



# Grid Clash Synchronization Protocol (GCSP)

## Developed By:

Yosuf Mohamed 22P0241  
Basmala Khaled 22P0270  
Fatma Saleh 22P0264  
Kirollous Ramzy 22P0194

## Contents

1	Introduction.....	1
1.1	Purpose of the Protocol .....	2
1.2	Use Case Summary .....	2
1.3	Assumptions .....	3
1.4	System Constraints .....	3
2	Protocol Architecture.....	4
2.1	Entities & Roles .....	<b>Error! Bookmark not defined.</b>
2.2	System Diagram .....	<b>Error! Bookmark not defined.</b>
2.3	Finite-State Machine.....	5
2.3.1	Client FSM.....	10
2.3.2	Server FSM .....	10
2.4	Sequence Flow Diagram .....	10
3	Message Formats .....	10
3.1	GCSP Header .....	10
3.2	EVENT Message Format.....	10
3.3	ACK Format.....	10
3.4	SNAPSHOT Format.....	10
4	Communication Procedures .....	10
4.1	Session Start Procedure .....	10
4.2	Event Transmission Procedure .....	10
4.3	Snapshot Synchronization Procedure.....	10
4.4	Error Handling Procedure.....	10
4.5	Shutdown Procedure .....	10
5	Reliability & Performance Features.....	10
5.1	Retransmission Timing Model .....	10
5.2	Probability of Event Delivery.....	10
5.3	Snapshot Redundancy.....	10
6	Experimental Evaluation Plan .....	10
6.1	Baseline .....	11
6.2	Impaired Conditions .....	11
6.3	Metrics Computed.....	11
7	Example Use Case Walkthrough.....	11

7.1	Annotated Message Trace.....	11
7.2	State Evolution Explanation .....	11

# 1 Introduction

The **Grid Clash Synchronization Protocol (GCSP)** is a lightweight, UDP-based communication protocol designed to support *real-time distributed interactive systems* operating under uncertainty, packet loss, and variable network delay. Similar to clinical monitoring systems that must deliver timely, reliable physiological data despite noise and intermittent sensor dropout, GCSP is engineered to maintain **state consistency, low latency, and resilience** during multi-user interactions on a shared grid-based environment.

GCSP enables multiple clients to acquire cells, receive authoritative snapshots, and maintain a consistent replicated state—without relying on heavy, connection-oriented transports such as TCP. The protocol explicitly targets scenarios where the timeliness of updates is more critical than perfect reliability, and where modest packet loss is acceptable. As in medical telemetry, **delayed information may become clinically irrelevant**, thus GCSP prioritizes *freshness* of data over full reliability, using redundancy and event-level acknowledgments to ensure essential operations succeed even under adverse conditions.

## 1.1 Purpose of the Protocol

Traditional transport mechanisms (e.g., TCP) introduce head-of-line blocking, retransmission delays, and strict ordering guarantees that degrade responsiveness during real-time interaction. Such characteristics are incompatible with fast-paced, multi-client simulations. GCSP introduces a custom reliability model optimized for the following needs:

- **Real-time responsiveness** — Clients must see state updates within milliseconds, not seconds.
- **Partial reliability** — Only critical messages (EVENT actions) require reliable delivery.
- **High-frequency state dissemination** — The server must broadcast snapshots at a fixed tick rate (20 Hz).
- **Loss tolerance** — The simulation should remain functional even with 5–10% packet loss.
- **Low overhead** — Header format and packet structure must remain compact to avoid exceeding MTU or increasing congestion.

GCSP is therefore not intended to replace TCP or RTP, but rather to fill a specific design niche comparable to “clinical real-time telemetry”: high-frequency monitoring where information is perishable and the latest values matter most.

## 1.2 Use Case Summary

The target application of GCSP is a *competitive grid acquisition game* in which each client attempts to claim cells on a 20×20 grid. The server maintains the authoritative state, processes client actions, and broadcasts snapshot updates at a stable 20 Hz refresh rate. Key requirements include:

- **Authoritative consistency:** Only the server decides final cell ownership.
- **Fast conflict resolution:** When multiple players claim the same cell, the first valid event wins.
- **Continuous state synchronization:** Clients always receive up-to-date grid states.
- **Measurement capability:** The protocol must support detailed performance logging (latency, jitter, retries).

This mirrors medical systems where multiple monitoring devices report readings, but only a central controller aggregates, validates, and synchronizes state across the network.

## 1.3 Assumptions

GCSP operates under a realistic set of conditions reflected in typical LAN and WLAN environments:

1. **Maximum packet size ≤ 1500 bytes**, respecting standard Ethernet MTU.
2. **Packet loss may reach 5–10%**, especially on congested wireless links.
3. **No global clock synchronization**; client timestamps are used relative to server receipt.
4. **Network delay may fluctuate ±20 ms** under minor load, higher under congestion.
5. **Clients may disconnect abruptly** without sending a shutdown message.
6. **Reordering is possible**, since UDP provides no ordering guarantees.

These assumptions justify design decisions such as redundant snapshot broadcasting and idempotent event handling.

## 1.4 System Constraints

GCSP must adhere to the following constraints:

- **Uses UDP exclusively.**  
The protocol cannot depend on stream semantics or guaranteed delivery.
- **Server tick frequency fixed at 20 Hz.**  
Higher frequencies risk bandwidth congestion; lower rates degrade responsiveness.
- **Limited per-message payload size.**  
Header must remain compact; payloads must avoid fragmentation.
- **Clients cannot depend on persistent state.**  
They must re-join gracefully after disconnects.
- **Server must scale to multiple clients without degrading tick rate.**  
CPU time for snapshot generation < 50 ms per interval.

## 2 Protocol Architecture

The Grid Clash Synchronization Protocol (GCSP) follows a **central-authority distributed architecture**

### 2.1.1 Architectural Entities

#### 2.1.2 Server

The server corresponds to a *central medical supervisor*, maintaining the authoritative state of the environment and providing consistent updates to all clients.

- **Server Responsibilities**
- Maintain the global grid ( $20 \times 20$ ) representing owned/unowned cells.
- Assign unique player identifiers upon client joining.
- Validate and apply EVENT messages (e.g., ACQUIRE\_REQUEST).
- Ensure **idempotent** processing of events to avoid double-application.
- Broadcast authoritative snapshots at a fixed rate (20 Hz).
- Send ACK responses to confirm event receipt.
- Detect and remove inactive clients using timeout-based heartbeats.
- Log all relevant metrics (latency, jitter, snapshot count) for evaluation.

#### 2.1.3 Clients

Clients act like remote medical sensors in a clinical network—collecting user actions, sending them reliably, and receiving continuous updates from the server.

- **Client Responsibilities**
- Initiate a JOIN handshake and obtain a player ID.
- Maintain local replicated game state based on snapshots.
- Transmit EVENT messages (cell acquisition attempts) reliably.
- Handle retries for EVENT messages until acknowledged.
- Maintain pending-events queue.
- Log latency and jitter for performance evaluation.

## 2.2 Communication Model

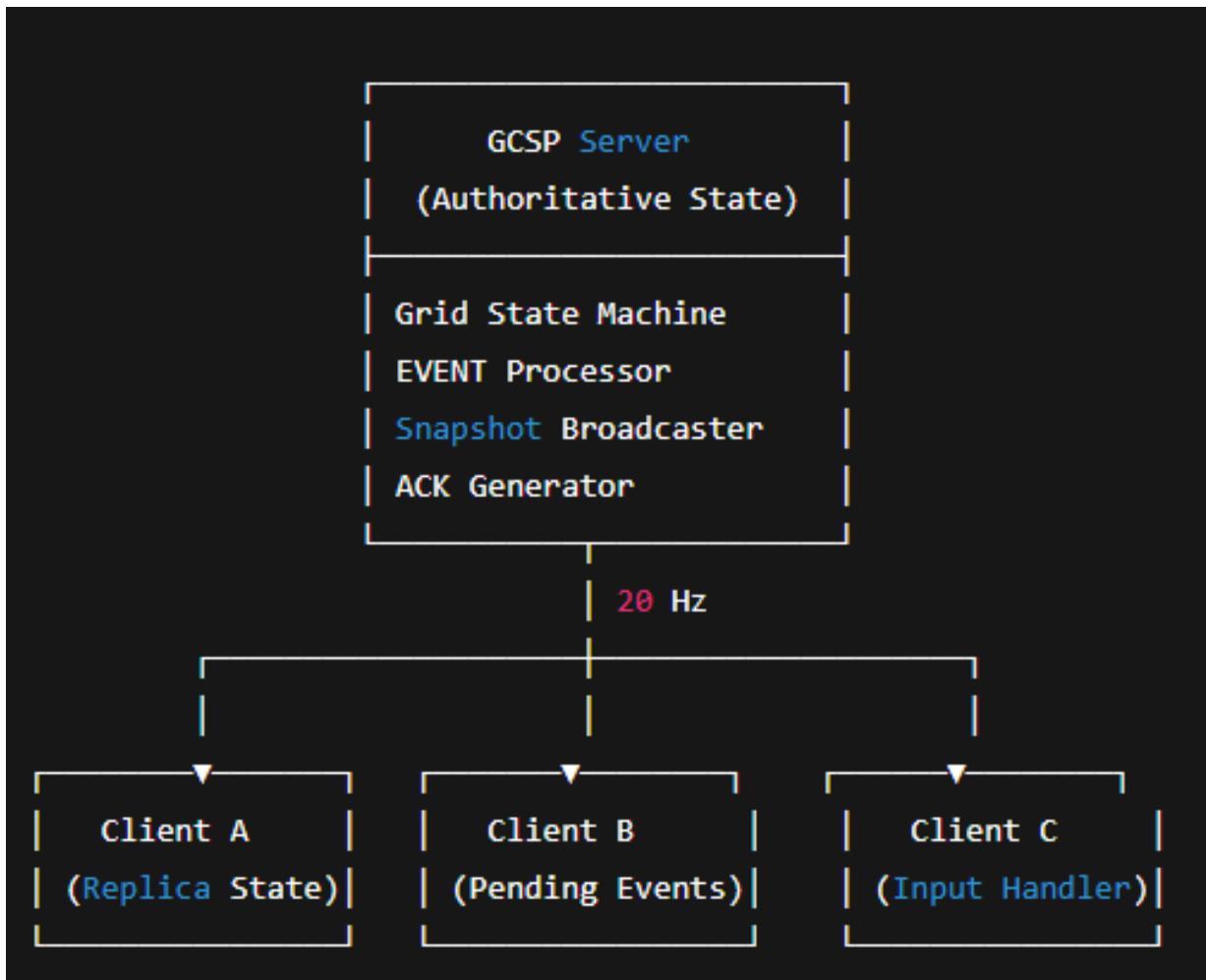
GCSP uses **UDP**, providing:

- Connectionless transport
- No delivery guarantees
- No ordering guarantees
- No retransmission at the transport layer

Therefore, the protocol enforces **application-layer reliability** only where needed (EVENT messages).

SNAPSHOT messages remain **fire-and-forget**, providing “freshness-first” information flow.

## 2.3 System Overview Diagram



## 2.4 Protocol Flow Overview

GCSP operates through three concurrent flows:

## 1. Session Establishment Flow

Client requests to join → server responds with JOIN\_ACK.

## 2. Reliable Event Flow

## **CLIENT → EVENT → SERVER**

**SERVER → EVENT\_ACK → CLIENT**

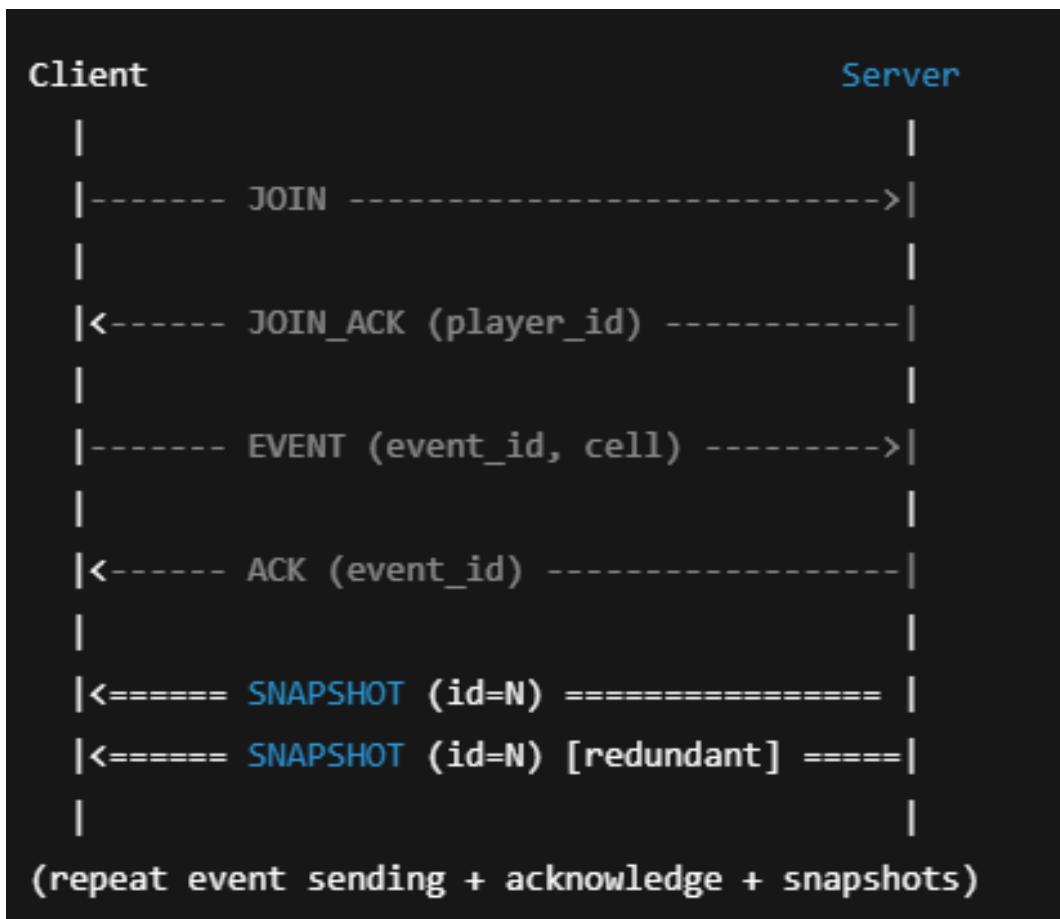
Retries occur until EVENT\_ACK is received.

### 3. Snapshot Synchronization Flow

## SERVER → SNAPSHOT → CLIENTS

Broadcast every 50 ms (20 Hz), sent twice for redundancy.

## 2.5 Detailed Sequence Diagram



## 2.6 Finite-State Machine

### 2.6.1 Client

#### **State 0 — DISCONNECTED**

- No communication established.
- Waiting for user input to start session.

#### **State 1 — JOIN\_SENT**

- Client sends JOIN request.
- Awaiting JOIN\_ACK.
- Timeout → resend JOIN.

#### **State 2 — ACTIVE**

- Client has player ID.
- Maintains snapshot-replicated state.
- Sends EVENT messages when user interacts.
- Retries pending events until ACKed.

#### **State 3 — WAITING\_FOR\_ACK**

- After sending an EVENT.
- If timeout: resend EVENT unless retries exhausted.

#### **State 4 — TERMINATED**

- User quits, or server timeout removes client.
- Cleanup occurs.

## 2.6.2 Server

### State 0 — IDLE

- Server running with no connected clients.

### State 1 — ACTIVE

- Clients joined.
- Receiving EVENT messages.
- Broadcasting SNAPSHOT packets every 50 ms.

### State 2 — TIMEOUT\_PROCESSING

- Detects clients with no communication for 5 seconds.
- Removes them safely.

### State 3 — SHUTDOWN

- Server terminates by operator.

## 2.7 Thread Architecture

The server uses **three concurrent threads**, analogous to separate clinical monitoring subsystems:

### 1. Receiver Thread

- Reads JOIN and EVENT messages
- Applies game logic
- Sends ACKs

### 2. Snapshot Broadcaster Thread

- Broadcasts snapshots at fixed frequency
- Implements redundancy (2 sends)

### 3. Client Cleanup Thread

- Removes inactive clients
- Preserves player\_id history for reconnection



2.7.1 Client FSM

2.7.2 Server FSM

## 2.8 Sequence Flow Diagram

# 3 Message Formats

3.1 GCSP Header

3.2 EVENT Message Format

3.3 ACK Format

3.4 SNAPSHOT Format

# 4 Communication Procedures

4.1 Session Start Procedure

4.2 Event Transmission Procedure

4.3 Snapshot Synchronization Procedure

4.4 Error Handling Procedure

4.5 Shutdown Procedure

# 5 Reliability & Performance Features

5.1 Retransmission Timing Model

5.2 Probability of Event Delivery

5.3 Snapshot Redundancy

# 6 Experimental Evaluation Plan

6.1 Baseline

6.2 Impaired Conditions

6.3 Metrics Computed

## 7 Example Use Case Walkthrough

7.1 Annotated Message Trace

7.2 State Evolution Explanation