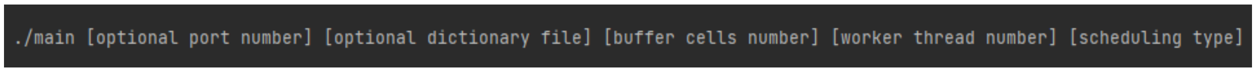
Alex Sawicki - Networked Spell Checker - ReadMe

User Manual

To use the Networked Spell Checker you must first start the server and then connect to the server on another client. To start the server you first type “make” to be sure that all of the code is up to date, then you type “./main”. Typing ./main will print the arguments that are needed to the screen and in what order you put them:



```
./main [optional port number] [optional dictionary file] [buffer cells number] [worker thread number] [scheduling type]
```

The arguments are provided in this fashion: [optional port number] [optional dictionary file] [buffer cells number] [worker thread number] [scheduling type]. The optional arguments are the port number and dictionary file, you can provide your own port number instead of using the default “2348” port that the program provides and you can provide your own dictionary instead of the default “dict” that the program uses. Other than the optional arguments, the buffer cells number, worker thread number, and scheduling type are required arguments. The buffer cells number allows the user to control how many cells the connections queue will hold. The worker thread number allows the user to control the amount of worker threads that the program will use in order to service the connections. The scheduling type allows the user to change whether they want the connections queue to be a sequential FIFO queue that the workers will consume from, or if they want the connections queue to become a priority queue, where the priority is assigned as a random number when it is added to the priority queue. Giving the value “1” to the scheduling type argument will make the connections queue a sequential FIFO queue, and will make the connections queue a priority queue if the value is “0”.

After you start the server program with your desired settings you can connect to the server using a client. The way I tested the client connecting to the server was using the command “nc localhost [port number]”. After you connect to the server using the client, you can type any word followed by enter, it will then return the same word with the words “MISSPELLED” or “OK” concatenated to the end of it to let the user know that the word sent to the server by the client is either misspelled or spelled correctly. After sending any amount of words the user desires, the user must type the word “exit” in order to sever the connection between the client and the server.

Design Aspects

The design of the program is broken into a few different portions: server thread, logger thread, worker threads, and user input.

The user input is the first thing in my program to be considered. There is a checkArgs() function that process the arguments and make sure the argument’s values are assigned correctly, and a getDict() function that opens the dictionary file and brings each word from the dictionary into a globally initialized unordered_set that will allow the server to check if the word taken from the client is within the dictionary. The reason why I used an unordered_set to hold the list of words in the dictionary was because I knew I didn’t want to keep on parsing the dictionary file each time the worker thread needed to check the dictionary. Therefore, I knew that a hashset from Java was a good data structure to use since it holds unique values and can tell you if the unique value is inside of the set, and I searched online and found the unordered_set data structure for c++ to use that fit the description that the hashset from Java. The checkArgs() function also uses a function called isNumber() that will return true if the string given to the function is a number and false otherwise, this is useful to tell the difference between a dictionary file and a port number argument.

Afterwards, the program spawns the logger thread, and the worker threads depending on how many the user input decided. The worker threads are in a never-ending while loop that will always check to see if the connection queue gets any connections, then it uses a mutex lock and will take the connection out of the queue and start to service it after it waits to see if there is something in the connection queue. The connection is a struct that holds the values of the priority, client file descriptor, and the time the client requested to be serviced by a worker thread. This was because I needed to give my worker thread multiple pieces of information to the worker threads and the queue would be the way to send the data so I let the queue be sorted by the priority value in a struct and send along other pieces of information in the struct. It services the connections by starting another while loop that will only end when the client sends the word “exit”, this is so the client can send as many words as they want without having to restart the connection each time. After the worker thread receives a word, it will check the word against the unordered_set that holds the list of words in the dictionary. Then it sends a string to the logger queue and sends back the words, concatenated with MISSPELLED or OK, to the client. At first I had sent the entire struct full of the connection information to the logger queue but quickly realized that there were problems with the timing data variables and that sending a string with the log information already fleshed out would be an easier option. Again, I used mutex locks whenever the worker thread accessed the globally called queue that the server thread produces to, this was to stop data corruption as only one thread could use the queue at a time to get their data. I also used condition variables to make sure that the worker threads would know when the connection queue was empty so that they would not try to access the connection queue with nothing in it. Furthermore, I signalled the server thread whenever the worker thread took something out of the connection queue so the server thread would know when the connection queue was no longer full and it would be safe to produce.

The logger thread then takes the string in the logger queue and uses it to append to an external file called logfile.txt, where each of the log values are held. I used the same mutex locks the worker thread did to add to the logger queue mutex locks when the worker threads tried to access the logger queue so the logger thread doesn’t try to retrieve from the queue while the worker threads are adding to it. Furthermore, I have the logger thread wait for the logger queue to no longer be empty and the worker thread signals the logger thread when it produces a log to the logger queue.

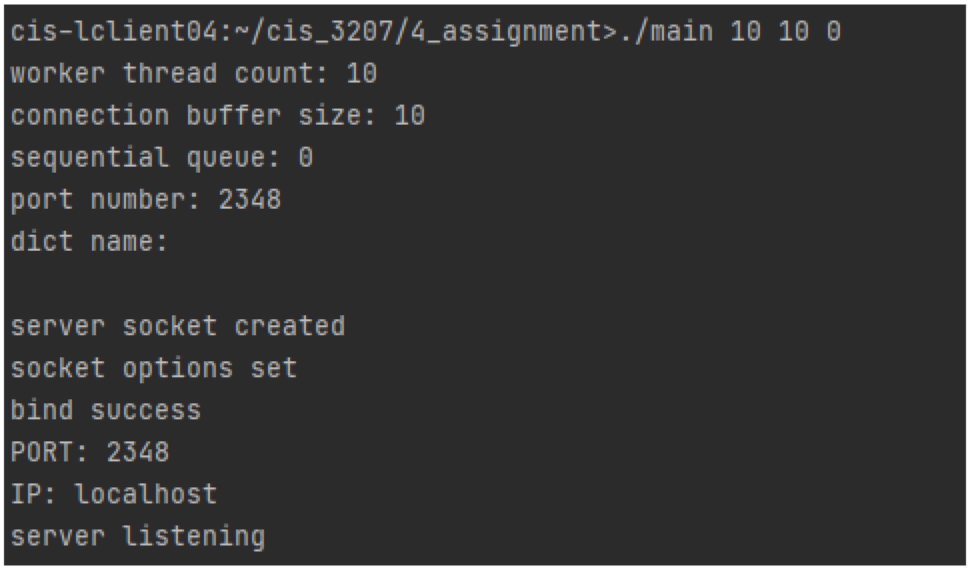
Finally, the server thread, which is also the original thread of the program, will open the network socket to start listening on the assigned port number and will then sit in a never-ending while loop to continuously check for any clients to connect to the network socket. Then it will use a mutex lock that the worker thread uses for accessing the global connection queue and will wait for the connection queue to not be full. Then it adds a connection to the connection queue and signals the worker thread that the connection queue is no longer empty. The connection it adds to the connection queue will have its priority changed depending on whether or not the user selects to have a sequential or priority queue, if it is sequential then each connection's priority will be equal to 0 and will be sorted as if it's FIFO. Otherwise, a random number generator will choose a number between 1 and 10 as it's priority and then add it to the connection queue to be sorted based on its priority. Then it will repeat these steps as it is in a never-ending while loop

Testing Process + Analysis

Starting off this project was hard for me because I hadn't ever done any network socket programming before and had to learn from scratch. However, after asking some friends for help and looking at some documentation online, I was able to get the network socket online without any threads and was able to send and receive messages through it. I then used my newfound understanding to implement a single server thread and a single worker thread to have them communicate between themselves. After learning how to use mutex locks and condition variables I was able to get not only the single server and worker threads but also an array of worker threads together and started implementing the connections queue while understanding how the various threads would communicate between each other.

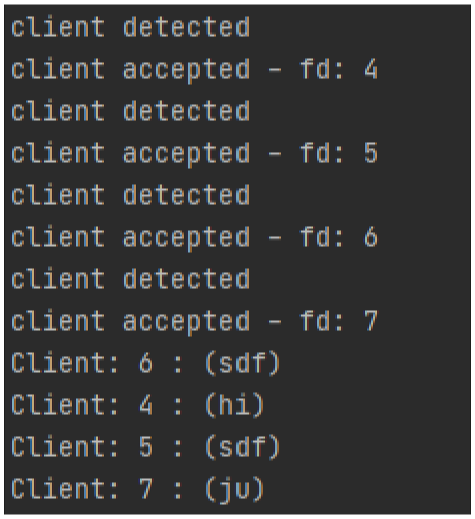
When testing my program I used many printf() statements in order to easily see the information that was being used and to see what clients are being connected to the server and what they are sending back and forth between each other.

One of the main things that I used for testing was printing out each of the argument settings to see if they worked. This includes the worker thread count, connection buffer size, sequential queue, port number, and dictionary name. Furthermore, I would printf() for each step in creating the network socket that would listen for clients. See below for how it looked like in the console during program startup.

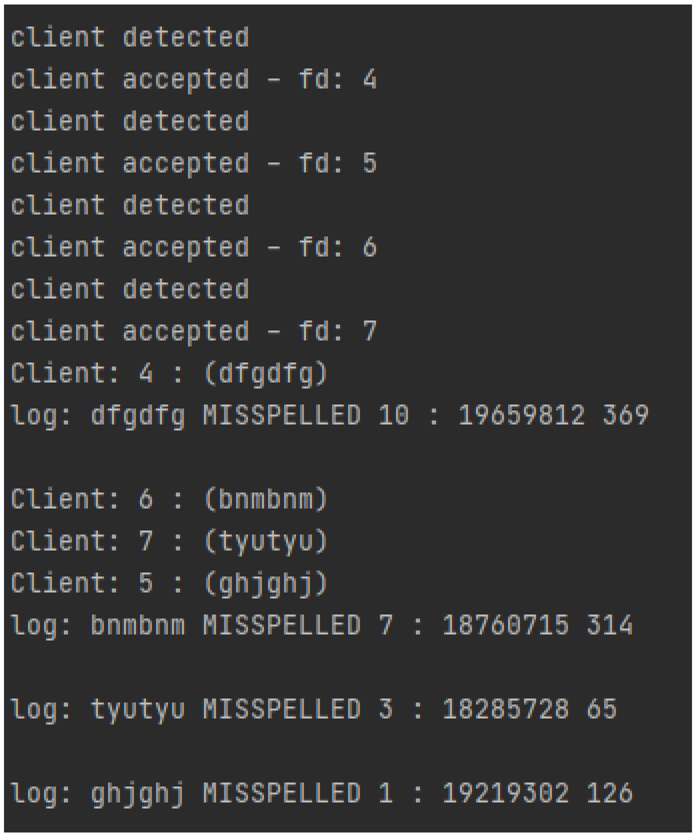


```
cis-lclient04:~/cis_3207/4_assignment>./main 10 10 0
worker thread count: 10
connection buffer size: 10
sequential queue: 0
port number: 2348
dict name:

server socket created
socket options set
bind success
PORT: 2348
IP: localhost
server listening
```

Another thing that I printf() out was each time a client connects to the server, what file descriptor the client has when it connects, and what words the client is sending to the server and which client sends the words. See below for how it looks like when multiple clients connect and each one sends one message.


```
client detected
client accepted - fd: 4
client detected
client accepted - fd: 5
client detected
client accepted - fd: 6
client detected
client accepted - fd: 7
Client: 6 : (sdf)
Client: 4 : (hi)
Client: 5 : (sdf)
Client: 7 : (ju)
```

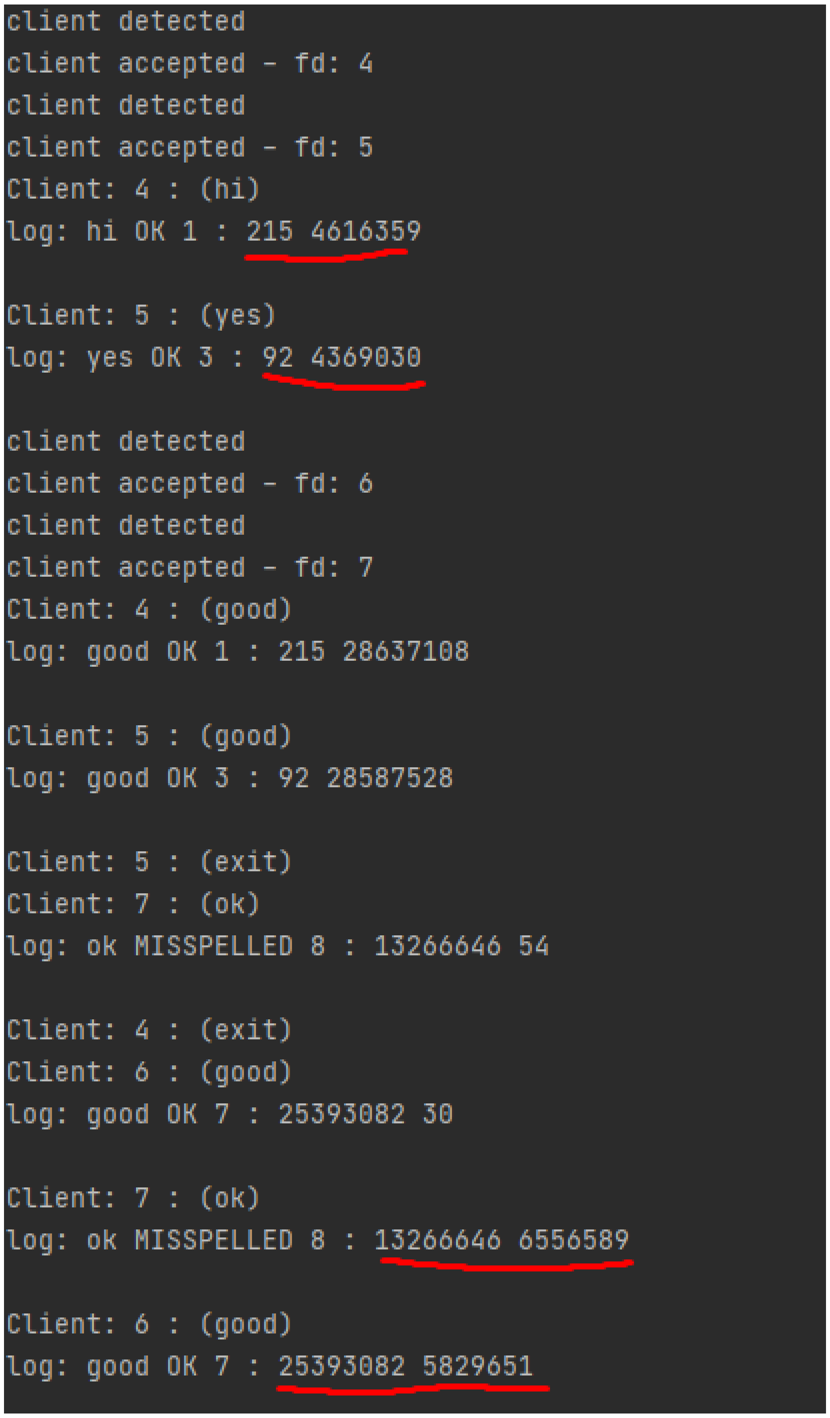
Then while testing the priority queue's usage, I needed to make sure that I could add multiple connections to the connection queue without the worker threads removing them immediately, therefore I used sleep(), inside of each worker thread I slept for 20 seconds on startup so that I could add multiple connections with different priorities, then I made sure to use a mutex lock on the worker thread in its entirety so that they couldn't switch between each other after already taking the connection out of the connection queue. Then the picture below is the output, where you can see that the priorities were taken out in order of the greatest to least, then were printed out inside of a mutex lock so that the order stays the same as when they were taken out of the queue.



```
client detected
client accepted - fd: 4
client detected
client accepted - fd: 5
client detected
client accepted - fd: 6
client detected
client accepted - fd: 7
Client: 4 : (dfgdfg)
log: dfgdfg MISSPELLED 10 : 19659812 369

Client: 6 : (bnmbnm)
Client: 7 : (tyutyu)
Client: 5 : (ghjghj)
log: bnmbnm MISSPELLED 7 : 18760715 314

log: tyutyu MISSPELLED 3 : 18285728 65

log: ghjghj MISSPELLED 1 : 19219302 126
```

Then after having a working product I started to test how different amounts of threads and connection queue sizes affected my program.

Using the connection size and thread size of 10 each, it was easy to see how each of the threads worked, with each one taking in a connection and servicing them. Afterwards I used a connection size of 2 and a thread size of 2, I let each of the clients use “nc localhost [port]” and after giving each of them a word, I exited two of the clients to let two other clients use the worker threads and the words that were given to the clients that weren’t being serviced became serviced when the worker threads were given the clients that hadn’t been serviced yet. Furthermore, I saw that their request service time and spell check time was a lot higher than the clients that had their spell check serviced right away. This was because they had been waiting to become serviced and had their words be spell checked. Check the picture below to see how the request service time is millions of microseconds longer than the first two clients and how the spell check time is around 2 million microseconds longer than the first two clients.



```
client detected
client accepted - fd: 4
client detected
client accepted - fd: 5
Client: 4 : (hi)
log: hi OK 1 : 215 4616359

Client: 5 : (yes)
log: yes OK 3 : 92 4369030

client detected
client accepted - fd: 6
client detected
client accepted - fd: 7
Client: 4 : (good)
log: good OK 1 : 215 28637108

Client: 5 : (good)
log: good OK 3 : 92 28587528

Client: 5 : (exit)
Client: 7 : (ok)
log: ok MISSPELLED 8 : 13266646 54

Client: 4 : (exit)
Client: 6 : (good)
log: good OK 7 : 25393082 30

Client: 7 : (ok)
log: ok MISSPELLED 8 : 13266646 6556589

Client: 6 : (good)
log: good OK 7 : 25393082 5829651
```