CISC 435 – Computer Networks – Fall 2018
Course Project

Due December 6th, 2018 before midnight

Student Name: Mitchell Mulder Student Number: 10137002

Component	Application	Sample cases	Report	Total
Score	/13	/2	/5	/20

- Please submit this as the cover page of your report.
- This is an individual project – you cannot work as groups. Plagiarism in any form will be treated as per "Queen's University Academic Integrity guidelines".
- Programming Language choices: Python, Java and C/C++ only.
- The report, sample test cases, and source code are to be uploaded to OnQ.

Deliverables

1. The report, detailing:
   a. Design challenges and model assumptions made
   b. Difficulties you faced and how you handled them (if you faced no difficulties discuss the scalability of your app and how you designed for it)
   c. Your system model with justifications
   d. Future work (if you had one more month, what would you do differently!)
   e. The procedure required to compile and run your program
2. The source file (Use comments to document/describe your code)
3. Sample test cases

CISC 435 Project

Design Challenges and Assumptions

When designing the client server application, I first ran into the problem of how to serve multiple clients at the same time. To do this concurrently, the server would need to use threads to handle the incoming connections to a maximum of three users. The way I implemented this on the server was inside a while loop I would accept connections and once a connection was made, I would then spawn a thread to handle all the client to server actions until the connection was closed.

One assumption made during the design of my project was that only a valid request of a cached image would be counted towards the client's quota. Thus, if a user made an invalid request or just wanted the list of cached images this would not count towards their quota. Also, if a user quota had a maximum of three requests, on the third request the connection will not be closed. The user will still be able to run any command; for example, list the cached images or exit. Not until the user tries to exceed their quota, will the connection be closed.

The second assumption I made was when a fourth client tried to connect to the server the connection was closed on the server's side. However, the client would not notice the socket was closed until the client tries to submit any command to the server. The broken pipe error will be caught and will notify the client that the server closed the socket.

Difficulties

One of the difficulties I had during this project was trying to implement a way to serialize a real image back to the client. When first trying to send the image in bytes over the socket the user would only get a part of the image. No matter how big I set the maximum bytes to receive I would not get the entire image. I searched around for a solution and found out my issue was with the socket protocol I was using TCP/IP was stream based. This means that for every socket send from the server does not necessarily match with one socket receive from the client. You can get around this by creating you own message-based protocol on top to ensure the entire message or in this case image is sent. The way to do this is to append a header with message length to the packet to tell the client how much data to receive. With this implementation I was able to send images from the server to the client through the socket connection.

Architecture

The way I designed my client server was to have a single server that was multithreaded to handle multiple concurrent client connections at one time. To do this, each time a new connection was made I incremented a counter to keep track of the open connections. Then I spawned a new thread to handle all the client's requests. Once the client was done the

connection was closed and the connection counter was decremented. I also assumed that there would not be any race conditions that could cause the shared connections counter to be inaccurate.

When the server is first started up it creates a history json file to keep track of all the requests from the active and past disconnected clients. When a client first connects to the server is it given a unique name and a random code. This is used to identify clients and determine what access level they have. The user can ask for this information with the list command. Whenever a client makes a valid request for a cached image it is added to that specific users log in the history file. It will also specify the image requested and the date and time when the request was made. If a user is a platinum user and wants to view the usage details, they only need to enter the usage command to the server and it will return the history json file to them. If the user is not a platinum user, the server will reply with an invalid command.

On the client side the design is a lot simpler since there was no need for multiple threads. When the client is started up it will connect with the server. The client will then prompt you to send commands to the server. Once a command is entered and submitted the client will send the command to the server through the opened socket connection. The server will handle the request and respond back. The response will then be printed out to you. If the request was for a cached image, then the client will stream the byte encoded image into a new file inside the client's cache folder and notify you the image has been downloaded. There is no need to create the client cache folder since the client program will handle that.

Future Work

One feature I would like to implement if I had more time would be to setup a simple kind of handshake protocol at the beginning of the client to server connection. The way I have implemented the connection process is that I assume the server accepts the client connection and if the server does not, then the client will not notice until they submit their first command and are prompted the connection was closed. This would happen if a fourth client tried to connect to the server, but the server would be over its maximum connections and would close the socket. A simple way to do this is to allow the server to first reply back to the client if the connection will be opened or if the server is over its maximum connections reply back that the connection is closed and close the socket. Then the client will know if its connection was accepted of closed by the server. Thus, if a client tried to connect to a server that was overloaded it would be notified the connection is closed and the client program would immediately notify the user and terminate.

Another useful feature to add to this client server project is logic to see if the cached images on the server have been updated the last time you requested for them. This would require the server to keep a timestamp on all the cached images when it was last updated. The client would also keep a timestamp on its cached images on when it downloaded them from the server. If a client requested an image that it already has, and it has not been updated since the last time the client downloaded the image, the server would not need to send the client anything. If the

client requested an image that it already has, and it has been updated since the last time the client downloaded, then the server would send a new image back. This would be a lot more efficient for the server if you would want to scale the application.

Setup

To run my project please use **Python 3.7.1**. Do not use Python 2.x or the project simply will not work.

To create the server to handle incoming requests from clients run this command inside the project folder:

```
python ./server/server.py
```

This will start the server and begin listening for connections. To exit just type control-c. To create clients to submit requests to the server run this command inside the project folder:

```
python ./client/client.py
```

This will create a client and prompt you with the list of available commands to submit to the server. Simply enter the commands and the server's response will be printed out.

Test Cases

To run automated test run this command inside the project folder:

```
python -m unittest -v testClientServer.py
```

Test Case 1: testClientServer

In this test case I start the server and three clients that ask for the list of the cached images and exit. This is just to test that the server can handle multiple clients concurrently.

Test Case 2: testMaxClient

In this test case I test to see if the server will allow more than its maximum of three concurrent connections. I start a server with four clients concurrently. The fourth client when trying to submit a command to the server should catch the broken pipe error and close the connection.

Test Case 3: testImageCache

In this test case I test to see if the image serialization from server to client is working. I bring up one server with one client. With the client I request for all three test images I have set up on the server and see if they are all downloaded into the client cache folder.

Test Case 4: testQuota

In this test case I test if the client quota is enforced by the server. I bring up one server and one client. I try to request for six images. Based on the client code I determine how many requests should go through before the server will close the connection. E.g. If the code is 108 then the maximum requests is three. Therefore, on the fourth request the server will not respond with a cached image but will close the connection.

Test Case 5: testPlatinumUser

In this test case I test if the platinum user can query for client usage. I bring up one server and one client. Then I query for clients' usage. If the client is not a platinum user, then I expect an invalid request. If the user is a platinum user, I expect the response to be usage details.