CITS3002 Networks Assignment Report
By Mitchell Cox (22233087) and Varun Jain (21963986)

How would you scale up your solution to handle many more clients?

Our solution could be scaled up by using the fork() function before our init_server() function. This would fork the server process and take all the players currently in the lobby to that forked server process. The lobby could then restart, accepting new players and repeating once lobby time has complete. This would allow for multithreading, optimizing performance if many players attempt to join as each thread would have a player limit.

We could also scale up our solution to run more optimally in sending and receiving messages, by having separate message handler threads. By introducing these, the server could continue running without select() and send() blocking the game logic code.

Alternative ways to scale up the solution to handle many more client would be to use functions such as poll, epoll and kqueue.

How could you deal with identical messages arriving simultaneously on the same socket?

To handle multiple players we used the fd_set struct from the “time.h” header to assign each player with their own individual socket. This helped simplify this issue as our server program would receive messages from the players on their own socket and could verify which socket and hence player the message came from. This helps alleviate issue of multiple players sending messages to the same socket as each socket is checked separately. This means that even if multiple players send a message simultaneously, their unique socket descriptor is checked separately.

If the same player sends identical messages simultaneously, as each player can only make one move per round, we only accept the first message from each player as their move in a round. The message is still received but we do not take any action with it. Once the round has complete, the server begins to accept messages from all clients once again.

If the identical messages arrive so close together, they may end up being received by the server as one merged message. This message will then be unreadable by our server program as it is not a valid move packet and the player will lose a life due to an invalid move. To deal with this, we could have implemented a check and instructed the player to remake his move. Provided the messages are received one after the other but merged and not the contents not shuffled around, another solution is to implement a different check which would be able to detect a merged message occurring due to this issue and discard the second

message. The server would then be able to parse the original message. Detection could have been made easier by including an end message character and instructing the server to discard anything sent after the end message character.

With reference to your project, what are some of the key differences between designing network programs, and other programs you have developed?

One key difference in designing a network program was ensuring all sockets are closed gracefully, whether it was closed from server-side or client-side. This had to be done as if the game or the server attempted to send a message to a closed socket, the server would encounter a segmentation fault.

Another key difference was considering the speed of the server program, sending out messages too quickly to the clients could sometimes cause multiple messages to be received as one on the client-side. We had to ensure that each client had time to process each message before another was sent.

The last key difference was to be aware of where the process blocking functions (e.g select()) were positioned in the code. This made an impact on our programming of the server as we had to be aware of where these functions were.

What are the limitations of your current implementation (e.g. scale, performance, complexity)?

The most significant limitation of our current implementation is that when there are 15+ players connected to the server, messages can take a short while to be sent out. The cause of this limitation could not be determined, it may have been caused by our code, the implementation of sockets in C or Python or the networking capabilities of our computers.

Another, albeit less probable, limitation with our current implementation is due to the upper limit of sockets set for the select() function. Currently in the unix header file “select.h” this upper limit is set to 1024 but this limit can be changed at a cost of performance and reliability. Another solution to this limitation would be to use the poll() function instead on our fd_set which has a higher limit of sockets.