**Design Document**

**Programming Team 1:**
Zaara Syeda - 998199765
Ghulam Umar - 998513988

**Originally worked with programming team below, but design has slightly diverged:**
**Programming Team 2:**
Saras Leelodharry - 998419064
Prerna Chandna -998584839

## Overall distributed algorithm functionality, components, interactions

Our distributed algorithm works using **UDP multicast** and at-least once message delivery. We use sequence numbers as an id for each packet sent by each client. Each client then maintains a sub-queue for all other clients and places the incoming packets into the appropriate clients queue by sequence number. When a client notices no gaps between two successive packets in a sub-queue for another client, it takes the head of that sub-queue and places it on the final queue. This final queue will be totally ordered since each client is cycling through the sub-queues in the same order and placing packets on the final queue only when no gaps are detected on the sub-queues. We also implement **dynamic joins** as described in the next section. In order to handle missing ticks, we implement **leader election.** Additionally, since we are using UDP, we implement **handling missing packets** due to the chance of networking dropping a packet. Moreover, we also implement **fault tolerance** as describe below. In order for our algorithm to work, we rely on an **assumption of bounded RTT time of 2 seconds.**

Our leader election algorithm runs a thread every 500ms, if no leader is detected the tread cycles through all the known clients and elects the leader with the lexicographically smallest name, we therefore **require all players to have unique names.**

## How a player locates, joins and leaves a game of Mazewar

**Locating:**

Since we are using UDP multicast, players do not need to locate the game through a naming service. Instead, they all communicate through the UDP multicast address assigned to the game.

**Joining:**

Our algorithm implements dynamic joins. Since we are using UDP multicast, each client joins the game by connecting to the multicast group address. Once the client is joined, it multicasts a freeze packet. This causes a momentary freeze in the game where the other clients are unable to move. During this time, the newly joined client listens for discovery packets from other clients which are continuously multicast with current location every 1 second. The newly joined client collects these locations and places all clients on its game board. Once all clients have been added, the new client adds itself to the board, multicasts an unfreeze packet which has its new location. All other clients upon receiving an unfreeze read the location of the new client and add it to their boards. The game now resumes with everyone synchronized.

**Leaving**:

To implement the leaving of a game, when a client presses key “q” a packet with type=ECHO_BYE is multicast. When receiving this packet, clients remove the leaving client off their boards and remove any data structures used for this client.

**If you handle failures: what happens when a process loses contact with another process?**

In order to implement fault tolerance, we have a scheduled thread which runs every 1.5seconds. This thread iterates through the clients that it knows about and detects missing discovery packets for them. Once it detects that multiple discovery packets have not been received, it assumes that this client has lost contact. Therefore, it multicasts a packet of type = ECHO_BYE on behalf of this client. Upon receiving this multicast packet, all other clients are able to remove this disconnected client from their game boards using the leaving code as described above.

**Timings of protocol events, and how they provide sufficient consistency for the game state**

Each event is multicast by the clients to the UDP multicast group address. Upon receiving the multicast packet, it is queued onto the sub-queue of the client who sent the packet. This queue is a priority queue based on sequence numbers. Therefore, each client waits for gaps to be filled in this queue for all other clients. Once the queues have no gaps, the messages can be added to the final totally ordered queue. This provides us sufficient consistency because each client iterates through the sub-queues in the same order which are all ordered according to sequence numbers. We also obtain consistency in firing events because we elect a leader to multicast tick packets every 200ms. Finally, we obtain consistency in killing a client by allowing only the leader to execute the kill code. The leader first kills the client off his board. It then uses the new coordinates produced by the respawn to create a packet of type=ECHO_KILL. It multicasts this packet to the other clients. Upon receiving this packet, the clients execute a specialized kill function which removed the victim off the board but spawns it to the location provided by the leader in the packet.

**Communication protocol – packet types and formats**

**Packet:**

Below is the the data members of our packet Class. Each multicast packet is sent with the corresponding type and event field.

```
public static final int ECHO_NULL    = 0;
public static final int ECHO_NEW = 102;
public static final int ECHO_BYE     = 300;       //used for exiting the game
public static final int DISCO   = 302;            //used to send discovery packets every second
public static final int LEADER   = 303;//used to report the result of leader election
public static final int REQUEST_MISSING   = 305;        //used to request for missing packet
public static final int RESPONSE_MISSING   = 306;       //used for responding with the missing packet

public int type = ECHO_NULL;

public static final int UP = 1;
public static final int DOWN = 2;
```

```
public static final int RIGHT = 3;
public static final int LEFT = 4;
public static final int FIRE = 5;
public static final int CONN = 6;
public static final int DIS = 7;
public static final int TICK = 8;
public static final int LOC = 9;
public static final int KILL = 10;
public static final int FREEZE   = 11;    //used for dynamically joining by sending freeze and unfreeze events
public static final int UNFREEZE   = 12;

int event;
int playerCount;
int x,y;
int missingIndex ;              //used to give index of missing packet that needs to be returned
int response;
Direction dir;
String player;
String killer;
String victim;
String leader;
String missingPackOwner;                //used to set which client's packet is missing

int packet_id;
String message;
```