CSE361 (UG2023) – Computer Networking

Semester (Fall 2025)

GAME SYNCHRONIZATION PROJECT REPORT Phase 1

Ahmed Wail	23P0335
Omar Shaker	23P0188
Youssef George	23P0227
Youssef Ahmed	23P0054
Zeyad Ahmed	23P0082

1.Projcet Proposal:

Assigned Scenario

In this project, we design and implement a custom network protocol to synchronize player positions and short-lived game events in a simplified multiplayer environment. The protocol must maintain low-latency synchronization between clients and a central server while tolerating moderate network loss and delay. The focus is on the network behavior and message exchange rather than building a full game, so a minimal 2D environment such as a grid-based system is sufficient.

In our implementation, this environment is represented by the Grid Clash game, where multiple players compete to claim cells on a shared grid. The server maintains the authoritative state and broadcasts periodic updates to all clients to keep their local views consistent.

Short Motivation

Real-time multiplayer games need quick and continuous communication between players and the server. Even small delays can cause desynchronization or lag. Using TCP in such systems often leads to noticeable latency because it enforces strict ordering and retransmits lost packets, which can slow gameplay.

To solve this, the implemented Violet Ascending Protocol (VAP-1) is built on UDP. UDP allows messages to be sent instantly without waiting for acknowledgments, keeping latency low. Although UDP does not guarantee delivery, the server and client code handle this at the application level through redundant state updates and simple acknowledgment messages.

This approach gives smoother, more responsive gameplay by prioritizing timely updates over perfect reliability. The result is a lightweight, real-time synchronization protocol suitable for fast-paced multiplayer scenarios where consistency and reaction time matter more than guaranteed delivery.

Proposed Protocol Approach

The Violet Ascending Protocol (VAP-1) uses a client–server architecture. The server acts as the authoritative source of truth. It listens for JOIN requests from clients, assigns them IDs, receives READY signals, and transitions through different states like WAITING_FOR_JOIN, WAITING_FOR_INIT, GAME_LOOP, and GAME_OVER.

Once the game starts, it sends snapshot messages containing the grid state, player scores, and timestamps.

Each client connects using UDP sockets, sends its join and ready messages, and waits for snapshot updates. When a snapshot arrives, it updates the client’s local grid to reflect the server’s authoritative view.

Every packet sent in the system follows the structure defined in the header file, which uses the format: !4s B B I I d H which uses big-endian format. The 4s corresponds to an array of char which is a string, the B corresponds to an unsigned char, the I corresponds to an unsigned int, the d corresponds to a double, and the H corresponds to an unsigned short, which totals up to a header size of 24 bytes.

This includes the following fields:
protocol_id (for example “VAP1”) (4 Bytes), version (1 Byte), msg_type (1 Byte), snapshot_id (4 Bytes), seq_num (4 Bytes), timestamp (8 Bytes), and payload_len (2 Bytes).

The payload, when present, is JSON-encoded and carries game-related information such as player actions, readiness, or current grid data.

Each broadcast the server sends can be either a full snapshot, containing the complete game state, or a delta snapshot, which only includes changes since the last update. To improve reliability, the server keeps a small history of the last two snapshots (K = 2), so that if a client misses a packet, it can still rebuild the latest state from the next snapshot.

The implementation uses non-blocking UDP sockets with the select module to efficiently check for new messages instead of relying on asyncio.

When the game ends, the server sends a MSG_LEADERBOARD message to all connected players showing their final ranks and scores before resetting for the next round.

This protocol balances speed, simplicity, and reliability. It provides a foundation for lowlatency, loss-tolerant multiplayer synchronization and demonstrates how real-time state management can be implemented efficiently using UDP and non-blocking communication.