CITS3002 Networking Project Report 2021
Michael Sargeant (22737938)
Developed on Ubuntu 20.04

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

I would implement a thread pool and task queue to scale up my solution to handle more clients. My solution currently avoids race conditions by using thread locking. It works by using the main process to run the game, a thread to listen for new connections, a thread for communication between server and each connected client, and lockable threads that are activated when a client is recognised as connecting or disconnecting. These threads are locked to prevent corruption of the data in the main game processing which relies on accurate lists of observers and players. Scaling up to include more many players and observers would overwhelm this system causing connection timeouts and corrupted data, and the server ultimately crashing. A thread pool with a queue could manage tasks more efficiently one after the other. And of course faster hardware would help if scaling up.

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

I don't believe they could arrive exactly simultaneously, as there would have to be some difference between them. However if simultaneous arrival was possible, some arbitrary order could be decided then both could be put in a buffer or queue and processed in turn. A log of past messages could easily detect if one had been sent twice, and then the second could be ignored. Alternatively, the first message could be processed and then the second ignored because it no longer accurately represents a response to the current game state. For example, if there were two disconnection notification messages for the same player, the second could easily be recognised as irrelevant because there is no one to disconnect.

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

Two key areas which differed from other programs I've developed are in the use of sockets and threads. The use of multiple sockets and threads, while not difficult to start using, results in having to deal with asynchronous events. This requires writing code to deal with unpredictable message types arriving in an unpredictable order and timing. Consequently more controls were needed to ensure the program progressed as intended, and also that race conditions were avoided.

In addition, other programs are not as concerned with efficient use of system resources as they used to be, except when dealing with huge sets of data. However, in network programming thrashing seems to be a problem that increases the processing burden, but could can only really be minimised as some features require variables shared by separate threads.

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

The most apparent limitation of my current implementation to me is the performance of the automatically determined move triggered by player inaction in games at a scale of more than 2 players. Unfortunately, I have found that after this process is triggered, it sometimes allows normal play to continue and sometimes doesn't even though a lock is not used.

I attempted to improve this by putting the cancellation of the timer as close to the detection of a move made as possible, but this didn't provide an absolute solution. The cause of this I think could be in that when the timer runs out for each move, a new timing thread is opened and closed on completion. The overhead of this could be affecting the ability of the main game thread to detect manual moves in games with 3 or 4 players. Although I didn't have the time to attempt it, I would like to see if consistently maintaining one timing thread would improve performance of auto-moves in large games.

Finally, as for complexity, I think I have made the code as clear as possible, although it could possibly been more readable by having the client data as a class/object instead of a dictionary within the game state class. However, I don't think this is a significant impediment.

Is there any other implementations outside the scope of this project you would like to mention?

I attempted to deal with some minor bugs that were not explicitly in the scope of the project. For instance simultaneous eliminations were causing crashes, and I was able to improve the elimination code to fix this.

Any other notable things to discuss.

There are constants defined at the top of server.py to control auto-restarting games and auto-move taking for idle players. As mentioned above, enabling automatic moves in games with 2+ players sometimes interferes with the the functionality of lower tiers. Also, I am unsure why but I was unable to gain any feedback from the test code, even though manual games ran properly.