# **Network Management with Socket API report**

Group 3: Thinh Nguyen, Suraj Mohan, Mark Musseau, Kyle Dortch

CNT4504 - Computer Network and distributed processing

# 

[**Network Management with Socket API report**](#_hqdedb25aae6) **0**

[**Abstract**](#_uppnn7gyv23u) **2**

[**Introduction**](#_kcxaf94qnnuf) **2**

[**Distributed Application Development**](#_krpaq3se8m88) **2**

[**Results and comparisons**](#_dvdlssywlkrc) **4**

[Test Bed](#_4avm1wl9bruf) 4

[Studies carried out](#_np7uqybixy48) 4

[Results](#_gqd41dwtvfw2) 5

[**Conclusions**](#_xeiehkwn04z8) **8**

[**References**](#_viwg2rmw3f4h) **9**

# 

# 

## 

## 

## 

## 

## 

# **Abstract**

We created Iterative and Concurrent servers that could be accessed by a client. The servers were designed to provide the client(s) with information about the server through a menu command. Tests provided the times required for the server to complete each request with a various number of clients.

# **Introduction**

Our project goal was to design a network and client program that would allow clients to access our server. Once a connection is established through socket programming the client will be displayed a menu with a list of commands numbered 1-7. These commands included host commands: Current date and time, uptime, memory use, netstat, current users and running processes, respectively, with the last command allowing the client to exit the server. The project was initially designed to use socketing to the server and multi-threading with the client, creating an iterative server. This would allow multiple clients to access the server but only one process can be executed at a time. After completion of our tests with this, we added multi-threading to the server for the second part of our project to allow for multiple processes to be executed from multiple clients, thus creating a concurrent server. The goal of this was to have a more efficient running server and allow the clients to receive their requests faster. We further tested the retrieval times for the client commands and recorded the results to compare the two servers. The server is designed to return the times used to execute each command to show if the server was responding in an efficient manner with various amounts of set clients and to ensure we were meeting our goals within the design. Results are expected to vary from iterative, concurrent and their reaction times to the various amounts of clients.

# **Distributed Application Development**

Distributed applications are applications or software that run on multiple systems simultaneously for a single task/job. Distributed applications are broken up into two components: the client software and the server software. The client software handles client(s) requests to complete a certain job and the server software processes the requests from the server-side [1]. When a typical user connects either directly to the server or indirectly through the internet with a URL, a connection (TCP, or SSH like in this class) request from the client is accepted by a server. The authorized client(s) then sends over requests of which it wants the server to do, can be from displaying information, or transferring files or services, etc. The server then sends over the requested information/service in a timely manner. After the contents are sent, the server will release the connection, free up its bandwidth to take in other requests. That is a simple design for a single-threaded server. These kinds of servers cannot handle multiple requests either from one client, or multiple clients. One way the server handles requests is to complete the job iteratively. The server will complete one requested task, then move on to completing the next request, either from the same client, or the next client in the queue.

However, nowadays’ web servers need to be able to handle as many requests a possible from many clients as fast as possible. The problem with simple iterative servers/system is that the bottleneck effect when multiple tasks is needed to be done but there’s only one system handling a limited amount of resources. If the current requested task is big and takes a lot of time to process, all other tasks will be blocked for a long period of time. Many solutions have been established, but the most efficient of all is to multithread the server. Regarding multithreading, the server’s front-end will accepts all incoming requests and k processing modules. A k+1 number of threads is created and they all have accessed to a cache inside the process’ address space. The idea is that whenever a request comes in, the server accepts it and builds a short record of it in cache, then hands the record down into one of the processing modules [2]. The advantage of this over single-threaded processes is while one or more processing modules (or requests) are blocked waiting for another task(s) to be completed, other modules can be actively working on other tasks. Modern web servers, after a client established a connection and ready to receive/transmit data, a list of steps is going to occur in the server: resolve the name of the requested page, perform access control, check the cache, fetch the requested information from either a disk or a program, determine the rest of the responses, return the response to the client, and finally make an entry in the server log [2].

In this project, we accomplished creating a server-client connection with the server being both single-threaded and multi-threaded to handle simple requests from a specified number of clients. These programs will not do anything complicated as nowadays modern web servers, but only do simple client-server connection via SSH and transmit information back and forward between the two machines. The expected outcome for this project is to prove that by using threads, requests are handled much faster and are not affected by the number of incoming requests. Hypothetically, iterative server is better at handling less (preferably just one) and simple requests from the client. On the other hand, concurrent server will be better at handling multiple requests at once, but will do slightly not good at too few requests. This project is done in Java by creating a socket object to every client connecting to the server (ex. Socket a = new Socket (IP address, port number)). The server will create a server socket by using the command (ServerSocket a = new ServerSocket (port number)). The IP address in the client socket call has to match the server’s IP address, and the port number from both side’s sockets has to be the same in order to have a connection. The server then accept() any incoming connection. In this experiment, the server and clients communicate through DataInputStream and DataOutputStream objects, which send requests from the client’s side to the server and responses back from the server to the client(s). The connections are then closed after every task (with single-threaded server) with the socket closed() method. In the multithreaded server, each time a client is connected, the server will create a thread object to handle the clients’ requests. This simple server program will be able to return the server’s statistic to the client(s) who requested it from a menu. The menu will include statistics such as the server’s current date and time, uptime, netstat, memory use, current connecting users, running processes, and an ability to quit the menu if the client(s) wants to. The mentioned functions are just the crucial functions that is needed to test out our hypothesis. However, in order to actually implement it, there are many more functions needed in order to create a simple client-server interactive program.

# **Results and comparisons**

### Test Bed

The specifications of this server are two Intel Xeon E5-2670 CPUs clocked at 2.60GHz with 8 cores and 16 threads each. This virtual machine has a total of 4 gigabytes of memory and 3 gigabytes of available memory. The operating system running on these machines is Linux Ubuntu 18.04.3 LTS (Bionic Beaver). The LAN used while working on this project is Osprey.

### Studies carried out

The metrics used in measuring the latency in milliseconds. They were measured by increasing the number of users and measuring the latency between asking for the command and returning results. We did this by recording the current time right after retrieving the output stream, which in this case was the user’s input and then subtracting that from the time recorded right after the server prints back the results. By doing such, we end up with the difference in time between the two actions, this is the latency between asking and getting back the data. After the user types in the IP address of the server, the client asks for the number of clients the user wishes to connect. By threading the client and using three for loops to first initialize the threads then perform the start function and finally join each other we ended up with an iterative server result. We then gradually increased the number of clients as we asked for 1-7 commands and recorded the latency times. This concluded in the graph of latency (ms) of an iterative server vs. different numbers of clients requesting the server to display the date and time and another graph comparing netstats. These would be commands 1 and 4 tested, respectively.

For project 2, we threaded the clienthandler class and had a server socket in which another socket would be established and a connection would be accepted in the main. There is also a new data input stream and output stream that would be created then threads would also be created then push these new streams and sockets into a new instance of clienthandler within the thread and then starts them in parallel. Now the server should run concurrently, the commands are then run again with the increasing number of clients and then the latency is then recorded. The result is our graph of latency (ms) of a concurrent server vs. different numbers of clients requesting the server to display the date and time and a final one comparing the netstats.

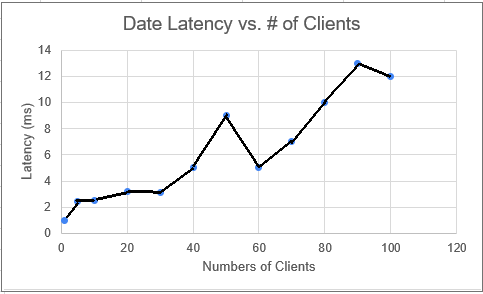
### Results

When comparing the performance between the iterative and concurrent servers it is evident that there is a strong correlation between latency and the number of clients on the server. As the number of clients increases so does the latency for all of the graphs, regardless of whether it is an iterative or concurrent server. Looking at the starting latency of a small process like calling for the date and time, you can see how parallel concurrence shines as the latency is much lower overall compared to the iterative server implementation. If you compare that to the graphs of netstats you can see that their overall latencies do not differ as much. This indicates that both server implementations don’t differ as much when processing more demanding requests. Although, later on in the graph as the clients increase it can be seen that the netstats latency of the iterative server grows linearly while the concurrent graph experiences an exponential growth pattern. In Figure 2, we did have an outlier between 40 and 60 clients which is possible because of the nature of the program, but this does not deviate from the overall message that the graph gives us.

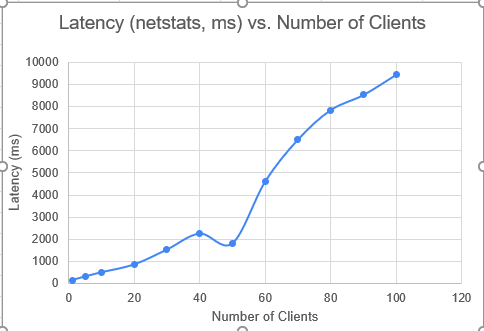
To conclude in terms of performance according to our graphs, one would preferably choose an iterative server over the concurrent one when dealing with smaller processes. The iterative server works well with smaller requests and is less work to implement. When the process is more demanding one would preferably choose a concurrent server rather than an iterative one, as the concurrent server still outshines even with the exponential latency growth in the middle because the overall latency is still so low as seen with the netstats graph of these two, Figures 3 and 4.

# 

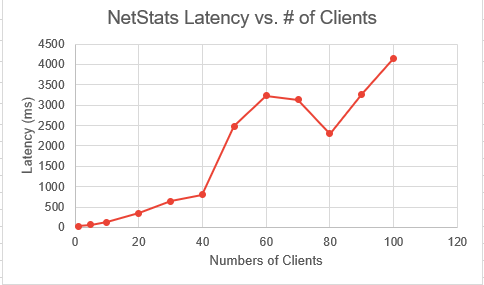
*Figure 1. Graph of latency (ms) of an iterative server vs. different numbers of clients requesting the server to display the date and time.*

**

*Figure 2. Graph of latency (ms) of a concurrent server vs. different numbers of clients requesting the server to display the date and time.*

**

*Figure 3. Graph of latency (ms) of an iterative server vs. different numbers of clients requesting the server to display host netstats.*

**

*Figure 4. Graph of latency (ms) of a concurrent server vs. different numbers of clients requesting the server to display host netstats.*

# **Conclusions**

While completing the projects and viewing the results, there seemed to be an evident difference in performance between the two, iterative v.s. concurrent servers. The concurrent server took longer to set up and program as we needed to establish parallel connections between the server and the client to handle multiple requests at once. The benefit of this was a lower overall latency compared to iterative servers for more demanding requests such as the host netstats as even with the spike, the concurrent server’s latency outperformed the iterative server demonstrated in Figures 3 and 4. Seen by the graph it is notable that the latency of the date and time commands in the iterative quadruples at 100 clients, whereas on a much more demanding request like netstats the latency goes up by ten times. So as a result if one didn’t want to go through the trouble of setting up a concurrent server if the request is simple they could go for an iterative server despite the increased lag, to save time programming. For this reason, an iterative server is preferred over a concurrent server for smaller programs, and a concurrent is preferred over an iterative one for larger programs. This was illustrated by our results of these two projects.

# **References**

[1] Rouse, M. (2019). *What is distributed applications (distributed apps)? - Definition from WhatIs.com*. [online] SearchITOperations. Available at: https://searchitoperations.techtarget.com/definition/distributed-applications-distributed-apps [Accessed 8 Nov. 2019].

[2]A.S. Tanenbaum and D.J. Wetherall. *Computer Network*, 5th edition. Boston, Massachusetts: Pearson Education. 2011., pp. 650-660, Accessed on December 01, 2019.