ECE419 Lab3 - Preliminary Design Document

Nayeem Zen, 998927924, nayeem.zen@mail.utoronto.ca
Warren Marivel, 998584651, warren.marivel@utoronto.ca

Naming Service:

We'll be using a centralized naming service to coordinate the list of players in the game. This naming service would always have a fresh state of the current list of players playing the game. When a new player joins the game, they announce their join to the naming service. The naming service will inform the joining player of all the existing players (prior to this join). Moreover, the naming service will also broadcast the new player's join to all the other players. This will be a centralized server, but will only coordinate the list of players playing a game at any point.

Total Order Multicast:

We'll use TOM to synchronize events among peers as per Lamport timestamps. Each event will be timestamped and put it a queue which is sorted according to Lamport's timestamps. Before pushing the event to the application, the sender will wait for an ACK for that event from all other peers. Once it receives all the ACKs, it will send a RELEASE message to all the clients to let them know that it is ok to render the event. This will maintain the causal order of events.

Bullet ticker:

To ensure that the bullet ticking event works consistently among all players in this non centralized version of the game, we'll be using a 'leader election' scheme. A leader at any point in the game, will be a player who will have the additional responsibility of broadcasting a tick event to all the other players to ensure that the missile ticking is consistent. A leader at any point will be a player with the lowest ID. A player knows all the other players playing a game at any point, hence the assignment of such a leader is always deterministic. In the event where a player with a lowest ID quits (and by extension the leader quits), the player with the next lowest ID assumes the ticking responsibility. When a leader announces a quit event, all the other players again know implicitly who the new leader is.

Dynamic Join/Quit:

To join a game, a client has to:

- Request the Naming Service for the list of players and their credentials
- Once it has the list of current players, it should send a join request to all players
- It should then wait for the other players to acknowledge him before being considered part of the game.
- Once the player is part of the game, it can spawn and start receiving and sending events.

To quit a game, a client has to:

- Send a "quit" request to the naming service
- Send a "quit" request to the other clients

Packet Structure

The packet structure shall consist of the following fields:

- Status codes (Event types, Error codes)
- Player coordinates
- Player score
- Lamport timestamp
- Client name (unique)
- Client id (unique)

Status codes indicate which events created the packet (e.g. move left, missile tick, etc) as well as to indicate any errors that may have occurred. Different types of errors are distinguished by unique error codes (e.g. -100 is timeout, -101 client already exists, etc).

Player coordinates and scores need to be sent with the packets because the system intends to support dynamic joins. The player coordinates and scores shall update any new clients joining with the current state of the game.

Lamport timestamps are needed in order to ensure causality and ordering of events. They shall act as the sequence number for the packets.

Client names shall be unique to distinguish packet from each player as there may be an overlapping lamport timestamp.

Fault Tolerance

The system shall resist failures of the naming service, clients and leader clients by implementing the following fault tolerance mechanisms:

- If any client stops receiving missile tick events for a specified time period, then a re-election must be carried out to elect a new leader

Evaluate the portion of your design that deals with starting, maintaining, and exiting a game – what are its strengths and weaknesses?

- If a player quits or joins the game, the game handles it gracefully by allowing players to continue to play the game until all the players have left. It does this by using the bully algorithm for leader election for electing a new leader to broadcast bullet fire events every 200ms.

Evaluate your design with respect to its performance on the current platform (i.e. ug machines in a small LAN). If applicable, you can use the robot clients in Mazewar to measure the number of packets sent for various time intervals and number of players. Analyze your results.

- The game is very responsive with events appearing simultaneously across computers in the EECG LAN. The game uses TCP sockets for communication so FIFO delivery of packets is guaranteed, so the game is not affected by network reordering. The totally ordered multicast algorithm uses 3(n-1) packets to per event on each client.

How does your current design scale for an increased number of players? What if it is played across a higher- latency, lower-bandwidth wireless network { high packet loss rates? What if played on a mix of mobile devices, laptops, computers, wired/wireless?

- In terms of the number of messages per event, the design increases linearly with the number of players since we exchange 3(n-1) messages per event across the network. - If played across a higher latency, we need to tweak the algorithm controlling the election process to designate the missile ticker. The algorithm relies on timeouts to know when to begin and end an election (bully algorithm).
- The game uses TCP/IP, which ensures packet delivery. At high packet loss rate, delivery of packets may be delayed, and the election algorithm's timeouts would have to be adjusted to account for this. The packet losses would not affect the game's states since we rely on lamport clock for ordering of events.

Evaluate your design for consistency. What inconsistencies can occur? How are they dealt with?

- Events broadcasted among the different players may arrive in a different order on each player. To deal with this, we order events using lamport timestamps.
- Some events may fail to reach a certain client in time. This would cause the client to miss on an event rendered by all the other clients. To account for this, we make sure that all clients receive the same events by having the clients acknowledge receipt of events.