Mini-RFC: GridClash Binary Protocol (GCBP v1)

Submitted to:

Dr. Karim Ahmed

Eng. Noha Wahdan

Submitted by :

Loaiy Mahmoud 23P0419

Omar Ayman23P0244

Ali Moataz 23P0280

Omar Osama 2300090

Jana Tamer 23P0173

Jana Mohamed 2301032

Mini-RFC: GridClash Binary Protocol (GCBP v1)

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.

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
+---------+ CONNECT_REQUEST +-----------+ | Client | ---------------------------> | Server | | | <---- WELCOME -------------| | | | -----> ACQUIRE_REQUEST ----->| | | | <------ GAME_STATE ----------| | | |
<------ GAME_OVER -----------| | +---------+ +-----------+
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.

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
checksum	4 (optional)	CRC32 for message integrity

Message Types:

Type	Code	Direction	Purpose
CONNECT_REQUEST	0x01	Client→Server	Initiate handshake
WELCOME	0x02	Server→Client	Confirm connection, assign ID
ACQUIRE_REQUEST	0x03	Client→Server	Request to claim a cell
ACQUIRE_RESPONSE	0x04	Server→Client	Grant/deny cell acquisition
PLAYER_MOVE	0x05	Client→Server	Update player position
GAME_STATE	0x06	Server→All	Broadcast grid snapshot
GAME_OVER	0x07	Server→All	End-of-game notice and results
ACK	0x08	Both	Confirm receipt of critical packet
HEARTBEAT	0x0A	Both	Liveness indicator

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 20Hz. To save bandwidth, **Delta Encoding** is used: only cells that have changed status (Claimed) since the last tick are included in the payload. Player positions are sent as absolute coordinates to correct drift.
- **Client -> Server:** Clients send MSG_PLAYER_MOVE (Type 0x06) whenever input is detected. These are sent unreliably; if a packet is lost, the next movement packet (sent ms later) will update the position.

4.3 Critical Event Handling (Reliable)

When a player attempts to claim a cell, **Selective Reliability** (Stop-and-Wait ARQ) is used:

1. **Client** sends MSG_ACQUIRE_REQUEST (Type 0x03) with a specific seq_num. It adds this request to a pending_requests queue.
2. **Client** starts a retry timer (100ms). If no ACK is received, it retransmits up to 10 times.
3. **Server** receives the request, processes the logic,
   and **immediately** sends MSG_ACK (Type 0x07) to stop client retransmission.
4. **Server** sends MSG_ACQUIRE_RESPONSE (Type 0x04) indicating success/failure. This message is also added to the Server's retry queue to ensure the client UI updates correctly.

5. Reliability & Performance Features

5.1 Selective Reliability Strategy

We categorize traffic into two classes:

1. **Class A (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.
- **Result:** Average packet size is kept small (< 400 bytes), well within the MTU limit.

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

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

6.3 Metrics Generation

- **Latency:** Calculated as $T_{rec} - T_{server\_ts}$

.

- **Position Error:** Euclidean distance between Server Authority Position (Ps) and Client Render Position (Pc).
  - $\text{Error} = \sqrt{(x_s - x_c)^2 + (y_s - y_c)^2}$

7. Limitations & Future Work

1. **Scalability:** The current server implementation uses Python threads. For >50 clients, asyncio or C++ would be required to avoid GIL contention.
2. **Security:** No encryption or cheat prevention (e.g., speed hacking validation) is implemented.
3. **Clock Sync:** We assume relatively synchronized clocks for latency calculation. A proper NTP-like offset calculation (Cristian's algorithm) would improve metric accuracy on separate machines.