MP1 Design Document

Protocol

Despite the overhead from SYN/ACK and FIN/ACK from TCP; I chose TCP instead of UDP as I did not wish to implement my own system to deal with packet loss and arrival order.

On the topic of TCP, I set the `TCP_NODELAY` and `TCP_CORK` socket options with `setsockopt()`.

With `TCP_NODELAY` our data gets sent immediately, which is good since our payload is not much larger than the headers. However, this left us susceptible to silly window syndrome and congestion collapse; these massively decreased the performance of the program in my tests.

After testing with `TCP_CORK`, I found a significant improvement in throughput over Nagle's and `TCP_NODELAY`; however, the latency from the 200ms ACK delay was very noticable.

Client-Server Communication

Command Mode Each command from the client begins with a 32-bit `MessageType`:

```
enum MessageType { CREATE, DELETE, JOIN, LIST, RESPONSE };
```

The `CREATE`, `DELETE`, `JOIN`, `LIST` types will be sent from client to server. These commands will be followed by a variable length null-terminated string from the client capped at 224 bytes.

The `RESPONSE` type sent from server to client. The data that follows the `RESPONSE` value depends on the client's message type:

- `CREATE` and `DELETE` is followed by a single 32-bit value from the `Status` enum
- `JOIN` is followed by two 32-bit values: `port` and `members`
- `LIST` is followed a null-terminated string.

For `LIST`, it would be better to send an integer with the string length before the string so we can know exactly how many bytes to read, thus improving performance and reliablity; but I ran out of time to implement this.

Chat Mode After sending the `JOIN` message, the client will await for the port number from the `RESPONSE` message from the server and establish a new connection on said port. Now in chat mode, the client will wait for user input; upon receiving input the client will send a variable length null-terminated string over the socket. The chat thread associated with this client will then multicast the message to the clients subscribed to the chatroom.

Server

Parallelization The server on the main thread will have a socket binded to the port specified on the command line. After accepting a client connection, the server creates a new thread to handle command messages from the client.

The command thread upon receiving `JOIN` message will create a new chatroom thread accepting connections to the room. When a client connects to the chatroom, a chat thread is created to handle chat messages from the client.

After reading about C10k and profiling with callgrind, I realized that the server wasted a lot of time through context switching between the chat threads. I attempted to use a single thread to `accept()` connections and `recv()/send()` chat messages with `epoll_wait()`. Stress testing with 6 GiB/s of input caused the TCP buffer to saturate causing `EAGAIN`; , the application was unable to recover after this. Due to the assignment deadline I did not have time to explore this issue, and instead I reverted to the multithreaded approach.

Database In an attempt to improve performance, I used the stl `unordered_map` to get O(1) access. `unordered_map` actually incurs a performance loss in the provided test cases due to the relatively large constant involved with the hashing function.

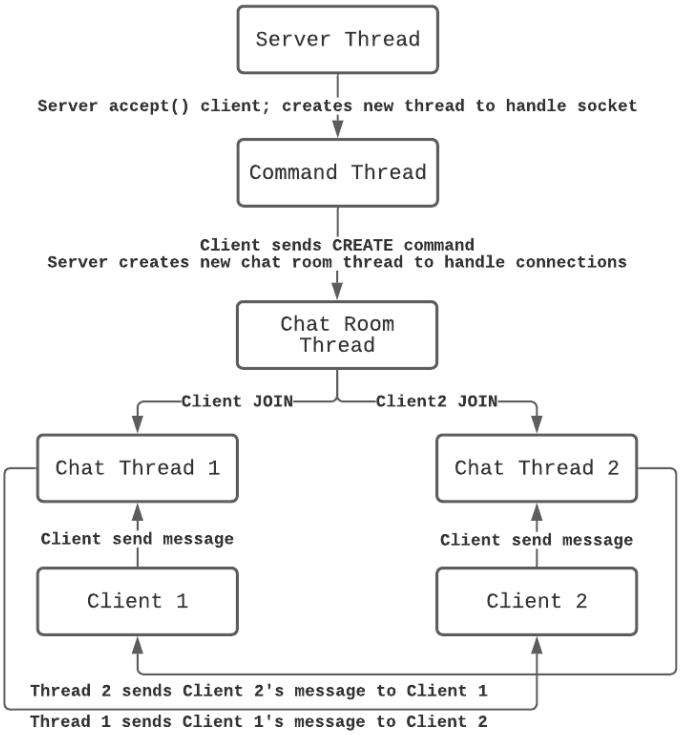

Figure 1: A diagram of the server behavior in response to commands from two connected clients.

Client

Chat Parallelization The client's chat mode uses two threads: one for reading from the socket, and one for reading from `stdin`.

I found this approach to be faster than using `epoll_wait()` on a singular thread; even if I enabled edge triggers with `EPOLLET`.

I used `select()` in the user input loop so that I can check to see if the connection is alive, if not then I can kill the thread immediately. Without this, `fgets()` will block the thread from dying until the user enters a new line character.

Known Issues

If we run `echo "create r1" | ./crc localhost 8080`, echo will close the pipe emitting an `EOF`, causing `get_command()` to run infinitely. This is because `get_message()` and `get_command()` in `interface.h` do not check if `fgets()` returns `NULL`.

I wanted to point this out as I don't know how the test script will work and I cannot modify `interface.h`.