GridClash

A Lightweight UDP Protocol for Real-Time Multiplayer Synchronization

Yousif Salah Mohammed
Student ID: 22P0232

Philopater Guirgis
Student ID: 22P0250

Hams Hassan
Student ID: 22P0253

Ahmed Lotfy
Student ID: 22P0251

Noha Elsayed
Student ID: 2201431

Adham Kandil
Student ID: 22P0237

Supervisor: Ayman Bahaa
Professor, Ain Shams University

Co-supervisor: Mina Fadi
Teaching Assistant, Ain Shams University

Faculty of Engineering
Computer Engineering & Software Systems
CSE351: Computer Networking

25, December 2025

Contents

List of Figures

List of Tables

1 Introduction

This document specifies the GridClash Update Protocol (GCUP), a lightweight UDP-based protocol for real-time multiplayer synchronization in the GridClash game, where players compete to claim cells on a shared grid.

1.1 Motivation and Use Case

Existing protocols like HTTP/TCP are unsuitable for real-time gaming due to head-of-line blocking and retransmission latency. General-purpose UDP libraries often lack the specific application-aware logic needed for this domain. GCUP is designed specifically for:

- **Low Latency:** Prioritizing fresh state over guaranteed delivery of outdated data.
- **High Interactivity:** Evaluating atomic cell acquisition requests with minimal overhead.
- **Bandwidth Efficiency:** Using delta compression to support frequent updates (20Hz) on constrained links.

1.2 Design Goals and Constraints

GCUP targets low-latency (≤ 50 ms) state synchronization for 4+ concurrent clients with minimal CPU overhead ($< 60\%$). The protocol uses selective reliability (ACK/NACK for critical events), delta encoding for bandwidth efficiency, and CRC32 checksums for data integrity over unreliable UDP transport.

Table 1.1: *Protocol Constraints*

Parameter	Value	Description
Max Packet Size	1200 bytes	MTU compatibility
Target Latency	≤ 50 ms	End-to-end latency
Concurrent Clients	4	Minimum supported
Update Frequency	20 Hz	Server broadcast rate
Heartbeat Timeout	10 s	Disconnect threshold

Protocol ID: GCUP (`0x47435550`). All multi-byte fields use big-endian encoding.

2

Protocol Architecture

GCUP uses a client-server model over UDP. The server maintains authoritative game state (grid ownership, player positions, scores) and broadcasts snapshots at 20 Hz. Clients render received state, send heartbeats (1 Hz), and issue acquire requests with retransmission.

2.1 Entities

The protocol defines two distinct entities:

1. **Game Server:** The authoritative source of truth. It maintains the global state (grid cells, player scores), resolves conflicting client actions (e.g., simultaneous claims), and creates periodic snapshots.
2. **Game Client:** A thin client responsible for rendering the game state, collecting user input, and predicting local state changes (optimistic UI) while awaiting server confirmation.

2.2 Communication Flow

1. **Handshake:** Client sends `CLIENT_INIT`. Server responds with `SERVER_INIT_RESPONSE` (player ID, position) or `SERVER_FULL` if at capacity.
2. **State Sync:** Server broadcasts `SNAPSHOT` (grid + player data with delta encoding) at 20 Hz. Clients discard outdated/duplicate snapshots by `snapshot_id`.
3. **Actions:** Client sends `ACQUIRE_REQUEST` for cell claims. Server responds with `ACK/NACK`. Clients retransmit on timeout with exponential backoff.
4. **Heartbeat:** Clients send `HEARTBEAT` every 1s. Server disconnects clients after 10s timeout.
5. **Termination:** Server broadcasts `GAME_OVER` when all cells claimed. Clients may send `NEW_GAME` to restart.

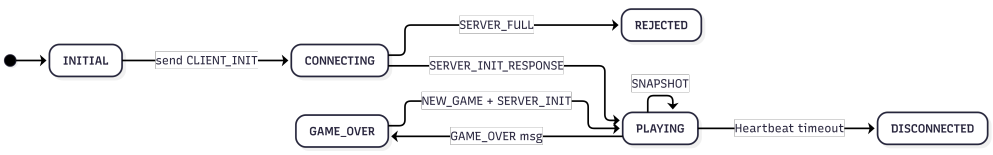

Figure 2.1: *Client FSM*

Figure 2.2: *Server FSM*

3 Message Formats

3.1 Header Format

Table 3.1: *Header Format (28 bytes, `!4sBBIIQHI`)*

Field	Size	Description
Protocol ID	4B	`"GCUP"` (`0x47435550`)
Version	1B	Protocol version (current: 1)
Message Type	1B	See Table 3.2
Snapshot ID	4B	Game state version (0 for non-snapshots)
Sequence Number	4B	Per-client packet sequence
Server Timestamp	8B	Milliseconds since epoch
Payload Length	2B	Payload size in bytes
Checksum	4B	CRC32 over header (field=0) + payload

Table 3.2: *Message Type Values*

Val	Name	Dir	Description
0	SNAPSHOT	S→C	Periodic state broadcast
1	HEARTBEAT	C→S	Keep-alive
2	CLIENT_INIT	C→S	Connection request
3	SERVER_INIT_RESPONSE	S→C	Connection confirm
4	ACQUIRE_REQUEST	C→S	Cell claim
5	GAME_OVER	S→C	Game end
6	NEW_GAME	C→S	Restart request
7	SERVER_FULL	S→C	Reject (full)
8	ACK	S→C	Positive ack
9	NACK	S→C	Negative ack

Table 3.3: *Payload Specifications*

Message	Format	Fields
CLIENT_INIT	`!B`	Client ID (placeholder)
SERVER_INIT_RESPONSE	`!Bii`	Player ID, Start X, Start Y
HEARTBEAT	`!B`	Client ID
ACQUIRE_REQUEST	`!BBQ`	Row, Column, Timestamp (ms)
ACK/NACK	`!I?`	Seq num, Success flag
GAME_OVER	`!BH`	Winner ID, Winner score
SNAPSHOT	`400s B + !BHiiii×N`	Grid (400B), Player count, Per-player: ID, Score, X, Y, dX, dY (17B each)

4 Communication Procedures

4.1 Session Start

1. Client sends `CLIENT_INIT` to the server (port 12000).
2. Server responds with `SERVER_INIT_RESPONSE` containing:
   - Player ID (0–3)
   - Spawn position

   if capacity allows.
3. If the server is at maximum capacity (4 clients), it responds with `SERVER_FULL`.
4. Upon receiving `SERVER_INIT_RESPONSE`, the client stores its assigned ID and begins receiving snapshots.

4.2 State Synchronization

1. **Server:**
   (a) Broadcasts `SNAPSHOT` every 50 ms (20 Hz).
   (b) Increments `snapshot_id`.
   (c) Serializes grid state (400B).
   (d) Calculates player deltas.
   (e) Computes CRC32 checksum.
   (f) Sends snapshot to all connected clients.
2. **Client:**
   (a) Validates checksum.
   (b) Discards snapshot if $\text{snapshot\_id}_{\text{recv}} < \text{snapshot\_id}_{\text{last}}$ (stale) or if `seq_num` is a duplicate.
   (c) Updates grid state, player positions, and scores.
   (d) Calculates latency as recv_time − server_timestamp.

4.3 Cell Acquisition

1. **Client:**
   (a) Validates the requested move.
   (b) Sends `ACQUIRE_REQUEST` containing row, column, and timestamp.
   (c) Starts RTO timer.
   (d) Queues the request until resolution.
2. **Server:**
   (a) Checks for duplicate `seq_num`.
   (b) Re-sends `ACK` if the request is a duplicate.
   (c) For new requests:
      i. Sends `ACK` if the cell is unclaimed or already owned by the requester.
      ii. Sends `NACK` if the cell is owned by another player.
   (d) Marks `seq_num` as processed.
3. **Client (Response Handling):**
   (a) On `ACK`, cancels the timer and updates RTT using: $\text{RTT} = 0.875 \times \text{RTT} + 0.125 \times \text{sample}$.
   (b) On `NACK`, retries the request immediately.

4.4 Connection Maintenance

Clients send `HEARTBEAT` every 1s. Server disconnects clients after 10s timeout.

4.5 Retransmission

RTO calculated as: $\text{RTO} = \text{RTT} + 4 \times \text{RTT\_dev}$ (min 100 ms, max 2000 ms). Exponential backoff: $\text{RTO} \times 2^{\text{retry\_count}}$. Max 3 retries before giving up.

4.6 Game Termination

Server broadcasts `GAME_OVER` when all 400 cells claimed or winner threshold reached. Clients may send `NEW_GAME` to reset.

5 Reliability & Performance

5.1 Selective Reliability

GCUP uses **selective reliability**: snapshots tolerate loss (newer data obsoletes old), while critical events (`ACQUIRE_REQUEST`) use ACK/NACK with retransmission. The server maintains per-client processed `seq_num` sets for duplicate suppression and idempotent ACK re-sends.

5.2 Adaptive Retransmission

RTT estimation follows TCP-style EWMA:

$$\text{RTT} = 0.875\,\text{RTT}_{\text{old}} + 0.125\,\text{RTT}_{\text{sample}}, \quad \text{RTT\_dev} = 0.75\,\text{RTT\_dev}_{\text{old}} + 0.25\,|\text{RTT}_{\text{sample}} - \text{RTT}|$$

RTO is computed as $\text{RTO} = \text{RTT} + 4 \times \text{RTT\_dev}$ (min 100 ms, max 2000 ms), with exponential backoff $\text{RTO} \times 2^{\text{retry\_count}}$ and a maximum of three retries.

5.3 Delta Encoding

Each snapshot carries absolute positions (X, Y) and deltas (dX, dY) from the previous snapshot, enabling single-packet loss recovery: $\text{pos}_{\text{recovered}} = \text{pos}_{\text{current}} - \text{delta}$. This adds 8 bytes per player in exchange for zero-latency recovery.

5.4 Stale Packet Rejection

Clients discard packets when $\text{snapshot\_id}_{\text{recv}} < \text{snapshot\_id}_{\text{last}}$ or when a duplicate `seq_num` is detected, ensuring monotonic state progression.

5.5 Visual Smoothing

Cursor motion is smoothed using linear interpolation: `lerp_factor = min(10.0 * dt, 1.0)`, reducing jitter while preserving responsiveness.

5.6 Data Integrity & Performance

- **CRC32:** Computed over header (checksum field set to zero) and payload using `binascii.crc32()`; corrupted packets are discarded.
- **Non-blocking I/O:** Enables single-threaded multi-client handling. A 1 ms sleep reduces CPU usage from 100% to below 10% while sustaining 20 Hz updates.
- **Memory bounds:** Latency tracking uses `deque(maxlen=1000)` to prevent unbounded growth.

6

Experimental Evaluation Plan

This chapter details the methodology for validating the GCUP protocol under various network conditions.

6.1 Test Environment

All tests are conducted on a standard Linux environment using `netem` for network emulation. The testbed supports concurrent execution of one server and four clients on a single machine, with traffic shaping applied to the loopback interface.

6.2 Test Scenarios

We define four core scenarios to evaluate performance against the acceptance criteria.

1. **Baseline (No Impairment):** Validates core functionality and resource usage.
   - *Criteria:* 20 updates/sec, latency $\leq$ 50ms, CPU $<$ 60%.
   - *Command:* None (default loopback).
2. **Packet Loss (LAN-like):** Simulates minor interference.
   - *Criteria:* Mean position error $\leq$ 0.5 units.
   - *Command:* `sudo tc qdisc add dev lo root netem loss 2%`
3. **Packet Loss (WAN-like):** Simulates poor connection quality.
   - *Criteria:* Critical events (cell acquisition) must eventually succeed (99% reliability).
   - *Command:* `sudo tc qdisc add dev lo root netem loss 5%`
4. **High Latency:** Simulates significant round-trip delay.
   - *Criteria:* System stability without disconnects.
   - *Command:* `sudo tc qdisc add dev lo root netem delay 100ms`

6.3 Metrics

Data is collected via client/server CSV logs and analyzed for:

- **Latency:** End-to-end delay ($T_{recv} - T_{sent}$).
- **Jitter:** Variance in inter-arrival times.
- **Perceived Error:** Euclidean distance between client displayed position and authoritative server position.
- **Convergence Time:** Time for all clients to reflect a state change.

7

Example Use Case Walkthrough

This section traces a critical game event: a client acquiring a cell, illustrating the protocol's handling of packet loss and reliability.

7.1 Scenario: Cell Acquisition with Loss

Context: Client 1 attempts to acquire cell $(5, 5)$ at time $t = 1000$ ms. The first request packet is lost due to simulated network failure.

1. **Initial Request ($T = 0$ms):** Client 1 sends `ACQUIRE_REQUEST` (Seq: 101, Cell: 5,5). It sets an RTO timer for 100 ms.
2. **Packet Loss:** The network dropps the UDP packet. Server never receives Seq 101.
3. **Timeout ($T = 100$ms):** Client 1's RTO timer expires. It enters the retransmission phase.
   - Backoff: New RTO $\leftarrow 200$ ms.
   - Retry count increments to 1.
4. **Retransmission ($T = 101$ms):** Client 1 resends `ACQUIRE_REQUEST` (Seq: 101, Cell: 5,5).
5. **Server Processing ($T = 120$ms):** Server receives Seq 101.
   - Checks ownership: Cell (5,5) is UNCLAIMED.
   - Grants ownership to Client 1.
   - Sends `ACK` (Seq: 101, Success: True).
6. **Ack Receive ($T = 140$ms):** Client 1 receives `ACK`.
   - Cancels RTO timer.
   - Marks cell as owned locally (prediction confirmed).
7. **State Broadcast ($T = 150$ms):** Server includes the new cell state in the next `SNAPSHOT` broadcast to all clients, ensuring global consistency.

8 Limitations & Future Work

8.1 Scalability & Bandwidth Efficiency

- **Full-State Broadcast Inefficiency:** The "Full Snapshot" model broadcasts the entire grid state at 20Hz, causing linear bandwidth growth ($O(GridSize)$). Large grids exceed the 1200-byte MTU, leading to fragmentation.
  Future Work: Implement **Delta Compression** to send only cells changed since the last acknowledged snapshot.
- **Lack of Interest Management:** All data is broadcast to all clients, wasting bandwidth on off-screen entities.
  Future Work: Implement **Area of Interest (AoI)** filtering to only send updates for entities within the client's viewport.

8.2 Reliability & Congestion

- **Static Transmission Rate:** Fixed 20Hz updates and RTO do not adapt to congestion, potentially worsening packet loss.
  Future Work: Implement **Adaptive Congestion Control** to dynamically adjust tick rate and RTO based on RTT and loss.
- **MTU Constraints:** The strict 1200-byte limit lacks application-layer fragmentation, making large states unplayable.
  Future Work: Design a multi-packet format to split logical snapshots across multiple UDP datagrams with reconstruction logic.

8.3 Security & Authentication

- **Lack of Session Security:** Cleartext UDP allows spoofing and replay attacks due to missing cryptographic validation.
  Future Work: Implement a challenge-response handshake for Session Tokens or encapsulate traffic in **DTLS**.
- **Trust Model Weaknesses:** Minimal server-side validation allows physically impossible moves.
  Future Work: Implement **Server-Side Sanity Checking** to validate velocity vectors against maximum limits.

8.4 State Synchronization

- **Lack of Prediction Reconciliation:** Naive interpolation causes rubber-banding when packets are lost.
  Future Work: Implement **Server Reconciliation** to re-simulate unacknowledged inputs over authoritative state.

This page intentionally left blank.

References

Eggert, L., G. Fairhurst, and G. Shepherd (Mar. 2017). *UDP Usage Guidelines*. RFC 8085. URL: https://www.rfc-editor.org/rfc/rfc8085.html.

Fiedler, Glenn (2015). *Snapshot Compression*. Gaffer on Games. URL: https://gafferongames.com/post/snapshot_compression/.

Postel, J. (Aug. 1980). *User Datagram Protocol*. RFC 768. URL: https://www.rfc-editor.org/rfc/rfc768.html.

Python Codes on LMS (2025). *Application Servers Samples, p2p-chat, Quick Crypto*. LMS. URL: https://lms.eng.asu.edu.eg/mod/folder/view.php?id=180666.

Schulzrinne, H. et al. (July 2003). *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550. URL: https://www.rfc-editor.org/rfc/rfc3550.html.

Valve Developer Community (2005). *Source Multiplayer Networking*. URL: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking.

Appendices

A PCAP Trace

A.1 Appendix: PCAP Trace

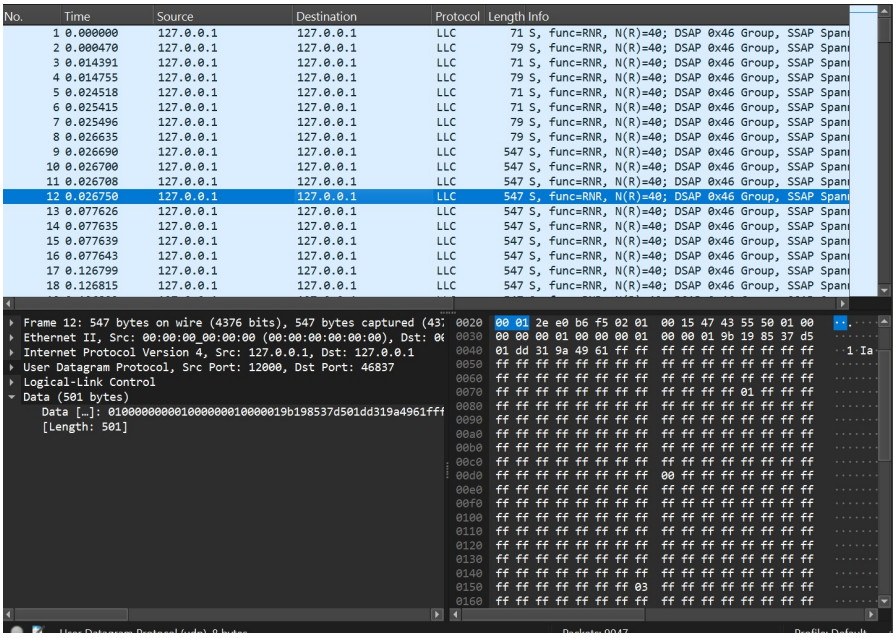

Figure A.1: *Wireshark capture showing UDP traffic with packet loss and retransmission.*

Protocol Configuration

Transport: UDP | **Protocol:** LLC/DSAP 0x46 | **Server:** 127.0.0.1:12000 | **Client Port:** 46837

Key Observations

1. **Session Setup:** Client initiates with a `hello`; server replies with client ID and position.
2. **Packet Types:**
   - Small (71–79 bytes): Control, ACK, and NACK messages
   - Large (547 bytes): Data and acquire responses
   - All frames use `func=RNR`, `N(R)=40` for flow control
3. **Communication Pattern:** Client acquire requests receive either ACKs (success) or NACKs (rejection) from the server.
4. **Reliability:** UDP with application-layer loss detection and request retransmission.
5. **Timing:**
   - First 8 packets exchanged within 27 ms
   - Burst at 26 ms (packets 9–12)
   - 51 ms gap after packet 12 suggests processing or timeout delay
6. **Frame 12:** Server sends a 547-byte packet with 501 bytes of payload, dominated by `0xff` patterns (padding or markers).