CPSC 416 Project 2 Proposal

Jerome Rasky, Madeleine Chercover, Raunak Kumar, Vaastav Anand

Introduction

Video games have always been an important aspect in the lives of people as they provide an escape from reality and life problems. in the last year or so, a new genre of video games, Battle Royale, has taken over the world by storm. Video games of this genre, like Fortnite and PUBG, are essentially last-man standing games where the last player wins the game. These games typically involve altercations between players and heavy interaction with the game map. These features place a lot of restirctions on maintaining the game state like validity and consistency of the world that has been modified and making sure the players that have already been eliminated are unable to modify, but maybe view, the game state. Additionally, like any network game these games also require that each palyer has very low latency.

We're interested in building a distributed 2D game of the Battle Royale genre for our term project. We feel that this area is interesting because it introduces real-time constraints into our distributed system. As mentioned before, such a game requires low latency whereas the blockchain can afford to take ten minutes to confirm transactions, players might be upset if it takes ten minutes to move at all in a game. In addition, we also intend to have a lot of distributed state in our game.

Background

To build an online real-time multiplayer game, there were initially two popular architectures, shown in Figure 1, which were used in industry. The first one is the peer-to-peer architecture in which all clients start in the same initial state and each client broadcasts every move to the all other nodes. With clients communicating directly, the gamestate can not advance until each client's move is received by every other client. The overall latency of the system is then dependent on the slowest client in the system. This system is also not tolerant to faulty clients, as each client will wait and decide themselves if a client has failed. The second architecture is the client-server architecture. In this architecture the gamestate is stored on a server and clients send updates to the server. This architecture reduces latency, for each client the latency is determined by the connection between that client and the server. However, this architecture is still too slow for real-time online multiplayer games, which lead to the introduction of client-side prediction. The client-side prediction allows the client to simulate its own version of the game whilst sending the results of the moves to the server. This essentially means that each client maintains its own gamestate but this can be overriden by the server. This architecture allows for the server to detect malicious clients as well because each action by the client is being validated on the server.

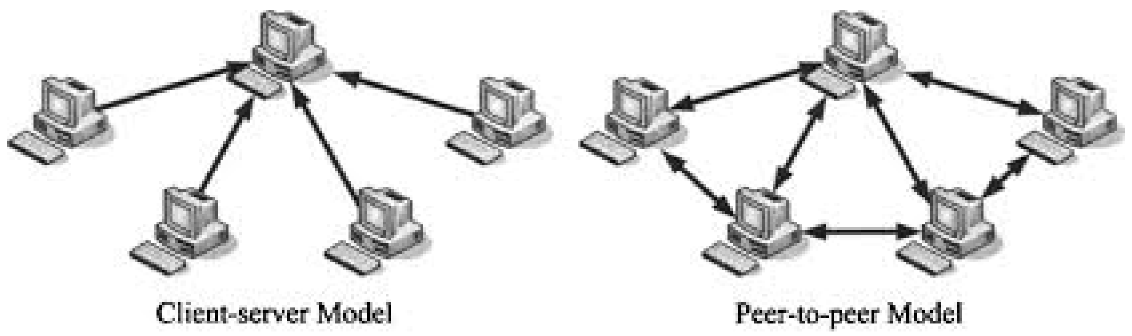

Figure 1: Network Architecture Models

Game Mechanics

The minimum possible game that we can build involves players moving around, aiming, and firing within a static, fixed-size map. We'll start with that, and build our way up by adding stage hazards, more player interactions, health, lives, more weapons, and any number of other features.

Network Architecture

When players make moves, the whole network has to find out about them. The simplest possible configuration is to have every player connected to every other player, and to have all players send all their moves to every other player every time they make a move. By using clock synchronization, and a stable tie breaker such as random number generator, we can ensure a global serial order of events. Whenever a player makes a move, they send their new position to the other players, and whenever they fire they send their position and angle that they are firing at.

Clients might be malicious, which means that every client has to validate the state at every step. With only movement and firing, those checks are reasonably simple. Each client verifies that the new position of a player is within the allowed movement speed, and that the position from which a player fires matches the players current position. Every client will likely have to keep events around in a buffer for a short while, to ensure that they only process events and update their state once there is a high likelihood that a global order has been reached.

Once we have those features built, we'll work on more interesting network topologies, like those that we built for project one. We'll also work on adding other kinds of interactions, and on optimizations to improve latency. Those optimizations might involve relaxing the global order requirement, and instead only ordering events that interact with eachother.

Packet Loss

We'll use UDP to communicate between clients. This means low latency, but it also means that packets might get lost. This makes validation more complicated: what if we lose two moves a player makes, and then only receive the third one? We'll have to reason about the likelihood that a certain update is valid, given the missing data. We can't be entirely sure about this update, but with the whole network put together and over time, even with lost packets, we can establish an ordering of actions.

Gameplay

We plan to build a game not unlike Tanks (Battle City), where players can move around and shoot at each other. The last player standing wins. Initially, our game will be very simple, without any health or stage hazards or anything. We'll grow our game mechanics as time goes on, adding things like stage hazards, status effects, and alternative goals. With this approach, we can grow the complexity to an almost arbitrary degree by adding features, but we can also scale back our ambitions if we run out of time.

Peer Discovery

We'll start with a central server to keep track of peers, much like in project one. Our server will eventually back the state of the game map, so that even if no players are connected, the world state is preserved.

Clock Synchronization

As the game is a real-time distributed system, we need to synchronize the clocks of all the clients as we do need to synchornize the events of multiple clients in order to resolve the altercations between players. We are planning to use the Berkeley Algorithm to synchornize our clocks with the server being automatically chosen as the master for the puposes of this algorithm. An example of this algorithm is shown in Figure 2.

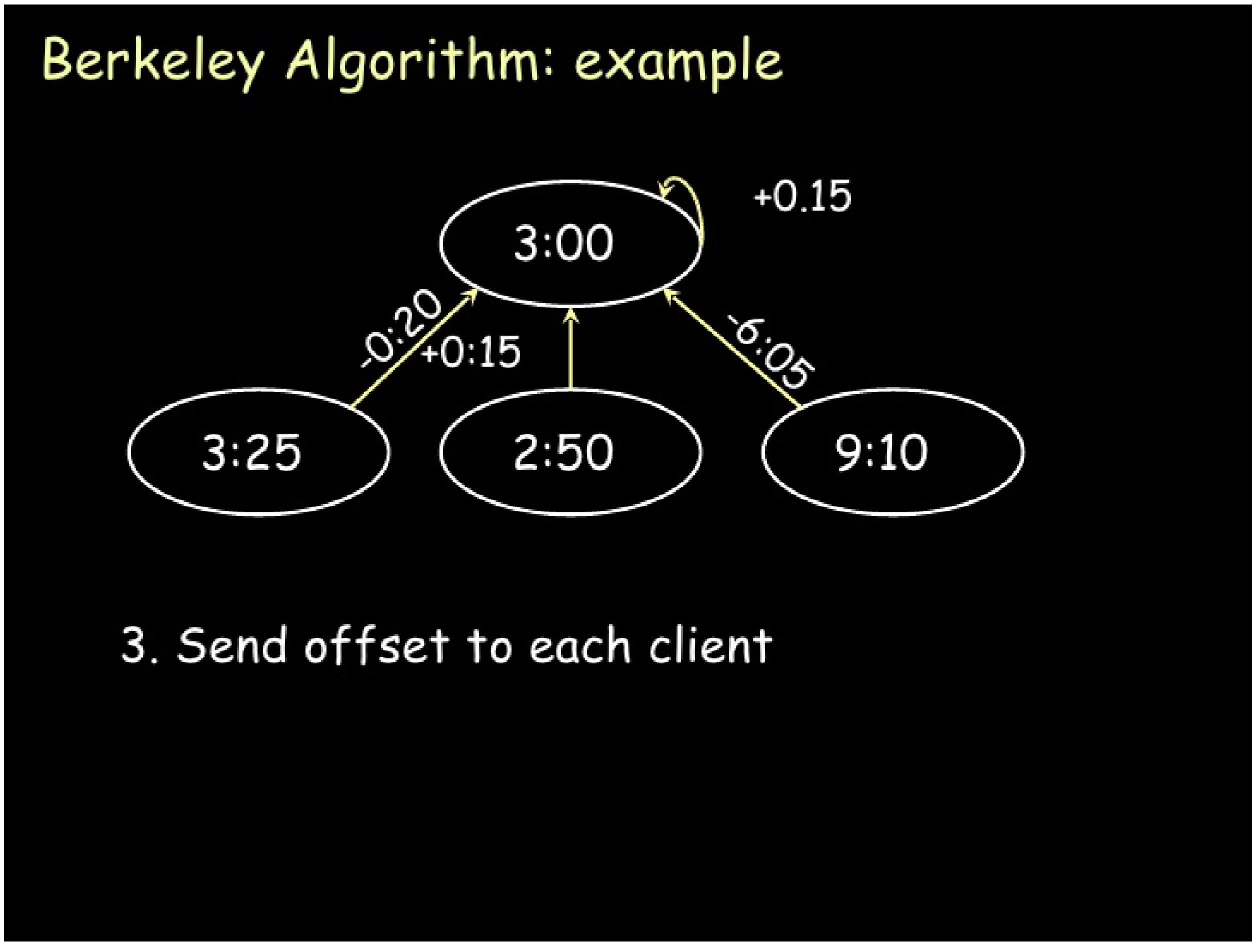

Figure 2: Clock Synchronization using Berkeley Algorithm

Player Disconnections

Players will send heartbeats to the other players that they are connected to. This doesn't scale well if every player is connected to every other player, but with a server we can connect clients in more interesting topologies like a hypercube or a taurus. In this configuration, players still send heartbeats to all the other players they are connected to, but those heartbeats aren't flooded through the whole network. Instead, once a player detects that one of the players it is connected to disconnects, it will flood an explicit disconnection message to the network. This means that every client will have to keep the whole graph of the network topology to detect partitions, but since each client has to keep the whole world state around, that's an acceptable cost.

Stat Collection

Stats are an important feature of any shooter game, let alone a Battle Royale game as it allows the players to see how well they have been doing in the game. We plan to keep track of player stats, such as a k/d (Kill / Death) amongst others by maintaining a Distributed Key/Value store using Conflict-Free Replicated Data Types. Here the key would be the username of the player (as this will be unique), and the value would be the interesting stats we would like to provide. This will provide a dimension of distributed systems design that is less latency-bound than the regular game mechanics.

The operations on the stats that we will provide are as follows:

- **get(username)** : Contacting the server to get the updated stats of a particular player
- **add(username, stats)** : Requesting the server to add a new pair of username, stats to the store
- **update(username, stats)** : Requesting the server to update the statistics of a particular player

Cloud as a Persistent Service Provider

We'll have a centralized server for peer discovery, which will be hosted on Azure. We also plan to have a "backup" client on Azure, so that the game is playable by only 1 client as well.

Technology stack

We'll use go for nearly everything. We'll use a simple 2D game engine, so that we don't have to work too hard on graphics. We'll use a games library to facilitate that development, but otherwise we'll stick to relatively standard go.

Limitations and Assumptions

- The server will only be hosting 1 game session at any point of time.
- The number of players that can join a game session has a maximum limit (which is 10).
- We assume that there won't be any Byzantine failures occuring in the system. (A Byzantine fault is any fault presenting different symptoms to different observers. A Byzantine failure is the loss of a system service due to a Byzantine fault in systems that require consensus.)

Stretch Goals

- Making the system Byzantine Fault tolerant. This will not be easy at all and as much as we would like to do this, it is probably not realistic to aim for this.
- Improving the game to have more features like stage hazards, status effects, and alternative goals which would require a more complicated distributed state than we have initially decided.

Development Plan

Deadline	Task
Mar 2	Project Proposal Draft Due
Mar 9	Project Proposal due; Finalize external libraries being used; develop a MVP for the game; learn about dinv, GoVector and ShiViz
Mar 16	Implement peer discovery, clock synchornisation and a basic version of the game that allows players to join and control their tanks
Mar 23	Implement shooting models and complete the game so that it has a winner; Also implement basic version of stats collection
Mar 30	Implement additional features for the game as time permits.
Apr 6	Stress-test the game with a mix of malicious and non-malicious users; Complete report with Dinv, GoVector and ShiViz
Apr 9-20	Project Demo on a date TBD

SWOT Analysis

Strengths

- Team members have worked with each other on the previous assignments of the course.
- Team members are diligent and punctual.
- All members are good at researching as well as solving potential issues.

Weaknesses

- There are very limited resources available for building a low-latency distributed game.
- None of the members have any prior experience with making multiplayer online games.

Opportunities

- We can make the game as complex as we would like given the time limitations that would allow us to expand on the distriuited system we are aiming to make.
- Conflict-Free Replicated Data Types is a very new concept for all of the members and would require a lot of work for a proper implementation.

Threats

- Commitment to exams or assignments from other courses may interfere with progress.
- None of us have any experience with GoVector, ShiViz or Dinv, all of which we are aiming to incorporate in our project.

Resources

https://www.cs.ubc.ca/~gberseth/projects/ArmGame/ARM%20Game%20With%20Distributed%20States%20-%20Glen%20Berseth,%20Ravjot%20%20%20%20%20%20Singh.pdf

http://www.it.uom.gr/teaching/distrubutedSite/dsIdaLiu/lecture/lect11-12.frm.pdf

https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Time-Clocks-and-the-Ordering-of-Events-in-a-Distributed-System.pdf

https://en.wikipedia.org/wiki/Berkeley_algorithm

http://pmg.csail.mit.edu/papers/osdi99.pdf

https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

https://en.wikipedia.org/wiki/Battle_City_(video_game)