1. Introduction

NetRush is a lightweight, real-time, UDP-based multiplayer game synchronization protocol designed for fast-paced gameplay where *speed matters more than reliability*. Unlike TCP, which enforces guaranteed delivery and ordering, NetRush accepts occasional packet loss to maintain low latency and smooth gameplay.

As long as players continuously receive updates, missing packets do not break the experience — the next snapshot replaces them. The goal of the protocol is to keep all players in a shared synchronized game world at high frequency with minimal delay and maximum responsiveness.

---

2. Protocol Architecture

2.1 Overview

NetRush follows a client–server model.

Component	Responsibility
Server	Maintains world state & broadcasts snapshots
Client	Sends input/actions & renders received updates

Update frequency: **20 snapshots per second (50ms interval)**
Transport layer: **UDP (IPv4)**

2.2 Network Data Flow

1. Client → Server:
   The player's device sends compact DATA packets containing current actions or movements.
2. Server → Clients:
   The server broadcasts updated world snapshots to everyone, ensuring all players see the same game state.
3. Occasional Acknowledgments:
   Only important messages (like joining or scoring) require ACKs, keeping most updates lightweight and fast.

2.3 Handling Reliability

UDP is unreliable — NetRush compensates by introducing **soft-reliability only** where needed:

Category	Behavior
Normal gameplay updates	No retransmission — new updates overwrite old ones
Critical events	Client resends if no ACK in **100 ms**
Max retry attempts	3 tries, then fail event

3. Message Format

Each packet in NetRush starts with a small header, followed by an optional payload depending on the message type.

Field	Why it is needed (Justification)	Why this size was chosen
protocol_id (4 bytes)	Ensures packets belong to NetRush and not another program or random UDP noise. Allows immediate early rejection without parsing.	4 bytes = readable ASCII token "NRSH" — common standard for protocol signatures.
version (1 byte)	Makes the protocol upgradeable. Clients using v1 still work when v2 adds features. Prevents mismatch crashes.	1 byte supports 0–255 versions — more than enough.
msg_type (1 byte)	Differentiates INIT vs DATA vs EVENT vs ACK. Required so both sides know how to parse payload.	1 byte supports up to 256 message types → scalable.

flags (1 byte)	Enables features without breaking older versions. Allows reliable messages, compression and encryption toggles.	1 byte = bitmask of 8 flags → lightweight and future-proof.
snapshot_id (4 bytes)	Used to determine if a world update is newer or older. Protects against out-of-order packets in UDP.	4 bytes allows 4.29B snapshots (at 20/s = ~7 years continuous).
seq_num (4 bytes)	Detects packet loss, duplicates & ordering. Required for retransmission logic and sliding window.	4 bytes avoids wrap-around for a long-running session.
server_timestamp (8 bytes)	Enables client interpolation, latency measurement, lag compensation & replay smoothing.	8 bytes UNIX epoch in milliseconds → valid for 292 million years.
payload_len (2 bytes)	Prevents buffer overflow, enables streaming and fragmented reassembly. Must know content length.	Max payload = 65535 bytes → enough for large state syncs.
checksum (4 bytes)	Validates integrity so corrupted or modified packets are discarded instead of breaking gameplay.	CRC32 = fast + hardware supported + reliable.
payload (variable)	Carries actual game data (state snapshots, events, player inputs). Flexible and content-dependent.	Variable size enables small realtime packets + large event packets.

3.2 Message Types

Message Type	Direction	Description
Init	Client → Server	Join request
Init_ACK	Server → Client	Join confirmation + assigned ID
Event	Client → Server	Critical gameplay action (cell capture)
ACK	Server → Client	Acknowledge Event delivery
HEARTBEAT	Client → Server	Keepalive signal
Snapshots	Server → Client	Game state updates (no explicit type)

3.3 Example Flow

1- **Client → INIT**
Player joins the game, sending their name and room code.

2- **Server → INIT_ACK**
Server assigns a unique PlayerID and provides the latest game snapshot.

3- **Client → DATA**
The player begins sending frequent DATA packets for movement or actions (~20 times per second).

4- **Server / Client → ACK**
Only critical messages (e.g., scoring, capture events) are acknowledged.
The client or server sends ACK for EVENT or important updates.

5- **Client → EVENT**
Critical gameplay actions such as scoring, flag capture, or special ability usage are sent.

4. Reliability & Packet Loss Handling

Although NetRush operates on UDP, the protocol introduces lightweight reliability for critical gameplay events while keeping normal movement updates free-flowing and low-latency.

4.1 Sequence Number Tracking

Each client maintains a sliding sequence window:

Field	Meaning
last_seq_received	Most recent valid DATA seq received
expected_next_seq	Used to detect missing packets
duplicate_count	Tracks ignored repeated packets

Processing logic:

```
if seq_num > last_seq_received:
  accept packet
  update(last_seq_received = seq_num)
elif seq_num == last_seq_received:
  ignore duplicate
else:
  stale/out-of-order -> discard
```

4.2 Snapshot ID Ordering

Snapshots always replace previous ones.
The client applies only packets where:

new_snapshot_id > current_snapshot_id

Older or duplicate state updates are not applied, keeping gameplay consistent even under packet loss.

4.3 Reliable EVENT Delivery

EVENT packets require confirmation.

If no ACK is received within 100ms, retransmission occurs:

Attempt	Delay
#1	100ms
#2	200ms
#3	300ms (final attempt)

5. Client-side Interpolation & Prediction

Network jitter may cause uneven packet arrival. Rendering directly from incoming DATA creates visible jumps → instead, the client renders using prediction.

5.1 Linear Interpolation

Concept:

- When the client has two snapshots, S_n (previous) and S_{n+1} (next), it can interpolate positions instead of waiting for the next packet.
- **Formula:**

$$\mathbf{pos}(\boldsymbol{t}) = \mathbf{pos}_{\boldsymbol{n}} + \boldsymbol{\alpha} \cdot (\mathbf{pos}_{\boldsymbol{n}+\mathbf{1}} - \mathbf{pos}_{\boldsymbol{n}})$$

Where α = fraction of time elapsed relative to snapshot interval

- ***Example:***
    - *Snapshot interval = 50 ms*
    - *Previous snapshot: player at (4,7)*
    - *Next snapshot: player at (6,10)*
    - *25 ms after S_n → α = 25 / 50 = 0.5*

$$pos(t) = (4{,}7) + 0.5 \cdot ((6{,}10) - (4{,}7)) = (5{,}8.5)$$

- ***Result:*** *The player is drawn* ***halfway between snapshots****, making movement smooth.*

FSM connection: *This happens in the PLAY state. Even if no new snapshot arrives yet, the client can interpolate between the last two snapshots.*

5.2 Movement Prediction if Packets Drop

If a packet is missing, the client assumes last known velocity continues temporarily:

Time without update	Render Behavior
<150ms	Predict forward using last (x,y,vel)
150–300ms	Slow prediction + visual smoothing
>300ms	Freeze → request state resync

For example

Time elapsed	Real server state	Client sees (predicted)	Notes
0 ms	(10,10)	(10,10)	Last valid snapshot received
50 ms	(11,12) — missing	(11,12)	Predict forward using last velocity
100 ms	(12,14) — missing	(12,14)	Still predicting normally
150 ms	(13,16) — missing	(12.75,14.5)	Slow prediction + visual smoothing begins
200 ms	(14,18) — missing	(13.25,15.25)	Smooth blending continues
250 ms	(15,20) — missing	(13.75,16)	Prediction slowing; client prepares to freeze if necessary
300 ms	(16,22) — missing	(14,16.5)	Threshold exceeded → freeze or request state resync
350 ms or more	(17,24)	(14,16.5)	Client freezes until next valid snapshot arrives

This prevents rubber-banding and keeps the gameplay visually smooth.

6. Error Handling & Failure Scenarios

Failure Condition	Protocol Response
Packet lost	Accept next snapshot (no resend)
EVENT lost	Retransmit w/timeout + ACK required
Snapshot gap detected	Log packet loss %
Server disconnect	Client retries reconnect INIT every 500ms
Duplicate packet	Drop automatically using seq check
Corrupted checksum	Discard → request state refresh

Clients additionally maintain gameplay statistics:

```
loss_rate = lost_packets / total_packets
avg_latency = Σ(rtt) / samples
jitter = maxΔrtt - minΔrtt
```

7. Experimental Testing & Evaluation Plan

Testing must prove that NetRush functions under real conditions, including delay, loss, and jitter.

7.1 Tools

Wireshark → capture traffic, verify seq/snapshot order
Linux netem → inject artificial impairment
CSV logging for metrics
PCAP

7.2 Planned Test Scenarios

Test	Condition	Expected Outcome
Baseline Local	0% loss, 3–5ms RTT	Smooth updates, no EVENT loss
Loss Tolerance	10–30% drop rate	Interpolation masks loss, movement stable
Latency Stress	100–300ms delay	Game still playable, EVENT reliable
Small Burst Loss	Drop 1–3 consecutive packets (<150 ms gap)	Client predicts forward using last (x,y,vel); smooth motion maintained
Medium Burst Loss	Drop 4–6 consecutive packets (150–300 ms gap)	Slow prediction + visual smoothing; linear interpolation applied to mask missing updates
High Burst Loss	Drop 7–10 consecutive packets (>300 ms / ≥350 ms gap)	Prediction fails; client freezes and requests state resync
High Jitter	±150ms variation	Visual smoothing remains consistent

Test Cases

Test Case 1: Baseline Local

- **Objective:** Verify normal gameplay under ideal network conditions.
- **Steps:**
    1. Run client and server on the same LAN or localhost.
    2. Ensure no packet loss or delay.
    3. Observe gameplay.
- **Expected Result:** Movement is smooth, all packets arrive in order, no EVENT loss.

Test Case 2: Loss Tolerance

- **Objective:** Test gameplay stability under moderate random packet loss.
- **Steps:**

1. Apply 10–30% random packet loss **(tc qdisc add dev eth0 root netem loss 15%).**
2. Run client/server.
3. Monitor movement and EVENT delivery.

- **Expected Result:** Movement remains smooth; interpolation compensates for missing updates; critical EVENT packets are acknowledged.

Test Case 3: Latency Stress

- **Objective:** Evaluate gameplay under high constant delay.
- **Steps:**
  1. Apply 150 ms network delay **(tc qdisc add dev eth0 root netem delay 150ms).**
  2. Run client/server.
  3. Observe movement and EVENT handling.
- **Expected Result:** Game is playable; movement is slightly delayed but smooth; EVENT packets are reliably delivered.

Test Case 4: Small Burst Loss

- **Objective:** Assess prediction under short burst packet loss.
- **Steps:**
  1. Drop 1–3 consecutive packets (<150 ms gap) using network simulation.
  2. Observe client prediction.
- **Expected Result:** Client continues smooth movement using last velocity; no noticeable visual jumps.

Test Case 5: Medium Burst Loss

- **Objective:** Check client behavior for moderate burst loss.
- **Steps:**
  1. Drop 4–6 consecutive packets (150–300 ms gap).
  2. Observe prediction and interpolation.
- **Expected Result:** Client uses slow prediction and linear interpolation to smooth missing updates; movement remains playable.

Test Case 6: High Burst Loss

- **Objective:** Verify client reaction to severe burst loss.
- **Steps:**
    1. Drop 7–10 consecutive packets (>300 ms / ≥350 ms gap).
    2. Observe client behavior.
- **Expected Result:** Client freezes temporarily and requests state resynchronization; EVENT packets handled reliably.

Test Case 7: High Jitter

- **Objective:** Test client stability under high variation in packet arrival times.
- **Steps:**
    1. Apply ±150 ms random delay **(tc qdisc add dev eth0 root netem delay 150ms 150ms)**.
    2. Run client/server.
    3. Observe movement smoothness.
- **Expected Result:** Client prediction and interpolation maintain smooth visuals despite RTT variations; no sudden jumps.

8. State Diagram