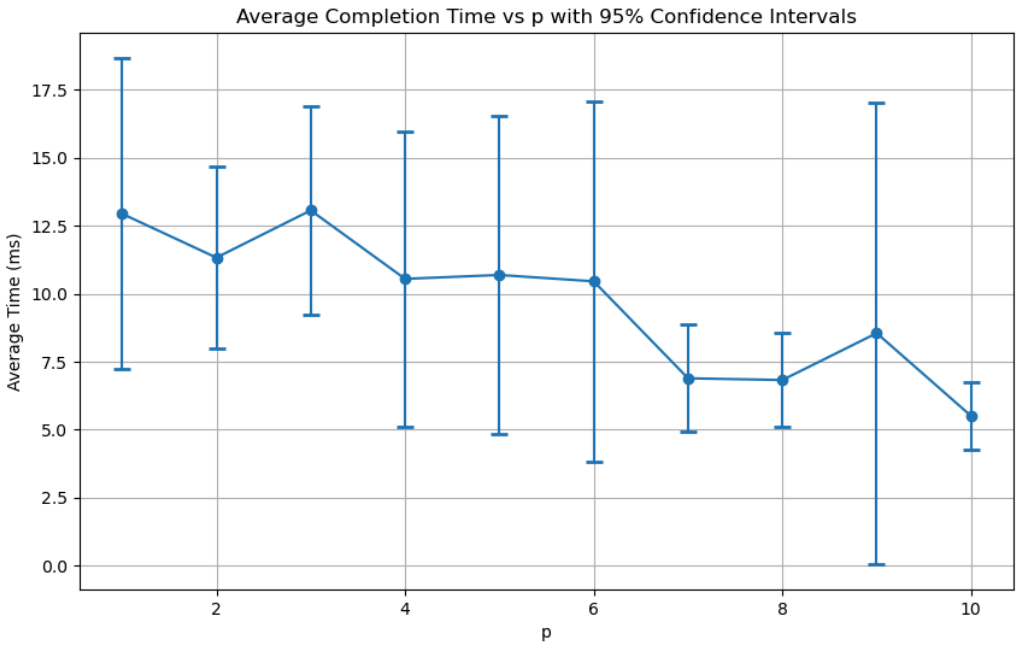
COL334 - Assignment 2 - Report

Jaskaran Singh Bhalla 2021TT11139
Basil Labib 2021TT11175

Part 1

In this part, we use the `socket` library provided in `<sys/socket.h>` to implement a simple server and client. The number of clients is 1.

PLOT



The parameters that we used to generate the above plot are as follows:

```
        "input_file": "words.txt",
        "k": 10,
        "num_clients": 1,
        "p": p,
        "server_ip": "127.0.0.1",
        "server_port": 3000,
```

We varied the value of p from 1 to 10 and calculated the average completion time for 10 runs for each value of p along with the confidence intervals at 95%.

Discussion

As we can see in the plot, the average completion time (in ms) decreases as the number of words in the packet increases. This is logical and can be explained by the fact that the

number of packets sent to the client for a given request decreases with increasing p. This, in turn, leads to less network usage and hence, a decrease in average completion time.

Network bandwidth: With the increase in network bandwidth, the ACT will decrease, ceteris paribus (while all other things being equal).

Latency: With the increase in latency, the ACT will increase, ceteris paribus.

k: With the increase in k, ACT will increase, ceteris paribus.

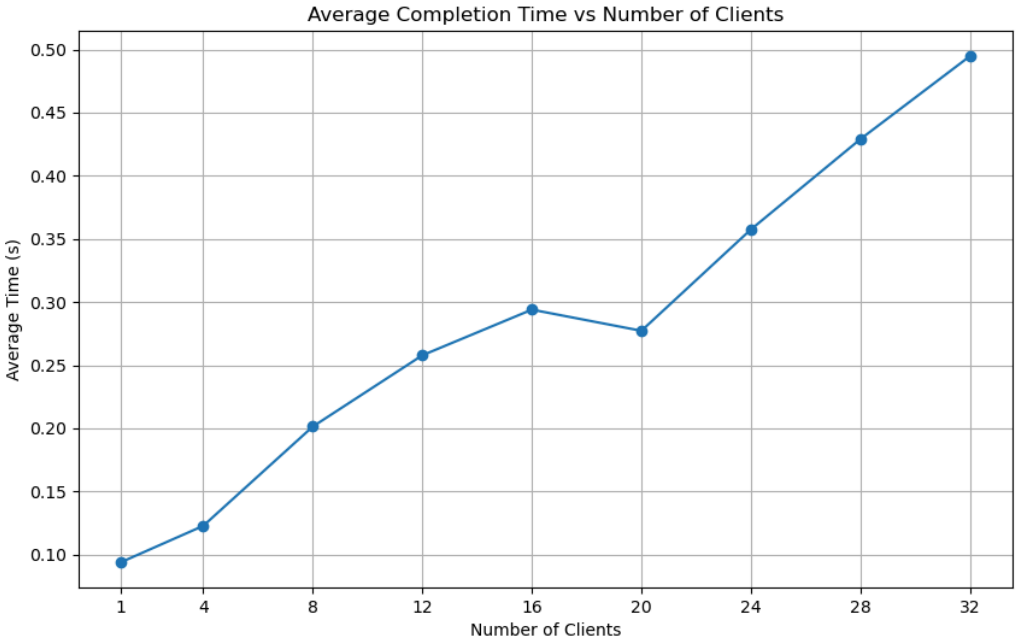
p: Already discussed above, but to reiterate, the ACT will decrease with an increase in p, ceteris paribus.

Part 2

In this part, we implemented a multi-threaded server with a `main` thread and multiple other threads spawned for each new client (a `client thread`) to serve each client in a multi-threaded fashion.

We used the `pthreads` library. See [here]

PLOT



As the number of clients requesting data from the server increases, so does the average completion time for those sets of clients. This contributes to a “high load” on the server.

Part 3

In this part, we implement various distributed client protocols to request a specific kind of server (called the grumpy server).

The grumpy server is defined as a server that will only serve one request at a time and send a special “HUH!” string to both clients if any new request comes in.

Implementation discussion

We use a memory-mapped struct object called `server_info` to decide if a collision occurred. We detect a collision if

1) the server status is set to BUSY. In this case, any new request is refused in the function `handle_client_thread(..)` which tries to lock a mutex.
2) the arrival time of the current serving request becomes less than the time at which the last collision was detected.

Moreover, we had to lock and unlock the mutex so that the fields `arrival_time` and `last_concurrent_request_time` were appropriately updated whenever a new request came on the main thread.

Machine-dependent compiler issues

My partner used a Mac, and I was running WSL2 on Win11. The issue was that the definition of a function pointer was different in both our cases. On Mac, the following is valid and compiles

```
    // void *_Nullable (*client_thread)(void *_Nullable);
```

However, for WSL2, the g++ compiler complains and asks us to revert to the standard C++11 syntax, which is as follows:

```
    void *(*client_thread)(void *);
```

Part 4

In this part, we implemented the two scheduling policies namely : FIFO and Round Robin (RR)

Implementation details

FIFO

The way we implemented FIFO was by maintaining a shared `request_queue` object which was shared between all the client serving threads on the server. Any new client request would lock this object in a mutex and push the current request which is a `Request` struct defined as follows:

```
struct Request
{
    int client_socket;
    int offset;
};
```

The main thread also spawns a new thread for serving requests (the scheduler thread). This thread locks the queue, pops the front of the queue and dispatches the Request to one of the client threads. It then unlocks the queue for other threads (new client threads / connections) to add their requests to.

In this way, we have implemented the FIFO scheduler.

Round Robin

In this implementation, if we are serving a client then any new requests by this client is queued by the scheduler thread to the end of the queue in a mutex lock.
Parallely, the request queue is being popped in a mutex by the scheduler and dispatched to one of the client threads for serving.

The server also has provision for responding to a control message `BUSY?`` with either a `IDLE` message or a `BUSY` message depending on the `server_info.status` value.

If there is a data request collision at the server, then the server cancels both requests by sending a `HUH!` message.

References

1. What is Multithreading? - Wikipedia
2. Concurrent Servers: Part 1 - Introduction - Eli Bendersky's website
3. Programming with Posix Threads.pdf - freebendy/ben-books