Mini-RFC: GridClash Binary Protocol (GCBP v1)

Submitted to:

Dr. Karim Ahmed

Eng. Noha Wahdan

Submitted by:

Loay Mahmoud 23P0419 Omar Ayman 23P0244

Ali Moataz 23P0280 Omar Osama 2300090

Jana Tamer 23P0173 Jana Mohamed 2301032

TABLE OF CONTENTS

1. Introduction

The GridClash Binary Protocol (GCBP v1) is a custom UDP-based application-layer protocol designed to synchronize player actions and global state in a low-latency, loss-tolerant multiplayer game called Grid Clash. The game involves multiple players competing on a shared 2D grid (e.g., 20×20 cells). Players click cells to claim them; conflicts are resolved by the server based on message timestamps. The protocol ensures near real-time synchronization across clients while remaining resilient to packet loss and network jitter. Goals: - Achieve sub-50 ms latency for real-time state updates. - Maintain synchronization consistency across 4 clients. - Handle transient network loss (≤5%) gracefully using selective reliability and redundancy. Assumptions and Constraints: - Transport: UDP (connectionless) - Maximum payload size: ≤ 1200 bytes - Typical update frequency: 20 Hz - No per-packet retransmissions (loss-tolerant design) - Runs cross-platform (Linux/Windows, Python 3

2. Protocol Architecture

Entities: - Server: Authoritative game state manager. Periodically broadcasts state snapshots and resolves conflicts. - Client: Represents an individual player. Sends actions (e.g., cell acquisition requests) and applies server updates. - Network: Unreliable medium with potential delay, jitter, and loss.

2.1 Data Flow Overview:

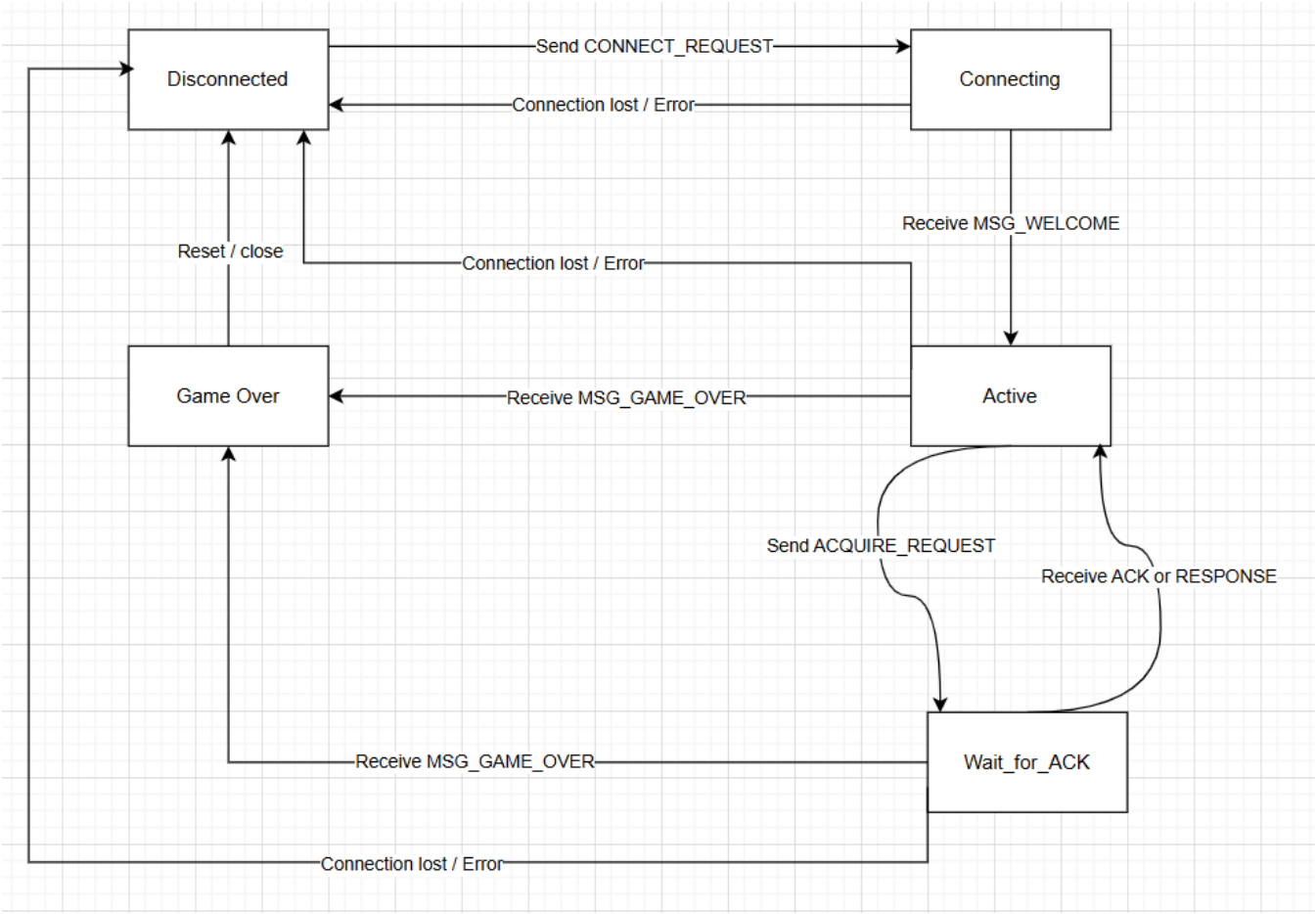
Client → Server: CONNECT_REQUEST → WELCOME Client → Server:

ACQUIRE_REQUEST(cell_id, timestamp) Server → Client:

ACQUIRE_RESPONSE(success/failure) Server → All:
GAME_STATE(snapshot_id, player_positions) Server → All:
GAME_OVER(winner_id) Both sides: HEARTBEAT, ACK for selective reliability

Reliability Enhancement: The protocol incorporates selective reliability by sending ACK messages for critical actions (e.g., ACQUIRE_REQUEST, GAME_OVER), while non-critical updates (e.g., GAME_STATE) rely on redundancy and sequence tracking.

2.2 Protocol Finite State Machine (FSM)



3. Message Formats

All messages are binary-encoded with fixed-size headers for minimal overhead. Messages begin with a 24-byte header, followed by an optional payload.

Field	Size (bytes)	Description
protocol_id	4	ASCII identifier, fixed to 'GRID'
version	1	Protocol version (1)
msg_type	1	Message type ID
snapshot_id	4	Unique snapshot identifier
seq_num	4	Sequential number for deduplication
server_timestamp	8	Milliseconds since epoch (server clock)
payload_len	2	Payload size in bytes

3.1 Message Types:

Type	Code	Direction	Purpose
WELCOME	0x01	Server → Client	Confirm connection, assign ID
GAME_STATE	0x02	Server→All	Broadcast grid snapshot
ACQUIRE_REQUEST	0x03	Client→Server	Request to claim a cell
ACQUIRE_RESPONSE	0x04	Server → Client	Grant/deny cell acquisition
GAME_OVER	0x05	Server→All	End-of-game notice and results
PLAYER_MOVE	0x06	Client→Server	Update player position
ACK	0x07	Both	Confirm receipt of critical packet
HEARTBEAT	0x09	Both	Liveness indicator
CONNECT REQUEST	0x0A	Client → Server	Initiate handshake

Example Header Encoding (Python struct format):

HEADER_FORMAT = '!4s B B I I Q H' # protocol_id, version, msg_type, snapshot_id, seq_num, timestamp, payload_len

4. Communication Procedures

4.1 Session Establishment

The protocol uses a stateless "Connect" mechanism to minimize overhead.

1. Client sends a CONNECT_REQUEST (Type 0x0A) or a raw string "CONNECT" to the server.
2. Server allocates a player_id (e.g., player_1) and responds with a MSG_WELCOME (Type 0x01) containing the Grid Size, initial Player Positions, and assigned ID.
3. Client receives MSG_WELCOME and initializes the game grid.

4.2 Normal Data Exchange (Ephemeral)

- **Server -> Client:** The server broadcasts MSG_GAME_STATE (**Type 0x02**) at a consistent frequency of 20Hz. To optimize bandwidth and stay well under the 1200-byte **MTU(Maximum Transmission Unit)** limit, **Delta Encoding** is used; the payload includes only the "dirty cells" (cells that changed ownership since the last broadcast). To correct any potential drift caused by UDP packet loss, player positions are sent as **absolute grid coordinates** in every snapshot, ensuring all clients converge on the server's authoritative state.
- **Client -> Server:** Clients transmit MSG_PLAYER_MOVE (**Type 0x06**) as soon as movement input is detected. These are categorized as **Class B (Ephemeral) traffic** and sent unreliably. If a packet is lost, it is not retransmitted because a newer position update will render the lost data obsolete within milliseconds, thus maintaining the lowest possible latency.

4.3 Critical Event Handling (Reliable)

When a player attempts to claim a cell, the protocol uses **Selective Reliability** via a **Stop-and-Wait ARQ** mechanism to ensure no actions are lost due to network interference:

1. **Request Phase:** The client transmits a MSG_ACQUIRE_REQUEST (**Type 0x03**) containing the target cell_id and a unique seq_num. This request is stored in the client's pending_requests queue.
2. **Client-Side Retry:** The client initiates a **100ms retry timer**. If a corresponding Acknowledgment (ACK) is not received from the server within this window, the client retransmits the request. This process repeats for a maximum of **10 attempts** before timing out.
3. **Acknowledgment Phase:** Upon receipt of the request, the server immediately validates the message and returns a MSG_ACK (**Type 0x07**). This specific packet signals the client to stop its retransmission timer for that sequence number.

4. **Authoritative Response:** After processing the game logic (checking cell ownership), the server transmits a MSG_ACQUIRE_RESPONSE (**Type 0x04**) to the client. To ensure the client UI updates correctly under lossy conditions, this response is also sent **reliably** (added to the server's retry queue), requiring a final ACK from the client to confirm delivery.

5. Reliability & Performance Features

5.1 Selective Reliability Strategy

We categorize traffic into two classes:

1. ClassA(Critical):ACQUIRE_REQUEST,ACQUIRE_RESPONSE, GAME_OVER.
    - Mechanism: Explicit ACK + Timeout Retransmission.
    - Rationale: Packet loss here results in game-breaking state desynchronization (e.g., a player thinks they own a cell they don't).
2. Class B (Ephemeral): GAME_STATE, PLAYER_MOVE, HEARTBEAT. Mechanism: Best-effort UDP (No Retransmission).
    - Rationale: Data is time-sensitive. A retransmitted movement packet arriving 200ms late is useless "jitter". We rely on the next update (50ms later) to correct state.

5.2 Timeout Calculation

- Target RTT: 50ms (Simulated WAN).
- Retry Interval: 100ms - 200ms.
- Justification: The retry interval is set to >2×RTT to prevent unnecessary retransmissions while ensuring recovery happens within human reaction time limits (<250ms perception threshold).

5.3 Bandwidth Optimization

- Binary Header: Fixed 24-byte header reduces overhead compared to text protocols.
- Delta Encoding: The MSG_GAME_STATE payload only contains dirty_cells (newly acquired cells). Static grid data is not re-sent.
- Payload Compression: All outgoing payloads are compressed using the **zlib** algorithm. This is particularly effective for JSON-serialized data, reducing total packet size by approximately 60%.
- Result: Average packet size is kept small (<400 bytes), well within the **MTU (Maximum Transmission Unit)** limit.

5.4 Design Justification and Comparative Analysis

- Unlike **MQTT-SN**, which is designed for periodic sensor data with a fixed publish/subscribe model, **GCBP v1** is optimized for high-frequency state synchronization. While **MQTT-SN** relies on a gateway and potentially heavy overhead for constrained devices, **GCBP** uses a stateless handshake and zlib compression to maintain a 20Hz update rate within a 1200-byte **MTU(Maximum Transmission Unit)**. Furthermore, GCBP implements a dual-class reliability system (Class A/B) which is not present in standard IoT protocols, ensuring that 'perishable' movement data does not cause head-of-line blocking.

6. Experimental Evaluation Plan

6.1 Testbed Setup

- OS: Linux (Ubuntu/WSL2)
- Tools: tc netem (Network Emulation), tcpdump (Capture), Python 3. Nodes: 1 Server, 4 Concurrent Clients (Localhost).

6.2 Scenarios & Parameters

- Baseline: No impairment. Expecting < 2ms latency, 0% loss.
- LAN Loss (2%): tc qdisc add dev lo root netem loss 2%. Verifies interpolation smoothing.
- WAN Loss (5%): tc qdisc add dev lo root netem loss 5%. Verifies Critical Event ARQ.
- High Latency: netem delay 100ms 10ms. Verifies Timestamp-based lag compensation.

6.3 Metrics Generation

- Latency: Calculated as Trec-Tserver_ts.
- Position Error: Euclidean distance between Server Authority Position (Ps) and Client Render Position (Pc). Error= (xs-xc)^2+(ys-yc)^2

7. Example Use Case Walkthrough

This section provides a wire-level analysis of a standard session using the GCBP v1 protocol. The trace was captured on the loopback interface (127.0.0.1) with four active clients.(example from baseline_run1)

7.1 Initial Handshake and Connection

The following trace shows the "Stateless Connect" sequence for the first client.

Packet #	Relative Time (s)	Source → Dest	Length (UDP)	Message Type (Inferred)	Description
1	0.0000	Client → Server	7	`CONNECT`	Raw string "CONNECT" sent to initiate.
2	0.0101	Server → Client	427	`MSG_WELCOME`	Server assigns `player_1` and sends initial grid.

Observation: The MSG_WELCOME packet is significantly larger than subsequent updates because it contains the full initial state of the 400-cell grid and player starting positions.

7.2 Critical Event Handling (Cell Acquisition)

This sequence demonstrates the **Selective Reliability** mechanism (Stop-and Wait ARQ) when a player attempts to claim a cell.

Packet #	Relative Time (s)	Source → Dest	Length (UDP)	Message Type	Description
4	0.0132	Client → Server	100	MSG_ACQUIRE_REQUEST	Player 1 requests cell 0_0.
5	0.0138	Server → Client	40	MSG_ACK	Server confirms receipt of request.
6	0.0142	Server → Client	83	MSG_ACQUIRE_RESPONSE	Server confirms claim success (Authoritative).

Observation: The Server issues an immediate MSG_ACK (Type 0x07) to stop the client's retransmission timer before processing the game logic. The actual result is then sent in a separate reliable ACQUIRE_RESPONSE.

7.3 Steady State: 20Hz Multi-Client Broadcast

Once four clients are connected, the server maintains synchronization by broadcasting snapshots.

Packet #	Relative Time (s)	Source → Dest	Length (UDP)	Message Type	Description
107	2.2555	Server → Client 1	382	`MSG_GAME_STATE`	Snapshot ID: 45 broadcast to C1
108	2.2555	Server → Client 2	382	`MSG_GAME_STATE`	Snapshot ID: 45 broadcast to C2
109	2.2557	Server → Client 3	382	`MSG_GAME_STATE`	Snapshot ID: 45 broadcast to C3
110	2.2557	Server → Client 4	382	`MSG_GAME_STATE`	Snapshot ID: 45 broadcast to C4
...	**~50ms later**				
111	2.3056	Server → Client 1	382	`MSG_GAME_STATE`	Snapshot ID: 46 broadcast to C1

Observation: The timestamps for Packets 107 and 111 show an interval of exactly **0.0501 seconds**, confirming the server successfully maintains the mandatory **20Hz update frequency**.

7.4 Delta Encoding and Compression

Compare the lengths of MSG_GAME_STATE at different stages:

- **Early Game (Packet 11):** Length = 382 bytes.
- **Mid Game (Packet 740):** Length = 384 bytes.
- **Late Game (Packet 1423):** Length = 386 bytes.

Analysis: As more cells are claimed, the "dirty cells" list in the Delta Update grows. However, due to **zlib compression** and **Delta Encoding**, the packet size remains extremely small (<400 bytes), well under the 1200-byte **MTU(Maximum Transmission Unit)** limit, ensuring efficiency even as game complexity increases.

APPENDIX (PCAP EXCERPT):

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E·
0010  00 6f 66 91 40 00 40 11  d5 ea 7f 00 00 01 7f 00   ·of·@·@· ........
0020  00 01 15 b3 a3 ae 00 5b  fe 6e 47 52 49 44 01 04   .......[ ·nGRID··
0030  00 00 00 00 30 00 00 00  79 3e 93 13 9b 01 00 00   ····0··· y>......
0040  3b 00 7b 22 63 65 6c 6c  5f 69 64 22 3a 20 22 30   ;·{"cell _id": "0
0050  5f 30 22 2c 20 22 73 75  63 63 65 73 73 22 3a 20   _0", "su ccess":
0060  74 72 75 65 2c 20 22 6f  77 6e 65 72 5f 69 64 22   true, "o wner_id"
0070  3a 20 22 70 6c 61 79 65  72 5f 31 22 7d            : "playe r_1"}
```

Hex Value	Protocol Field	Meaning
47 52 49 44	protocol_id	ASCII for **"GRID"**
01	version	Protocol Version 1
04	msg_type	x04 (**MSG_ACQUIRE_RESPONSE**)
00 00 00 00	snapshot_id	Snapshot ID 0
30 00 00 00	seq_num	Sequence #48 (Little Endian)

3b 00	payload_len	59 bytes of data follows
7b 22 63 65...	**Payload**	JSON string: {"cell_id": "0_0"...}