NetRush Protocol Mini-RFC

Protocol: NetRush v1.0 | **Signature:** NRSH | **Date:** December 24, 2025

---

1. Introduction

NetRush is a UDP-based real-time multiplayer game synchronization protocol for Grid Clash, a competitive grid capture game where 2-8 players claim cells on a 5×5 grid.

Why a new protocol?

- TCP: Head-of-line blocking unacceptable for real-time gameplay
- MQTT: Pub/sub model unsuitable for authoritative server architecture
- Existing protocols (ENet, RakNet): Prohibited by assignment requirements

Design constraints:

- Max packet: 1200 bytes (avoids fragmentation)
- Update rate: 20 Hz (50ms intervals)
- Acceptable loss: 0-30% for state updates
- Transport: UDP over IPv4
- No external game networking libraries

---

2. Protocol Architecture

Client-Server Model:

Entity	Role
Server	Authoritative state; processes claims; broadcasts 20 Hz snapshots
Client	Sends claim events; renders state with prediction/smoothing

Communication Flow:

```
Client           Server           Client
  |--INIT---------->|                 |
  |<-INIT_ACK(ID)---|                 |
  |                 |<--INIT--------------|
  |                 |---INIT_ACK(ID)----->|
  |--EVENT(claim)-->|                 |
  |<-ACK(confirm)---|                 |
  |<-SNAPSHOT-------|---SNAPSHOT--------->|
  (repeats 20 Hz)
  |<-GAME_OVER------|---GAME_OVER-------->|
```

Finite State Machines:

Client: DISCONNECTED → CONNECTING (send INIT, retry on timeout) → PLAYING (recv snapshots, send events) → GAME_OVER

Server per client: IDLE → CONNECTED (recv INIT, assign ID, broadcast @ 20Hz) → GAME_OVER (grid full, send winners 3×)

---

3. Message Formats

3.1 Binary Header (28 bytes)

Field	Offset	Size	Type	Description
protocol_id	0	4B	ASCII	"NRSH" signature
version	4	1B	uint8	Protocol version (1)
msg_type	5	1B	uint8	Message type (see §3.2)
snapshot_id	6	4B	uint32	Snapshot/sequence ID
seq_num	10	4B	uint32	Packet sequence number
timestamp_ms	14	8B	uint64	Unix epoch milliseconds
payload_len	22	2B	uint16	Payload bytes (max 1172)
checksum	24	4B	uint32	CRC32 of header+payload

Struct format: `!4sBBIIQHI` (network byte order)

3.2 Message Types

Value	Name	Direction	Purpose
0	INIT	C→S	Connect/keep-alive
1	INIT_ACK	S→C	Assign player ID
2	SNAPSHOT	S→C	State update (full/delta)
3	EVENT	C→S	Critical action (claim)
4	ACK	S→C	Confirm EVENT
5	GAME_OVER	S→C	Announce winners

3.3 Payload Formats (JSON, optional zlib compression)

Compression flag: First payload byte: `0x00`=uncompressed, `0x01`=zlib

INIT: `{}` (empty)

INIT_ACK: `{"client_id": <int>}`

SNAPSHOT:

```json
{
  "full": <bool>,
  "changes": [[row,col,owner], ...],
  "grid_enc": "r,c,owner;..." (if full),
  "redundant": [{"snapshot_id": <int>, "changes": [...]}]
}
```

EVENT: `{"cell": <int>, "client_id": <int>, "ts": <int>}`

ACK: `{"cell": <int>, "owner": <int>}`

GAME_OVER: `{"winner": [<int>, ...], "final_grid_enc": "..."}`

Grid encoding (sparse): Only claimed cells: `"0,0,1;0,1,2"` = cell(0,0)→player1, cell(0,1)→player2

---

4. Communication Procedures

Session Start:

1. Client sends INIT, waits 500ms for INIT_ACK, retries up to 10×
2. Server assigns ID, sends INIT_ACK, adds client to broadcast list
3. Client enters PLAYING state, listens for snapshots

Normal Exchange:

1. Server broadcasts SNAPSHOT every 50ms (20 Hz):
    - Full grid every 10th snapshot (500ms)
    - Delta changes otherwise
    - Includes last 2 snapshots as redundancy
2. Clients receive, validate checksum, apply changes with deduplication (seen_ids set)

Critical Events (Claims):

1. Client clicks cell → sends EVENT, adds to retransmission queue (timeout: 500ms, max retries: 3)
2. Server processes claims (timestamp-ordered), sends ACK with confirmed owner
3. Client receives ACK → removes from queue, updates cell state

Keep-Alive: Client sends INIT every 3s (no explicit timeout enforcement)

Termination: Grid full → server computes winners → sends GAME_OVER 3× (reliability) → stops broadcast

---

5. Reliability & Performance Features

5.1 Reliability Model

Data	Method
SNAPSHOT	Unreliable (next replaces lost)
EVENT	Reliable (timeout+retry, ACK)
INIT/INIT_ACK	Reliable (client retry)
GAME_OVER	Reliable (sent 3×)

5.2 Retransmission

EVENT timeout: Fixed RTO = 500ms (conservative for home networks, typical RTT 5-100ms)
Max retries: 3 attempts, then fail

5.3 Redundant Updates

Mechanism: Each SNAPSHOT includes changes from last K=2 snapshots
Benefit: Recovers from burst loss ≤100ms (2 packets)
Deduplication: Clients track `seen_ids` (rolling window, max 500)

5.4 Delta Compression

- **Full snapshot:** Every 10th update (500ms), zlib compressed, ~100-300 bytes
- **Delta snapshot:** Only changed cells, uncompressed, ~20-80 bytes
- **Bandwidth savings:** 78% reduction vs. full snapshots every update

5.5 Sparse Grid Encoding

Before: 25 cells × 12 bytes = 300 bytes (dense JSON)
After: Only claimed cells: "r,c,owner;..." = ~60 bytes (10 claims)
Savings: 80% for typical mid-game state

---

6. Experimental Evaluation Plan

6.1 Metrics

Metric	Method
Latency	server_ts - client_recv_ts
Jitter	σ(inter-packet arrival)
Loss rate	gaps in snapshot_id / expected
Bandwidth	bytes/sec, logged by server
CPU	psutil.cpu_percent()

6.2 Test Scenarios

Baseline: Localhost, 0% loss, <5ms RTT

Moderate Loss: 10% random loss via `tc qdisc add dev lo root netem loss 10%`
High Latency: 150ms delay via `tc qdisc add dev lo root netem delay 150ms 20ms`
Burst Loss: 20% loss w/ 50% correlation via `tc qdisc add dev lo root netem loss 20% 50%`
Combined: 100ms delay + 15% loss + 2Mbps limit:

```bash
tc qdisc add dev lo root handle 1: tbf rate 2mbit burst 32k latency 400ms
tc qdisc add dev lo parent 1:1 netem delay 100ms 10ms loss 15%
```

6.3 Automated Testing

Script: `run_experiment.sh <scenario> <duration>`

```bash
#!/bin/bash
SCENARIO=$1; DURATION=${2:-60}
# Setup netem (varies by scenario)
python3 server.py &
sleep 2
python3 client.py & python3 client.py &
sleep $DURATION
killall python3
tc qdisc del dev lo root
mv *_log.csv results/$SCENARIO/
```

Analysis: `analyze_results.py` loads CSVs, computes metrics, generates plots (latency over time, bandwidth)

6.4 Expected Results

Scenario	Latency	Loss	Outcome
Baseline	1-5ms	0%	Smooth gameplay
Moderate Loss	5-10ms	~10%	Redundancy compensates
High Latency	145-175ms	0%	Playable, claims confirm <700ms
Burst Loss	5-10ms	~20%	Smoothing masks loss
Combined	90-110ms	~15%	Functional, occasional artifacts

7. Example Use Case Walkthrough

Scenario: Player 1 claims cell(0,0), Player 2 claims cell(0,1)

Time	Event	Packet
0ms	P1 starts	-
5ms	P1→INIT	`INIT(sid=0,seq=0)`
8ms	S→P1	`INIT_ACK(cid=1)`
175ms	P1 clicks (0,0)	-
176ms	P1→EVENT	`EVENT(cell=0,cid=1,ts=176)`
200ms	Server processes	Grid[0][0]=1
200ms	S→P1	`ACK(cell=0,owner=1,seq=3)`
200ms	S→All	`SNAPSHOT(sid=3,changes=[[0,0,1]])`

Packet hexdump (INIT):

```
4e 52 53 48 01 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 05 00 02 00 7b 7d xx xx xx xx
```

- `4e52 5348` = "NRSH"
- `01` = version, `00` = INIT type
- `7b7d` = JSON `{}`
- `xxxxxxxx` = CRC32 checksum

---

8. Limitations & Future Work

Current Limitations:

1. No authentication (vulnerable to spoofing) → add shared secret in INIT
2. Fixed RTO=500ms (suboptimal for varied networks) → implement adaptive RTO (Karn's algorithm)

3. No encryption (plaintext) → add DTLS or AES
4. Single-threaded server → multi-threaded claim processing
5. Hardcoded 5×5 grid → negotiate size in INIT_ACK
6. Redundancy K=2 only → adaptive based on measured loss
7. No reconnection logic → add session tokens

Future Enhancements:

- Adaptive redundancy: `K = f(loss_rate)`
- Client prediction: extrapolate positions during loss
- Forward error correction (FEC): Reed-Solomon codes
- Congestion control: monitor RTT, adjust update rate
- NAT traversal: STUN/TURN integration
- Anti-cheat: server-side timestamp validation, rate limiting

---

9. References

1. RFC 768 - User Datagram Protocol (https://www.rfc-editor.org/rfc/rfc768)
2. RFC 6298 - Computing TCP's Retransmission Timer (timeout reference)
3. Gaffer on Games - "UDP vs TCP" (https://gafferongames.com/post/udp_vs_tcp/)
4. Valve Source Multiplayer Networking (delta compression techniques)
5. Python struct module documentation (https://docs.python.org/3/library/struct.html)
6. Python zlib module documentation (https://docs.python.org/3/library/zlib.html)