RFC – EchoSync Protocol (ESP)

1. Introduction

The EchoSync Protocol (ESP) is a custom UDP-based synchronization protocol built for the real-time multiplayer game "Grid Clash".
It provides low-latency, partially reliable communication between clients and the central game server.
In Grid Clash, players compete on a shared 20×20 grid to claim cells by clicking them.
The server acts as the authoritative source of truth — it receives player actions, resolves conflicts, and continuously broadcasts snapshots representing the current grid state.
ESP avoids TCP because retransmission and congestion control cause latency spikes unsuitable for fast-paced games.
Instead, it builds a lightweight reliability layer on top of UDP, handling:

- **Fragmentation a**nd reassembly of large packets,
- **Sequence-based acknowledgment** and **retransmission**,
- **Periodic synchronization** of state through snapshots and incremental updates.

1.1 Assumptions & Constraints:

- Reliability Mechanism: Redundant updates (include last K updates per packet),
- Transport Layer: UDP
- Maximum packet size: ≤ 1200 bytes (fragments larger payloads automatically)
- Update rate: 20–60 Hz
- Expected packet loss: 2–5%
- Target latency: ≤ 50 ms

1.2 Repo & Demo Video:

- **GitHub Repo**
  https://github.com/okhadragy/EchoSync-Multiplayer-Game-Protocol
- **Demo Video**
  https://www.youtube.com/watch?v=Y9ZhF8ZhnWk

---

2. Protocol Architecture

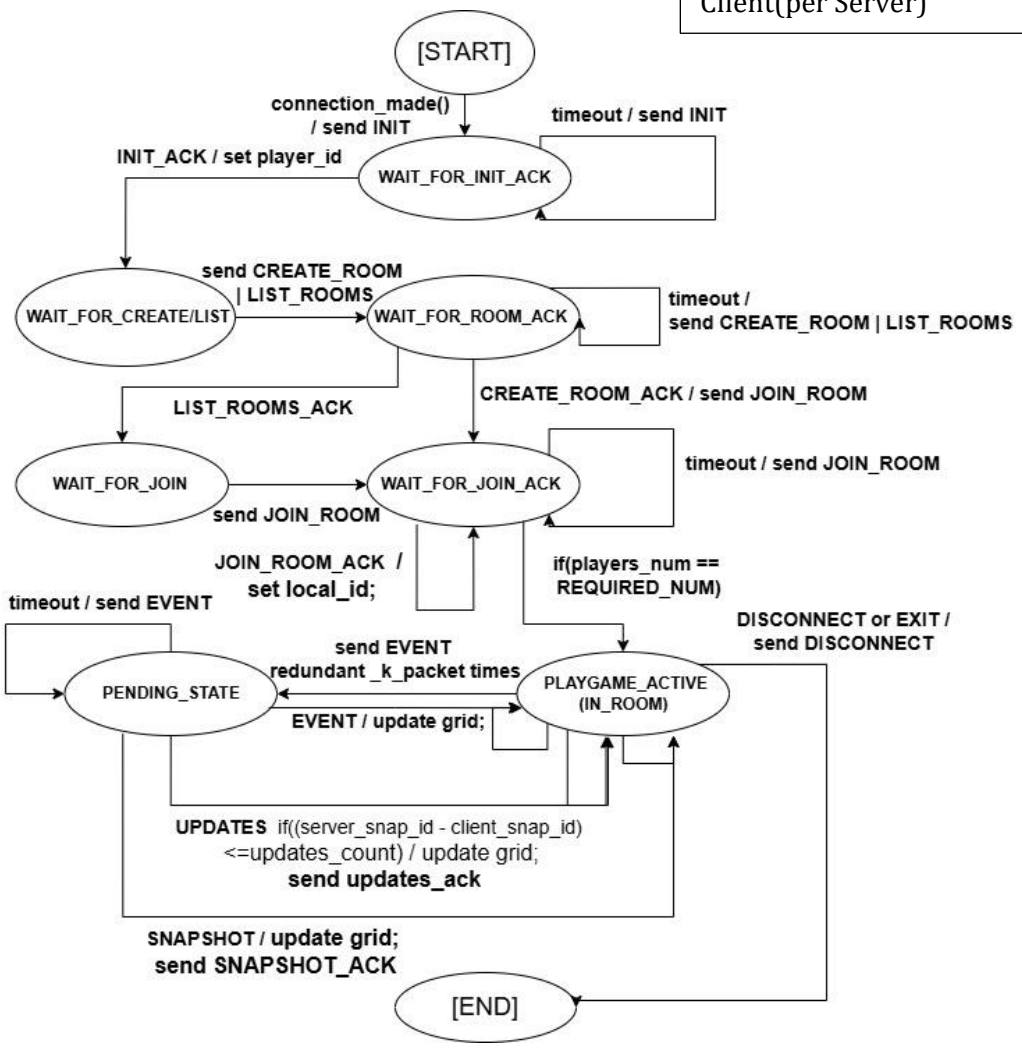
The EchoSync Protocol follows a centralized client–server architecture, where the server manages the global game state, and clients synchronize based on snapshots and updates.

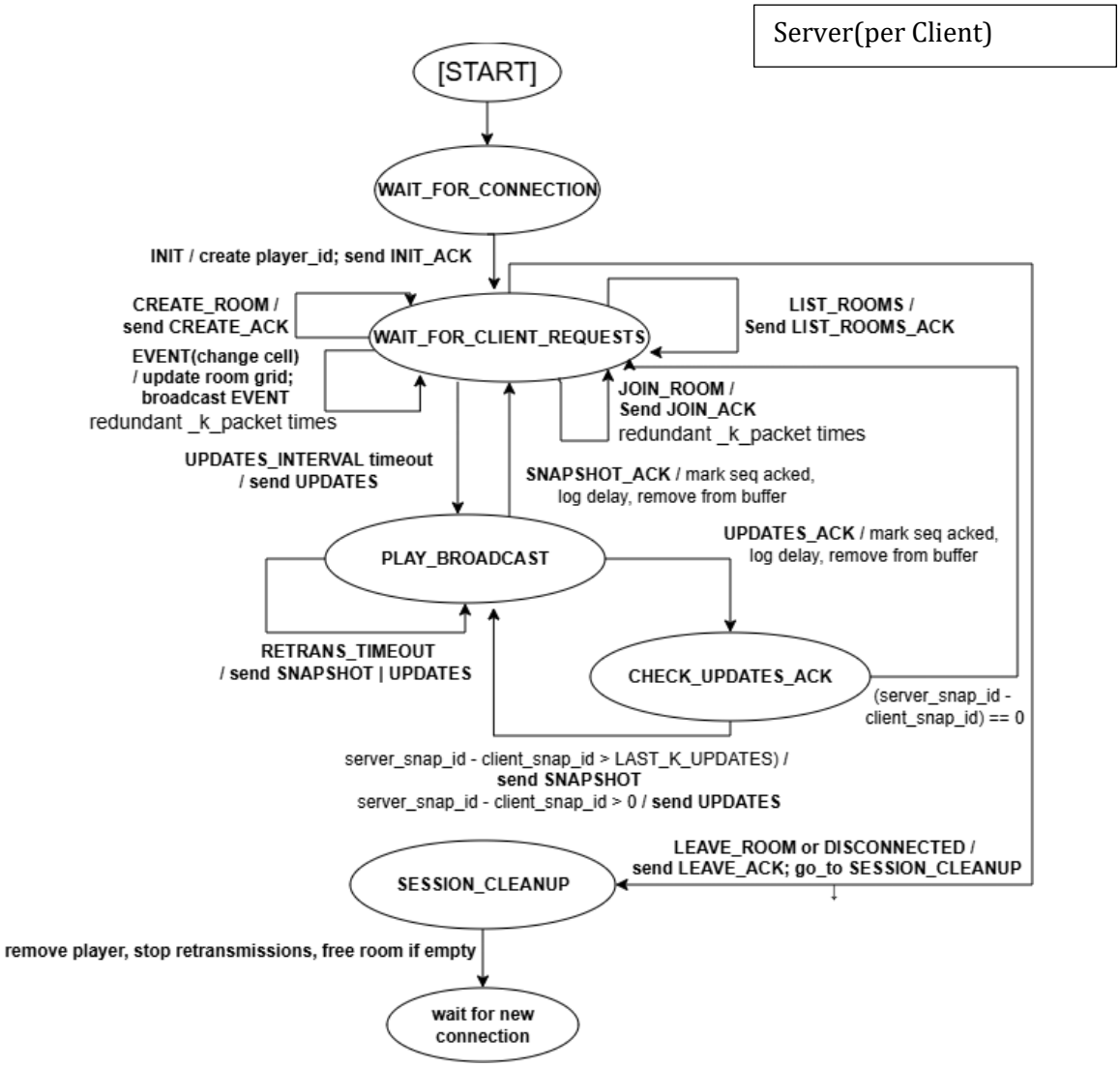
2.1 Entities

- **Server:**
  - Maintains the authoritative grid state and list of active rooms with up to 4 concurrent players.
  - Assigns each player a unique player_id and manages player registration.
  - Periodically sends UPDATES messages (Most Recent Update) to all players in each room.
  - If client is too late send Sapshot(Complete Grid State)
  - Tracks acknowledgments (ACKs) to measure latency and retransmit lost packets if necessary.
  - Cleans up disconnected clients and removes inactive rooms automatically.
- **Clients:**
  - Connect to the server using an INIT → INIT_ACK handshake.
  - Send CREATE_ROOM or JOIN_ROOM requests to enter a game session.
  - Once inside a room, they render the grid and continuously receive SNAPSHOT and UPDATES packets.
  - On each cell click, they send EVENT messages (cell claim requests).
  - Respond to each SNAPSHOT or UPDATE with an ACK (to confirm receipt and assist in latency tracking).
  - Handle retransmission and fragment reassembly for large payloads.

Finite State Machine

Client(per Server)





3-Message Formats

Each ESP packet is composed of a fixed-length header (32 bytes) followed by a variable-length payload (0–1168 bytes), keeping the total packet size under 1200 bytes to avoid fragmentation. The header ensures correct packet identification, ordering, synchronization, and integrity validation across UDP transport.

3.1 Header Structure(ASCII lay-out)

Field	Size(Bytes)	Offset	Justification	Type(Struct Format)
Protocol_id	**4**	**0-3**	**Constant ASCII `"ESP1"` identifying the protocol version**	**4s**
Version	**1**	**4**	**Currently `1`**	**B**

Msg_type	**1**	**5**	**Message category (e.g., `0=INIT`, `1=INIT_ACK`, `2=CREATE_ROOM`)**	**B**
Snapshot_id	**4**	**6-9**	**Snapshot identifier, used to track game state updates**	**I**
Seq-num	**4**	**10-13**	**Sequence number for ordering and loss detection.**	**I**
Timestamp	**8**	**14 – 21**	**for synchronization and latency measurement**	**Q**
Payload_len	**2**	**22 – 23**	**length of the payload data following the header**	**H**
Packet_id	**4**	**24 – 27**	**unique packet identifier used for retransmission tracking and debugging**	**I**
Checksum	**4**	**28 – 31**	**CRC32 checksum for verifying packet integrity.**	**I**

3.2-Sample message:

This message is an INIT_ACK packet sent by the server to confirm a client's initialization request, acknowledging sequence 42 with integrity verified by a checksum.

Field	Value
Protocol_id	b'ESP1'
Version	1
Msg_type	1(INIT_ACK = 1)
Snapshot_id	0
Seq-num	42
Timestamp	173,066,111,234.568 (ms)
Payload_len	8
Packet_Id	1001
Checksum	0x4A8F12CD

4. Communication Procedures

4.1 Connection Establishment

Step-by-step sequence:

1. **Client → Server: INIT**
    - **Client sends empty INIT packet**
    - **seq_num = 1, no payload**
2. **Server → Client: INIT_ACK**
    - **Server assigns unique player_id**
    - **Payload: seq_num (4B) + player_id (4B)**
    - **Client stores player_id for future messages**

Example Trace:

```
[T=0.000s] Client sends INIT (seq=1, pkt_id=1)
[T=0.015s] Server receives INIT
[T=0.016s] Server sends INIT_ACK (seq=1, player_id=1001)
[T=0.031s] Client receives INIT_ACK, stores player_id=1001
```

4.2 Room Creation and Joining

Creating a Room:

```
Client → Server: CREATE_ROOM
Server → Client: CREATE_ACK
  Payload: seq_num (4B) + room_id (1B)
```

Joining a Room:

```
Client → Server: JOIN_ROOM
  Payload: room_id (1B)

Server validates:
  - Room exists
  - Room has space (< 4 players)
  - Assigns local_id (1-4)

Server → All room players: JOIN_ACK
  Broadcasting updated player list to all participants
  New player receives full grid SNAPSHOT
```

4.3 Normal Game Operation

Client Action Flow:

```
 Player clicks cell → Client sends EVENT (event_type=0, cell_idx)
 Server validates: cell is free, player is in room, room has 4 players
 ? If valid:
    • Server updates grid
    • Increments snapshot_id
    • Broadcasts EVENT to all room players (with 3× redundancy)
```

Server Broadcast Cycle (20.7 Hz):

Every 48.3ms, server sends UPDATES to all players containing the last K=3 updates (most recent 3 grid changes). This allows clients to catch up if they missed 1-2 packets.

Client Response:

- Client sends UPDATES_ACK with received seq_num
- If client's snapshot_id is behind:
    - Server calculates gap: gap = room.snapshot_id - client.snapshot_id
    - If gap ≤ 10: Send UPDATES with missing deltas
    - If gap > 10: Send full SNAPSHOT (client too far behind)

4.4 Error Recovery

Scenario 1: Packet Loss

- Server maintains unacked packets dictionary
- Every 200ms, retransmit unacked packets (max 5 retries)
- After 5 failed attempts, packet is dropped

Scenario 2: Client Desynchronization:

```
IF client.snapshot_id < server.snapshot_id THEN
  gap = server.snapshot_id - client.snapshot_id
  IF gap <= 10 THEN
    Send UPDATES with missing deltas
  ELSE
    Send full SNAPSHOT
  END IF
END IF
```

Scenario 3: Stale Pending Cell

- Client marks cell as "pending" when clicked
- If no confirmation after 100ms → retry EVENT
- Prevents stuck UI states

4.5 Disconnection

Graceful Disconnection:

Client → Server: DISCONNECT

Server cleanup procedure:
1. Remove player from room.players

2. Free their grid cells (set to 0)
3. Remove their updates from queue
4. Broadcast SNAPSHOT to remaining players
5. If room empty → delete room

Ungraceful Disconnection (timeout):

- Server detects no activity after fragment_timeout (5 seconds)
- Automatically executes cleanup procedure

5. Reliability & Performance Features

5.1 Fragmentation and Reassembly

Problem: While SNAPSHOT payloads (400 bytes) + header (32 bytes) = 432 bytes fit in one packet, large payloads like JOIN_ACK with many players could exceed the MTU(Max transmission unit).

Fragmentation Algorithm:

When payload > **MAX_DATA (1168 bytes):**

1. Calculate fragment_count = [payload_len / 1168]
2. FOR each fragment i FROM **0** TO **fragment_count-1:**
   a. Extract slice: data[i*1168 : min((i+1)*1168, payload_len)]
   b. Assign seq_num = base_seq + i
   c. Set pkt_id (same for all fragments of this message)
   d. Compute CRC32 checksum over (header + fragment_data)
   e. Transmit fragment
3. END FOR

Reassembly Algorithm:

On receiving fragment:
1. Store in fragments[(client_addr, pkt_id)][seq_num] = payload
2. Track received_bytes and expected_bytes
3. IF received_bytes >= expected_bytes THEN
   a. Verify all seq_nums are contiguous
   b. Concatenate fragments in sequence order
   c. Return complete payload
   d. Delete fragment entry
4. ELSE
   Wait for more fragments
5. END IF

Background cleanup:
Remove incomplete fragments after 5 second timeout

Example:

```
Payload = 2000 bytes, MAX_DATA = 1168 bytes

Fragment 1:
  seq=10, pkt_id=50, payload_len=1168
  data = payload[0:1168]

Fragment 2:
  seq=11, pkt_id=50, payload_len=832
  data = payload[1168:2000]

Receiver assembles when both seq=10 and seq=11 received
```

5.2 Redundant Updates Strategy

ESP implements redundant updates as the primary reliability mechanism:

1. K=3 Redundancy in UPDATES:

- Each UPDATES packet contains the last 3 grid changes
- If packet N is lost, packet N+1 still carries the change from N

2. 3× Packet Transmission for Critical Messages:

- JOIN_ACK, LEAVE_ACK, and EVENT messages sent 3 times immediately
- Compensates for 2-5% loss without retransmission delay
- Cost: 2× additional bandwidth for critical messages only

3. Rolling Update Window:

- Server maintains queue of last 10 updates per room
- Client can request missing updates via **UPDATES_ACK**
- If gap > 10 updates, full SNAPSHOT sent

Justification:

- Adds minimal bandwidth overhead
- Eliminates need for complex NACK mechanism
- Effective for 2-5% loss environments
- Reduces latency compared to retransmission-only approaches

5.3 Retransmission Timers

Configuration:

```
RETRANS_TIMEOUT = 100 ms
MAX_TRANSMISSION_RETRIES = 5
```

Retransmission Algorithm:

```
FOR each unacked_packet IN unacked_packets:
  IF packet.sent_count >= MAX_RETRIES THEN
    Remove packet from unacked_packets
    Log: "Dropped after 5 retries"
    CONTINUE
  END IF

  IF (current_time - packet.last_sent) > RETRANS_TIMEOUT THEN
    Retransmit packet
    packet.last_sent = current_time
    packet.sent_count++
  END IF
END FOR
```

RTO Selection:

- Fixed 100ms based on target latency ≤ 50ms
- Allows 2× RTT margin before retransmission

5.4 Sequence Number Management

Per-Player Sequences:

- Server maintains independent seq_num counter for each player_id
- Prevents sequence collisions in multi-client scenarios
- 32-bit sequence space allows 4 billion packets before wraparound

Ordering and Discard Rules:

```
On receiving packet:
1. IF packet.snapshot_id < client.current_snapshot_id THEN
   Discard (outdated state)
2. ELSE IF packet.snapshot_id == client.current_snapshot_id THEN
   Accept (current state update)
3. ELSE
   Accept and update client.current_snapshot_id
4. END IF
```

Duplicate Detection:

- Clients track most recent seq_num per message type
- Packets with seq_num ≤ last_seen_seq are duplicates
- ACK sent but payload ignored

5.5 Checksum Validation

CRC32 Calculation:

```
Checksum Algorithm:
1. Create header with checksum field = 0
2. Concatenate: header_zeroed + payload
3. Compute: checksum = CRC32(header_zeroed + payload) & 0xFFFFFFFF
4. Insert checksum into header[28:31]
```

Validation On Receive:

```
On packet reception:
1. Extract received_checksum from header bytes [28:31]
2. Zero out header bytes [28:31]
3. Compute calculated_checksum = CRC32(header_zeroed + payload)
4. IF calculated_checksum ≠ received_checksum THEN
    Discard packet silently (corruption detected)
5. ELSE
    Accept packet
6. END IF
```

Justification:

CRC32 detects 99.9999% of bit errors with minimal CPU overhead (~50 CPU cycles per packet).

5.6 Justification: Comparison with Known Protocols

How EchoSync Protocol (ESP) Differs from Known Protocols (e.g., MQTT-SN)

While both the EchoSync Protocol (ESP) and MQTT-SN are designed for communication over unreliable transports like UDP, their goals and design trade-offs are fundamentally different. This table highlights the core differences:

Feature	**ESP (EchoSync Protocol)**	**MQTT-SN (Message Queuing Telemetry Transport for Sensor Networks)**
Primary Use Case	Real-time multiplayer game state synchronization	Lightweight publish/subscribe messaging in constrained IoT networks
Communication Pattern	Centralized client–server with continuous snapshot and update pushes	Broker-centric pub/sub with asynchronous message distribution
Quality of Service	Partial reliability built for low latency (no strict delivery guarantees)	Supports defined QoS levels (0, 1, 2) for message delivery assurances
Retransmission Strategy	Redundant updates + limited retransmit timers	Retransmissions based on protocol QoS level
Packet Overhead & Design	Fixed minimal header (~32 bytes) plus application payload	Structured control packets (e.g., CONNECT, SUBSCRIBE) with additional fields
Synchronization Model	Server authoritative state & incremental deltas	Topic-based message distribution, with no notion of global state synchronization
Latency Bias	Designed to provide low and stable latency (~≤50 ms)	MQTT-SN's reliability mechanisms may introduce variable delays
Intended Network Loss Profile	Tailored for mild packet loss (2–5%) and frequent broadcasts	Designed for intermittent connectivity in low-power networks

Key Differences Explained

1) **Purpose & Philosophy**
   ESP is purpose-built for **real-time games** where the *latest state matters more than guaranteed delivery of all historical packets*. By contrast, MQTT-SN is optimized for **event messaging in sensor networks**, where reliable delivery and message ordering per topic are essential.

2) **State vs Events**
   ESP focuses on maintaining *global authoritative state snapshots* and *incremental updates* to all clients, tolerating occasional missed packets. MQTT-SN focuses on delivering **individual messages** reliably via a broker.

3) **Reliability Model**
   ESP uses **redundant update inclusion** (last K events) and periodic retransmissions for essential messages instead of complex QoS protocols. MQTT-SN uses multi-level QoS to guarantee delivery, which incurs extra protocol overhead and delays.

4) **Latency vs Guarantee Trade-off**
   ESP trades strict delivery guarantees for low latency and smoothing behavior in high-frequency update loops. MQTT-SN's reliability mechanisms can introduce retransmission delays and circuitous flows via a broker, unsuitable for sub-50 ms requirements.

*That's why, ESP is not a variant of MQTT-SN and is not designed to replace generic messaging protocols. It is tailored to **game synchronization** with minimal overhead, bounded jitter, and soft reliability suited for interactive systems.*

5.7 Implemented Smoothing Technique to Reduce Perceived Jitter

Client perceived jitter fluctuations in the time between displayed game updates—**can negatively** impact user experience even when raw packet loss and latency are within acceptable bounds. To address this, ESP incorporates a *simple client-side **interpolation buffer*** that **smooths display** updates between received server states.

***Smoothing Method*: Time-based Linear Interpolation with Buffering**

Goals:

- Deliver smoother visual updates at a consistent render rate (e.g., 60 FPS) even when server snapshots arrive at an irregular interval (e.g., 20–60 Hz).
- Reduce spikes and jitter in client perception caused by variable network delay.

Buffering Strategy

Each client maintains a small *render buffer* of the last few snapshots received from the server:

1. **Server Update Arrival**
   - Each UPDATES or SNAPSHOT contains a timestamp and snapshot_id.

- The client appends it to a *sorted ring buffer* of recent snapshots B.

2. **Render Timer**
   - The client runs a fixed-rate render loop, e.g., 60 frames per second (~16.7 ms/frame).
   - For each frame, the client selects two adjacent snapshot states from buffer B whose timestamps bracket the *target render time*.
3. **Linear Interpolation**
   - If snapshot(s) are missing for current render time, compute an interpolated grid state:
     - For each cell:

```
interp_value = previous_state + (next_state – previous_state) *
```

     - Where fraction is:

```
(render_time – prev_timestamp) /
(next_timestamp – prev_timestamp)
```

4. **Buffer Management**
   - Only keep a bounded snapshot history (e.g., last 5 server updates).
   - Drop old snapshots beyond buffer capacity.

6. Experimental Evaluation Plan

6.1 Test Scenarios

Scenario	netem Command	Acceptance Criteria
Baseline	None	Server sustains 20 updates/sec per client; average latency ≤50ms; average CPU <60%
Loss 2% (LAN-like)	tc qdisc add dev eth0 root netem loss 2%	Mean perceived error ≤0.5 units; 95th percentile ≤1.5 units; graceful interpolation
Loss 5% (WAN-like)	tc qdisc add dev eth0 root netem loss 5%	Critical events reliably delivered ≥99% within 200ms; system remains stable
Delay 100ms (WAN)	tc qdisc add dev eth0 root netem delay 100ms	Clients continue functioning; redundancy prevents visible misbehavior

6.2 Metrics Collection

Per-Packet Logging (CSV format):

```
client_id,   snapshot_id,
 seq_num,  server_timestamp_ms,
 recv_time_ms,   latency_ms,
jitter_ms,   grid_state,
 cpu_percent,   bandwidth_per_client_kbps
```

Metric Definitions:

Metric	Formula	Unit
Latency	recv_time - server_timestamp	milliseconds
Jitter	|inter_arrival(n) - inter_arrival(n-1)|	milliseconds
Bandwidth	(bytes_received × 8) / (interval_s × 1000)	kbps
CPU Usage	psutil.cpu_percent() sampled per update	percent

Statistical Reporting:

- Mean, median, 95th percentile for latency and jitter
- Bandwidth utilization over 60-second test window
- CPU usage averaged over test duration
- Cell Claim Latency: Time from client EVENT to server confirmation
- State Consistency: Percentage of clients with identical grid state at T+1s intervals

6.3 Automation Scripts

Test Execution Flow:

1. Install Dependencies:
pip install -r requirements.txt
2. Run the server:
python server.py
3.Run the client:
Python client.py
4. Run the ALL Test Cases (Multi-client Simulation):
bash run_all_tests.sh
This script will:

- Install the required libraries
- Run 4 test scenarios: baseline, loss2, loss5, delay100
    - Create test folders in full_run_<timestamp> folder
    - Run test command
    - Start the server
    - Start a client that creates a room with a random name and joins it
    - Start 3 more clients and join the created room (total 4 players required to start)
    - All clients begin clicking cells randomly to simulate gameplay
    - Run the test for 60 seconds
    - Collect raw metrics in .full_run_<timestamp>/{scenario}/results_raw/
    - Generate merged metrics and summarised results in .full_run_<timestamp>/{scenario}/results/
    - Generate performance plots in .full_run_<timestamp>/{scenario}/plots/
    - Collect clients and server logs in .full_run_<timestamp>/{scenario}/logs/
    - Collect PCAP file and logs in .full_run_<timestamp>/{scenario}/pcaps/
    - Collect NETEM list file in .full_run_<timestamp>/{scenario}/netem_list.txt
    - Generate tests comparison plots** in `.full_run_<timestamp>/plots/`

🗁 Output Structure
At the end, you'll get:

```
full_run_<timestamp>/
└── <scenario>/              # e.g., baseline, loss2, loss5, delay100
    ├── pcaps/
    │   └── tcpdump_<scenario>.log
    │   └── <scenario>.pcap
    ├── logs/
    │   ├── client1_stdout.log
    │   ├── client2_stdout.log
    │   ├── client3_stdout.log
    │   ├── client4_stdout.log
    │   └── server_stdout.log
    ├── results_raw/
    │   ├── client_1_metrics.csv
    │   ├── client_2_metrics.csv
    │   ├── client_3_metrics.csv
    │   ├── client_4_metrics.csv
    │   └── server_metrics.csv
    ├── results/
    │   ├── metrics.csv
    │   └── summary.csv
```

```
    ├── plots/
    │   ├── latency_cdf.png
    │   ├── snapshots_per_sec.png
    │   ├── latency_timeseries.png
    │   ├── jitter_timeseries.png
    │   ├── cpu_timeseries.png
    │   ├── bandwidth_timeseries.png
    │   ├── per_client_snapshots.png
    │   └── latency_histogram.png
    └── netem_list.txt
└── plots/
    ├── compare_mean_latency.png
    ├── compare_mean_jitter.png
    ├── compare_mean_error.png
    ├── compare_avg_cpu.png
    ├── compare_bandwidth_vs_updates.png
    └── compare_bandwidth_vs_loss.png
```

7. Example Use Case Walkthrough

7.1 Trace Example:

Player 1 creates a room, and players 2, 3, and 4 join the same room. Each player continuously clicks on random cells, generating activity within the session. After one minute, the server and all clients are automatically terminated. The trace files, including server logs, client logs, and network captures, are generated during this process as a record of the test run.

The following are sample outputs from the generated trace files:

Server:

```
2025-11-27 10:22:20,866 [SERVER] Running (Ctrl+C to stop)
2025-11-27 10:22:22,832 [SERVER] Connected player 1 from ('127.0.0.1', 33041)
2025-11-27 10:22:22,832 [SERVER] Created room 1 named 'Room_157'
2025-11-27 10:22:22,833 [SERVER] Sent join ack for player 1 as local id 1
2025-11-27 10:22:22,834 [SERVER] Player 1 joined room 1 successfully as local id 1
2025-11-27 10:22:25,834 [SERVER] Sent Ignore Event (players < required number of
room players) to ('127.0.0.1', 33041)
2025-11-27 10:22:27,838 [SERVER] Connected player 2 from ('127.0.0.1', 33831)
2025-11-27 10:22:27,839 [SERVER] Sent room list to ('127.0.0.1', 33831)
2025-11-27 10:22:27,840 [SERVER] Sent join ack for player 1 as local id 1
2025-11-27 10:22:27,841 [SERVER] Sent join ack for player 2 as local id 2
2025-11-27 10:22:27,841 [SERVER] Player 2 joined room 1 successfully as local id 2
2025-11-27 10:22:28,341 [SERVER] Connected player 3 from ('127.0.0.1', 57586)
2025-11-27 10:22:28,342 [SERVER] Sent room list to ('127.0.0.1', 57586)
2025-11-27 10:22:28,344 [SERVER] Sent join ack for player 1 as local id 1
2025-11-27 10:22:28,344 [SERVER] Sent join ack for player 2 as local id 2
2025-11-27 10:22:28,344 [SERVER] Sent join ack for player 3 as local id 3
2025-11-27 10:22:28,345 [SERVER] Player 3 joined room 1 successfully as local id 3
2025-11-27 10:22:28,841 [SERVER] Sent Ignore Event (players < required number of
room players) to ('127.0.0.1', 33041)
2025-11-27 10:22:28,847 [SERVER] Connected player 4 from ('127.0.0.1', 39149)
2025-11-27 10:22:28,848 [SERVER] Sent room list to ('127.0.0.1', 39149)
2025-11-27 10:22:28,849 [SERVER] Sent join ack for player 1 as local id 1
2025-11-27 10:22:28,850 [SERVER] Sent join ack for player 2 as local id 2
2025-11-27 10:22:28,850 [SERVER] Sent join ack for player 3 as local id 3
2025-11-27 10:22:28,850 [SERVER] Sent join ack for player 4 as local id 4
2025-11-27 10:22:28,851 [SERVER] Player 4 joined room 1 successfully as local id 4
# AFTER ROOM IS READY SERVER STARTS SENDING UPDATES
2025-11-27 10:22:28,855 [SERVER] Updates Sent Player ID:1, Seq_num:11
2025-11-27 10:22:28,856 [SERVER] Updates Sent Player ID:2, Seq_num:8
2025-11-27 10:22:28,856 [SERVER] Updates Sent Player ID:3, Seq_num:7
2025-11-27 10:22:28,857 [SERVER] Updates Sent Player ID:4, Seq_num:6
.
.
.
# BROADCAST EVENT TO ROOM PLAYERS
2025-11-27 10:22:30,845 [SERVER] Sent Event (Type: 0, Room (ID:1, Name:
Room_157), Player local id:2, Cell index:383) to ('127.0.0.1', 33041)
2025-11-27 10:22:30,845 [SERVER] Sent Event (Type: 0, Room (ID:1, Name:
Room_157), Player local id:2, Cell index:383) to ('127.0.0.1', 33831)
2025-11-27 10:22:30,846 [SERVER] Sent Event (Type: 0, Room (ID:1, Name:
Room_157), Player local id:2, Cell index:383) to ('127.0.0.1', 57586)
2025-11-27 10:22:30,846 [SERVER] Sent Event (Type: 0, Room (ID:1, Name:
Room_157), Player local id:2, Cell index:383) to ('127.0.0.1', 39149)
2025-11-27 10:22:30,862 [SERVER] Updates Sent Player ID:1, Seq_num:53
2025-11-27 10:22:30,863 [SERVER] Updates Sent Player ID:2, Seq_num:50
2025-11-27 10:22:30,863 [SERVER] Updates Sent Player ID:3, Seq_num:49
2025-11-27 10:22:30,864 [SERVER] Updates Sent Player ID:4, Seq_num:48
.
.
.
2025-11-27 10:23:20,867 [SERVER] Test duration ended, server stopped
```

Player 1:

```
2025-11-27 10:22:22,831 [Client] Sending INIT
2025-11-27 10:22:22,832 [Client] Creating room: Room_157
2025-11-27 10:22:22,832 [Client] Running (Ctrl+C to stop)
2025-11-27 10:22:22,832 [Client] Got player_id = 1
2025-11-27 10:22:22,833 [Client] Room created -> id 1
2025-11-27 10:22:22,833 [Client] Joining room 1
2025-11-27 10:22:22,834 [Client] Joined room 1 as local id 1
2025-11-27 10:22:22,834 [Client] Room players: {1: (1, (109, 193, 114))}
2025-11-27 10:22:22,834 [Client] Snapshot #0 seq #4 received & ACKed
2025-11-27 10:22:25,833 [Client] Cell 95 → PENDING (ownership requested)
2025-11-27 10:22:28,841 [Client] Cell 203 → PENDING (ownership requested)
2025-11-27 10:22:30,845 [Client] Cell 383 CONFIRMED for player 2
2025-11-27 10:22:30,845 [Client] Cell 383 CONFIRMED for player 2
2025-11-27 10:22:30,846 [Client] Cell 383 CONFIRMED for player 2
2025-11-27 10:22:31,349 [Client] Cell 328 CONFIRMED for player 3
2025-11-27 10:22:31,350 [Client] Cell 328 CONFIRMED for player 3
2025-11-27 10:22:31,350 [Client] Cell 328 CONFIRMED for player 3
2025-11-27 10:22:31,843 [Client] Cell 270 → PENDING (ownership requested)
2025-11-27 10:22:31,846 [Client] Cell 270 CONFIRMED for you
2025-11-27 10:22:31,847 [Client] Cell 270 CONFIRMED for you
2025-11-27 10:22:31,847 [Client] Cell 270 CONFIRMED for you
2025-11-27 10:22:31,851 [Client] Cell 380 CONFIRMED for player 4
2025-11-27 10:22:31,851 [Client] Cell 380 CONFIRMED for player 4
2025-11-27 10:22:31,851 [Client] Cell 380 CONFIRMED for player 4
2025-11-27 10:22:33,853 [Client] Cell 384 CONFIRMED for player 2
2025-11-27 10:22:33,854 [Client] Cell 384 CONFIRMED for player 2
2025-11-27 10:22:33,854 [Client] Cell 384 CONFIRMED for player 2
.
.
.
2025-11-27 10:23:22,834 [Client] Test duration ended, client stopped
2025-11-27 10:23:22,835 [Client] Disconnecting...
```

Player 2:

```
2025-11-27 10:22:27,837 [Client] Sending INIT
2025-11-27 10:22:27,838 [Client] Requesting room list
2025-11-27 10:22:27,838 [Client] Running (Ctrl+C to stop)
2025-11-27 10:22:27,839 [Client] Got player_id = 2
2025-11-27 10:22:27,839 [Client] Available Rooms:
2025-11-27 10:22:27,839  - 1: Room_157 (1 players)
2025-11-27 10:22:27,839 [Client] Auto-joining room 1
2025-11-27 10:22:27,840 [Client] Joining room 1
2025-11-27 10:22:27,841 [Client] Joined room 1 as local id 2
2025-11-27 10:22:27,841 [Client] Room players: {1: (1, (109, 193, 114)), 2: (2, (53, 200, 106))}
2025-11-27 10:22:27,842 [Client] Snapshot #0 seq #4 received & ACKed
2025-11-27 10:22:30,844 [Client] Cell 383 → PENDING (ownership requested)
2025-11-27 10:22:30,845 [Client] Cell 383 CONFIRMED for you
2025-11-27 10:22:30,846 [Client] Cell 383 CONFIRMED for you
2025-11-27 10:22:30,846 [Client] Cell 383 CONFIRMED for you
2025-11-27 10:22:31,350 [Client] Cell 328 CONFIRMED for player 3
2025-11-27 10:22:31,350 [Client] Cell 328 CONFIRMED for player 3
2025-11-27 10:22:31,350 [Client] Cell 328 CONFIRMED for player 3
2025-11-27 10:22:31,847 [Client] Cell 270 CONFIRMED for player 1
2025-11-27 10:22:31,847 [Client] Cell 270 CONFIRMED for player 1
2025-11-27 10:22:31,848 [Client] Cell 270 CONFIRMED for player 1
.
.
.
2025-11-27 10:23:27,846 [Client] Test duration ended, client stopped
2025-11-27 10:23:27,846 [Client] Disconnecting...
```

Player 3:

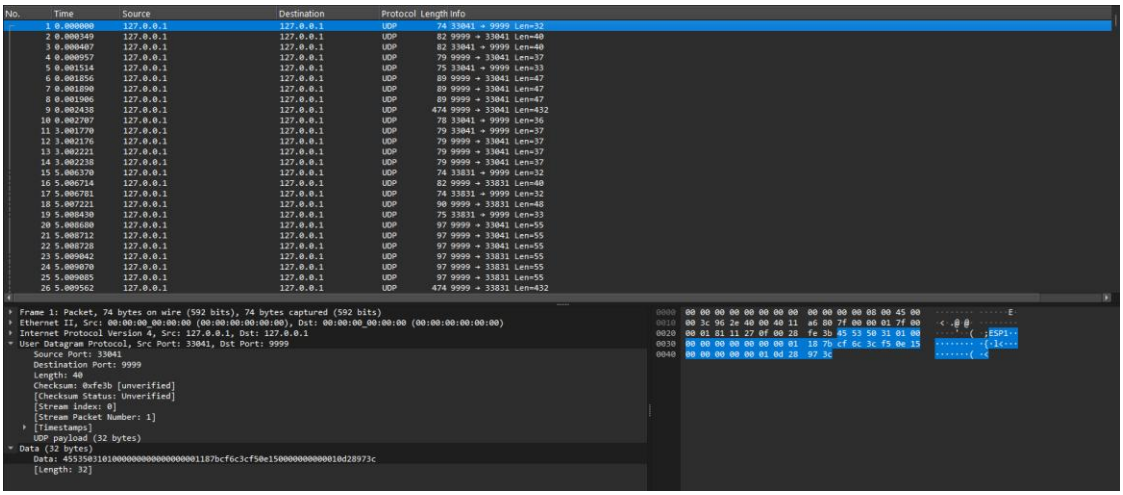
```
2025-11-27 10:22:28,341 [Client] Sending INIT
2025-11-27 10:22:28,341 [Client] Requesting room list
2025-11-27 10:22:28,342 [Client] Running (Ctrl+C to stop)
2025-11-27 10:22:28,342 [Client] Got player_id = 3
2025-11-27 10:22:28,342 [Client] Available Rooms:
2025-11-27 10:22:28,343  - 1: Room_157 (2 players)
2025-11-27 10:22:28,343 [Client] Auto-joining room 1
2025-11-27 10:22:28,343 [Client] Joining room 1
2025-11-27 10:22:28,344 [Client] Joined room 1 as local id 3
2025-11-27 10:22:28,345 [Client] Room players: {1: (1, (109, 193, 114)), 2: (2, (53, 200, 106)), 3: (3,
(79, 160, 172))}
2025-11-27 10:22:28,345 [Client] Snapshot #0 seq #4 received & ACKed
2025-11-27 10:22:30,846 [Client] Cell 383 CONFIRMED for player 2
2025-11-27 10:22:30,846 [Client] Cell 383 CONFIRMED for player 2
2025-11-27 10:22:30,847 [Client] Cell 383 CONFIRMED for player 2
2025-11-27 10:22:31,348 [Client] Cell 328 → PENDING (ownership requested)
2025-11-27 10:22:31,350 [Client] Cell 328 CONFIRMED for you
2025-11-27 10:22:31,351 [Client] Cell 328 CONFIRMED for you
2025-11-27 10:22:31,351 [Client] Cell 328 CONFIRMED for you
2025-11-27 10:22:31,847 [Client] Cell 270 CONFIRMED for player 1
2025-11-27 10:22:31,848 [Client] Cell 270 CONFIRMED for player 1
2025-11-27 10:22:31,848 [Client] Cell 270 CONFIRMED for player 1
.
.
.
2025-11-27 10:23:28,348 [Client] Test duration ended, client stopped
2025-11-27 10:23:28,348 [Client] Disconnecting...
```

Player 4:

```
2025-11-27 10:22:28,846 [Client] Sending INIT
2025-11-27 10:22:28,847 [Client] Requesting room list
2025-11-27 10:22:28,847 [Client] Running (Ctrl+C to stop)
2025-11-27 10:22:28,848 [Client] Got player_id = 4
2025-11-27 10:22:28,848 [Client] Available Rooms:
2025-11-27 10:22:28,848  - 1: Room_157 (3 players)
2025-11-27 10:22:28,848 [Client] Auto-joining room 1
2025-11-27 10:22:28,849 [Client] Joining room 1
2025-11-27 10:22:28,851 [Client] Joined room 1 as local id 4
2025-11-27 10:22:28,851 [Client] Room players: {1: (1, (109, 193, 114)), 2: (2, (53, 200, 106)), 3: (3, (79, 160,
172)), 4: (4, (217, 183, 204))}
2025-11-27 10:22:28,851 [Client] Snapshot #0 seq #4 received & ACKed
2025-11-27 10:22:30,846 [Client] Cell 383 CONFIRMED for player 2
2025-11-27 10:22:30,847 [Client] Cell 383 CONFIRMED for player 2
2025-11-27 10:22:30,847 [Client] Cell 383 CONFIRMED for player 2
2025-11-27 10:22:31,351 [Client] Cell 328 CONFIRMED for player 3
2025-11-27 10:22:31,351 [Client] Cell 328 CONFIRMED for player 3
2025-11-27 10:22:31,351 [Client] Cell 328 CONFIRMED for player 3
2025-11-27 10:22:31,848 [Client] Cell 270 CONFIRMED for player 1
2025-11-27 10:22:31,848 [Client] Cell 270 CONFIRMED for player 1
2025-11-27 10:22:31,848 [Client] Cell 270 CONFIRMED for player 1
2025-11-27 10:22:31,849 [Client] Cell 380 → PENDING (ownership requested)
2025-11-27 10:22:31,852 [Client] Cell 380 CONFIRMED for you
2025-11-27 10:22:31,852 [Client] Cell 380 CONFIRMED for you
2025-11-27 10:22:31,853 [Client] Cell 380 CONFIRMED for you
.

.
2025-11-27 10:23:28,850 [Client] Test duration ended, client stopped
2025-11-27 10:23:28,850 [Client] Disconnecting...
```

7.2PCAP file :



7.3PCAP Explanation:

Port Allocation Table:

Entity	Port	IP Address
Server	9999	127.0.0.1
Player 1	33041	127.0.0.1
Player 2	33831	127.0.0.1
Player 3	57586	127.0.0.1
Player 4	39149	127.0.0.1

Key Packet Flow Analysis:

Packet #	Time (s)	Source Port	Dest Port	Length	Message Type	Description	Corresponding Log Entry
1	0.000000	33041	9999	32	INIT	Player 1 initiates connection	[Client] Sending INIT
2	0.000349	9999	33041	40	INIT_ACK	Server assigns player_id=1	[SERVER] Connected player 1
3	0.000407	33041	9999	40	CREATE_ROOM	Player 1 creates "Room_157"	[Client] Creating room: Room_157
4	0.000957	9999	33041	37	ROOM_CREATED	Server confirms room creation	[SERVER] Created room 1
5	0.001514	33041	9999	33	JOIN_ROOM	Player 1 joins room 1	[Client] Joining room 1
6-8	0.001856 - 0.001906	9999	33041	47	JOIN_ACK	Join acknowledgments	[SERVER] Sent join ack for player 1
9	0.002438	9999	33041	432	SNAPSHOT	Initial game state snapshot #0	[Client] Snapshot #0 seq #4 received

10-14	0.002707 - 0.002238	33041	9999	33-37	ACK	Player 1 acknowledges messages	Client processing
15	5.006370	33831	9999	32	INIT	Player 2 initiates connection	[Client] Sending INIT (Player 2)
16	5.006714	9999	33831	40	INIT_ACK	Server assigns player_id=2	[SERVER] Connected player 2
17	5.006781	33831	9999	32	LIST_ROOMS	Player 2 requests room list	[Client] Requesting room list
18	5.007221	9999	33831	48	ROOM_LIST	Server sends available rooms	[SERVER] Sent room list
19-22	5.008430 - 5.008728	9999	33041/ 33831	33-55	JOIN_UPDATE	Server broadcasts Player 2 join	[SERVER] Sent join ack for player 2
23-25	5.009042 - 5.009085	9999	33831	55	JOIN_ACK	Join acknowledgments to Player 2	[Client] Joined room 1 as local id 2
26	5.009562	9999	33831	432	SNAPSHOT	Full game state to Player 2	[Client] Snapshot #0 seq #4 (Player 2)

8. Limitations & Future Work

8.1 Current Limitations

1. No Congestion Control:

-Fixed 20.7 Hz broadcast rate regardless of network conditions

-May overwhelm clients on constrained networks

- Does not adapt to increasing RTT or loss rates

2. Fixed Retransmission Timeout:

- 100ms RTO does not adapt to measured RTT variations

- May cause premature retransmissions on high

3. No Security:

- No authentication or integrity protection beyond CRC32

- Vulnerable to replay attacks, packet injection, and man-in-the-middle

8.2 Future Work

1. Adaptive Update Rate (Smart Speed Control):

Making the server adjust its speed based on how well clients are keeping up:

```
Check each client's network quality every second:
  - Measure their round-trip time (how long packets take)
  - Count how many packets they're missing

If a client is struggling (high latency OR losing many packets):
  → Slow down to 10 updates per second for that client
  → Send larger snapshots less frequently

If a client has excellent connection (low latency AND no loss):
  → Speed up to 30 updates per second
  → Give them smoother, more responsive gameplay
```

2. Dynamic Retransmission Timer (Karn's Algorithm):

Learn from each successful packet delivery to calculate the perfect timeout**:**

```
Every time a client acknowledges a packet:
  1. Measure how long it took (sampleRTT)
  2. Update our running average:
     estimatedRTT = 87.5% × old_estimate + 12.5% × new_sample
  3. Track how much the times vary (deviation)
  4. Calculate smart timeout:
     RTO = estimatedRTT + (4 × deviation)

Example:
  - Local network (5ms average) → RTO becomes ~25ms
  - Internet (80ms average) → RTO becomes ~150ms
```

Will lead to: Faster retransmission on good networks, fewer unnecessary retries on slow networks.

3. Security Enhancements:

Packet Authentication (Prove packets are real): Adding a cryptographic signature to each packet

```
- Client and server agree on a shared secret key using Diffie-Hellman
For every packet sent:
1.Calculate authentication code:
                    HMAC = special_hash(secret_key + packet_data)
2. Attach this 32-byte "signature" to the packet When receiving:
 - Recalculate HMAC and compare
 - If they don't match → reject the packet (it's fake or tampered)
```

Cost: Extra 32 bytes per packet (~7% overhead for typical packets)

Benefit: Prevents packet forgery and tampering. An attacker can't create fake packets without the secret key.

9. References

RFC 768 – User Datagram Protocol (UDP)
https://www.rfc-editor.org/rfc/rfc768

RFC 1071 – Computing the Internet Checksum
https://www.rfc-editor.org/rfc/rfc1071

RFC 6298 – Computing TCP's Retransmission Timer
https://www.rfc-editor.org/rfc/rfc6298
(Referenced for Karn's algorithm and RTO calculation)

Valve Developer Community – Source Multiplayer Networking
https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking
(Industry best practices for game state synchronization)

Gaffer on Games – Reliability and Flow Control
https://gafferongames.com/post/reliability_and_flow_control/
(Authoritative guide on UDP reliability for games)

Python struct module – Binary Data Handling
https://docs.python.org/3/library/struct.html
(Reference for binary packing format strings)

CRC32 Algorithm – Cyclic Redundancy Check
https://en.wikipedia.org/wiki/Cyclic_redundancy_check
(Checksum algorithm specification)