# README — Multiplayer Game State Synchronization

## Dependencies

• Python 3.9 or higher
• matplotlib (for plotting performance graphs)
All other modules used are part of the Python standard library.
To install matplotlib once before running:
pip install matplotlib

## Overview

This project implements a simple UDP-based protocol to synchronize the state of a small multiplayer game between a central server and multiple clients.
The goal is to keep all players' game views consistent with minimal delay, even if some packets are lost or arrive late.
The focus is on protocol behavior and timing, not on graphics or full gameplay. The implemented protocol, called Violet Ascending Protocol (VAP-1), manages player joins, readiness messages, and periodic state updates (snapshots) sent from the server to clients.
Each update sent by the server can be either a full snapshot (complete grid state) or a delta snapshot (only changes since the last one). This allows faster and more efficient synchronization.

## Build Instructions Requirements:

• Python 3.9 or newer
• matplotlib (used only for plotting collected metrics)
• Standard Python libraries: socket, json, struct, time, enum, dataclasses, csv No build or compilation is required.
Place the following files in the same directory: server.py, client.py, header.py, run_baseline.sh, collect_metrics.py, plot_metrics.py.

## Run Instructions

## How to Run Tests

To run the baseline local test scenario:
bash run_baseline.sh
This script automatically:
• Starts the server in the background
• Launches four client instances concurrently
• Records all message exchanges in log files • Stops the server and prints short summaries Expected console output:
Player joined

Player ready
Initial snapshot sent
Snapshot received
After the test completes, you will find:
• server_log.txt and client log files (client1_log.txt … client4_log.txt)
• metrics.csv (collected latency and jitter measurements)
• latency_timeseries.png and jitter_hist.png (graphs generated automatically)

## For manual Run:

1. Start the server: python server.py
   The server listens on UDP port 8888 and uses non-blocking UDP sockets with the select module to handle multiple clients efficiently.
2. Start clients (in separate terminals): python client.py
   Each client sends a JOIN and READY message to the server and then receives periodic state snapshots (full or delta).
3. Observe logs:
   The terminal will show messages such as “Player joined”, “Player ready”, and “Snapshot sent”.
   During automated tests, all output is saved in text log files for later analysis.

## Design Mechanisms

• UDP communication is used for fast, low-latency message delivery without TCP’s retransmission overhead.
• The server and clients communicate through regular non-blocking UDP sockets using the select module (no asyncio).
• Each message follows a fixed header format: !4s B B I I d H (24 bytes total).
• Payloads are JSON-encoded and carry player actions, readiness, or grid updates. • The server can send both full and delta snapshots depending on synchronization needs.
• Redundant snapshots (K = 2) are stored to help recover from packet loss.
• The server moves through defined states: WAITING_FOR_JOIN, WAITING_FOR_INIT, GAME_LOOP, and GAME_OVER.
• When the game ends, the server broadcasts a MSG_LEADERBOARD message to show final player scores before resetting for the next round.
• Logs are later processed by Python scripts to compute latency and jitter statistics.

## Baseline Scenario

The Phase 1 baseline test verifies that the basic INIT and DATA exchanges work correctly in a local environment using four concurrent clients.

This establishes a functional prototype before adding network impairments such as delay or packet loss in later phases (handled by Linux netem).

## Operational Notes

- Maximum packet size (header + payload) is 1200 bytes.
- Target average latency is below 50 ms.
- The system supports up to 4 local clients in this phase.
- All logs, metrics, and graphs are saved automatically in the working directory after each test.