**Concurrent Socket Server**

Shyam Rajendren

CNT4504 - Computer Networks & Distributed Processing

Professor Scott Kelly

Due Date: 08/03/19

**Introduction**

The purpose of this project is to become familiar with the way clients and servers interact with one another. This specific project helps to show the interaction between a multi-threaded client and server. Through the coding of such a client and server, the student becomes familiar with the characteristics of the two which helps further their understanding.

The goals of this project include:

* Analyzing the turn-around time for individual client requests and seeing how increasing the number of client requests impacts the turn-around time.
* Analyzing the total turn-around time for all client requests and seeing how increasing the number of client requests impacts the total turn-around time.
* Analyzing the average turn-around time for the client requests and seeing how increasing the number of client requests impacts the average turn-around time.
* Contrasting the average turn-around time results contrast from the iterative server and the concurrent server.
* Determining if there are any situations in which one would use a concurrent server over an iterative server and vice versa.

This document will discuss the general design of the client and server programs and the various decisions that were made while designing the two programs. The basic operation of the multi-threaded client and server will be discussed as well. The method used to test and collect data will be discussed along with the various charts and graphs that visualize the results of these tests. These charts and graphs will be analyzed and a conclusion will be drawn from this. Finally, any lessons learned while writing the programs and collecting the data will be discussed.

**Client-Server Setup and Configuration**

The client program is designed to accept the hostname and port number from the user and relay this information to the server program which in turn creates a socket server from this information. The client then accepts one of six commands from the user (Date and Time, Uptime, Memory Use, Netstat, Current Users, and Running Processes) along with the number of requests of this command the user would like. This information is relayed to the server which then executes the command. The individual turn-around time, total turn-around and average turn-around time are calculated on the client side. The above information along with the results of the command is displayed to the user along with each individual thread id. Once this is done, the user can choose to send another request or exit from the program.

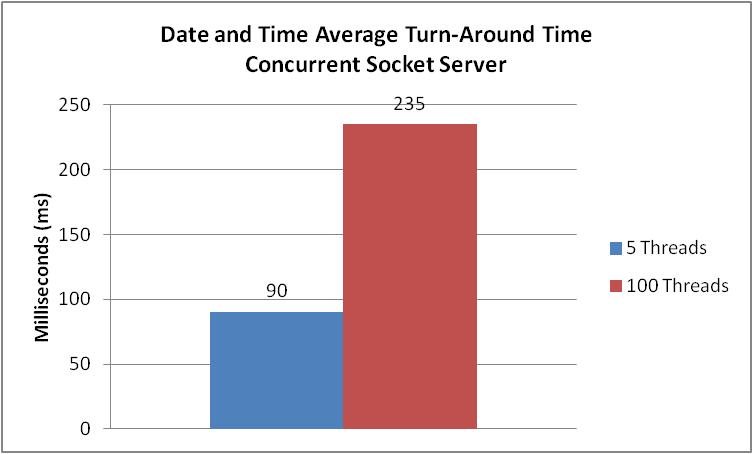
It was decided that while writing the programs that a custom class called “Clients” would be created on the client side which could create a new thread for each request. These threads would contain all the previously discussed data and would be displayed to the user once all threads were completed. This allowed for the data to be displayed in a simple and neat manner. A custom class called “CustomThread” was created on the server side. This class would run a particular command depending on the user request and send this information back to the client. It was decided not to store the turn-around times on the server side as it was deemed easier to both calculate and store this data on the client side.

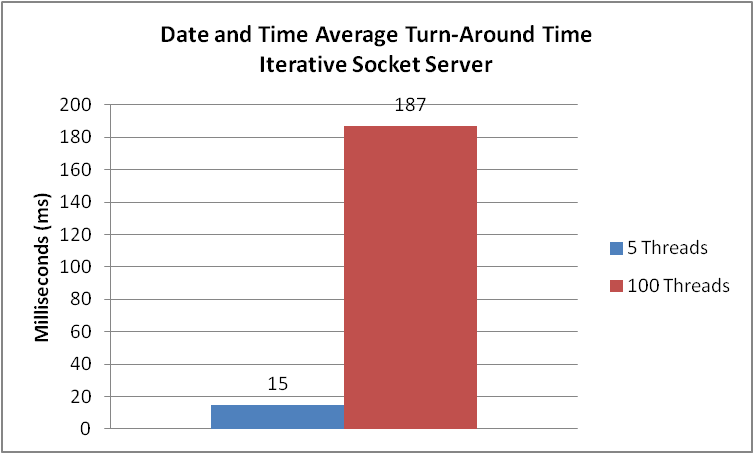
In order to run the multi-threaded client and server program, the user must first run the Server program. Once running, the user will run the Client program and input the host name of the server along with the desired port number. Once this is done, a connection is established between the client and server. The user will then be prompted by the Client to choose a command along with the number of requests. Once the user inputs the necessary information, the results of the request will be displayed. The user then may make another request or exit the program.

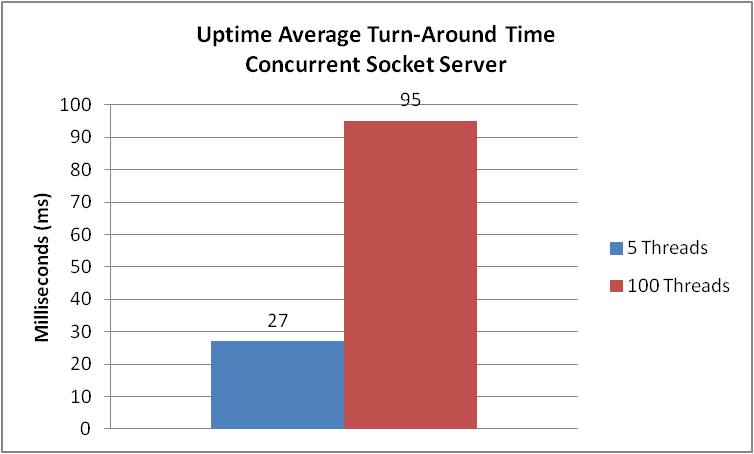
**Testing and Data Collection**

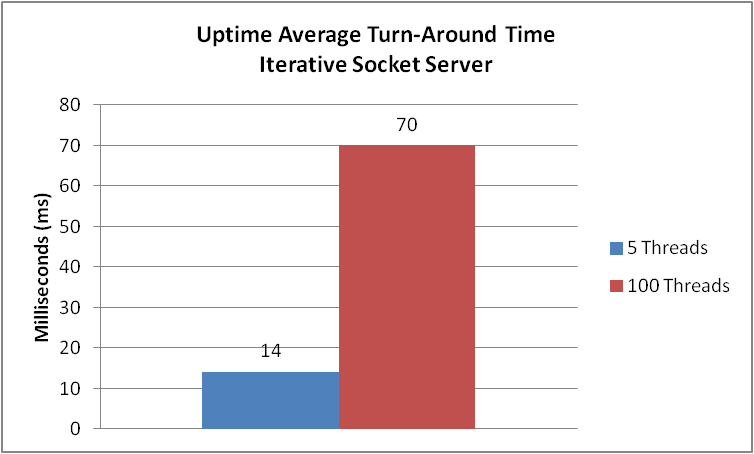
For each of the six commands, a 5 threaded request and a 100 threaded request was made, and the average turn-around time was noted. This was done for both the Iterative Server and Concurrent Server, and these results were tabulated on bar graphs. Finally, a line graph was compiled for each command comparing the average turn-round time between the Iterative Server and the Concurrent Server.

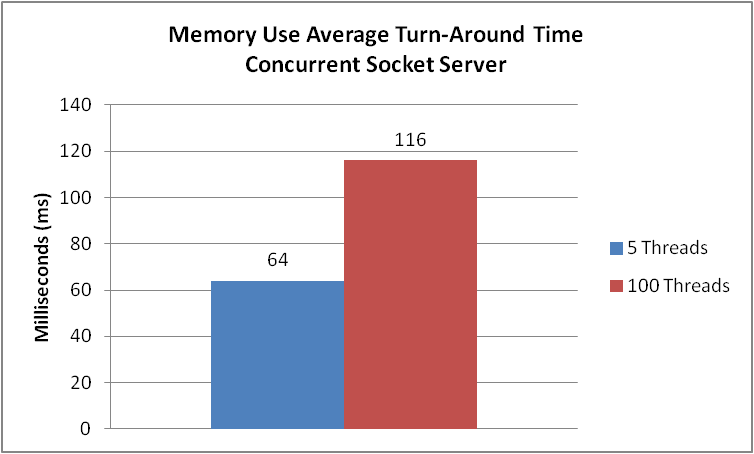
Here are the results of the data collection:

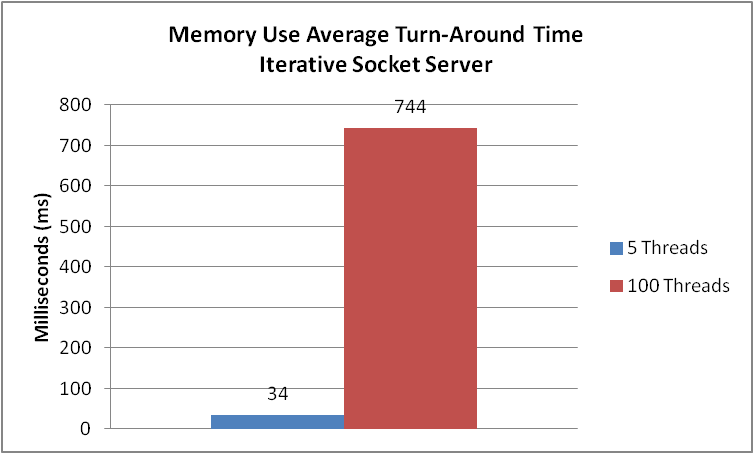


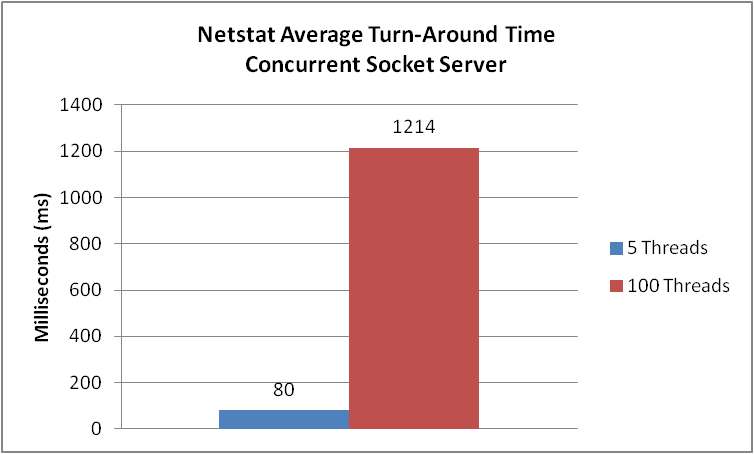


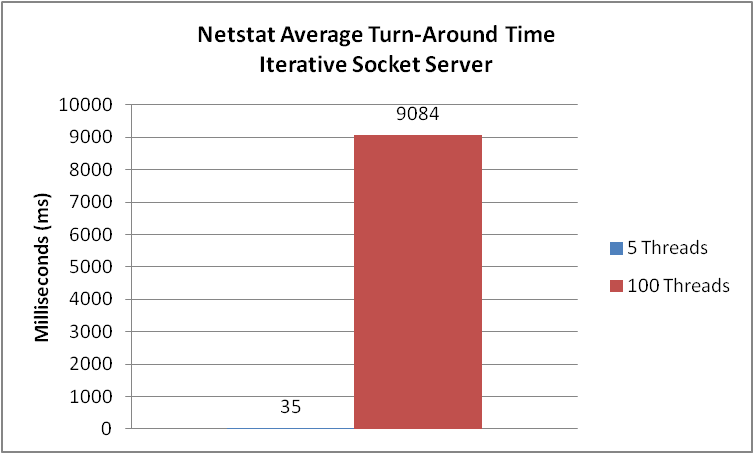


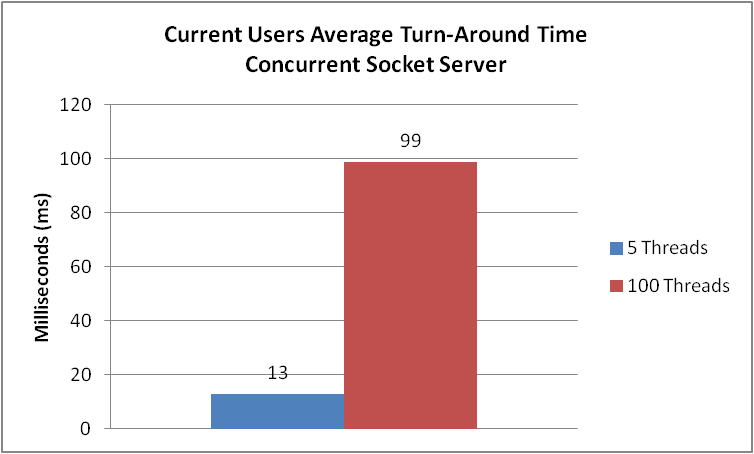


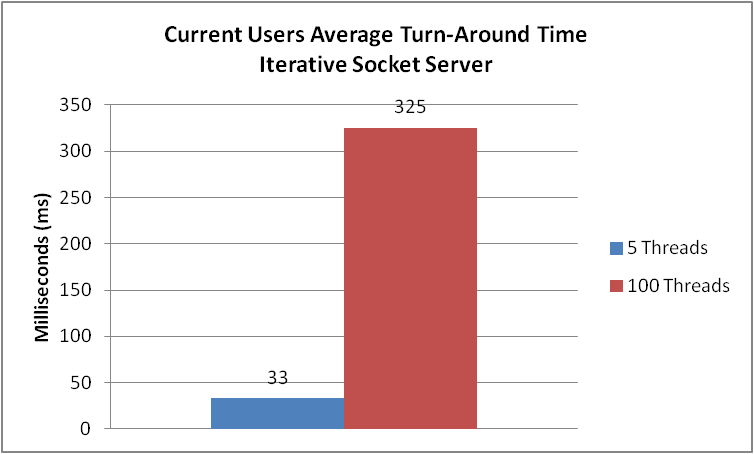


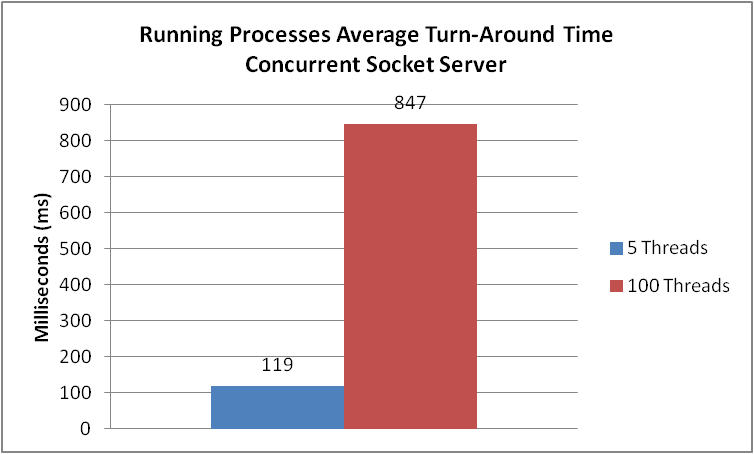


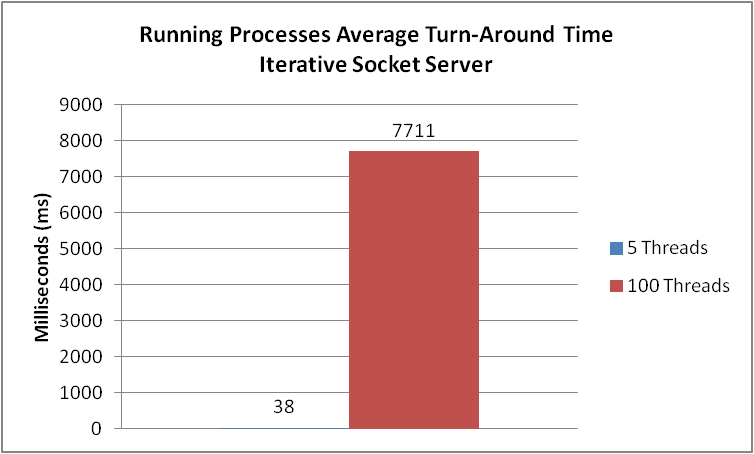


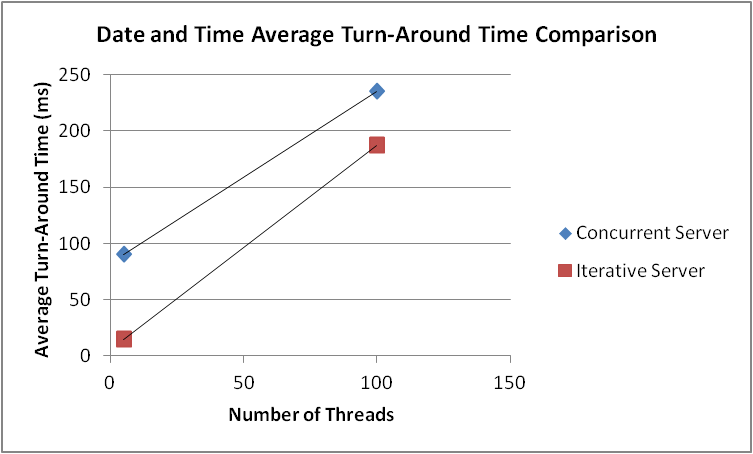


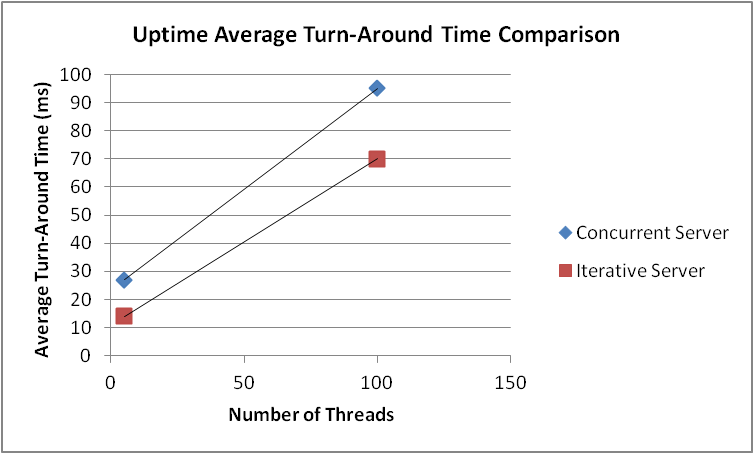


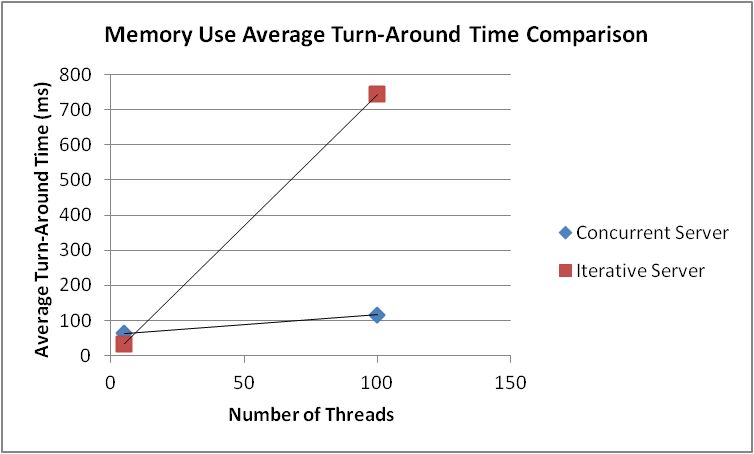


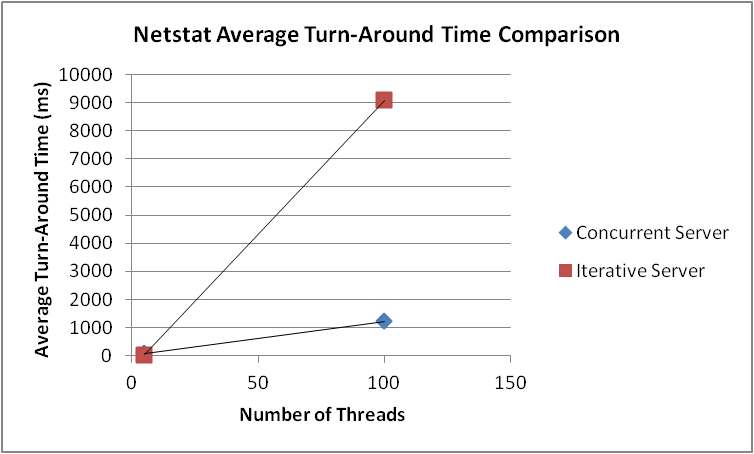


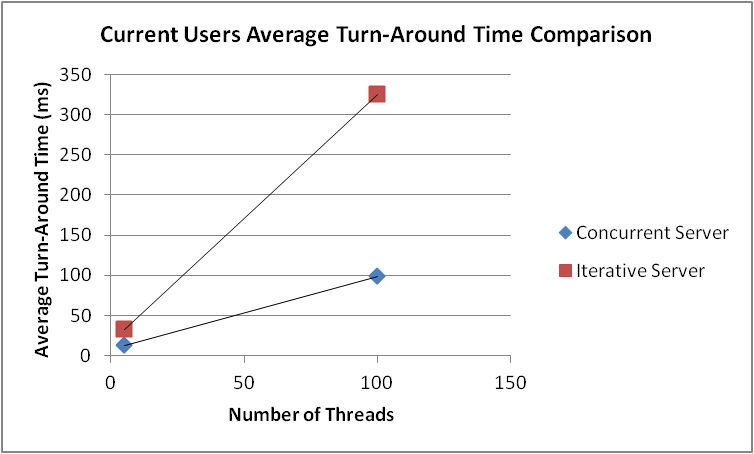


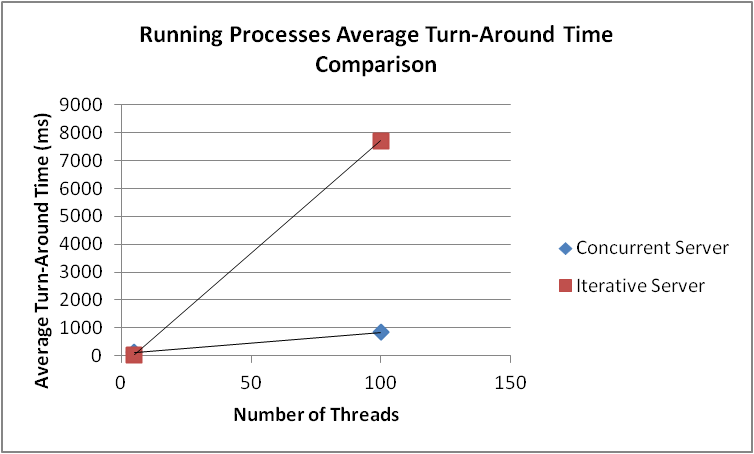












**Data Analysis**

As one would expect increasing the number of clients led to an increase in individual, total and average turn-around time for both the concurrent and iterative servers. However, the most interesting inferences we can draw from this data is when we compare the average turn-around times of the iterative and concurrent servers for each operation. We can see for Date and Time, and Uptime. The Iterative Server performs better than the Concurrent Server. However, we see a stark difference when we look at the other four operations. While the Iterative Server performs at the same level or even better than the Concurrent Server when there are just 5 threads, we can see that the Concurrent Server performs vastly better when there are 100 threads.

If we look at the two operations in which the Iterative Server outperformed the Concurrent Server (Date and Time, and Uptime), we can see that they are relatively simple commands that would not take a lot of processing power. As such, we can say that we can use an Iterative Server over a Concurrent Server if the operation to be executed is relatively simple. Also, since the Iterative Server mostly performed better than the Concurrent Server when the number of threads was 5, we can also say that an Iterative Server may be used when the numbers of requests are very less. Inversely, we can that we should use a Concurrent Server over an Iterative Server if the operation to be executed is complex or if a large number of requests must be processed.

**Conclusion**

The results given by the two types of servers can be attributed to their very nature. An Iterative Server handles a single request at a time and must wait for one request to finish before starting the next one. This is mirrored by our results which shows the Iterative Server taking less time to process a fewer number of requests and also taking less time when the operation to be executed is relatively simple. A Concurrent Server in contrast can handle multiple requests at a time and each requested can be completed at different times regardless of the order in which they are received. We can see this in our results which shows the Concurrent Server performing better when there are a greater number of requests or when the operation to be executed is time complex. In conclusion, we can say the Iterative Server is better equipped to handle simple and fewer requests while the Concurrent Server is more suited for more numerous and complex operations.

**Lessons Learned**

The Concurrent Server was relatively easier to write as we had previously completed the Iterative Server. The main problem lied in trying to multi-thread the client and server. This was eventually done by creating a thread for each request and having the client-side store the results and accompanying data. This data would then be displayed when all requests had been completed. This project really got home the pros and cons of a Concurrent and Iterative Server. By comparing the average turn-around times, I was able to clearly see the scenarios in which each type of server performed best in.