Computer Networking Project

Adham Walid Said Zaki 23P0024

Amir Tamer Abdelreheim 23P0248

Moaz Ahmed Fathy 23P0049

Mohamed Wael Badra 23P0059

Mostafa Amr Nabil 23p0206

Basem Walid Talaat 23p0246

1-Proposal:

1. Assigned Scenario

This project focuses on designing and implementing a custom communication protocol to support a real-time competitive multiplayer game. The game is a grid-claim challenge where multiple players connect to the same server and race to click cells on a shared grid. Once a player clicks a free cell successfully, that cell becomes permanently claimed by that player and visually updates for everyone. When the grid is filled the player with the most claimed cells wins the game.

The theme is intentionally simple so the main focus remains on network communication performance, synchronization, and state consistency. The server maintains full authority over the game state, while clients only send user input and display updates received from the server. This ensures that players never disagree about who owns which cell.

2. Motivation

Real-time interactive systems are one of the most practical and important applications of network communication. Almost all modern games, teamwork applications, and collaborative tools depend on fast and synchronized data exchange. Although the click-to-claim mechanic looks simple, it immediately exposes essential networking challenges:

- **Latency: If Player A clicks first but Player B's update arrives sooner, how do we fairly decide the winner?**
- **Packet loss: If a claim update is lost, do we freeze the game waiting for it?**
- **State synchronization: How do we guarantee that every player sees the exact same grid?**
- **Fairness: How are simultaneous actions resolved?**
- **Scalability: Can more players join without breaking performance?**

This project gives hands-on experience in solving these issues the way professional multiplayer games do. It bridges theory with real networking engineering, teaching how protocols are designed, tested, and optimized under real constraints.

We are also motivated to use a game scenario because it creates instant, observable feedback. If the protocol fails, the game visually breaks… which is the best teacher

3. **Proposed Protocol Approach (Grid Clash) UDP is chosen over TCP because this is a real-time competitive game where speed matters more than perfect reliability. TCP delays updates due to retransmissions and ordering rules, which would freeze gameplay and ruin fairness. UDP allows data to flow continuously with very low latency. If a packet is lost, the next update fixes the state anyway because the server sends full state snapshots regularly. The design accepts small imperfections in exchange for smooth gameplay.**

UDP provides:

- **Lower latency**
- **No blocking on lost packets**
- **Better performance with many players**
- **Control over how to handle delays and reordering**

This matches real multiplayer games, making the project realistic and educational.

2-Mini RFC Draft:

1. Reason for Protocol (Explanation)

The protocol is needed to ensure that every player in the game sees the exact same grid state at the same time while actions are happening quickly and possibly at the same moment. Without a custom protocol managing how information travels between players and the server, there would be chaos: two players could claim the same cell, some devices might show outdated information, and packet delays could give unfair advantages. The protocol provides strict rules for how data is packaged, labeled, delivered, and handled, making sure that the server stays the single source of truth and that clients always stay synchronized.

Since UDP does not automatically handle ordering, reliability, or timing, the protocol adds simple mechanisms to detect lost, late, or duplicate packets and ensures that the newest valid update always takes priority. This eliminates the need to wait for missing data and keeps the game fast and competitive. In short, the protocol is the backbone that keeps gameplay fair, responsive, and consistent for all players, no matter how messy the network conditions get.

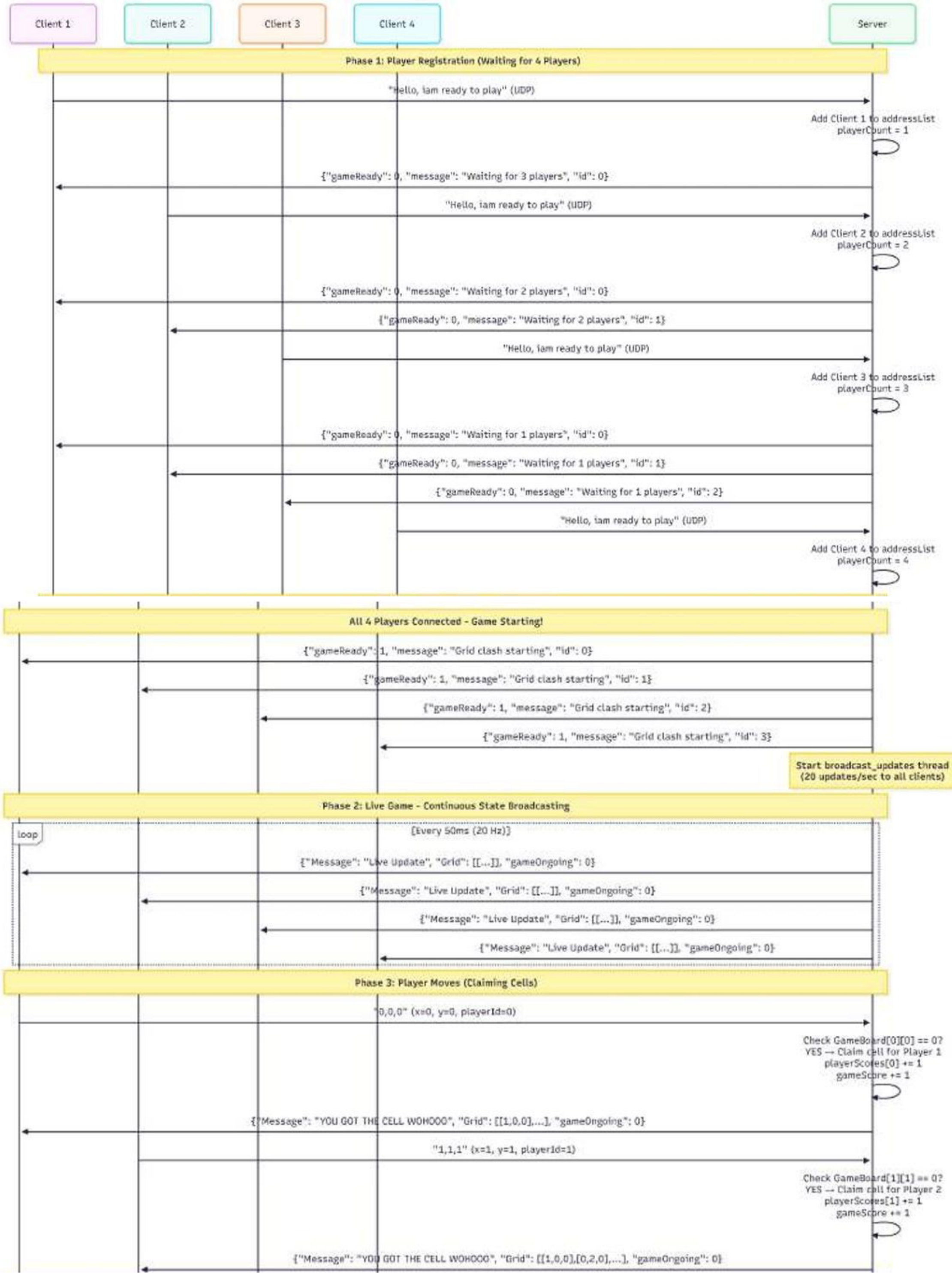
2. Protocol Architecture

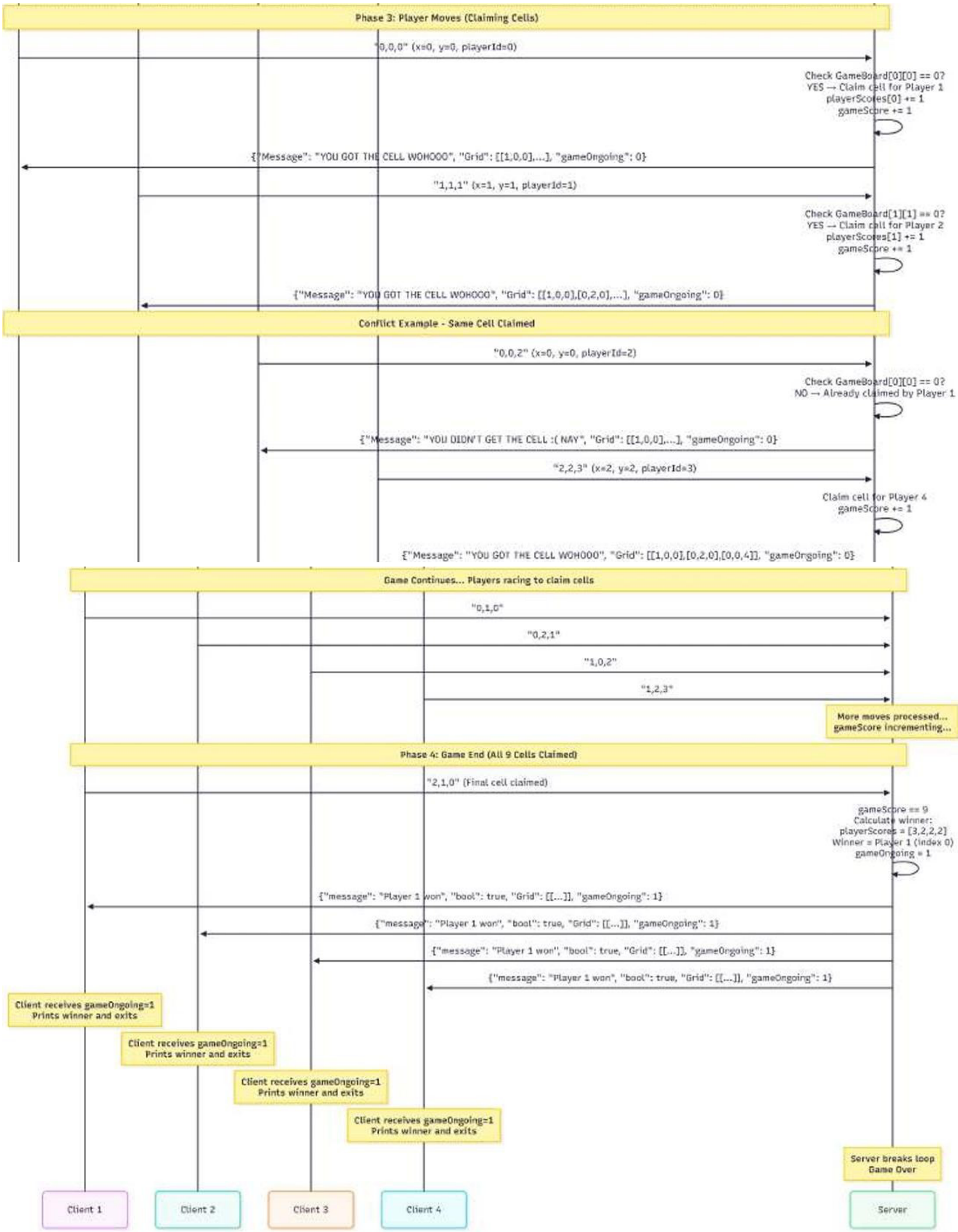
The protocol follows a server-authoritative architecture in which a central server is responsible for always maintaining the complete and correct game state. Clients do not communicate with each other directly. Instead, each client sends its input events (such as clicking a grid cell) only to the server. The server determines which player successfully claims the cell based on the order in which the packets arrive and then updates the official state of the grid.

After processing any input, the server broadcasts updated game snapshots to all connected clients. These snapshots contain the full current state of the grid, ensuring that every player always sees the same information. Even if some updates are lost due to network issues, the next snapshot received will correct the client's state.

This design makes the server the single source of truth, prevents cheating or conflicting states between players, and avoids the complexity of peer-to-peer synchronization. UDP provides low-latency delivery of messages, while the protocol's header fields handle ordering and filtering of packets to maintain smooth and consistent gameplay.

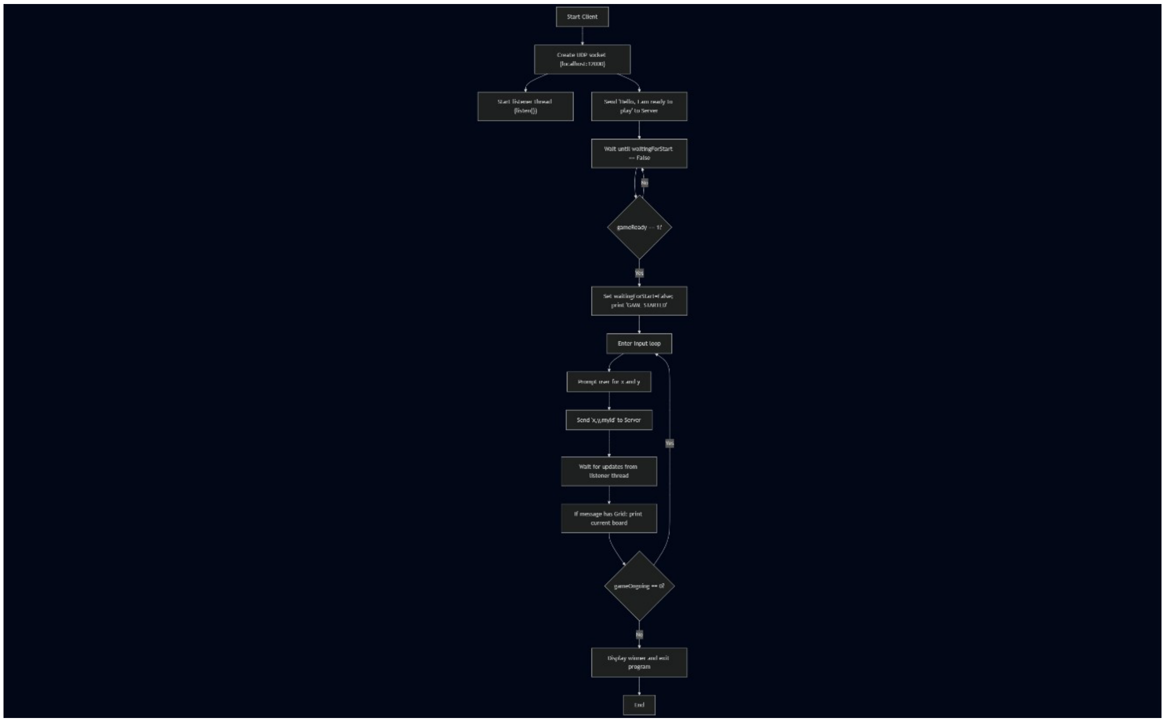
Sequence Diagram for game: (sorry for the low quality)



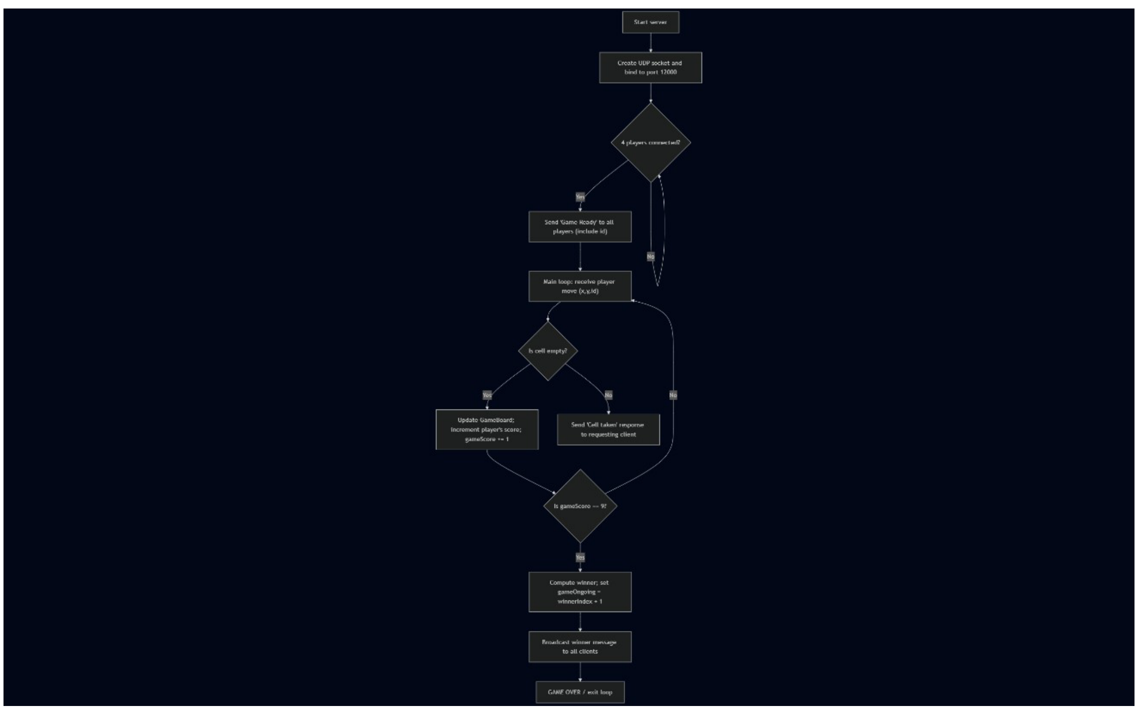

Phase 3: Player Moves (Claiming Cells)
"0,0,0" (x=0, y=0, playerId=0)
Check GameBoard[0][0] == 0?
YES → Claim cell for Player 1
playerScores[0] += 1
gameScore += 1
{"Message": "YOU GOT THE CELL WOHOOO", "Grid": [[1,0,0],...], "gameOngoing": 0}
"1,1,1" (x=1, y=1, playerId=1)
Check GameBoard[1][1] == 0?
YES → Claim cell for Player 2
playerScores[1] += 1
gameScore += 1
{"Message": "YOU GOT THE CELL WOHOOO", "Grid": [[1,0,0],[0,2,0],...], "gameOngoing": 0}
Conflict Example - Same Cell Claimed
"0,0,2" (x=0, y=0, playerId=2)
Check GameBoard[0][0] == 0?
NO → Already claimed by Player 1
{"Message": "YOU DIDN'T GET THE CELL :(NAY", "Grid": [[1,0,0],...], "gameOngoing": 0}
"2,2,3" (x=2, y=2, playerId=3)
Claim cell for Player 4
gameScore += 1
{"Message": "YOU GOT THE CELL WOHOOO", "Grid": [[1,0,0],[0,2,0],[0,0,4]], "gameOngoing": 0}
Game Continues... Players racing to claim cells
"0,1,0"
"0,2,1"
"1,0,2"
"1,2,3"
More moves processed...
gameScore incrementing...
Phase 4: Game End (All 9 Cells Claimed)
"2,1,0" (Final cell claimed)
gameScore == 9
Calculate winner:
playerScores = [3,2,2,2]
Winner = Player 1 (index 0)
gameOngoing = 1
{"message": "Player 1 won", "bool": true, "Grid": [[...]], "gameOngoing": 1}
{"message": "Player 1 won", "bool": true, "Grid": [[...]], "gameOngoing": 1}
{"message": "Player 1 won", "bool": true, "Grid": [[...]], "gameOngoing": 1}
{"message": "Player 1 won", "bool": true, "Grid": [[...]], "gameOngoing": 1}
Client receives gameOngoing=1
Prints winner and exits
Client receives gameOngoing=1
Prints winner and exits
Client receives gameOngoing=1
Prints winner and exits
Client receives gameOngoing=1
Prints winner and exits
Server breaks loop
Game Over
Client 1
Client 2
Client 3
Client 4
Server

Flow Charts (For Prototype):

Client:



Server:



3. Message Formats

Every Grid Clash packet begins with a fixed-size header that allows the receiver to validate, order, and correctly interpret incoming data. The payload that follows depends on the message type. Two message types are currently supported: EVENT messages sent from clients to the server, and SNAPSHOT messages sent from the server to all clients.

***Header Table+Message(Payload):**

Field	Size	Sender	Purpose	Role in the Game
protocol_id	4 bytes	Both	Identifies GridClash packets	Non-game packets are ignored
version	1 byte	Both	Protocol compatibility	Prevents errors if protocol updates
msg_type	1 byte	Both	Distinguishes message content	0 = EVENT, 1 = SNAPSHOT
snapshot_id	4 bytes	Server	Detects older game states	Client applies only the newest grid update
seq_num	4 bytes	Both	Detects duplicates, loss, and reordering	Receiver keeps the latest valid message
server_timestamp	8 bytes	Server	Measures delay and staleness	Late snapshots discarded
payload_len	2 bytes	Both	Length of payload section	Supports variable data sizes
payload	Variable	Both	Action or full game state	EVENT: clicked cell, SNAPSHOT: updated grid or other indicators to establish connection

🧠 1. Overview

Every message exchanged follows a consistent structure that ensures synchronization, latency tracking, and error detection.

The communication is bi-directional:

- **Server → Client: Sends game state updates (snapshots), notifications, and results.**
- **Client → Server: Sends player actions (cell selections) and readiness signals.**

🛰 2. Server → Client Message Format

Example: Regular Snapshot Message


```
{
  "protocol_id": 1234,
  "version": 1,
  "msg_type": "SNAPSHOT",
  "snapshot_id": 42,
  "seq_num": 84,
  "server_timestamp": "2025-11-04T21:45:38.123456",
  "payload_len": 2048,
  "payload": {
    "Message": "YOU GOT THE CELL WOHOOO",
    "Grid": [
      [1, 2, 0],
      [0, 1, 0],
      [2, 0, 1]
    ],
    "gameOngoing": true,
    "id": 1,
  }
}
```

Payload Structu re

Payload Field	Type	Description
Message	***String***	**Status message or feedback from the server.**
Grid	***2D Array (3×3)***	**Current game board showing player moves (0 = empty).**
gameOngoing	***Boolean***	**Indicates if the match is still active.**
id	***Integer***	**ID of the player associated with this update.**

Example: Game Over Message


```
{
  "protocol_id": 1234,
  "version": 1,
  "msg_type": "SNAPSHOT",
  "snapshot_id": 99,
  "seq_num": 0,
  "server_timestamp": "2025-11-04T21:50:00.000000",
  "payload_len": 2048,
  "payload": {
    "Message": "Player 2 won",
    "Grid": [
      [2, 1, 2],
      [1, 2, 1],
      [2, 1, 2]
    ],
    "gameOngoing": false,

  }
}
```

🎮 3. Client → Server Message Format

Clients send two types of messages:

1. Connection / Readiness Message
2. Game Action Message

3.1 Connection / Readiness Message

When a client first connects, it sends a plain-text message to indicate readiness to play.

Hello, I am ready to play

This notifies the server to register the player and assign them an ID.

3.2 Game Action Message

Once the game starts, each player sends their move in the following format:

x,y,id

Example

1,2,0

Field Descriptions

Field	Type	Description
x	*Integer*	X-coordinate of the selected cell (row index).
y	*Integer*	Y-coordinate of the selected cell (column index).

id	***Integer***	**Unique player ID assigned by the server.**

Client Message Scenarios

Scenario	Example Message	Explanation
Player joins	**"Hello, I am ready to play"**	**Client announces readiness.**
Player selects an empty cell	**"0,1,2"**	**Player with ID 2 claims cell (0,1).**
Player selects an occupied cell	**"2,2,1"**	**Request rejected with a failure message in server response.**

---

4. Example Communication Flow

Step	Sender	Message Type	Example / Description
1	**Client**	**Connection**	**"Hello, I am ready to play"**
2	**Server**	**Acknowledgement**	**{"gameReady": 0, "message": "Waiting for more players", "id": 0}**
3	**Server**	**Game Start**	**{"gameReady": 1, "message": "Grid clash starting", "id": 0}**
4	**Client**	**Game Action**	**"1,2,0"**
5	**Server**	**Snapshot**	**{"Message": "YOU GOT THE CELL WOHOOO", "Grid": [[...]], ...}**
6	**Server**	**Game Over**	**{"Message": "Player 3 won", "gameOngoing": false}**

Phase 2:

Changes:

In phase 2 along with the making of the GUI for the game we applied many changes to the protocol of the game to introduce reliability and optimization to improve the performance of the game.

1. Updated Message Format and Communication Flow

To support the new optimization strategies, the communication protocol has been expanded to include four distinct message types. The DELTA message is the primary method for game state updates when clients are synchronized.

How it works:

We added two new dictionaries on the server side: client_acks and grid_history (referred to as prevGrids in your logic).

The server now receives an acknowledgment (ACK) from the client after sending a snapshot. This acknowledgment contains the Snapshot ID of the grid the client last received. When the server receives this ACK, it stores the Snapshot ID for that specific client in a dictionary, mapping their address to their latest confirmed game state.

Then, in the broadcast updates thread, the Snapshot IDs for each client are checked, which is where the Delta Encoding logic begins. The server saves the grid state after every update in the grid_history dictionary, mapping a Snapshot ID to a specific grid.

When broadcasting updates, the server checks each client's previous grid state:

1. **Standard Delta: If the client's acknowledged grid is the same as the immediate previous grid in our history, it means the client is missing only one update. We calculate the specific changes needed to update their grid and send only those changes (a "Delta").**
2. **Custom Delta: If the client is behind by more than one update (missing several Snapshot IDs) but their state is still stored in the grid_history, we retrieve their old grid. We then calculate the changes required to bring that specific old grid up to the current state and send those changes.**
3. **Full Snapshot: Finally, if the client is too far behind and their last acknowledged grid has been removed from the history, we send them the Full Grid with the message type SNAPSHOT.**

To address the input lag issues, we introduced the INFO message type. This is a lightweight response sent immediately by the Main Thread when a player submits a move. It contains a confirmation message (e.g., "Nice move!") and the player ID but intentionally excludes the grid data to prevent buffer bloat. Finally, the client now sends an ACK (Acknowledgment) message containing the snapshot_id it just received and the update is handled by the broadcast updates thread. This critical addition allows the server to track exactly which state the client possesses, enabling accurate delta calculations.

Sender	Message Type	Payload Structure (JSON)	Description
Server	**DELTA**	{"Changes": [[r, c, v], ...], "gameOngoing": bool, "timestamp": float}	**Partial Update:** Sent when the client is synced. Contains only the list of changed cells [row, col, value] to save bandwidth.
Server	**SNAPSHOT**	{"Grid": [[...], ...], "gameOngoing": bool, "timestamp": float}	**Full Update:** Sent as a fallback if the client is lagging or just joined. Contains the full 20x20 grid.
Server	**INFO**	{"Message": "Nice move!", "id": 1, "timestamp": float}	**Move Confirmation:** Immediate lightweight response to a player's move. Does **not** contain grid data, preventing buffer bloat.
Client	**ACK**	{"msg_type": "ACK", "snapshot_id": 105, ...}	**Acknowledgment:** Sent by the client upon receiving a SNAPSHOT or DELTA, telling the server "I have state #105".

Test:
These are the commands we used to run certain tests on the game.

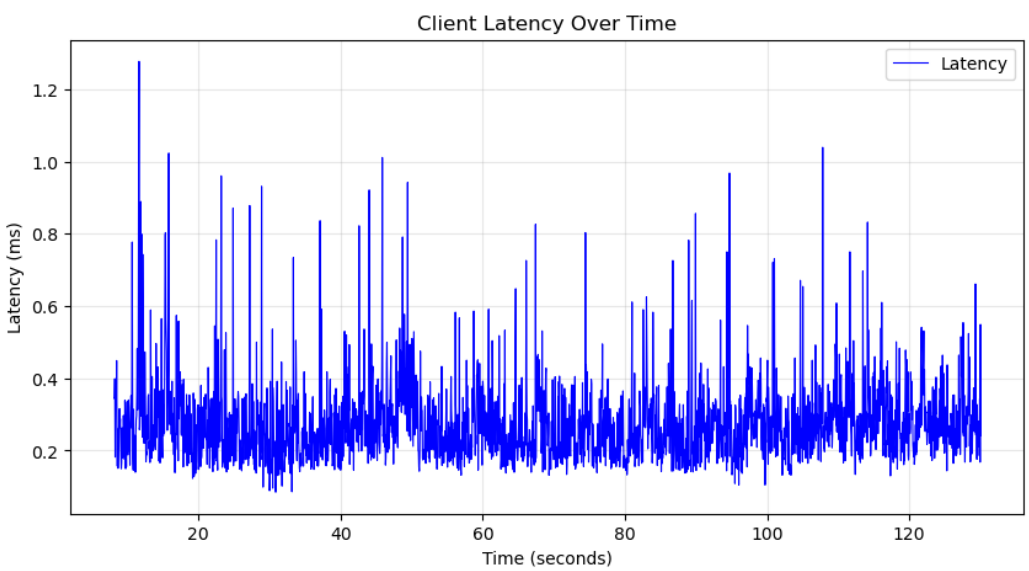
 # To add 100ms delay:
sudo tc qdisc add dev lo root netem delay 100ms

 # To add 5% packet loss:
sudo tc qdisc change dev lo root netem loss 5%

 # To add 2% packet loss:
sudo tc qdisc change dev lo root netem loss 2%

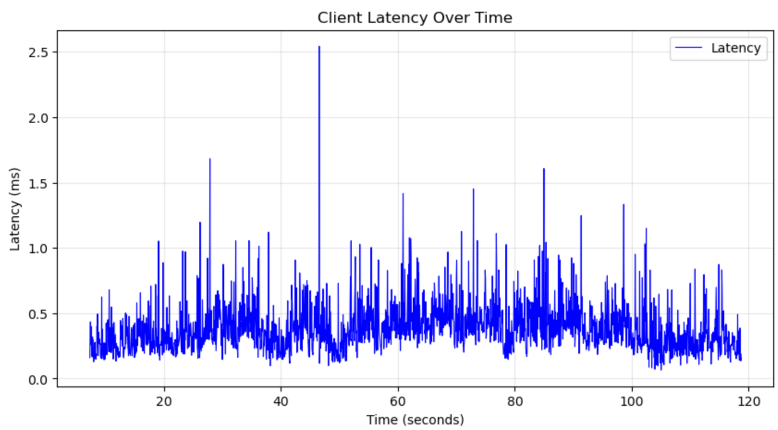
Results:

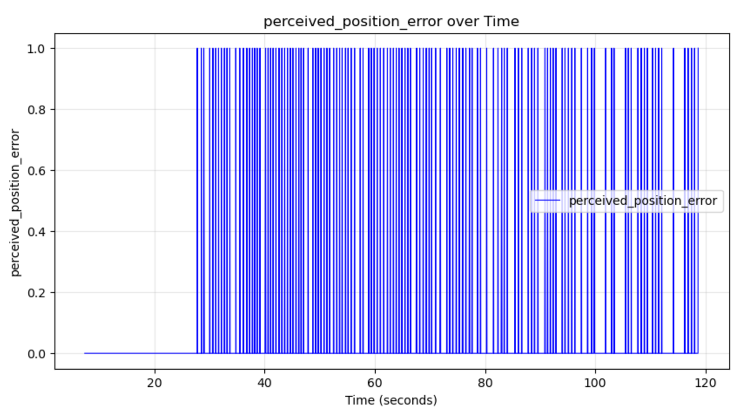
Metric	Value
Average CPU Utilization	1.76%
Max CPU Utilization	10.00%
Average Client Latency	0.20 ms



2.Loss 2%

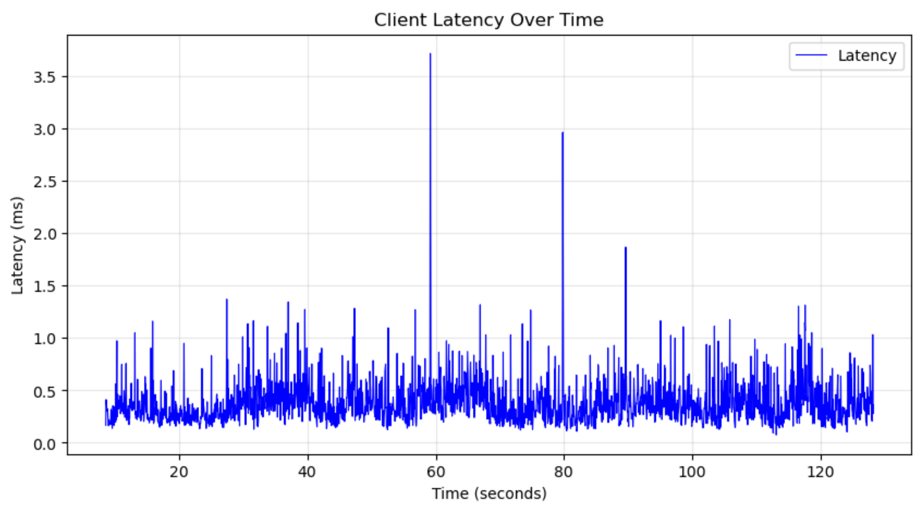
Metric	Value
Average CPU Utilization	2.48%
Max CPU Utilization	10.00%
Average Client Latency	0.30 ms
Average Perceived Position Error (MPE)	0.062
95th Percentile MPE	1.0





3.Loss 5%

Metric	Value
Average CPU Utilization	2.38%
Max CPU Utilization	10.00%
Average Client Latency	0.29ms
Average Perceived Position Error (MPE)	0.059
Reliability	99.95%



4.Delay 100ms

Metric	Value
Average CPU Utilization	1.95%
Max CPU Utilization	10.00%
Average Client Latency	100.4ms
Average Perceived Position Error (MPE)	0.059
95th Percentile MPE	1.0