CITS3002 Project Report 2021

Kane Alexander 22710428

# Player Scalability

The server was designed with scalability in mind, as all the given game variables are held within a single ‘gamestats’ object and only the socket addresses have a global scope. This ultimately allows multiple independent game objects to be populated in the same run time, all accessing the array of global connections. In other words, the server implementation could easily be expanded to run multiple concurrent games using the global list of all clients connected to the server. The server functionality itself was also constructed as a class, so expanding the number of servers is a possibility to reach an even larger magnitude of clients. These scalable characteristics are not current employed as per the project description, but the class orientated approach used tends towards logical scalability.

# Limitations

Scaling the server by adding more game objects would work for a substantial number of connections, however, would ultimately fail if the number of connections increases to disproportionate amounts. This is because the actual games themselves, although contained within independent objects, communicate through the same socket. The controlling while loop of the server uses the selector module to monitor all connections, and act accordingly to the data received. This is equivalent to a single threaded server, with the downfall that if multiple connections are all sending lots of data, a backlog could form.

This could result in a higher latency as the server has to work through the packets one by one. In the worst-case scenario, the kernel could start dropping connections if the server cannot handle the incoming data fast enough. Employing multithreading would allow concurrent communication between multiple clients and could increase the performance of a single server dramatically.

To avoid this problem with the current server design, the server dumps all data sent from players whom it is not their current turn (with the exception of clients leaving and joining). This keeps the server backlog essentially always empty; and avid testing of up to 15 clients could not produce any latency problems.

# Multiple Messages

A single threaded server makes dealing with concurrent messages relatively straight forward as only one can be acted on at a time. If two of the same messages are received, the server will execute the message first selected by the selector module and update the game accordingly. If the message is from the player whose turn it currently is, the ‘gamestat’ object will attempt to be updated with the move (only successfully if the move is valid). The repeat message will then be ignored and dumped as the server is updated and acknowledges that the client has already made its game move (or attempt to update again if the first move was invalid).

If the messages are from a client whose turn it is not currently; then the messages are immediately dumped as to not cause a backlog. Client joining and disconnecting messages are exceptions to this and will not be immeadetly dumped. Instead, the first message will be executed, and the second duplicate ignored as the connections array would have already been updated.

# Key Differences

One of the key differences I found when designing and developing this network program versus other programs I have developed is the unpredictable nature of the clients. Typically, projects require a standard set of inputs, often user or even creator defined, whilst the socket connection must be prepared for a complete array of different messages. These messages must be dealt with correctly for the server to satisfy its role in hosting the game. The program must be prepared for potentially malicious or advantageous client messages that try to manipulate the game state, and proper checks must be done rigorously.

Furthermore, another key difference was the style of programming, a move from procedural to event driven programming. The final design of the server was ultimately procedural as it revolves around an infinite while loop that executes messages as they arrive but draws heavy parallels to true event driven programming. This style is unlike other programs I have developed and requires a different mindset when thinking about how the program is truly intended to function.

The final major difference that I encountered when researching and designing this program was the concept of multithreading and multiprocessing. The broad concept of having multiple instances of the program running instantaneously opens a whole new world of both insanely increased efficiency and potentially hazardous errors.

# Implementations Outside of the Project

# Notable Points

I completed the coding aspect of this project before the release of the helper video on sockets and threads and the subsequent select document. As a result, my implementation was not one that utilized multithreading as my interpretation of the project outline was that a single threaded server would be suffice; the outline desiring a maximum of 4 player concurrent players per game and only 1 game running.

I did contact Matt to clarify that a single threaded server would be adequate, and he assured me with a proper implementation it would be. As a result, I decided to stick to my current single threaded server that used the selectors module. In the future I will likely improve the server to accommodate multithreading as the benefits gained in efficiency seem worth exploring.

Kane Alexander