Design Document (HW4)

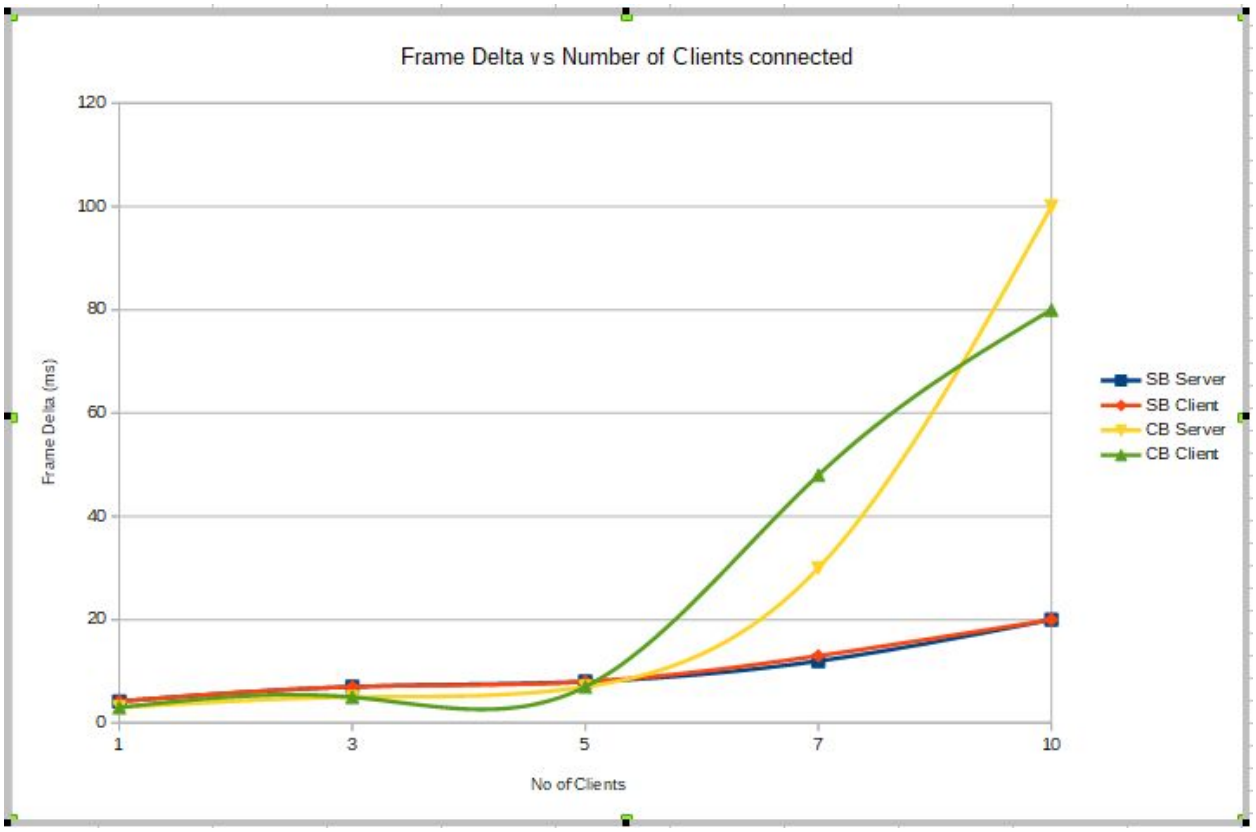
Karan Rakesh

Part 1 : I implemented an event management system by using an Event representation as mentioned in class. I implemented move,death, spawn,recording start/end, and character collision. I used timestamp as the only metric since it was sufficient to chronologically order the queues for my particular use cases. Also, to send messages across the network I implemented a eventToStr() as well as a reverse strToEvent() which converted the events into a form that was transmitted across the zmq sockets, as I had trouble integrating any json library. I decided to go with a server based implementation which required me to significantly refactor my code because upto this point all of the processing was happening on the client side and only the updates in position were shared across the network. But since I would require to anyways be able to share events across the network for the part 3, it made sense to do all the code changes up front. I made a queue on the client side to store events and sent all the information on the queue to the server in batches. The server received this data and formed events which were then sent to its own event handling queue which executed the events as they came. I tried with implementing an event handler in my game object model purely but it required too many parameters to be sent over as parameters. So I implemented the handling to carried out in the same scope as the server implementation so that they could both make changes to the character and client position maps that were implemented. This also required the use of multiple mutex locks to prevent memory read/ access violations. My present design felt extensible since it can take any event from an arbitrary number of clients and handle it without any adverse performance impact (as measured in section 3).

Part 2 : This part of the assignment proved more challenging than expected. I went ahead with Dr Roberts suggestion of using a queue that stores events and then replaying them in order to achieve a replay. So I built a recorder object that stores the state of the game at the moment a client requested a replay to start recording. This went on till the client asked to stop recording. There were many bugs that had to be solved. Initially the client was not teleporting correctly since if it moved to another screen, it did not shift back to its own screen. Then there was no way to force the timeline to hit the same time line points that the recording had. So I had to implement a system that dequeued and ran all the events that occurred up to this point in time, so that it was as close to the required implementation as possible. Then there was the issue of the client running through the entire queue before the recording stop event was processed in the queue and hence it seemed like nothing had occurred when in reality all the events had processed so fast that it was not noticeable. So for that I added a flag to ensure only after it was handled could the playback start. Another major hurdle was how to handle the teleportation and synchronisation of the platforms with the replay since they are run on separate threads and need to sync with the objects to ensure an object doesn’t suddenly hover in the air while the platform is at some other point because the sync is off. Also ran into some errors while dequeuing the recorded queue, which had to be fixed apart from various other bugs. I also avoided any event chains occuring in my implementation to not have to deal with handling them

at all. Overall it didn’t require much effort to leverage the event system to get replays up and running since I made the required changes to queue as well as handle events so the replay only basically required me to jump back in time and repeat the sequence of events. I tried my best to iron out most of the kinks but there are two bugs that I must mention are not handled in my code:
Bug : I tried to implement the speed change, but given the server side implementation in mine, I tried incorporating separate keys to speed up or slow down the timeline based on the key press but that didn't workout since we were told to implement speed changes by altering the step size, but this will not work in this case since the changing of step size will alter the measured game time so either it processed all events in one go or the opposite. So I tried implementing a separate timeline implementation where the timeline stores the last time measured and the only stores how much time has elapsed since then. But this again ran into issues while calculating the time that elapsed rather than the overall timestamp so I had to drop the implementation and hence server speed up and slow down isn’t functional in my output. Though I have attached files with my try at fixing the timeline and the server in tester_myTime.h and tester_server.cpp.
Bug : I couldn’t figure out why this occurred, but on the very first time you run a replay on a particular client it sometimes takes too long or skips over the entire replay very quickly. Then on recording the replay following that, it seems to work perfectly in sync. I tried debugging it for a while but then had to move on to other more important components in the interest of time.

Part 3: So, since I implemented the server-centric implementation. I used this as a template with my older code that performed computation on the client side to implement a distributed system. It involved a lot of code moving from the server code to the client side but it was not too hard since all the components were already implemented in the server-centric model and hence they only had to be placed and called at the right intervals. What I decided to implement here, to make the least amount of implementation changes is that I sent the events to the server and then instead of repacking them into events I just added them to the queues reserved for each connected client and then when the client requested for a message, all the event strings queued up in the client queue were sent to it, following which the repacking them into events, followed by queueing and handling were carried out locally itself rather than occurring on the server and the updates being sent to each client. The critical difference between this distributed implementation and the previous homework' implementations which also did computation on the client side is that in the older ones, the client only computes its own “event”/positions and then updates the server which maintains a global state and sends it to the other clients. In the new implementation. Every client gets all the events raised by its peers as well as itself to be queued and handled on the client side itself. This causes massive redundancies even if we do basic collision detection n death detection etc in advance, since every event is handled n times where n is the number of clients. Hence this is expected to have a massive performance impact. And even with my gaming laptop I was able to notice a massive spike in the frame delta as the number of clients increased. I have attached the graphs below.



You can see as the no of clients went passed 5, the number of events to be handled grew at n-squared complexity an it soon grew too big to handle the whole application was hanging and was unuseable at that point. Meanwhile the server implementation which grew at the order n complexity is slowly and steadily increasing. This shows that a multithreaded server implementation seems to be the best choice in the given scale and circumstance.

I have attached a README detailing the instructions for execution. This assignment was pretty challenging and the refactoring involved took a lot of time and effort. Probably could have been easier if we were forced to implement it on the server itself from the beginning. Similar to the prev assignment where a major chunk of time was lost in making the code multithreaded as suddenly that became a requirement even though it wasn't necessary from the implementation mentioned in HW2. But there was a lot to learn in this assignment, and it helped sharpen my debugging skills too!