address, it existed. For example, https://www.somewebsite.com is a website that is in the proper format but does not exist. Due to limitations of the "Django" package, my software can only determine if it is in the proper format and cannot determine if it actually exists. This is due to the fact that the **isValidURL** function (using the "Django" package) on the runServer.py class checks to ensure that the URL is in the correct format but does not ensure that it is valid. This allows us to also make the assumption that, if presented in the correct format, the URL is valid.

In addition, the server would have the ability to handle more than one client but for the sake of this project we assumed a maximum of three concurrent connections would be made. I also assumed that the TCP implemented in Python guaranteed in-order delivery of packets, this was specifically important in the **myreceive** function in the Client.py class. Lastly, I am operating under the assumption that the random generator for the access number would actually be random.

Obstacles I faced when deciding the structure of the application were how many files would be necessary for the architecture of the program. I knew that at least two were needed, one to run the client-side and one to run the server-side. Additional files came into the framework to run the server and client, as well as store the client information. Secondary challenges that presented themselves were if it should object-oriented or not. The software's object-oriented nature kept things encapsulated and tidy. It was also very convenient to work within a class with its local attributes. Issues arose when I wanted to use the class' attributes in another class. More specifically, the challenge of how the server and client class would communicate the usage details became increasingly challenging under the original two class architecture.

Technical Difficulties

Getting the server and client to connect presented itself as my first challenge. After running the application in the initial stages I found that there was no communication between the two interfaces. I later learned that the communication protocols being used by the client was UDP and the server was TCP. This was caused when I was creating the sockets in both classes. The server used SOCK_STREAM and the client used SOCK_DGRAM. Upon setting both to TCP (SOCK_STREAM), a connection between the two was able to be established.

Secondly, getting the usage details from the client class to the server. Originally, I thought to pass the whole Client object through the connection. A module in Python called "pickle" could achieve object serialization to pass it as a “bytes” object. When I implemented this, the result was disastrous, making the code increasingly tangled. My second approach was much better. I created the “ClientInfo” class, which contained all the usage details and is directly accessible by the Server class. This removed the need to pass objects from the Client to the Server through the connection.

Another issue arose was when I wanted to implement the bonus of actually pulling up a webpage in a default browser. The problem was, on the client side, I could not distinguish between messages the server wanted to print to the console and the html for the requested webpage. I solved this by adding a Null indicator, ‘\xff’, to the end of the console messages. Then on the client side, I created a **separateMessage** function that took in the decoded string representation of the data sent from the server, and split it into the console message and html message, using the ‘\xff’ as the point to split the two.

Furthermore, difficulties in testing the platinum ability to look at other users usages through https://www.clientusage.com special address was overcome by introducing a new optional parameter in the ClientInfo constructor. This parameter allowed me to do these required tests since it forces a client to have the specified category no matter what their random access number was.

System Model

The system can be broken down in 6 files: Server.py, Client.py, ClientInfo.py, runClient.py, and runServer.py,

3 classes: Server.py, Client.py, and ClientInfo.py

1. Server interacts with ClientInfo. When a client creates a new connection, Server calls on ClientInfo to keep track of the category, access number, and usage details.
2. The original Client class was split into two so that server object could access the information more easily. As mentioned above, this removed the need to do object serialization

2 executables: runClient.py, runServer.py

1. Uses the methods in Client and Server to establish a connection, listen for connections (Server), request connection (Client), and send messages (Client)

1 text file: urlContents.html

1. This will only exist after the application is running and the client requests a real URL. It contains the real HTML content of the requested URL, and is then opened in the client’s default browser.

Future Work

In the future, I could find a way to merge the Client.py and runClient.py files to make things neater. Although having more files worked for my intents and purposes, they are not necessary, and could be condensed. I would also add updates every time the user used a valid URL to let them know how many URLs they had remaining. Lastly, I would create a portal on the default browser explaining to the users how the system worked and how to connect. That would make the project fully polished. All in all, excellent experience working on this, my software is small-scale, but I still find it to be comparable to systems such as TConnect wifi service provided by the Toronto Transit Commission (TTC) that I grew up with. Taking the time to read the various libraries and modules allowed me to truly understand this, other future work could be to do research in the field to develop safer, and more efficient algorithms to implement Hotspot/Relay or Proxy.

Instructions for Running

1. Ensure Python 3.6.0 is installed on your computer.

2. Make sure the “Django” package is installed. Pycharm ⇒ Preferences ⇒ Project ⇒ Project Interpreter ⇒ + ⇒ type in “django” ⇒ install package
3. Make sure you are connected to the internet
4. Run the server first from the command line. For example:
“python3 /Users/LydiaNoureldin/PycharmProjects/CISC435_Project/runServer.py”
5. Open a new terminal session for each client you want to add, and run the client. For example:
“python3 /Users/LydiaNoureldin/PycharmProjects/CISC435_Project/runClient.py”
6. Input the URLs in the format “https://www.somewebsite.com”
7. Quit when you are ready to terminate