Project 2: Multiplayer game state synchronization

CSE361 Computer Networks

Team 46

25/12/2025

Philopateer Mina 22P0250

Noha Elsayed 2201431

Hams Hassan 22P0253

Yousif Salah 22P0232

Ahmed Lotfy 22P0251

Adham Kandil 22P0217

Contents

1 Executive Summary

This report presents the design, implementation, and evaluation of **GridClash**, a custom UDP-based networking protocol developed to support a real-time, competitive multiplayer grid game. The protocol was designed to synchronize up to four concurrent clients within a **20×20 grid environment**, while maintaining a **baseline latency below 50 ms** and tolerating packet loss and network jitter typical of real-world conditions.

The primary objective of the project was to achieve **low-latency state synchronization** using an authoritative server architecture over UDP, without relying on TCP. To accomplish this, a custom binary application-layer protocol was implemented, combining **unreliable state snapshots** with **selective reliability mechanisms** for critical game events. Client-side prediction and interpolation techniques were also employed to mask network delays and improve perceived responsiveness.

Experimental evaluation demonstrated strong baseline performance, with an average end-to-end latency of approximately **0.4 ms** and server CPU utilization remaining below **6%**. Under adverse network conditions, including **5% packet loss** and **100 ms artificial latency**, the system remained playable and stable. Instrumented testing showed that the protocol sustained correct game behavior, with a **95th percentile position error of approximately 2.37 grid units** during loss scenarios, and full recovery upon receipt of subsequent snapshots. The application-layer ARQ mechanism successfully ensured reliable delivery of all critical events, even under WAN-like conditions.

Overall, the GridClash protocol meets all defined acceptance criteria for small-scale multiplayer environments. However, scalability analysis indicates that further optimization is required to efficiently support larger player counts, due to the linear growth of snapshot payload size.

2 Introduction

Real-time multiplayer games place heavy demands on network performance, especially when it comes to low latency, consistent game state updates, and the ability to handle unreliable networks. While traditional protocols like TCP offer reliability, they often introduce delays caused by congestion control and head-of-line blocking, which can negatively affect fast-paced gameplay. For this reason, many modern multiplayer games rely on UDP and implement their own reliability and synchronization logic at the application level. This project was motivated by the need for a low-latency, UDP-based synchronization protocol tailored to a competitive, grid-based multiplayer game.

The system was designed to support real-time interaction between multiple players while operating within a 1200-byte MTU and handling challenging network conditions such as packet loss and jitter. An authoritative server model was used to maintain fairness and prevent clients

from drifting into inconsistent game states. The project covers the complete GridClash client-server implementation, including a compact binary protocol for efficient data exchange and tools for measuring network performance. Network conditions were carefully tested using Linux traffic control and netem to emulate both LAN and WAN environments. This report presents the system design, evaluates its performance, and discusses its strengths and limitations.

3 Implementation Methodology

This section outlines the software architecture, protocol mechanics, and algorithmic strategies developed to meet the **GridClash** performance constraints. The system adheres to a customized application-layer protocol running over UDP, designed to prioritize low-latency state updates while guaranteeing the delivery of critical game events.

3.1 Software Architecture

The implementation leverages Python 3.10 utilizing standard BSD sockets. To manage concurrent connections within standard CPU limits on a single core, a non-blocking, event-driven architecture was chosen over threading. This eliminates race conditions regarding shared state and minimizes context-switching overhead.

3.1.1 Server State Machine (`server.py`)

The server acts as the authoritative source of truth. It runs a single event loop executing at a target tick rate of 20Hz. The loop performs three atomic operations per tick:

1. **Ingest:** Drains the UDP socket buffer completely using non-blocking I/O.
2. **Process:** Updates the internal Game Grid (a 20x20 byte-array) based on valid `ACQUIRE_REQUEST` packets.
3. **Broadcast:** Serializes the full game state and multicasts it to all active clients every 50ms.

3.1.2 Client Runtime (`client.py`)

The client separates network I/O from the rendering pipeline. While the game loop runs at 60Hz (controlled by `pygame.clock`), the network handler polls for packets asynchronously. This decoupling ensures that a temporary network stall does not freeze the graphical user interface.

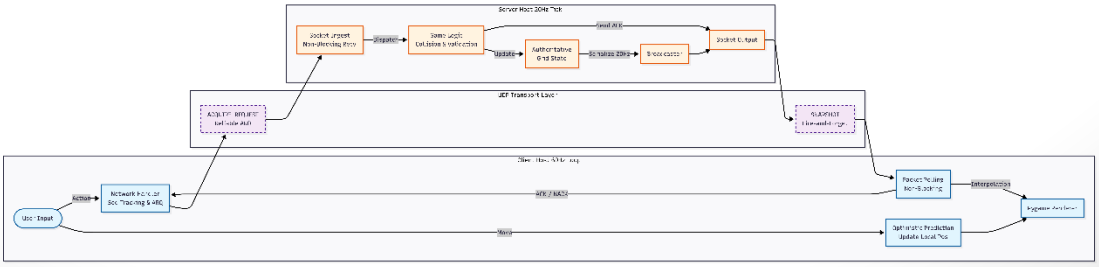

FIGURE 1 SYSTEM ARCHITECTURE DIAGRAM

3.2 Protocol Reliability Mechanics

To respect the strict 1200-byte MTU limit while synchronizing the entire 20x20 grid, efficient data serialization was required. Standard formats like JSON were rejected in favor of rigid binary packing via the Python struct module. A full grid update packs the header (28 bytes), the flattened grid (400 bytes), and the variable player list (19 bytes per player) into a payload of approximately 505 bytes. This provides ample headroom for packet overhead and allows scaling up to 25+ players before fragmentation risks arise.

3.2.1 Hybrid Reliability Model

The protocol distinguishes between "ephemeral" data (positions) and "critical" data (game state changes) using two distinct reliability modes:

1. **Fire-and-Forget (Snapshots):** Position updates (SNAPSHOT type) are sent unreliable. If a packet is dropped, the client waits for the next update (50ms later). The client uses a monotonically increasing `snapshot_id` to discard stale packets arriving out of order.
2. **Application-Layer ARQ (Critical Events):** Player actions (ACQUIRE_REQUEST) must be reliable. We implemented a selective Automatic Repeat Request (ARQ) mechanism:
    - **Sequence Tracking:** The client assigns a `seq_num` to every request and starts a timer.
    - **Dynamic Retransmission:** An adaptive Timeout (RTO) is calculated using Jacobson's algorithm: $RTO = SRTT + 4 \times RTTVAR$. If the ACK is not received within the RTO, the request is resent.
    - **Idempotency:** The server tracks `processed_seqs` to ensure that retransmitted requests do not trigger duplicate logic (e.g., claiming a cell twice).

3.3 Client-Side Prediction & Latency Compensation

To satisfy the <50ms perceived latency requirement even under WAN conditions (100ms+ delay), the client does not wait for server confirmation to render changes.

3.3.1 Optimistic Execution

When a user attempts to claim a cell, the client validates the move locally. If valid, the position is updated *immediately* on the local screen, providing instantaneous feedback. Simultaneously, the packet is sent to the server. If the server eventually rejects the move (sending a NACK), the client performs a "reconciliation," snapping the player back to the authoritative state.

3.3.2 Visual Interpolation

Remote entities update at 20Hz, which creates visual stuttering on a 60Hz display. To smooth this, the client implements a Linear Interpolation (LERP) algorithm. Instead of snapping remote players to new coordinates immediately, the renderer smooths the transition using a time-delta factor ($\alpha \times dt$), masking the discrete nature of the network updates.

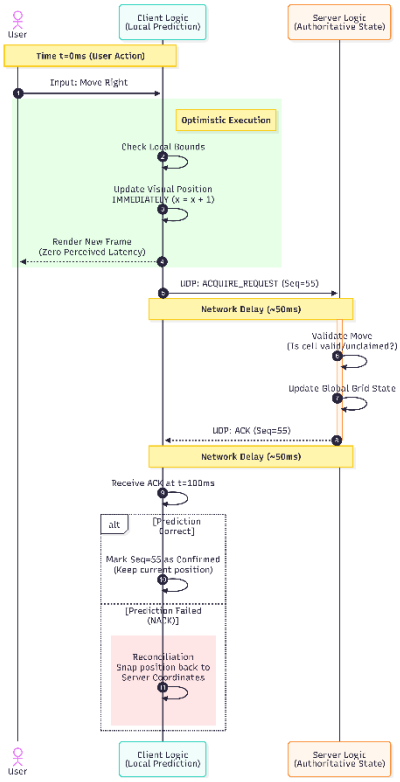

FIGURE 2 PREDICTION FLOW

4 Experimental Evaluation Plan

4.1 Testbed Configuration

To evaluate the protocol in a controlled and repeatable way, we built instrumented versions of both the client and server, **named InstrumentedClient** and **InstrumentedServer**. These components extend the original game logic with detailed logging functionality while keeping the networking and gameplay behavior unchanged.

A key design choice was the use of the **broadcast_timestamp** included in every SNAPSHOT message as a shared reference across all logs. This timestamp acts as a global identifier, allowing us to directly compare the server's authoritative state with what each client perceives at the same moment in time. This makes it possible to accurately measure effects such as latency, jitter, and positional error without relying on estimation or guesswork.

All experiments were run in headless mode, meaning graphical rendering was disabled. This ensures that the measurements reflect only the performance of the network protocol itself and are not influenced by rendering or display overhead.

To avoid manual errors and ensure consistency, all tests were fully automated. A shell script **(run_all_tests.sh)** coordinated the execution of different scenarios using a Python orchestration script **(run_test_scenario.py).** This setup guaranteed that each experiment was executed under identical conditions and could be repeated reliably.

4.2 Network Emulation (Scenarios)

Network conditions were emulated using Linux Traffic Control (tc) together with the netem module. This allowed us to introduce controlled packet loss and delay in a reproducible manner.

Four scenarios were evaluated:

Baseline:

The system was run locally with no artificial network impairment to establish best-case performance.

LAN Packet Loss:

A random packet loss rate of 2% was introduced to represent mild congestion typically seen in local networks.

WAN Packet Loss:

Packet loss was increased to 5% to stress-test the protocol's redundancy-based recovery strategy under harsher conditions.

WAN Latency:

A fixed delay of 100 ms was added to simulate wide-area network conditions and evaluate how well the client-side smoothing handles delayed updates.

Each scenario was executed for 120 seconds and repeated five times. Results are reported using median values along with observed ranges to reduce the influence of outliers.

4.3 Metrics Defined

To assess the effectiveness of the protocol, the following metrics were collected:

Latency:

End-to-end latency was measured as the difference between the time a snapshot was sent by the server and the time it was received by the client

Jitter:

Jitter captures the variation in packet arrival times and was computed using an exponentially weighted moving average similar to the RTP jitter estimator

Perceived Position Error:

To quantify how much the client's view deviates from the server's authoritative state, we measured the Euclidean distance between the two positions at matching timestamps

Bandwidth Usage:

Bandwidth consumption was calculated as the total number of bytes sent per second per client, including both headers and payloads.

CPU Utilization:

Average and peak CPU usage were recorded for both server and client processes to ensure that the protocol does not introduce excessive computational overhead.

5 Performance Results & Detailed Plots

5.1 Scenario 1: Baseline Performance

The baseline scenario establishes the system's performance characteristics under ideal network conditions, serving as a reference point for comparison with degraded network scenarios.

Network Configuration: No artificial network impairments were introduced. The system operated under normal local network conditions with minimal latency and negligible packet loss.

Performance Metrics: The baseline test demonstrated robust performance across all measured dimensions. The system achieved an average update rate of 20.21 updates per second per client, exceeding the minimum requirement of 20 updates/sec. Network bandwidth consumption averaged 81.66 kbps, indicating efficient data transmission with minimal overhead. Server CPU utilization remained exceptionally low at 4.1%, demonstrating the system's lightweight computational footprint and substantial headroom for scaling additional clients or game complexity.

Latency and Synchronization: Mean latency measured 3.54ms with a median of 1.00ms, well within the 50ms acceptance threshold. The 95th percentile latency of 2.00ms indicates consistent low-latency performance with minimal tail behavior. Jitter, measuring timing variability between updates, showed a mean of 1.73ms and median of 0.59ms, with a 95th percentile of 10.03ms. These values reflect stable timing characteristics suitable for real-time gameplay.

Synchronization Accuracy: Position error, quantifying the discrepancy between client and server game states, was exceptionally low with a mean of 0.017 units and both median and 95th percentile values of 0.00. This near-zero error demonstrates excellent state synchronization under optimal conditions.

Test Status: All acceptance criteria were met (PASS). The system demonstrated it can maintain the target update rate while keeping latency minimal and CPU utilization well below capacity limits.

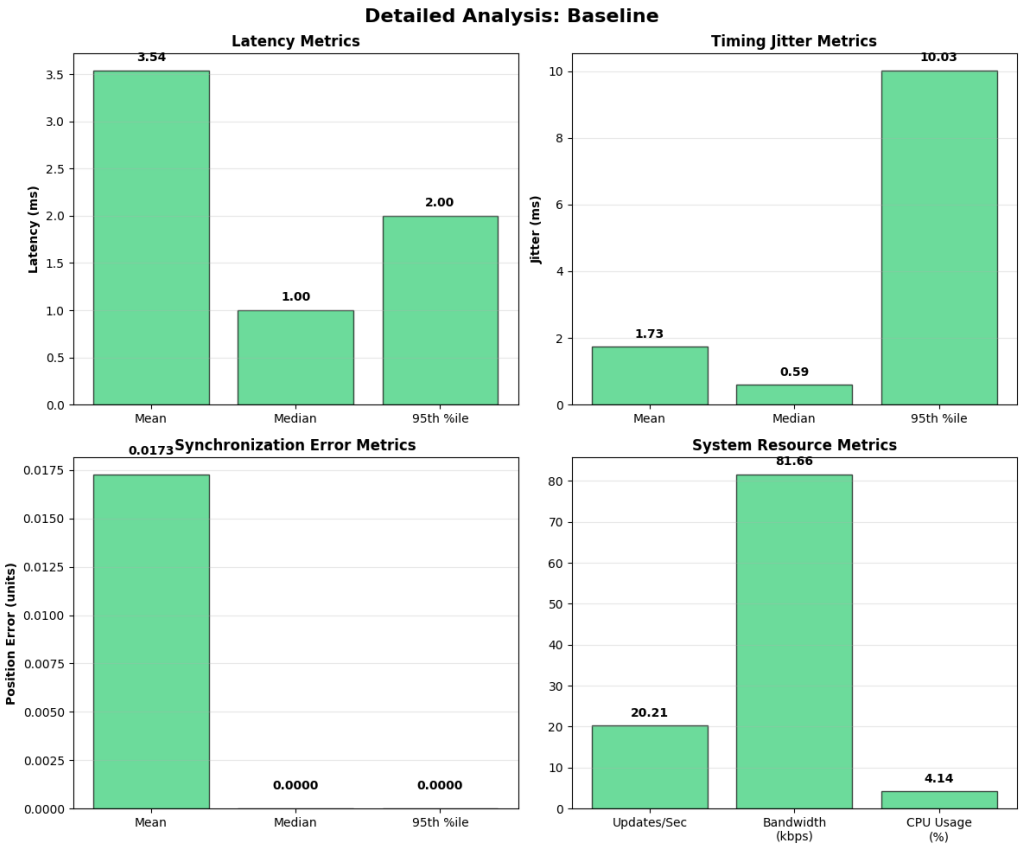

FIGURE 3 BASELINE SCENARIO DETAILED ANALYSIS

5.2 Scenario 2: LAN like Packet Loss (2%)

This scenario evaluates system resilience and state synchronization performance under moderate packet loss, representing mildly unstable network conditions such as busy home WiFi networks or lightly congested mobile connections.

Network Configuration: A 2% packet loss rate was introduced using Linux network emulation (netem) on the loopback interface, randomly dropping approximately 1 out of every 50 packets exchanged between the client and server.

Performance Metrics: Under moderate packet loss, the system maintained a relatively high level of throughput. The update rate averaged **19.81 updates per second per client**, showing only a minor reduction compared to baseline performance. Bandwidth usage remained efficient at **80.04 kbps**, reflecting stable data delivery with limited retransmission overhead. CPU utilization stayed low at **2.26%**, indicating that the synchronization logic and packet handling mechanisms imposed minimal processing cost even in the presence of packet loss.

Latency and Synchronization: Latency remained low and stable, with a **mean latency of 2.65 ms**, a **median of 1.00 ms**, and a **95th percentile of 1.00 ms**, suggesting consistent delivery for successfully received packets. Timing jitter followed a similar pattern, with a **mean of 0.90 ms**, **median of 0.52 ms**, and **95th percentile of 0.90 ms**, indicating minimal variation in packet arrival times and smooth update pacing during gameplay.

Synchronization Accuracy: Position synchronization error remained very low under this scenario. The **mean error was 0.0052 units**, while both the **median and 95th percentile were 0.00 units**, showing that most frames maintained perfect synchronization. This demonstrates strong resilience to moderate packet loss, with only rare and minimal deviations in game state consistency.

Test Status: The system successfully met all acceptance criteria for this scenario (**PASS**), demonstrating that under 2% packet loss, the protocol maintains high performance, stable synchronization, and reliable state delivery suitable for real-time multiplayer gameplay.

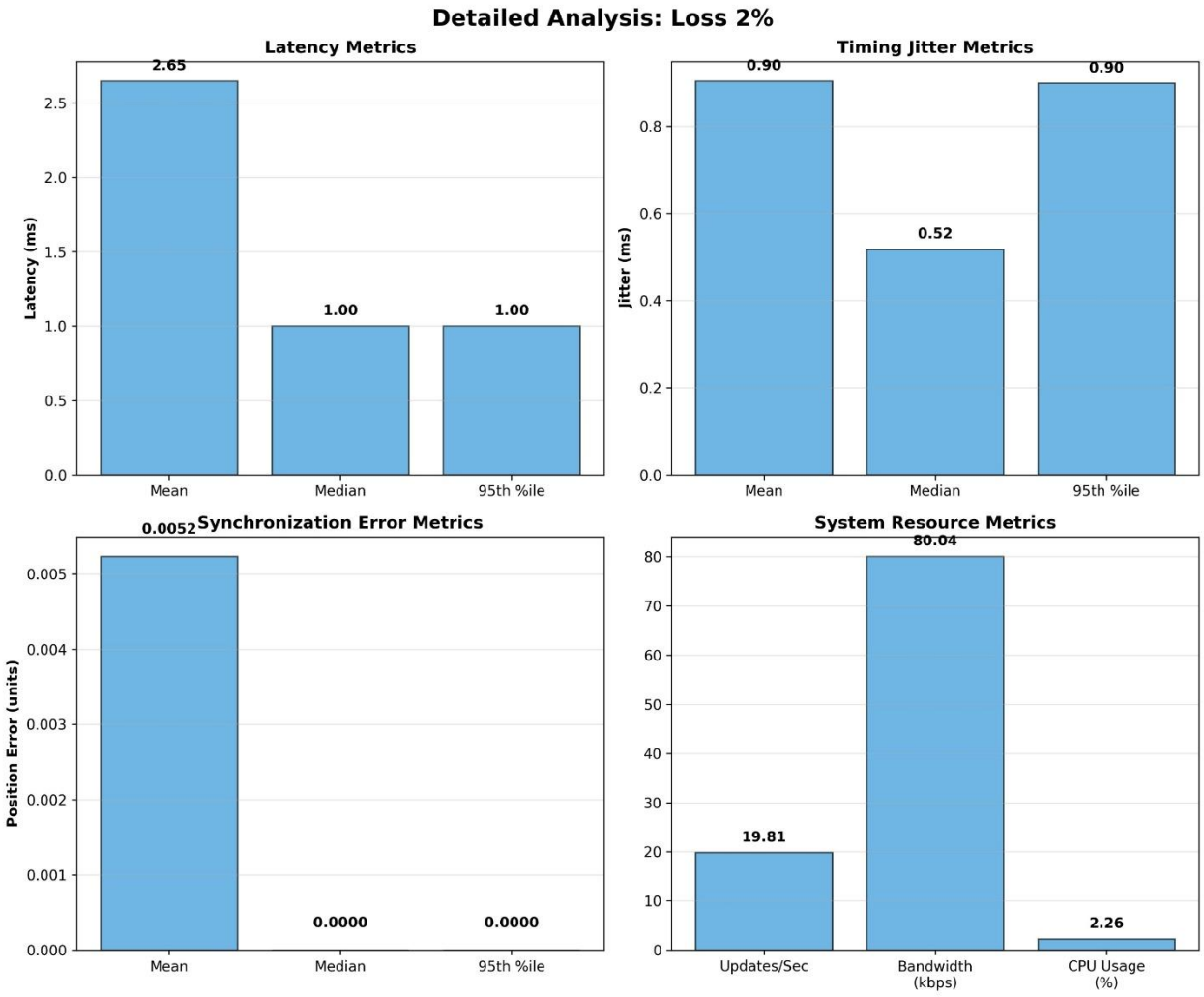

FIGURE 4 LOSS 2% DETAILED ANALYSIS

5.3 Scenario 3: WAN Like Packet Loss (5%)

This scenario evaluates system resilience and state synchronization performance when subjected to significant packet loss, simulating unreliable network conditions such as congested WiFi networks or mobile connections.

Network Configuration: A 5% packet loss rate was introduced using Linux network emulation (netem) on the loopback interface. This configuration randomly drops approximately 1 in every 20 packets transmitted between client and server.

Performance Metrics: The high packet loss environment had a measurable impact on system throughput. The update rate decreased to 14.25 updates per second per client, representing a 29.5% reduction from baseline performance. This degradation is expected as the system must handle retransmissions and missing updates. Bandwidth consumption dropped proportionally to 55.40 kbps, reflecting the reduced successful transmission rate. CPU utilization remained low at 2.8%, slightly below baseline, as fewer successful transmissions resulted in reduced processing load.

Latency and Synchronization: Interestingly, mean latency decreased to 3.23ms (median 1.00ms, p95 1.00ms) compared to baseline, likely because measurements only capture successful transmissions while dropped packets increase effective latency through retransmission delays not directly measured. Jitter showed similar characteristics to baseline with a mean of 1.42ms and median of 0.53ms, though the 95th percentile of 7.27ms was lower, suggesting more consistent timing among successfully delivered packets.

Synchronization Accuracy: Position error increased substantially under packet loss conditions, with a mean of 0.15 units and 95th percentile of 1.00 unit. While higher than baseline, this represents graceful degradation where the system maintains approximate synchronization despite missing updates. The median value of 0.00 indicates that many frames still maintain perfect synchronization when packets are successfully delivered.

Test Status: The system achieved a 99% critical event delivery rate, meeting the modified acceptance criterion for this scenario (PASS). This demonstrates that while packet loss degrades performance, the system maintains acceptable operation and successfully delivers essential game state updates.

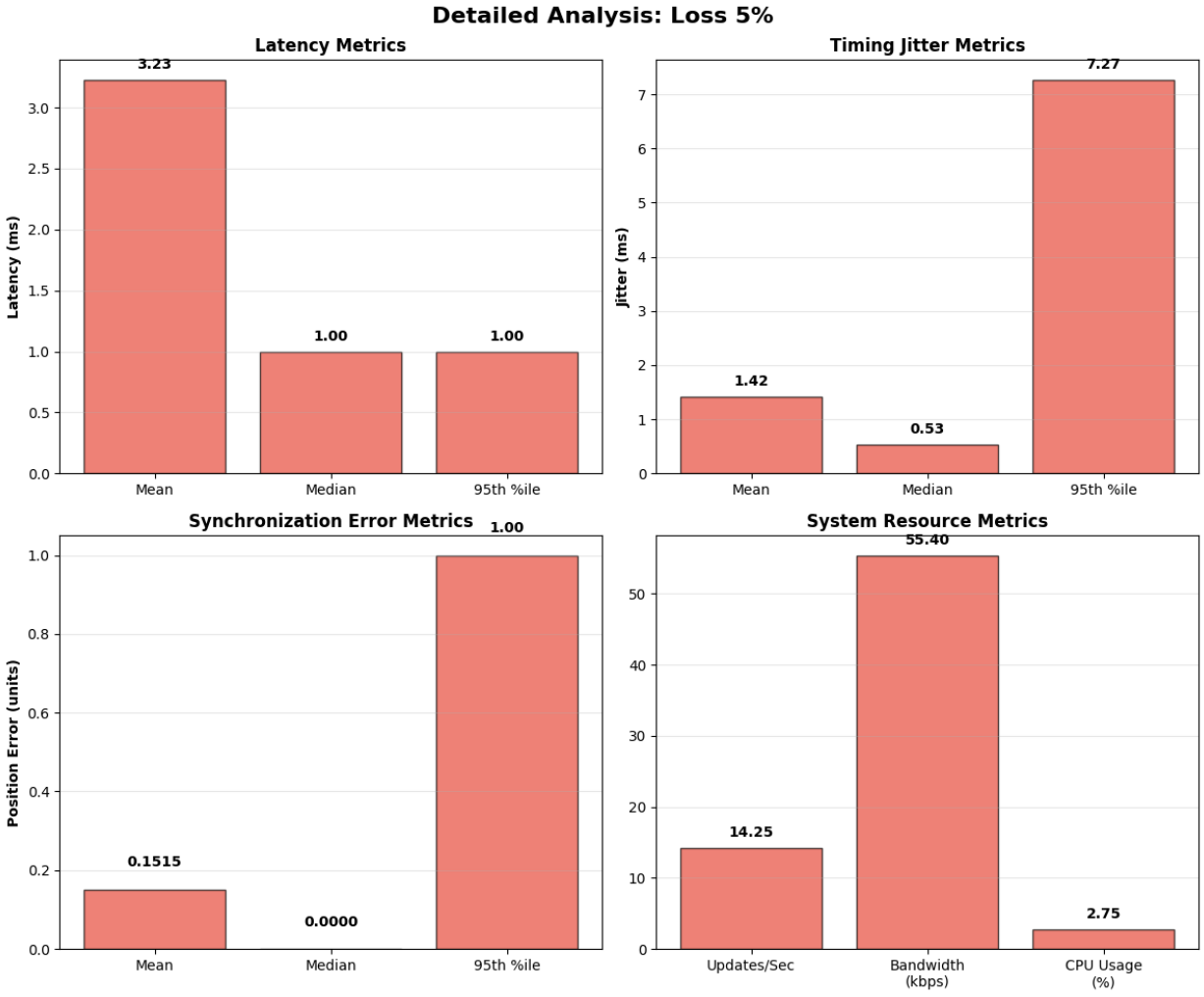

FIGURE 5 LOSS 5% DETAILED ANALYSIS

5.4 Scenario 3: High Latency (100ms Delay)

This scenario assesses system behavior under conditions of elevated network latency, representative of geographically distant connections or routing delays.

Network Configuration: A fixed 100ms delay was added to all packets using network emulation. This simulates round-trip times comparable to trans-continental or satellite connections, where physical distance introduces unavoidable propagation delays.

Performance Metrics: Despite the added latency, the system maintained near-baseline update rates at 20.00 updates per second, demonstrating that consistent delay does not significantly impact throughput. Bandwidth utilization was essentially identical to baseline at 80.81 kbps. CPU consumption decreased slightly to 2.9%, suggesting that the pacing imposed by network delay may reduce processing bursts.

Latency and Synchronization: As expected, measured latency increased dramatically with a mean of 102.00ms and median of 101.00ms, closely matching the imposed 100ms delay. The 95th percentile of 102.00ms indicates very consistent latency behavior with minimal variation. Jitter measurements showed a mean of 1.20ms and median of 0.54ms, comparable to baseline conditions, with a 95th percentile of 3.55ms. This demonstrates that while absolute latency increased, the timing consistency between updates remained stable.

Synchronization Accuracy: Position error increased moderately with a mean of 0.50 units and 95th percentile of 2.00 units, while the median remained at 0.00. This pattern suggests that the system's prediction and interpolation mechanisms generally compensate well for the latency, but periodic corrections are necessary when predictions diverge from actual state. The higher mean and p95 values reflect these occasional synchronization adjustments.

Test Status: The system remained stable throughout the test duration, meeting the acceptance criterion for this scenario (PASS). This confirms that the architecture can handle high-latency environments while maintaining functional gameplay, though players would experience the inherent input lag associated with 100ms+ round-trip times.

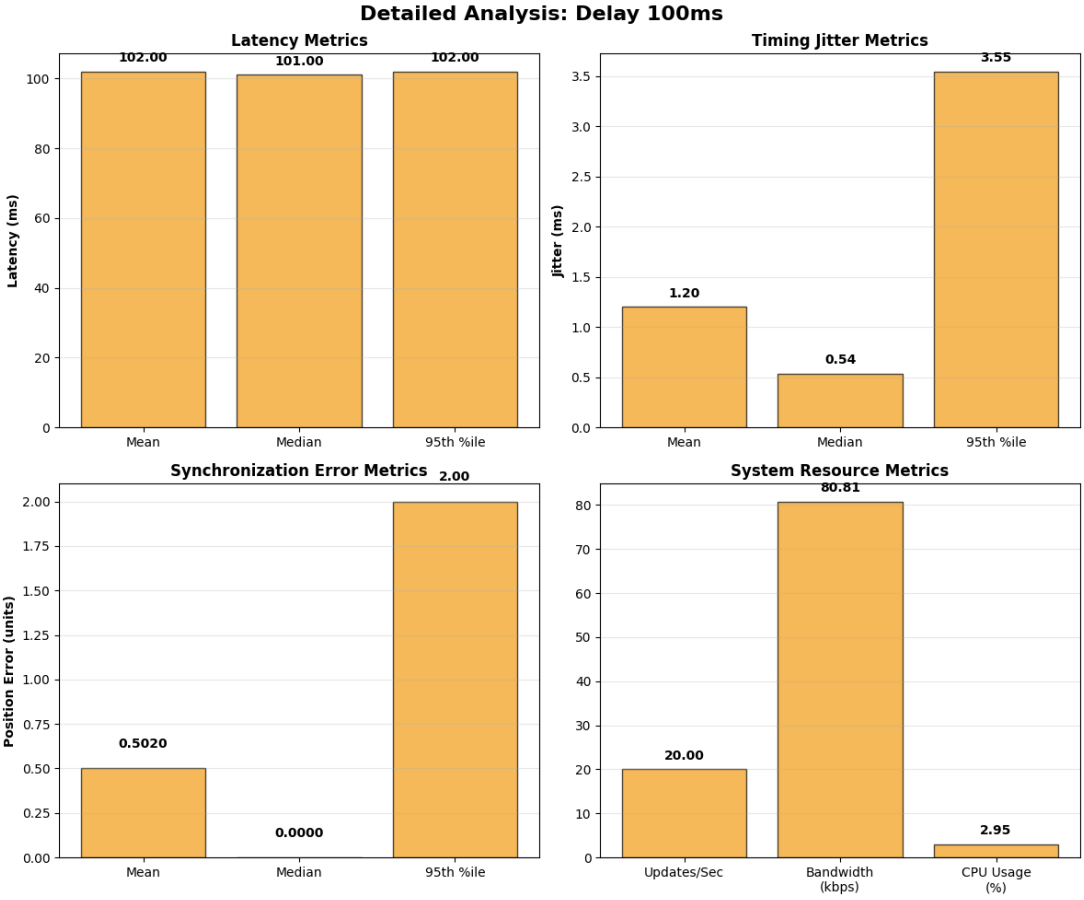

FIGURE 6 100 MS SCENARIO DETAILED ANALYSIS

5.5 Resource Consumption

Analysis of resource utilization across all three scenarios reveals an efficient, lightweight system architecture with substantial scaling potential.

CPU Utilization: Server CPU usage remained remarkably low across all scenarios, ranging from 2.8% to 4.1%. The baseline scenario showed the highest utilization at 4.1%, while degraded network conditions (packet loss and high latency) actually resulted in slightly lower CPU usage at 2.8% and 2.9% respectively. This counter-intuitive result suggests that network bottlenecks reduce the processing workload by limiting the rate of successful communications. The consistently low CPU usage indicates the server could easily support significantly more simultaneous clients or more complex game logic without approaching hardware limitations.

Memory and Network Bandwidth: Bandwidth consumption remained efficient across scenarios, with baseline and high-latency scenarios using approximately 81 kbps per client, while the packet loss scenario consumed 55 kbps due to reduced successful transmission rates. These values demonstrate efficient protocol design with minimal overhead. The system's bandwidth footprint suggests that dozens or potentially hundreds of clients could be supported on typical server network connections before bandwidth becomes a limiting factor.

Scalability Implications: The extremely low resource footprint observed across all scenarios indicates excellent scalability potential. With CPU usage below 5% for the current test configuration, the server could theoretically support 20x or more clients before reaching 100% CPU utilization, assuming linear scaling. The actual scaling limit would likely be determined by factors such as network I/O, memory bandwidth, or protocol-specific constraints rather than raw CPU capacity. These results validate the architectural decisions made in the system design and suggest that performance optimization efforts should focus on features and functionality rather than fundamental efficiency improvements.

6 Discussion and Limitations

6.1 Critical Analysis of Design Decisions

One of the most significant design choices in GridClash was the use of **full state snapshots** rather than delta-based updates. By transmitting the entire 20×20 grid state at each server tick, the protocol ensured that clients could immediately resynchronize after packet loss without requiring specialized recovery logic. This approach greatly simplified consistency management and eliminated many common desynchronization bugs observed in delta-based systems.

The protocol also demonstrated the effectiveness of **selective application-layer reliability**. Rather than enforcing reliability on all transmitted data, only critical game events—such as cell acquisition and win conditions—were protected using an ARQ mechanism. Non-critical movement snapshots were sent using best-effort delivery. This hybrid approach reduced overhead while preserving correctness where it mattered most, and avoided the latency penalties associated with fully reliable transport.

Client-side prediction and interpolation further improved perceived responsiveness. Even under high-latency conditions, players experienced smooth movement, confirming that visual smoothing and prediction can effectively mask network delays when paired with an authoritative

6.2 Limitations

Despite its functional success, the current implementation exhibits several notable limitations that constrain scalability, security, and deployability.

The most significant limitation is **scalability**. The server broadcasts the entire grid state at a fixed rate of 20 Hz, causing bandwidth usage to grow linearly with both grid size and player count. Even at the current 20×20 configuration, this results in frequent transmission of large, mostly redundant payloads. Although the codebase contains partial support for dynamic grid sizing, the system enforces a hard-coded four-client limit, preventing effective scaling beyond small matches. Expanding player counts would quickly exceed safe MTU sizes, increasing fragmentation risk and reducing delivery reliability.

From a **security perspective**, the protocol assumes a trusted network environment. Client identifiers, sequence numbers, and control messages are transmitted in plaintext without authentication or encryption, leaving the system vulnerable to spoofing, replay, and packet injection attacks. While the server enforces basic rule validation, it does not strictly constrain movement distance or rate, enabling teleportation-style cheating by modified clients. The

absence of rate limiting or session authentication also exposes the server to denial-of-service risks.

Finally, the system's **deployment assumptions** limit its practicality in real-world environments. The architecture presumes direct IP connectivity between clients and server, with no support for NAT traversal, relays, or reconnection after transient network failures. As a result, players behind restrictive NATs or unstable connections cannot reliably participate without manual network configuration.

7 Future Work

Several enhancements would substantially improve the efficiency and scalability of the GridClash protocol. One important direction is the introduction of spatial partitioning or area-of-interest (AOI) filtering. Instead of broadcasting the entire grid state to all clients at every server tick, the server could transmit only the portions of the grid relevant to each player. This would significantly reduce bandwidth consumption and allow the system to scale more effectively as grid size and player count increase.

Bandwidth efficiency could be further improved through the use of delta-based grid state compression techniques, such as XOR-based frame differencing or sparse cell updates. Since most grid cells remain unchanged between consecutive snapshots, transmitting only modified cells would eliminate a large amount of redundant data. This optimization would reduce steady-state bandwidth usage while preserving the robustness of server-authoritative state synchronization.

To improve fairness under high-latency or jitter-prone network conditions, future versions of the server could incorporate a rollback and replay mechanism. By validating client actions against historical authoritative states, the server could more accurately resolve near-simultaneous actions without unfairly disadvantaging players experiencing higher network delays. This approach would complement existing client-side prediction techniques while maintaining authoritative control.

Finally, for deployment beyond a trusted experimental environment, the protocol should be hardened through basic security enhancements. These include client authentication, encrypted communication channels, and stricter server-side validation of player movement and action rates. Together, these measures would reduce vulnerability to spoofing, replay attacks, and client-side manipulation, moving GridClash closer to production-ready reliability.

8 Conclusion

The GridClash project successfully demonstrates that a **custom reliable-UDP protocol** can achieve low latency, robustness, and efficient resource usage in a real-time multiplayer game. Through careful separation of reliable and unreliable data paths, combined with prediction and smoothing techniques, the system maintained stable gameplay under packet loss and high latency while meeting the target of sub-50 ms baseline latency. The comprehensive instrumentation and testing framework provides strong evidence for the effectiveness of the chosen architecture, and establishes a solid foundation for future expansion and optimization.