Edward Yamamoto (22709905)

All functionality specified in the tasksheet was successfully implemented.

Key Differences

Networking problems often require multiple services running in parallel to achieve some bigger functionality.

If this project only required us to run a game with static clients (no netowking), the project would have been trivial. But by allowing players to join and leave as they please, the project became a parallel problem rather than linear.

This meant that I had to design code that did multiple things at once by using threads. Checking for connecting clients, running a game, and checking for disconnected clients all had to occur at the same time. Whereas other programming units at UWA only considered sequential execution of code.

Identical Messages

Assuming the content of these identical messages arriving from a single socket could be decoded:

- Introduce an arbitration process
  - The client needs to add a timestamp value to the content of the message
  - The message with the latest timestamp is prioritised and the rest is discarded

If the content could not be decoded, then all the server would have to do is ignore the message and wait for the socket to transmit again. It may be possible to extend the client class by allowing it to recieve data from the server. Then the server could inform the client that the message was unreadable.

Scaling

Client Updates

The main target of horizontaly scaling this program would be finding ways to parallelize the computation. Thus allowing multiple devices or threads to balance the load.

Limitation

The current program, when run on my local macheine was load tested with 500 clients connecting at the same time. After profiling the activity, the main bottleneck seemed to be the propagation of board updates to the users.

While the game was sending these board updates to all of the users, the game was effectively stalled. This is because I had only tasked a single thread to inform users of the updates that had occured.

Solution

To scale this server horizontaly, it would be necessary to make the propagation of updates to all connected clients occur in parallel. Essentially, the messages being sent to each client are identical, so it may be possible to package all the updates and schedule another devices to deliver the message to a single, or pool of clients. Some considerations though,

- The people playing the game must be prioritised when getting updates as their data needs to be as concurrent as possible
- Spectators are low-priority for receiving messages as they have no effect on playing the game

Client Joining

Joining clients are handled one at a time. Their connections are recieved, and then they are sent all the information reqired to participate.

Limitation

When a burst of clients join, processing the clients in a linear fashion means that it takes a long time for a client to be accepted

Solution

The client accepting handler would need to be able to listen and accept multiple clients at once. This may be difficult as it then becomes necessary to syncronise the id numbers that each handler has assigned as to not use duplicates.

This is a more difficult form of vertical scaling, but would reduce the average time for a client to join.

Memory Locks + Barriers

Limitation

My multithreading code does not implement memory locks or barriers. Normally, I would protect accesses to global variables inside a thread to prevent race conditions. But the program runs fine without them

Solution

If this were to be more reliable, memory locks need to be implemented on global varibales. This would ensure a error-free experience when adding large amounts of clients - by preventing corruption to server data.