CSE361 (UG2023) – Computer Networking

Mini-RFC

12/12/2025

<u>Submitted by:</u>

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

Multiplayer Game State Synchronization

Phase 1

Section 1:

Introduction

The Violet Ascending Protocol (VAP-1) is a small and efficient network protocol built on UDP. It is designed to keep multiple players synchronized with a game server in real time. The main goal is to share player positions and short game events quickly, even if some packets are lost on the way.

VAP-1 focuses on speed and low latency, which are more important for real-time games than perfect reliability. It lets players see updates almost instantly instead of waiting for packet retransmissions like in TCP.

Use Case

This protocol is used for a simple multiplayer environment called *Grid Clash*. In this setup, several players connect to one central server and play on a shared 2D grid. The server keeps track of the full game state (like which cells are claimed and each player's score) and sends frequent updates to all clients so that everyone sees the same grid.

Why a New Protocol Is Needed

Existing protocols don't fit well for this kind of real-time communication.

- TCP is reliable but too slow for live updates because it waits for lost packets to be resent.
- UDP is faster but doesn't define how to structure or manage messages for games.

VAP-1 uses UDP for speed but adds its own structure to make communication organized and consistent. It defines message types like JOIN, READY, and SNAPSHOT, includes small headers for each message, and adds simple reliability features such as sending the latest and previous snapshots together so the client can recover if one is lost.

Assumptions and Constraints

- All messages use UDP sockets.
- One server handles all connected clients.
- The maximum packet size is about 1200 bytes to avoid fragmentation.
- The server sends snapshots around every 50 milliseconds.
- The system can handle 5**–**10**%** packet loss without major issues.
- The game state is small enough to fit into a single packet.
- Clients assume the server is disconnected if no update is received for 2 seconds.

Section 2:

Protocol Architecture

Entities:

Server: authoritative game process that accepts joins, tracks players, and broadcasts snapshots. Implemented as the GameServer class with states WAITING_FOR_JOIN, WAITING_FOR_INIT, GAME_LOOP, GAME_OVER.

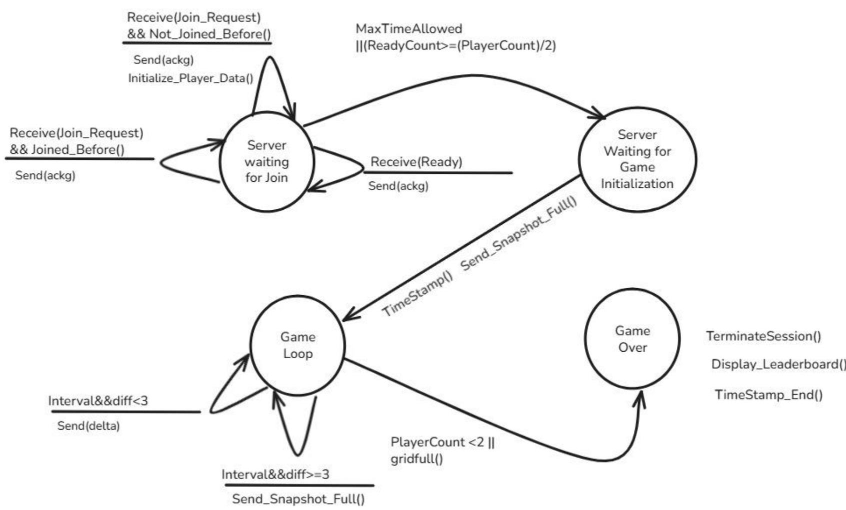
Client: a player instance that sends JOIN and READY messages and applies SNAPSHOT updates. A client runs a UDP socket and follows the handshake then listens for snapshots.

Sequence Flow:

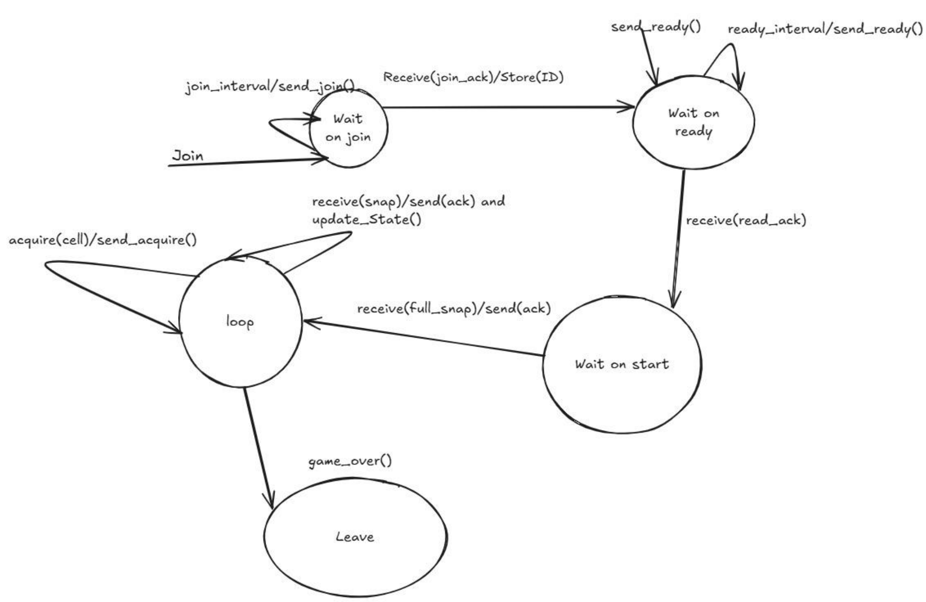
Message Type Name	Direction
MSG_JOIN_REQ	Client → Server
MSG_JOIN_ACK	Server → Client

MSG_READY_REQ	Client → Server
MSG_READY_ACK	Server → Client
MSG_START_GAME	Server → Client
MSG_SNAPSHOT_FULL	Server → Client
MSG_SNAPSHOT_DELTA	Server → Client
MSG_SNAPSHOT_ACK	Client → Server
MSG_ACQUIRE_EVENT	Client → Server
MSG_END_GAME	Server → Client
MSG_TERMINATE	Server → Client
MSG_LEADERBOARD	Server → Client
MSG_ACQUIRE_ACK	Server→ Client

Finite State Machine For Server:



Finite State Machine For Client:



Section 3:

Message Formats:

Field	Size(Bits)	Description
protocol_id	32	4-byte ASCII identifier unique to your protocol(VAP)
version	8	Protocol version
msg_type	8	Message type (JOIN, READY, SNAPSHOT
snapshot_id	32	Snapshot identifier used for ordering and redundancy
seq_num	32	Sequence number for tracking message order
timestamp	64	Server-side timestamp (seconds since start using monotonic clock)
payload_len	16	Length of the variable-size payload (in bytes)

Total Header Size: 24 Bytes or 192 Bits

The Header Format is: !4s B B I I d H

!: big-endian format

4s: array of char(String)

B: unsigned char

I: unsigned int

d: double

H: unsigned short

Message Type Name	**Value**	**Description**
MSG_JOIN_REQ	1	Client asks to join the game
MSG_JOIN_ACK	2	Server accepts the join and assigns player ID
MSG_READY_REQ	3	Client signals readiness
MSG_READY_ACK	4	Server confirms client is ready
MSG_START_GAME	5	Server announces game start
MSG_SNAPSHOT_FULL	6	Server sends full snapshot
MSG_SNAPSHOT_DELTA	7	Server sends delta snapshot
MSG_SNAPSHOT_ACK	8	Client returns the snapshot acknowledgement
MSG_ACQUIRE_EVENT	9	Client sends a game action (e.g., claiming a grid cell)
MSG_END_GAME	10	Server signals the end of the match

MSG_TERMINATE	11	Server closes the session
MSG_LEADERBOARD	12	Final scores and results broadcast
MSG_ACQUIRE_ACK	13	Confirms that the server successfully received a player’s cell-capture attempt.

Section 4:

1.Join Phase (Session Start)

Client:

- Repeatedly sends JOIN_REQ every 0.25s until JOIN_ACK.
- On JOIN_ACK → stores player_id → moves to WAIT_FOR_READY.

Server:

- On new JOIN_REQ → create player entry + send JOIN_ACK.
- On duplicate JOIN_REQ → resend same JOIN_ACK.

2. Ready Phase

Client:

- Sends READY_REQ until receiving READY_ACK.
- On READY_ACK → moves to WAIT_FOR_STARTGAME.

Server:

- On READY_REQ → mark player ready + send READY_ACK.
- When all 4 players are ready → transition to INIT.

3. Initial Snapshot (Start Game)

Server:

- Builds full snapshot (snapshot_id=0).
- Sends SNAPSHOT_FULL to all clients → enters GAME_LOOP.

Client:

- Waits for full snapshot → applies it → sends SNAPSHOT_ACK.
- Enters IN_GAME_LOOP.

4. Game Loop (Normal Data Exchange)

Server:

- Sends periodic snapshots (~25 FPS):
  - SNAPSHOT_DELTA for small changes
  - SNAPSHOT_FULL if too many changes
- Handles:
  - SNAPSHOT_ACK
  - ACQUIRE_EVENT → updates grid + scores → sends ACQUIRE_ACK.

Client:

- Applies snapshots (ignores old ones).
- Sends SNAPSHOT_ACK.
- Sends ACQUIRE_EVENT for capture attempts (resend on timeout).
- Enters GAME_OVER on receiving LEADERBOARD.
- ## 5. Error Recovery

  We send the snapshots continuously from the server to all clients such that any errors can be overwritten till the clients and the server become fully in sync.

6. Game Over (Shutdown)

Server:

- Sends LEADERBOARD → enters GAME_OVER → waits for END_GAME.

Client:

- Receives leaderboard → prints results → sends END_GAME.
- Closes socket and stops the FSM.

Section 5:

The network timing parameters were chosen to balance responsiveness, reliability, and efficiency in the game's client–server communication. The system runs at a 50 ms tick rate (TICK = 0.05), providing smooth real-time updates without overloading the CPU.
JOIN_RESEND and READY_RESEND are each set to 250 ms to ensure that critical connection messages are resent to overcome packet loss but not so frequently that they create unnecessary network traffic.
ACQUIRE_RESEND was previously set to 150 ms but has been decreased to 60 ms because the acquire/lock step during game start is highly time-sensitive, and reducing the interval allows faster recovery from dropped packets and smoother synchronization between players.
Finally, START_TIMEOUT is set to 2 seconds to prevent the client from waiting indefinitely during initialization while still allowing enough time to handle network jitter. Together, these values create a stable and responsive communication flow suitable for low-latency gameplay

Retransmission Improvements in the Game Server

In the new version of the game server, several improvements were implemented to enhance the reliability of communication between clients and the server. These changes focus on critical events such as player actions, game-over notifications, and leaderboard delivery. The primary goal is to ensure that all important messages are successfully received by the clients, even in the presence of packet loss.

Player Actions (Acquire Events):

In the old server, when a client sent an acquire event to claim a cell, the server would update the grid and the player's score but did not send any acknowledgment back to the client. This could result in the client being unaware of whether the action was successfully registered, potentially causing repeated actions or inconsistencies. In the new server, the server sends an explicit acknowledgment (MSG_ACQUIRE_ACK) to the client for each acquire event. This allows the client to confirm that the server received the action, reducing unnecessary retransmissions and improving reliability.

Game Over Acknowledgment:

Previously, the server ignored any client messages during the game-over state. As a result, clients had no way of confirming that the server had registered the end of the game. The new server handles MSG_END_GAME messages from clients, removing acknowledged players from the active list. This ensures that clients properly receive confirmation of the game's conclusion and prevents dangling connections or repeated game-over messages.

Leaderboard Delivery:

In the old server, the final leaderboard was sent only once. If the packet was lost, clients would never receive the results. The new server repeatedly retransmits the leaderboard for a short period (e.g., 3 seconds) until all clients acknowledge receipt or the time expires. During this period, the server continues to process network events, allowing late or delayed clients to receive the leaderboard. This mechanism ensures reliable delivery of final scores and player rankings.

CPU and Packet Monitoring:

The new server also logs CPU usage and packet timestamps during snapshot broadcasts.

While this does not directly implement retransmission, it allows the server operator to monitor system load and timing, which can affect packet delivery. This information helps identify delays or potential issues that could trigger retransmissions.

Snapshot Handling:

The snapshot mechanism (sending full or delta updates of the game grid) was retained, but the retransmission logic was refined. Now, if a client is behind by one or more snapshots, the server can send the combined missed deltas instead of a full snapshot, allowing clients to resynchronize efficiently.

Section 6:

Baselines:

- **Local baseline:** Run the server and four clients on the same machine or local network without any artificial network impairment. This measures the ideal system performance under minimal latency and zero packet loss.
- **Network-impaired baselines:** Introduce controlled packet loss or latency to simulate real-world network conditions. This allows evaluation of system robustness and event reliability.

Metrics

1. **Latency:** Time between server sending a snapshot and client receiving it (ms).
2. **Jitter:** Variation in latency between consecutive snapshots (ms).
3. **Update rate:** Number of snapshots received per second by clients (Hz).
4. **Perceived position error:** Euclidean distance between server-reported and client-observed positions.
5. **Bandwidth:** Average per-client network usage (kbps).
6. **CPU usage:** Server utilization (%) during the game session.
7. **Event reliability:** Percentage of critical events acknowledged within 200 ms.

Measurement Methods

- **Logging:** Server and clients print timestamped events, snapshots, and acknowledgments to log files.
- **Metrics extraction:** Scripts parse logs to compute latency, jitter, update rate, bandwidth, and position error.
- **Statistical analysis:** Compute mean, median, 95th percentile for each metric.
- **Plotting:** Generate time series plots of latency, jitter histograms, and position error trends.

Network Condition Simulation

- 2% packet loss: sudo tc qdisc add dev lo root netem loss 2%
- 5% packet loss: sudo tc qdisc add dev lo root netem loss 5%
- 100ms delay: sudo tc qdisc add dev lo root netem delay 100ms
- Reset rules: sudo tc qdisc del dev lo root

Automation Scripts

- Sudo ./run_all_tests.sh baseline
- Sudo ./run_all_tests.sh loss2
- Sudo ./run_all_tests.sh loss5
- Sudo ./run_all_tests.sh delay100

Section 7:

Scenario: A client joins the game, moves to a cell (action), receives a snapshot, and the game ends.

1	0.000000000	127.0.0.1	127.0.0.1	UDP	66	58908 → 8888 Len=24
2	0.000455421	127.0.0.1	127.0.0.1	UDP	82	8888 → 58908 Len=40
3	0.000475558	127.0.0.1	127.0.0.1	UDP	66	8888 → 58908 Len=24
4	0.002872688	127.0.0.1	127.0.0.1	UDP	66	8888 → 58908 Len=24
5	0.004226881	127.0.0.1	127.0.0.1	UDP	66	58908 → 8888 Len=24
6	0.004759677	127.0.0.1	127.0.0.1	UDP	66	8888 → 58908 Len=24
7	0.006438270	127.0.0.1	127.0.0.1	UDP	66	58908 → 8888 Len=24
8	0.006485099	127.0.0.1	127.0.0.1	UDP	66	8888 → 58908 Len=24
30	1.507825009	127.0.0.1	127.0.0.1	UDP	1367	8888 → 58908 Len=1325
31	1.508000648	127.0.0.1	127.0.0.1	UDP	1367	8888 → 53372 Len=1325
32	1.508234213	127.0.0.1	127.0.0.1	UDP	1367	8888 → 49281 Len=1325
33	1.508486974	127.0.0.1	127.0.0.1	UDP	1367	8888 → 44922 Len=1325
34	1.508565041	127.0.0.1	127.0.0.1	UDP	66	58908 → 8888 Len=24
35	1.509080091	127.0.0.1	127.0.0.1	UDP	66	49281 → 8888 Len=24
36	1.509143804	127.0.0.1	127.0.0.1	UDP	66	53372 → 8888 Len=24
37	1.509879442	127.0.0.1	127.0.0.1	UDP	66	44922 → 8888 Len=24
38	1.513980877	127.0.0.1	127.0.0.1	UDP	82	53372 → 8888 Len=40
39	1.514059204	127.0.0.1	127.0.0.1	UDP	82	8888 → 53372 Len=40
40	1.526048051	127.0.0.1	127.0.0.1	UDP	83	44922 → 8888 Len=41
41	1.527261147	127.0.0.1	127.0.0.1	UDP	83	8888 → 44922 Len=41
42	1.536902396	127.0.0.1	127.0.0.1	UDP	83	44922 → 8888 Len=41
43	1.538046423	127.0.0.1	127.0.0.1	UDP	83	8888 → 44922 Len=41
44	1.539893082	127.0.0.1	127.0.0.1	UDP	84	53372 → 8888 Len=42
45	1.540033915	127.0.0.1	127.0.0.1	UDP	84	8888 → 53372 Len=42
46	1.550215933	127.0.0.1	127.0.0.1	UDP	145	8888 → 58908 Len=103
47	1.550441699	127.0.0.1	127.0.0.1	UDP	145	8888 → 53372 Len=103
48	1.550663492	127.0.0.1	127.0.0.1	UDP	145	8888 → 49281 Len=103
49	1.550995076	127.0.0.1	127.0.0.1	UDP	145	8888 → 44922 Len=103
50	1.551696964	127.0.0.1	127.0.0.1	UDP	66	58908 → 8888 Len=24
51	1.552195213	127.0.0.1	127.0.0.1	UDP	66	53372 → 8888 Len=24
52	1.552199892	127.0.0.1	127.0.0.1	UDP	66	49281 → 8888 Len=24
53	1.552479786	127.0.0.1	127.0.0.1	UDP	66	44922 → 8888 Len=24
54	1.590436359	127.0.0.1	127.0.0.1	UDP	99	8888 → 58908 Len=57
55	1.590684745	127.0.0.1	127.0.0.1	UDP	99	8888 → 53372 Len=57
56	1.590960498	127.0.0.1	127.0.0.1	UDP	99	8888 → 49281 Len=57
16235	80.613746699	127.0.0.1	127.0.0.1	UDP	66	44922 → 8888 Len=24
16236	80.637132473	127.0.0.1	127.0.0.1	UDP	84	53372 → 8888 Len=42
16237	80.637653330	127.0.0.1	127.0.0.1	UDP	84	8888 → 53372 Len=42
16238	80.642444868	127.0.0.1	127.0.0.1	UDP	271	8888 → 49281 Len=229
16239	80.642691872	127.0.0.1	127.0.0.1	UDP	271	8888 → 58908 Len=229
16240	80.642940835	127.0.0.1	127.0.0.1	UDP	271	8888 → 44922 Len=229
16241	80.643337835	127.0.0.1	127.0.0.1	UDP	271	8888 → 53372 Len=229
16242	80.646157220	127.0.0.1	127.0.0.1	UDP	271	8888 → 49281 Len=229
16243	80.646376212	127.0.0.1	127.0.0.1	UDP	271	8888 → 58908 Len=229
16244	80.646578510	127.0.0.1	127.0.0.1	UDP	271	8888 → 44922 Len=229
16245	80.646594909	127.0.0.1	127.0.0.1	UDP	69	49281 → 8888 Len=27
16246	80.646792980	127.0.0.1	127.0.0.1	UDP	271	8888 → 53372 Len=229
16247	80.647294376	127.0.0.1	127.0.0.1	UDP	69	58908 → 8888 Len=27
16248	80.647526945	127.0.0.1	127.0.0.1	UDP	69	53372 → 8888 Len=27
16249	80.647667042	127.0.0.1	127.0.0.1	UDP	69	44922 → 8888 Len=27
16250	80.755140787	127.0.0.1	127.0.0.1	UDP	228	8888 → 58908 Len=186
16251	80.755341776	127.0.0.1	127.0.0.1	UDP	228	8888 → 44922 Len=186
16252	80.755518355	127.0.0.1	127.0.0.1	UDP	228	8888 → 53372 Len=186

Scenario for player 1 at port 58908

Time	Action
0	Client sends MSG_JOIN_REQ to server
0.00045	Server sends MSG_JOIN_ACK to client
0.0044	Client sends MSG_READY_REQ to server
0.0047	Server sends MSG_READY_ACK to client
1.507	Server sends initial Full Snapshot to call clients
1.5085	Client sends MSG_SNAPSHOT_ACK to server
1.55	Client sends MSG_ACQUIRE_EVENT to server
1.59	Server sends Delta Snapshot to client
80.642	server sends MSG_LEADERBOARD to client
80.647	Client sends MSG_END_GAME to server