# Distributed Musi-Q Project Report

Zoe Cui, Anh Thu Pham, Tyler Williams

May 10th, 2019

## Project Overview

The Musi-Q Project aims to help Computer Science students bond and escape from their studious silence by allowing them to share their various music tastes in the classroom. This project utilizes concepts of parallelism with threads, threads synchronization, and network and distributed system, from the Operating System course. Musi-Q allows a user to establish themself as a server and others users to join the network as clients. Clients can add songs to the shared music queue maintained by the server and the server will play music on the server's computer when the queue is not empty and prompt the clients with appropriate feedback. To achieve these desired functionalities, we have a client.c file to connect to the server using server's name and port ID, to read from and write to the server. We have a server.c file to run musicHandler thread that plays the songs in the music queue, listen for connections, and assign each client to a clientHandler thread as connections are made. Moreover, we have a populate.h, playSong.h and a local music library with mp3s to enable server's computer to play the music.

To evaluate the success of our implementation, we designed an experiment to match the intended scope of our program. Our experiment evaluates all the functions of Musi-Q that a client can use: join server, print library, print queue, and add songs. The experiment tests whether Musi-Q can play music while performing these operations with varying number of clients connected and length of song queue. The experiment also times how long each operation takes with each permutation of number of connected clients and queue length. We found that Musi-Q takes more time join server, print queue, and add song as we add more clients and more songs to the server. On the other hand, Musi-Q remained relatively consistent in the time that it takes to print library. Even though there trend of increasing time for most of the operations under experimentation, all of the operations worked while music played. Additionally, the increased time was not noticeably longer by humans. Overall, we consider Musi-Q to be a success.

## Design and Implementation

Our project is divided into two major components, server.c and client.c, which both include the playSong.h and populate.h libraries.

Server.c consist of a `typedef struct node`, `typedef struct queue`, a global lock, a `void queue_put(queue_t* queue, char * element)` function, a `void queue_take(queue_t* queue)` function, a `void queue_print(queue_t* queue, FILE* to_client)` function, a `void * musicHandler ()` thread function, a `void * clientHandler(void* arg)` thread function, and an `int main()` function. server.c's main function handles opening a port to listen to connections from clients. The opened port is printed on the client side UI, which allows clients to view which port they need to connect to. The global music queue is initialized in main so songs can be added to it later and the later called musicHandler function will have a valid global queue to query. A thread is then started with the function musicHandler to complete this query. A while loop that never ends is run to continuously accept new clients. When a new connection is made, the clientHandler is run with the client's file stream as input.

Queue_put, queue_take, and queue_print are all functions that operate on the queue. Queue_put takes as input a song name and the global music queue. The global lock is locked to keep other operations from being run on the queue while it places a song at the end of the global music queue, after which the global queue is unlocked. Queue_take works similarly as it takes the global music queue as input. It then locks the global lock, removes the first element from the queue, then unlocks the global lock. As we never need to do anything with the element that we remove from the queue, the function can simply return void. Queue_print also locks the global lock, then, for every song in the queue, sends its properly formatted song name and artist to the client. This design makes it easier for the client code to receive the state of the global queue as it abstracts the working of sending queue status to the client.

musicHandler starts a while loop that runs forever, constantly querying the global music queue for songs. This design feature causes the musicHandler function to never return, constantly querying the global music queue for new additions. When new additions/songs are found, the musicHandler calls the playSong function with the name of the first song in the global music queue and removes this song from the global music queue using the queue_take function after the song is played.

clientHandler takes as input the client socket field descriptor. It then sets up a receiving and sending file stream to communicate with the client. A while loop is then ran until the client sends the message quit, which closes file streams and connections to the client socket and cause the function to return. As long as the client does not send a quit message, we either add songs to the queue if they are valid, send messages to the client saying they should enter a valid song, or print the queue using the previously described print_queue function.

Threads and thread functions are used for playing music and handling separate clients so that separate clients have individual independent connection with the server and the global queue and so music playing can be done on its own.

Client.c's consists only of an `int main(int argc, char** argv)` function. `argv` consists of the username, servername, and port number which are used to connect to the server started with server.c. Two file streams are then opened, one to receive information from the server and another to send information to the server. A prompt is then displayed to the client, informing them how to communicate with the music queue server. Two char arrays are created, read message and write message, to hold communication messages between the client and the server. These char array lengths are declared on the stack as we do not use them outside of main function. We limit client and server messages to 255 characters. A while loop is then started, continuously checking for client input. If the client ever types quit, quitting messages will be sent to the clients and server, file streams will be closed between the quitting client and server, the server socket will be closed, and the initially started client program will end. Otherwise, client messages will be lowercased to account for client input variability and ran through a series of if statements. If the client inputs an empty string, we will give them a prompt asking for a properly formatted command. If the client types “view library”, `printLibrary()` will be called and the contents of the song library will be printed to the client. If the client types “add <song>”,  the library will be queried to check if the song resides in the library. If the song resides in the music library it is added to the global queue through client server communication. If the song does not reside in the queue, we simply prompt the user that their input is invalid. This causes song request that do not reside in the music library to never be added to the global music queue. If the user types “view queue”, this command is sent to the server to print all songs in the global music queue. If the user types any other string, we prompt the user to enter valid input.

As discussed, playSong.h is used by the client and server side code to make playing music, viewing the library, and checking if strings are in the library easier. playSong.h is a library consisting of the following four functions: `void removeSpacesandLowerCase(char *str)`, `void playSong(char *song)`, `bool inLibrary(char *song)` and `void printLibrary()`. These functions are defined in a separate library to abstract functionality from the client and server side code, making their code more readable. `void removeSpacesandLowerCase(char *str)` is designed to remove spaces, and newlines as well as lowercase all characters in the input string. Its motivation is that users will send input strings that contain extra spaces or capital letters that our music player will not be able it interpret as songs in our library, even if the input string contains a proper song name. For example, our music player will not find a song in our library called ‘F  o rma T i o n    ’, but after this string is formatted with our function it will be of the form ‘formation’ and our music player can now successfully play this song. `void playSong(char *song)` is a large abstraction that makes playing music extremely easy for the server. `playSong` creates a local songs library and removes all spaces from the input string using the aforementioned `removeSpacesandLowerCase` function. A for loop is then ran for the amount of songs in the library, checking if the input string is in the library. If the song is found in the created songs library, `fork()` is called. This will cause the child process to trigger an `execvp()` call that plays music. The parent process will wait until the child process finishes, give an exit message stating the song finished playing, and end the outer for loop as its corresponding song was played. As this function is ran in a thread, it is fine to block here because the global queue can still be queried and manipulated on the client side. If the input was not found in the library, but it is a substring of some number *n* songs in the library, those *n* songs will be collated in a possible songs array. If the input string was found not to be in the library, but it was a substring of some number *n* songs in the library, the songs the input string is a substring of will be printed; if it was not found in the library and was not a substring of any songs in the library, a prompt asking the user for valid input will be printed. The motivation for this is that users may enter strings that are similar to song names and should get some useful feedback if they enter similar strings. Due to the scope and time limits of the project we were not able to implement a similarity function to give even more useful feedback. `bool inLibrary(char *song)` returns true if the input string is in the songs library and false otherwise. As a side effect, if the function returns false it also prints the songs that the input string is a substring of or enter a valid song. The motivation behind this feature is that it will print useful messages on client’s interface while returning false so invalid songs are not added to the queue, causing unnecessary traffic on the server side. `void printLibrary()` simply prints the song title and corresponding artist of all songs in the library.

populate.h is a library designed to make adding songs to the library easier. It consists of a `TOTALSONGS` constant, a `typedef struct Song`, and a `void populateLibrary(Song* songs)` function. The `TOTALSONGS` constant is defined here to provide one central variable to keep track of the total number of songs in the library without having to use non identifying variables throughout our code. This constant also allows us to change the number of songs in our local library by only changing one definition. The `typedef struct Song` consist of a `char title[50]`, `char filename[50]`, and a `char artist[50]`. All chars are limited to a size of 50 to eliminate the need to malloc space on the heap and later need to free; we set the size at 50 because all of our library specific filenames, song titles, and artist names, would reasonably fit inside this char array without worrying about overflow. The `char title[50]` and `char artist[50]` are used to print properly formatted song titles and artist names in other parts of the program and the `char filename[50]` is used as a proper format string to give to our music player to play the song. The `void populateLibrary(Song* songs)` function takes an array of `Song`'s and populates them with the corresponding song title, song artist, song filename. The motivation for not making this function `Song* populateLibrary()` is once again the use of malloc. While we could have used malloc to allocate space for the `Song*` array, we would have to be cognizant later of making sure we freed the `Song*` array.

## Evaluation

### Experimental Design

Since our project aims to provide a way for Grinnell College Computer Science students to add songs to a central queue and play music, we conducted all of our experimental testings in the computer labs on Noyce Third, which has Debian GNU/Linux 9 operating system. Our experiments required speaker-enabled machines. For the Linux OS, we enabled sound by calling `alsamixer` in the terminal and toggled the ‘`m`’ key to unmute the sound. We also used the up and down arrow to set the desired volume.

We designed our experiments to match the intended purpose of our project—to allow Computer Science students in Grinnell to play music while studying and doing work. We do not anticipate the use of our program during class time, so we only tested for the scope of usage that we expect on evenings. On most nights, there are at most ten desktops in use in a single room due to the nature of partnerwork in the department and students working on their personal machines. Hence, we tested for up to ten clients. For all of the tests, we used the professor’s desktop in the front of the room; the desktop that is connected to the projector is also connected to the overhead speakers, which makes the most sense to be the central server to share music. We ran our experiment involving ten clients in Noyce 3818 and the rest of the experiments in 3819. We used multiple desktops and multiple terminals per desktops to conduct the experiment, and chose a terminal and a desktop at random to perform the operation under examination.

If experiments and evaluation should be replicated, be sure to git clone from `https://github.com/tylerdamonwill/distributed_queue.git`. We prepared our experimentation by pulling from the github repository to get the most updated version of the code. Notice that at the time of this experiment, we use the most up to date versions of string.h, unistd.h, pthread.h, ctype.h, sys/types.h, sys/socket.h, netinet/in.h, netdb.h, and our local libraries. To evaluate the success of our program, we tested the time that each basic operations within the program took. We used the `gettimeofday` method from the `<sys/time.h>` and stored the start and stop time before and after an operation in a global variable struct. Then, we printed the difference between the start and stop time in microseconds. We recorded the time that each operation took under different experimentation conditions in an excel sheets.

We tested each operation that a client user can perform with a server, namely join server, print library, print queue, and add song. The join server operation tests whether clients can successfully connect to the server with a varying number of existing clients and queue length, which is important to gauge if our distributed networks was implemented successfully. A successful connection means a client has full capabilities of adding songs to the queue, viewing the queue, and viewing the library after they connect. The print library operation tests whether clients can view the contents of the library. It also tests whether our library was populated correctly. The print queue operation tests whether clients can view all the songs that they and other clients have added in correct order. The operation also tests if a song is successfully removed from the queue after it has been played. The add song operation tests whether clients can add songs that exist in the library to the queue and have the correct song play in the correct order. All of these operations are only successful if they can be done while there is music already playing on the server. We also tested whether these operations can be done simultaneously and with multiple clients trying to use Musi-Q at randomly chosen points in the experiment with various number of clients connected and queue lengths.

For each of these operations, we measured the time that it took the program to perform for one, two, five, and ten clients. We chose a minimum of one client because the project should not play music or perform any interesting tasks without any clients. We chose a maximum of ten clients to match our intended purpose for the project. We also timed each level of client connections for each basic operations at queue lengths of zero, one, and five. For the case with ten client connections, we timed each operation an additional iteration with a queue length of ten. We timed each permutation of operation, number of clients, and queue length three times. After each iteration of permutation, we restarted the server and all of the clients for a fresh experiment. See Figure 1 for an example of our data collection table.

| Server Details | | Length of Basic Operations (in microseconds) | | | |
|---|---|---|---|---|---|
| Number of Clients | Queue Length | Join Server | Print Library | Print Queue | Add Song |
| 1 | 0 | 761 | 117 | 171 | 167 |
| 1 | 0 | 231 | 111 | 80 | 83 |

| | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 201 | 99 | 134 | 58 |
| Average | | 397.6666667 | 109 | 128.3333333 | 102.6666667 |

Figure 1: Example evaluation data for one client connection and queue length of zero

## Discussion

After we collected all of our data, we computed the average length of time for each permutation of operation, number of clients, and queue length from the three times that we recorded. We compiled the data and our findings are as follows.

The join server operation seems to be affected by both the length of the queue and the number of clients already connected to the server. As the number of existing client connections increases, the time that it takes to join the server also increases. It seems that the number of existing client connections affect the time it take to join server more than the queue length. It takes a new client roughly 4,826 microseconds to join the server with ten existing clients and a queue length of ten, which is approximately 0.005 seconds. Join server works at our expected scale since 0.005 seconds is noticeable by most humans. Interestingly, we found that the first time we attempt to join the server from any given desktop takes longer than any consequent time. After we realized this quirk in our program, we only record the time following the first that a client connects to the server so as to not skew our data.

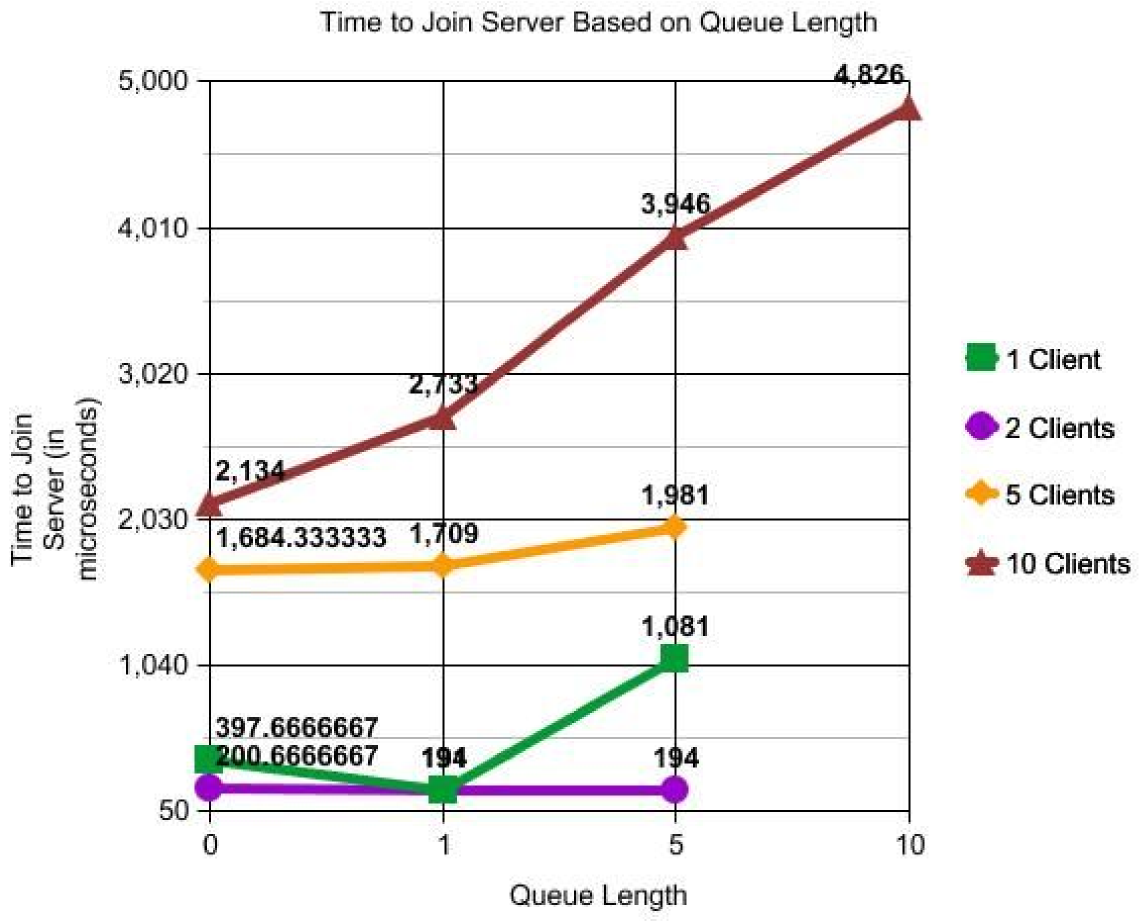

Figure 2: Average time to join the server in microseconds with varying number of existing client connections and queue lengths.

The add song operation behaves similarly to the join server operation. As the number of existing client connection increases and the queue length increases, the time that it takes for any client to add a song to the queue increases. That correlation is expected, as our song queue is implemented as a linked list. The server would have to traverse the entirety of the linked list to add a new song to the end of the queue. The program took approximately 2,487 microseconds to add a new song to a queue of ten songs and a server with ten connected clients.

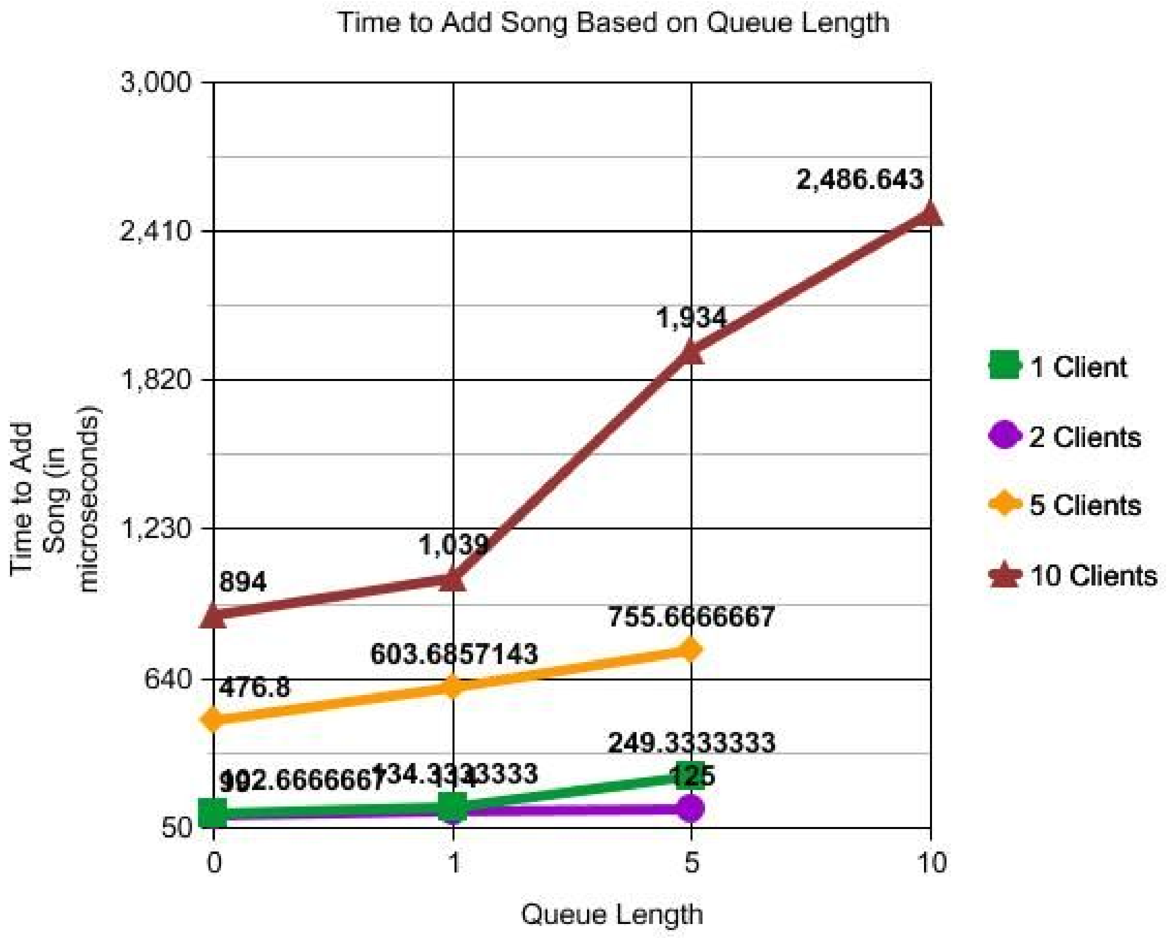

Figure 3: Average time to add songs to Musi-Q in microseconds with varying number of existing client connections and queue lengths.

The print library operation does not necessarily take more time as the existing client connections and queue length increases. Since print library is a function in playSong.h and is called in each client's local program, the operation does not rely on other client connections or the queue. As seen in Figure 4, the time for Musi-Q to print library is variable and mostly around 200 microseconds regardless of the number of existing client connections and queue lengths. The time to print library is possibly more dependent on system processes and scheduling.

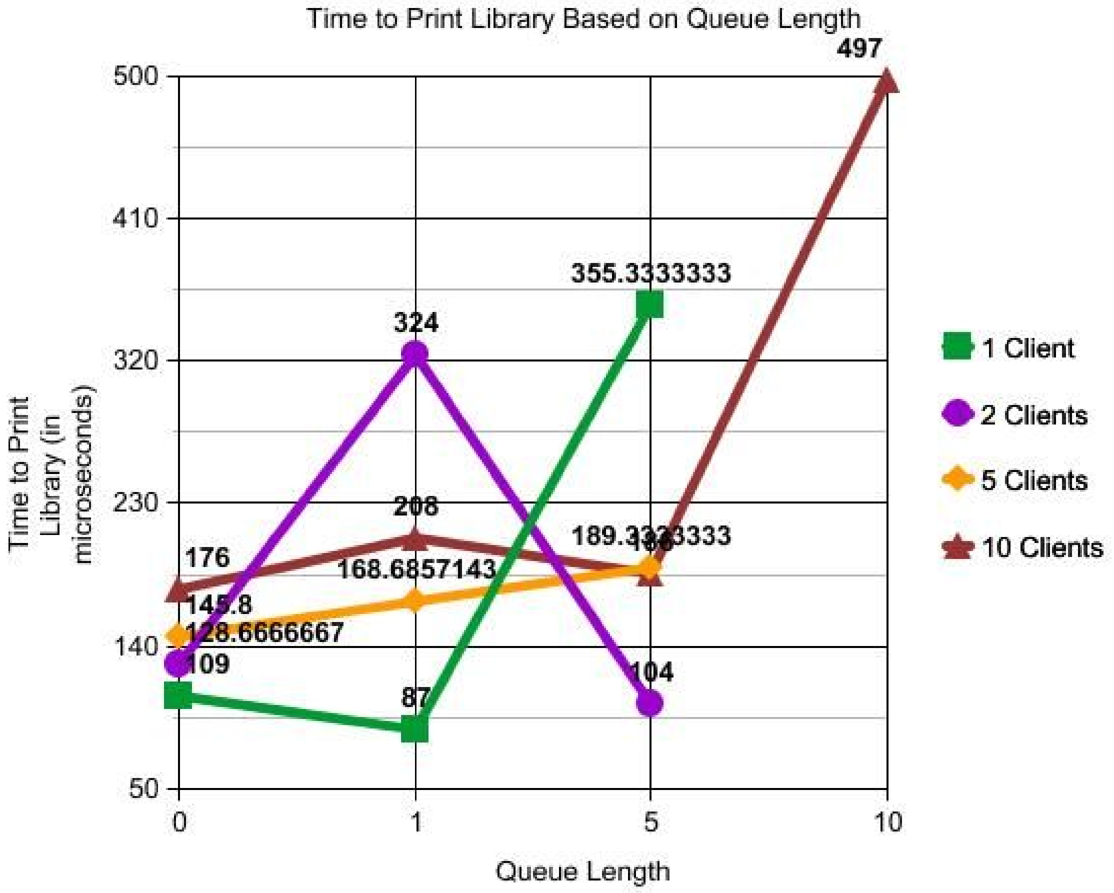

Figure 4: Average time to print library in microseconds with varying number of existing client connections and queue lengths.

The print queue operation behaves similarly to join server and add song operation. Generally, as the length of the queue increases and the number of existing client connection increases, the time Musi-Q takes to print queue also increases. At ten connected clients and a queue with ten songs, Musi-Q took approximately 2,768 microseconds to print the queue.

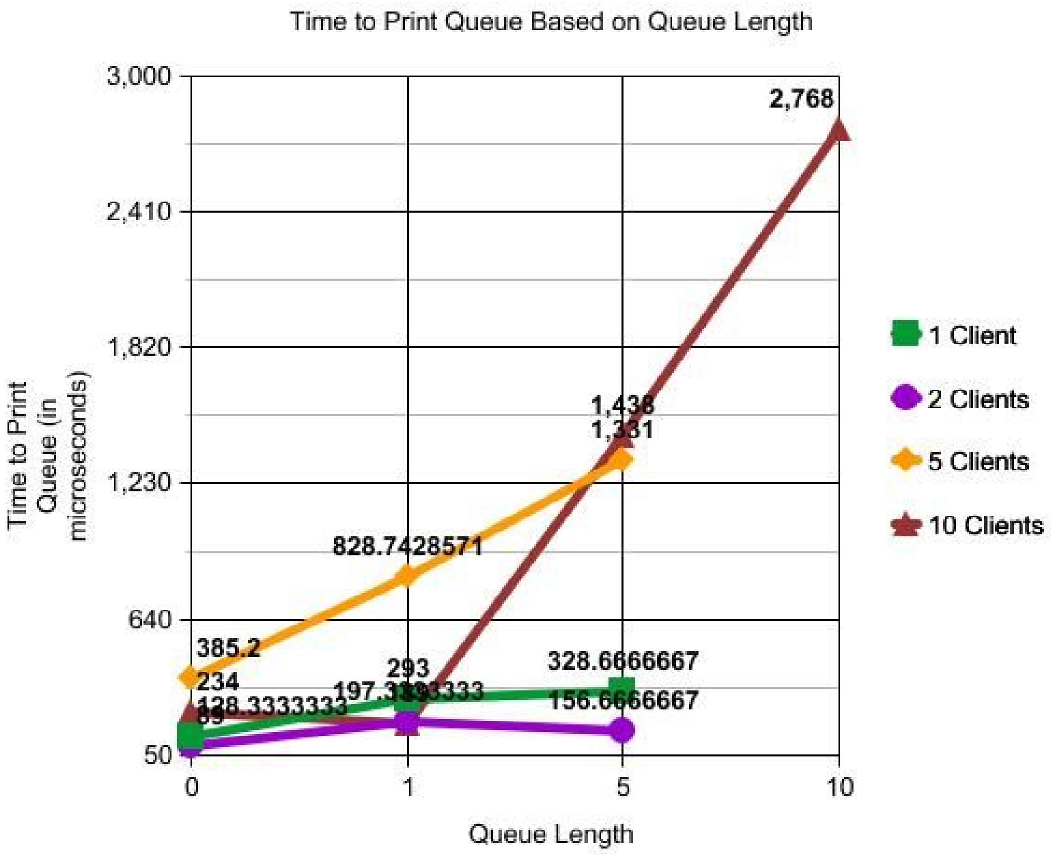

Figure 4: Average time to print library in microseconds with varying number of existing client connections and queue lengths.

Overall, the Musi-Q successfully added songs, allowed clients to join the server, printed queue, and printed library according to our metrics. Music played for the entirety of our experimentation, except when we restarted our servers and clients to reset our experiments. Although there is a general trend that the time Musi-Q takes to perform an operation increases as there are more clients connected and more songs on the queue, each operation still performed at a fast pace for most humans. After our thorough evaluation of the Musi-Q, we consider the Musi-Q to be definitely usable at the small scale that we intend it to exist. Since Musi-Q can only play music on one central server, the program is limited to run in one room for most social scenarios. However, we belief that the Musi-Q should work across rooms, should Computer Science Grinnellians desire that.

If given more time, we would love to perform larger scale experimentation to fully understand the scopes and limitations of Musi-Q. We would also love to implement options for the user to play, pause, skip, and stop song playing. Additionally, we would like to try connecting our project to Spotify API again. We believe these options would make the program much more user-friendly for our fellow classmates. We hope that our classmates enjoy the work that we've put into this program.