R-Type Network Protocol Specification

Binary Protocol for UDP Communication
Version 1.0

R-Type Development Team

November 24, 2025

Abstract

This document specifies the binary network protocol used for client-server communication in the R-Type multiplayer game. The protocol is designed to operate over UDP, providing efficient real-time data transmission with optional reliability mechanisms. This specification describes packet structures, data serialization formats, and communication patterns required for implementing compatible R-Type clients and servers.

Contents

1 Introduction

1.1 Purpose

This document defines the R-Type network protocol, a binary protocol designed for real-time multiplayer game communication. The protocol enables multiple clients to connect to a central authoritative server, exchange game state information, and synchronize gameplay events.

1.2 Scope

This specification covers:

- Packet structure and format
- Data serialization and quantization techniques
- Message types and their payloads
- Reliability mechanisms over UDP
- Communication flows and patterns

1.3 Requirements

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

1.4 Protocol Characteristics

- **Transport Layer**: UDP (User Datagram Protocol)
- **Format**: Binary (network byte order - big-endian)
- **Maximum Packet Size**: 1200 bytes (recommended to avoid fragmentation)
- **Protocol Version**: 1.0

2 General Packet Structure

2.1 Packet Format

Every packet in the R-Type protocol consists of two main parts:

Packet Structure

```
+--------------------------------------------+
|  Header (12 bytes)                         |
+--------------------------------------------+
|  Payload (variable, 0-1188 bytes)          |
+--------------------------------------------+
```

2.2 Packet Header

The packet header is a fixed-size structure present in all protocol messages. It provides essential metadata for packet processing, validation, and ordering.

```
struct PacketHeader {
    uint16_t magic;           // Protocol identifier: 0x5254 ('RT')
    uint8_t  packet_type;     // Packet type identifier
    uint8_t  flags;           // Control flags
    uint32_t sequence_number; // Monotonic sequence number
    uint32_t timestamp;       // Milliseconds since connection
};
// Total size: 12 bytes
```

Listing 1: Packet Header Structure

2.3 Header Fields Description

Field	Size	Description
magic	2 bytes	Protocol magic number (0x5254). Used for packet validation. Packets with invalid magic MUST be discarded.
packet_type	1 byte	Identifies the packet type. Valid range: 0x01-0x7F. See Section 4 for type definitions.
flags	1 byte	Bitfield for packet control flags. See Section 2.4.
sequence_number	4 bytes	Monotonically increasing packet sequence number. Used for duplicate detection, ordering, and acknowledgment.
timestamp	4 bytes	Timestamp in milliseconds since connection establishment. Used for latency measurement and interpolation.

2.4 Packet Flags

The flags field is an 8-bit bitfield used to specify packet processing requirements.

```
enum PacketFlags {
    FLAG_RELIABLE    = 0x01,  // Requires acknowledgment
    FLAG_COMPRESSED  = 0x02,  // Payload is compressed
    FLAG_ENCRYPTED   = 0x04,  // Payload is encrypted
    FLAG_FRAGMENTED  = 0x08,  // Part of fragmented message
    FLAG_PRIORITY    = 0x10,  // High priority processing
    FLAG_RESERVED_5  = 0x20,  // Reserved for future use
    FLAG_RESERVED_6  = 0x40,  // Reserved for future use
    FLAG_RESERVED_7  = 0x80   // Reserved for future use
};
```

Listing 2: Packet Flags Definition

2.4.1 Flag Descriptions

- **FLAG_RELIABLE (0x01)**: When set, the receiver MUST send an acknowledgment. The sender MUST retransmit if no ACK is received within the timeout period.
- **FLAG_COMPRESSED (0x02)**: Indicates the payload is compressed. The receiver MUST decompress before processing.

- **FLAG_ENCRYPTED (0x04)**: Indicates the payload is encrypted. Implementation-specific.
- **FLAG_FRAGMENTED (0x08)**: Indicates this packet is part of a larger fragmented message.
- **FLAG_PRIORITY (0x10)**: Suggests priority processing. Implementation MAY prioritize these packets.

3 Packet Types

Packet types are organized into functional categories. Each category occupies a specific range of type identifiers.

3.1 Type Ranges

Range	Category	Description
0x01-0x0F	Connection	Session establishment and management
0x10-0x1F	Input	Player input and control messages
0x20-0x3F	World State	Game world and entity state updates
0x40-0x5F	Game Events	Gameplay events and actions
0x60-0x6F	Game Control	Game lifecycle and state transitions
0x70-0x7F	Protocol Control	Reliability and connection maintenance

4 Connection Management Packets (0x01-0x0F)

4.1 CLIENT_CONNECT (0x01)

Sent by the client to initiate a connection to the server.

```
struct ClientConnect {
    PacketHeader header;          // type = 0x01
    uint8_t protocol_version;     // Protocol version (current: 1)
    char player_name[32];         // UTF-8 player name (null-terminated
        )
    uint32_t client_id;           // Unique client identifier
};
// Total size: 49 bytes
```

Listing 3: CLIENT_CONNECT Packet

Fields:

- `protocol_version`: Protocol version supported by the client. Current version is 1.
- `player_name`: Player's display name, null-terminated UTF-8 string. Maximum 31 characters + null terminator.
- `client_id`: Randomly generated unique identifier for this client instance.

Usage:

1. Client generates unique `client_id`
2. Client sends CLIENT_CONNECT to server
3. Server responds with SERVER_ACCEPT or SERVER_REJECT

4.2 SERVER_ACCEPT (0x02)

Sent by the server to accept a client connection.

```
struct ServerAccept {
    PacketHeader header;           // type = 0x02
    uint32_t assigned_player_id;   // Server-assigned player ID
    uint8_t max_players;           // Maximum players in game
    uint32_t game_instance_id;     // Current game instance ID
    uint16_t server_tick_rate;     // Server update rate (Hz)
};
// Total size: 23 bytes
```

Listing 4: SERVER_ACCEPT Packet

Fields:

- `assigned_player_id`: Unique player ID assigned by server. Client MUST use this ID in all subsequent communications.
- `max_players`: Maximum number of players supported (typically 2 to 4).
- `game_instance_id`: Identifier for the game instance the player joined (can be useful for rejoining and future lobby management).
- `server_tick_rate`: Server simulation tick rate in Hz (typically 60).

4.3 SERVER_REJECT (0x03)

Sent by the server to reject a connection attempt.

```
struct ServerReject {
    PacketHeader header;           // type = 0x03
    uint8_t reason_code;           // Rejection reason code
    char reason_message[64];       // Human-readable reason
};
// Total size: 77 bytes
```

Listing 5: SERVER_REJECT Packet

Reason Codes:

- 0x00: Server full
- 0x01: Incompatible protocol version
- 0x02: Invalid player name
- 0x03: Banned client
- 0xFF: Generic error
- More to come with the network track

4.4 CLIENT_DISCONNECT (0x04)

Sent by either client or server to terminate the connection.

```
struct ClientDisconnect {
    PacketHeader header;           // type = 0x04, FLAG_RELIABLE
    uint32_t player_id;            // Disconnecting player ID
    uint8_t reason;                // Disconnect reason
};
// Total size: 17 bytes
```

Listing 6: CLIENT_DISCONNECT Packet

Reason Codes:

- 0x00: Normal disconnect
- 0x01: Timeout
- 0x02: Kicked by server
- 0x03: Client error

Note: This packet MUST have the FLAG_RELIABLE flag set.

4.5 HEARTBEAT (0x05)

Sent periodically by clients to maintain the connection.

```
struct Heartbeat {
    PacketHeader header;           // type = 0x05
    uint32_t player_id;            // Player ID
};
// Total size: 16 bytes
```

Listing 7: HEARTBEAT Packet

Usage:

- Client SHOULD send every 1-2 seconds
- Server SHOULD disconnect clients that don't send heartbeat within 10 seconds

5 Player Input Packets (0x10-0x1F)

5.1 PLAYER_INPUT (0x10)

Contains player input state. Sent frequently (typically every frame or when input changes).

```
struct PlayerInput {
    PacketHeader header;           // type = 0x10
    uint32_t player_id;            // Player ID
    uint16_t input_flags;          // Bitfield of input states
    int16_t aim_x;                 // Optional: aim direction X
    int16_t aim_y;                 // Optional: aim direction Y
};
// Total size: 24 bytes
```

Listing 8: PLAYER_INPUT Packet

Input Flags Bitfield:

```
Bit 0:  MOVE_UP
Bit 1:  MOVE_DOWN
Bit 2:  MOVE_LEFT
Bit 3:  MOVE_RIGHT
Bit 4:  ACTION_SHOOT
Bit 5:  ACTION_SPECIAL
Bit 6:  ACTION_6 (Reserved)
Bit 7:  ACTION_7 (Reserved)
Bits 8-15: Reserved for more bonus actions
```

Example - Packing Input:

```
uint16_t pack_input(bool up, bool down, bool left,
                    bool right, bool shoot, bool special) {
    return (up << 0) | (down << 1) | (left << 2) |
           (right << 3) | (shoot << 4) | (special << 5);
}
```

6 World State Packets (0x20-0x3F)

6.1 WORLD_SNAPSHOT (0x20)

Contains the current state of all entities in the game world. Sent regularly by the server (typically 20-60 times per second).

```
struct EntityState {
    uint32_t entity_id;          // Unique entity identifier
    uint8_t entity_type;         // Entity type code
    int16_t pos_x;               // Quantized X position
    int16_t pos_y;               // Quantized Y position
    int16_t vel_x;               // Quantized X velocity
    int16_t vel_y;               // Quantized Y velocity
    uint8_t health;              // Current health (0-255)
    uint8_t state_flags;         // Entity-specific state flags
};
// Total size: 16 bytes per entity
```

Listing 9: Entity State Structure

```
struct WorldSnapshot {
    PacketHeader header;         // type = 0x20
    uint32_t world_tick;         // Server simulation tick
    uint16_t entity_count;       // Number of entities in this packet
    EntityState entities[];      // Variable-length array
};
// Total size: 18 + (16 * entity_count) bytes
```

Listing 10: WORLD_SNAPSHOT Packet

Entity Type Codes:

- 0x00: Player
- 0x01: Enemy (generic)
- 0x02: Enemy (snake pattern)
- 0x03: Enemy (boss)

- 0x10: Projectile (player)
- 0x11: Projectile (enemy)
- 0x20: Powerup
- 0x30: Obstacle
- 0x40: Background element

Position Quantization:

Positions are stored as 16-bit integers representing game world coordinates. The conversion from floating-point to quantized format is:

```
// Quantize position (float to int16)
int16_t quantize_position(float pos, float world_min,
                              float world_max) {
    float normalized = (pos - world_min) / (world_max - world_min);
    return (int16_t)(normalized * 65535.0f);
}

// Dequantize position (int16 to float)
float dequantize_position(int16_t quantized, float world_min,
                              float world_max) {
    float normalized = (float)quantized / 65535.0f;
    return world_min + normalized * (world_max - world_min);
}
```

Recommended World Bounds:

- X: 0 to 2048 units
- Y: 0 to 1536 units

6.2 ENTITY_SPAWN (0x21)

Notifies clients of a new entity entering the game world.

```
struct EntitySpawn {
    PacketHeader header;           // type = 0x21, FLAG_RELIABLE
    uint32_t entity_id;            // New entity's unique ID
    uint8_t entity_type;           // Entity type code
    int16_t pos_x;                 // Initial X position
    int16_t pos_y;                 // Initial Y position
    uint8_t variant;               // Entity variant/subtype
    uint8_t initial_health;        // Starting health
    int16_t initial_velocity_x;    // Initial X velocity
    int16_t initial_velocity_y;    // Initial Y velocity
};
// Total size: 26 bytes
```

Listing 11: ENTITY_SPAWN Packet

Note: This packet MUST have the FLAG_RELIABLE flag set to ensure delivery.

6.3 ENTITY_DESTROY (0x22)

Notifies clients that an entity has been destroyed or removed.

```
struct EntityDestroy {
    PacketHeader header;          // type = 0x22, FLAG_RELIABLE
    uint32_t entity_id;           // Destroyed entity's ID
    uint8_t destroy_reason;       // Reason for destruction
    int16_t final_pos_x;          // Final X position (for effects)
    int16_t final_pos_y;          // Final Y position (for effects)
};
// Total size: 21 bytes
```

Listing 12: ENTITY_DESTROY Packet

Destroy Reason Codes:

- 0x00: Killed by player
- 0x01: Killed by enemy
- 0x02: Out of bounds
- 0x03: Timeout/despawn
- 0x04: Level transition

6.4 ENTITY_UPDATE (0x23)

Updates specific attributes of an entity without sending full state.

```
struct EntityUpdate {
    PacketHeader header;          // type = 0x23
    uint32_t entity_id;           // Entity to update
    uint8_t update_flags;         // Which fields are updated
    int16_t pos_x;                // Updated X position (if flag set)
    int16_t pos_y;                // Updated Y position (if flag set)
    uint8_t health;               // Updated health (if flag set)
    uint8_t shield;               // Updated shield (if flag set)
    uint8_t state_flags;          // Updated state (if flag set)
    int16_t velocity_x;           // Updated X velocity (if flag set)
    int16_t velocity_y;           // Updated Y velocity (if flag set)
};
// Total size: 26 bytes
```

Listing 13: ENTITY_UPDATE Packet

Update Flags:

- Bit 0: Position updated
- Bit 1: Health updated
- Bit 2: Shield updated
- Bit 3: State flags updated
- Bit 4: Velocity updated
- Bits 5-7: Reserved

7 Game Event Packets (0x40-0x5F)

7.1 PLAYER_HIT (0x40)

Notifies that a player has been hit and taken damage.

```
struct PlayerHit {
    PacketHeader header;          // type = 0x40, FLAG_RELIABLE
    uint32_t player_id;           // Player who was hit
    uint32_t attacker_id;         // Entity that caused damage
    uint8_t damage;               // Damage amount
    uint8_t remaining_health;     // Health after damage
    uint8_t remaining_shield;     // Shield after damage
    int16_t hit_pos_x;            // Hit location X
    int16_t hit_pos_y;            // Hit location Y
};
// Total size: 29 bytes
```

Listing 14: PLAYER_HIT Packet

7.2 PLAYER_DEATH (0x41)

Notifies that a player has died.

```
struct PlayerDeath {
    PacketHeader header;          // type = 0x41, FLAG_RELIABLE
    uint32_t player_id;           // Player who died
    uint32_t killer_id;           // Entity that killed player
    uint32_t score_before_death;  // Player's score
    int16_t death_pos_x;          // Death location X
    int16_t death_pos_y;          // Death location Y
};
// Total size: 30 bytes
```

Listing 15: PLAYER_DEATH Packet

7.3 SCORE_UPDATE (0x42)

Updates a player's score.

```
struct ScoreUpdate {
    PacketHeader header;          // type = 0x42
    uint32_t player_id;           // Player whose score changed
    uint32_t new_score;           // Updated total score
    int16_t score_delta;          // Change in score (can be negative)
    uint8_t reason;               // Reason for score change
};
// Total size: 23 bytes
```

Listing 16: SCORE_UPDATE Packet

Score Change Reasons:

- 0x00: Enemy killed
- 0x01: Boss killed
- 0x02: Powerup collected
- 0x03: Level completed
- 0x04: Bonus points

7.4 POWERUP_PICKUP (0x43)

Notifies that a player collected a powerup.

```
struct PowerupPickup {
    PacketHeader header;          // type = 0x43, FLAG_RELIABLE
    uint32_t player_id;           // Player who picked up powerup
    uint32_t powerup_id;          // Powerup entity ID
    uint8_t powerup_type;         // Type of powerup
    uint8_t duration;             // Effect duration (seconds, 0=
       permanent)
};
// Total size: 22 bytes
```

Listing 17: POWERUP_PICKUP Packet

Powerup Types:

- 0x00: Speed boost
- 0x01: Weapon upgrade
- 0x02: Force (R-Type signature weapon)
- 0x03: Shield
- 0x04: Extra life
- 0x05: Invincibility

7.5 WEAPON_FIRE (0x44)

Notifies that an entity fired a weapon.

```
struct WeaponFire {
    PacketHeader header;          // type = 0x44
    uint32_t shooter_id;          // Entity that fired
    uint32_t projectile_id;       // New projectile entity ID
    int16_t origin_x;             // Fire origin X
    int16_t origin_y;             // Fire origin Y
    int16_t direction_x;          // Direction vector X (normalized
       *1000)
    int16_t direction_y;          // Direction vector Y (normalized
       *1000)
    uint8_t weapon_type;          // Weapon type fired
};
// Total size: 31 bytes
```

Listing 18: WEAPON_FIRE Packet

Weapon Types:

- 0x00: Basic shot
- 0x01: Charged shot
- 0x02: Spread shot
- 0x03: Laser beam
- 0x04: Missile
- 0x05: Force shot

8 Game Control Packets (0x60-0x6F)

8.1 GAME_START (0x60)

Notifies all clients that the game is starting.

```c
struct GameStart {
    PacketHeader header;          // type = 0x60, FLAG_RELIABLE
    uint32_t game_instance_id;    // Game instance identifier
    uint8_t player_count;         // Number of players
    uint32_t player_ids[4];       // Player IDs (up to 4)
    uint8_t level_id;             // Starting level
    uint8_t difficulty;           // Difficulty setting
};
// Total size: 36 bytes
```

Listing 19: GAME_START Packet

Difficulty Levels:

- 0x00: Easy
- 0x01: Normal
- 0x02: Hard
- 0x03: Insane

8.2 GAME_END (0x61)

Notifies clients that the game has ended.

```c
struct GameEnd {
    PacketHeader header;          // type = 0x61, FLAG_RELIABLE
    uint8_t end_reason;           // Reason game ended
    uint32_t final_scores[4];     // Final scores for all players
    uint8_t winner_id;            // Winning player (if applicable)
    uint32_t play_time;           // Total game time (seconds)
};
// Total size: 34 bytes
```

Listing 20: GAME_END Packet

End Reasons:

- 0x00: Victory (all levels completed)
- 0x01: Defeat (all players dead)
- 0x02: Timeout
- 0x03: Host disconnect
- 0x04: Server shutdown

8.3 LEVEL_COMPLETE (0x62)

Notifies clients that the current level has been completed.

```c
struct LevelComplete {
    PacketHeader header;           // type = 0x62, FLAG_RELIABLE
    uint8_t completed_level;       // Level that was completed
    uint8_t next_level;            // Next level to load (0xFF=game end)
    uint32_t bonus_score;          // Completion bonus
    uint16_t completion_time;      // Time taken (seconds)
};
// Total size: 22 bytes
```

Listing 21: LEVEL_COMPLETE Packet

8.4 LEVEL_START (0x63)

Notifies clients that a new level is starting.

```c
struct LevelStart {
    PacketHeader header;           // type = 0x63, FLAG_RELIABLE
    uint8_t level_id;              // Level identifier
    char level_name[32];           // Level name
    uint16_t estimated_duration;   // Estimated time (seconds)
};
// Total size: 47 bytes
```

Listing 22: LEVEL_START Packet

9 Protocol Control Packets (0x70-0x7F)

9.1 ACK - Acknowledgment (0x70)

Acknowledges receipt of a reliable packet.

```c
struct Acknowledgment {
    PacketHeader header;           // type = 0x70
    uint32_t acked_sequence;       // Sequence number being ACKed
    uint32_t received_timestamp;   // When packet was received
};
// Total size: 20 bytes
```

Listing 23: ACK Packet

Usage:

1. When receiving a packet with FLAG_RELIABLE set, the receiver MUST send an ACK
2. The sender MUST retransmit if ACK is not received within timeout period (typically 500ms-1000ms)
3. Maximum retransmission attempts: 5

9.2 PING (0x71)

Measures round-trip time to server.

```c
struct Ping {
    PacketHeader header;           // type = 0x71
    uint32_t client_timestamp;     // Client's current timestamp
};
// Total size: 16 bytes
```

Listing 24: PING Packet

9.3 PONG (0x72)

Response to PING packet.

```
struct Pong {
    PacketHeader header;           // type = 0x72
    uint32_t client_timestamp;     // Original client timestamp from
       PING
    uint32_t server_timestamp;     // Server's timestamp when received
};
// Total size: 20 bytes
```

Listing 25: PONG Packet

RTT Calculation:

```
uint32_t calculate_rtt(uint32_t ping_sent_time,
                       uint32_t pong_received_time) {
    return pong_received_time - ping_sent_time;
}
```

10 Reliability Mechanism

10.1 Overview

While UDP is an unreliable protocol, certain game events require guaranteed delivery. Our R-Type protocol implements a selective reliability mechanism.

10.2 Reliable Packet Handling

10.2.1 Sender Responsibilities

1. Set FLAG_RELIABLE in packet header
2. Store packet in retransmission queue
3. Start timeout timer (typically 500ms)
4. If ACK received, remove from queue
5. If timeout expires, retransmit up to MAX_RETRIES (5) times
6. If all retries exhausted, consider connection lost

```
void send_reliable(Packet packet) {
    packet.header.flags |= FLAG_RELIABLE;
    packet.header.sequence_number = next_sequence++;

    // Store for potential retransmission
    retransmit_queue.add(packet);

    // Send packet
    udp_send(packet);

    // Start timeout timer
    start_timer(packet.header.sequence_number, TIMEOUT_MS);
}

```

```
void on_timeout(uint32_t sequence) {
    Packet packet = retransmit_queue.get(sequence);
    if (packet.retry_count < MAX_RETRIES) {
        packet.retry_count++;
        udp_send(packet);
        start_timer(sequence, TIMEOUT_MS);
    } else {
        // Connection lost
        handle_connection_lost();
    }
}

void on_ack_received(uint32_t acked_sequence) {
    retransmit_queue.remove(acked_sequence);
}
```

Listing 26: Reliable Send Pseudocode

10.2.2 Receiver Responsibilities

1. Check if packet has FLAG_RELIABLE set
2. Send ACK immediately
3. Check sequence number against last received
4. Discard if duplicate (already processed)
5. Process if new

```
void on_packet_received(Packet packet) {
    if (packet.header.flags & FLAG_RELIABLE) {
        // Send acknowledgment immediately
        send_ack(packet.header.sequence_number);
    }

    // Check for duplicate
    if (packet.header.sequence_number <= last_processed_sequence) {
        // Duplicate packet, discard
        return;
    }

    // Process new packet
    last_processed_sequence = packet.header.sequence_number;
    process_packet(packet);
}

void send_ack(uint32_t sequence) {
    Acknowledgment ack;
    ack.header.magic = 0x5254;
    ack.header.packet_type = 0x70;
    ack.header.sequence_number = next_sequence++;
    ack.acked_sequence = sequence;
    ack.received_timestamp = get_current_time_ms();

    udp_send(ack);
}
```

Listing 27: Reliable Receive Pseudocode

10.3 Packets Requiring Reliability

The following packet types MUST be sent with FLAG_RELIABLE:

- CLIENT_DISCONNECT (0x04)
- ENTITY_SPAWN (0x21)
- ENTITY_DESTROY (0x22)
- PLAYER_HIT (0x40)
- PLAYER_DEATH (0x41)
- POWERUP_PICKUP (0x43)
- GAME_START (0x60)
- GAME_END (0x61)
- LEVEL_COMPLETE (0x62)
- LEVEL_START (0x63)

11 Data Serialization and Optimization

11.1 Byte Order

All multi-byte integer fields MUST be transmitted in network byte order (big-endian). Implementations MUST convert between host and network byte order appropriately.

```c
#include <required.h>  // POSIX systems

// Serialization (host to network)
void serialize_header(PacketHeader* header, uint8_t* buffer) {
    uint16_t* buf16 = (uint16_t*)buffer;
    uint32_t* buf32 = (uint32_t*)(buffer + 4);

    buf16[0] = htons(header->magic);
    buffer[2] = header->packet_type;
    buffer[3] = header->flags;
    buf32[0] = htonl(header->sequence_number);
    buf32[1] = htonl(header->timestamp);
}

// Deserialization (network to host)
void deserialize_header(const uint8_t* buffer, PacketHeader* header) {
    const uint16_t* buf16 = (const uint16_t*)buffer;
    const uint32_t* buf32 = (const uint32_t*)(buffer + 4);

    header->magic = ntohs(buf16[0]);
    header->packet_type = buffer[2];
    header->flags = buffer[3];
    header->sequence_number = ntohl(buf32[0]);
    header->timestamp = ntohl(buf32[1]);
}
```

Listing 28: Byte Order Conversion

11.2 Position Quantization

To reduce bandwidth, floating-point positions are quantized to 16-bit integers.

```
// Configuration
const float WORLD_MIN_X = 0.0f;
const float WORLD_MAX_X = 2048.0f;
const float WORLD_MIN_Y = 0.0f;
const float WORLD_MAX_Y = 1536.0f;

// Clamp helper function
float clamp(float value, float min, float max) {
    if (value < min) return min;
    if (value > max) return max;
    return value;
}

// Quantize float position to int16
int16_t quantize_position_x(float x) {
    float normalized = (x - WORLD_MIN_X) / (WORLD_MAX_X - WORLD_MIN_X);
    normalized = clamp(normalized, 0.0f, 1.0f);
    return (int16_t)(normalized * 65535.0f);
}

int16_t quantize_position_y(float y) {
    float normalized = (y - WORLD_MIN_Y) / (WORLD_MAX_Y - WORLD_MIN_Y);
    normalized = clamp(normalized, 0.0f, 1.0f);
    return (int16_t)(normalized * 65535.0f);
}

// Dequantize int16 to float position
float dequantize_position_x(int16_t quantized) {
    float normalized = (float)quantized / 65535.0f;
    return WORLD_MIN_X + normalized * (WORLD_MAX_X - WORLD_MIN_X);
}

float dequantize_position_y(int16_t quantized) {
    float normalized = (float)quantized / 65535.0f;
    return WORLD_MIN_Y + normalized * (WORLD_MAX_Y - WORLD_MIN_Y);
}
```

Listing 29: Position Quantization Implementation

11.3 Velocity Quantization

Velocities are quantized to signed 16-bit integers.

```
const float MAX_VELOCITY = 500.0f;  // Game units per second

int16_t quantize_velocity(float velocity) {
    float normalized = velocity / MAX_VELOCITY;
    normalized = clamp(normalized, -1.0f, 1.0f);
    return (int16_t)(normalized * 32767.0f);
}

float dequantize_velocity(int16_t quantized) {
    float normalized = (float)quantized / 32767.0f;
    return normalized * MAX_VELOCITY;
}
```

Listing 30: Velocity Quantization

11.4 Direction Vectors

Normalized direction vectors are stored as 16-bit fixed-point values.

```
struct Direction {
    int16_t x;  // Range: -1000 to 1000 (represents -1.0 to 1.0)
    int16_t y;
};

Direction quantize_direction(float dx, float dy) {
    // Normalize the vector
    float length = sqrt(dx * dx + dy * dy);
    if (length > 0.0001f) {
        dx /= length;
        dy /= length;
    }

    Direction dir;
    dir.x = (int16_t)(dx * 1000.0f);
    dir.y = (int16_t)(dy * 1000.0f);
    return dir;
}

void dequantize_direction(Direction dir, float* dx, float* dy) {
    *dx = (float)dir.x / 1000.0f;
    *dy = (float)dir.y / 1000.0f;
}
```

Listing 31: Direction Vector Quantization

12 Communication Flows

12.1 Connection Establishment

```
CLIENT                        SERVER
   |                             |
   |---(0x01) CLIENT_CONNECT------>|
   |                             |
   |                        [Validate connection]
   |                             |
   |<--(0x02) SERVER_ACCEPT--------|
   |                             |
   |                        Connection established
   |                             |
   |---(0x05) HEARTBEAT----------->|
   |                             |
   |<--(0x20) WORLD_SNAPSHOT-------|
   |                             |
```

12.2 Rejection Flow

```
CLIENT                        SERVER
```

```
   |                              |
   |---(0x01) CLIENT_CONNECT------>|
   |                              |
   |                         [Reject: server full]
   |                              |
   |<--(0x03) SERVER_REJECT--------|
   |                              |
   |                         Connection refused
```

12.3 Normal Gameplay Loop

```
CLIENT                          SERVER
   |                              |
   |---(0x10) PLAYER_INPUT-------->|
   |                              |
   |                         [Process input]
   |                         [Update game state]
   |                              |
   |<--(0x20) WORLD_SNAPSHOT-------|
   |                              |
   |                         [Render frame]
   |                              |
   |---(0x10) PLAYER_INPUT-------->|
   |                              |
   |<--(0x21) ENTITY_SPAWN---------|
   |---(0x70) ACK----------------->|
   |                              |
   |<--(0x20) WORLD_SNAPSHOT-------|
   |                              |
   |---(0x10) PLAYER_INPUT-------->|
   |                              |
   |<--(0x44) WEAPON_FIRE----------|
   |<--(0x20) WORLD_SNAPSHOT-------|
   |                              |
   (continues...)
```

12.4 Entity Destruction Flow

```
CLIENT                          SERVER
   |                              |
   |---(0x10) PLAYER_INPUT-------->|
   |        [SHOOT pressed]       |
   |                              |
   |                         [Spawn projectile]
   |                              |
   |<--(0x44) WEAPON_FIRE----------|
   |<--(0x21) ENTITY_SPAWN---------|
   |---(0x70) ACK----------------->|
   |---(0x70) ACK----------------->|
   |                              |
   |<--(0x20) WORLD_SNAPSHOT-------|
   |                              |
   |                          [Collision detected]
   |                          [Enemy destroyed]
   |                               |
   |<--(0x22) ENTITY_DESTROY-------|
   |<--(0x42) SCORE_UPDATE---------|
   |---(0x70) ACK----------------->|
   |---(0x70) ACK----------------->|
   |                               |
```

12.5 Player Death and Respawn

```
CLIENT                          SERVER
   |                               |
   |                          [Player takes damage]
   |                               |
   |<--(0x40) PLAYER_HIT-----------|
   |---(0x70) ACK----------------->|
   |                               |
   |                          [Health reaches 0]
   |                               |
   |<--(0x41) PLAYER_DEATH---------|
   |---(0x70) ACK----------------->|
   |                               |
   |                          [Wait respawn timer]
   |                               |
   |<--(0x21) ENTITY_SPAWN---------|
   |        [Player respawns]      |
   |---(0x70) ACK----------------->|
   |                               |
   |<--(0x20) WORLD_SNAPSHOT-------|
   |                               |
```

12.6 Graceful Disconnection

```
CLIENT                          SERVER
   |                               |
   |---(0x04) CLIENT_DISCONNECT--->|
   |        [FLAG_RELIABLE]        |
   |                               |
   |                          [Remove player]
   |                          [Notify others]
   |                               |
   |<--(0x70) ACK------------------|
   |                               |
   |                          Connection closed
```

13 Security Considerations

13.1 Input Validation

All received packets MUST be validated before processing:

```
bool validate_packet(const uint8_t* buffer, size_t length) {
```

```cpp
    // Minimum size check
    if (length < sizeof(PacketHeader)) {
        return false;
    }

    PacketHeader header;
    deserialize_header(buffer, &header);

    // Magic number validation
    if (header.magic != 0x5254) {
        return false;
    }

    // Packet type validation
    if (header.packet_type < 0x01 || header.packet_type > 0x7F) {
        return false;
    }

    // Size validation based on packet type
    size_t expected_size = get_expected_packet_size(header.packet_type)
        ;
    if (length < expected_size) {
        return false;
    }

    return true;
}
```

Listing 32: Packet Validation

13.2 Rate Limiting

Servers SHOULD implement rate limiting to prevent abuse:

- Maximum packets per second per client: 120
- Maximum connection attempts per IP per minute: 10
- Maximum reliable packet retransmissions: 5

13.3 Sequence Number Validation

```cpp
bool is_sequence_valid(uint32_t received_seq, uint32_t last_seq) {
    // Allow for some reordering (window of 100)
    const uint32_t MAX_SEQUENCE_WINDOW = 100;

    // Handle sequence number wraparound
    if (received_seq < last_seq) {
        // Check if it's a wraparound or an old packet
        uint32_t diff = last_seq - received_seq;
        return diff > (UINT32_MAX - MAX_SEQUENCE_WINDOW);
    }

    // Forward sequence check
    uint32_t diff = received_seq - last_seq;
    return diff <= MAX_SEQUENCE_WINDOW;
}
```

Listing 33: Sequence Number Validation

13.4 Buffer Overflow Prevention

```c
void safe_string_copy(char* dest, const char* src, size_t dest_size) {
    if (dest_size == 0) return;

    size_t i;
    for (i = 0; i < dest_size - 1 && src[i] != '\0'; i++) {
        dest[i] = src[i];
    }
    dest[i] = '\0';
}
```

Listing 34: Safe String Copy

13.5 Denial of Service Prevention

Servers MUST implement the following protections:

- Limit maximum packet size (1200 bytes)
- Timeout idle connections (10 seconds without heartbeat)
- Limit maximum entities per snapshot (64)
- Validate all array bounds
- Implement connection backlog limits

14 Performance Optimization

14.1 Snapshot Optimization

To reduce bandwidth usage for WORLD_SNAPSHOT packets:

```c
// Only send entities that changed since last snapshot
struct DeltaSnapshot {
    PacketHeader header;
    uint32_t base_tick;          // Reference tick
    uint16_t changed_entity_count;
    EntityState changed_entities[];
};
```

Listing 35: Delta Compression

14.2 Update Frequency Recommendations

- **PLAYER_INPUT**: Every frame or on change (30-60 Hz)
- **WORLD_SNAPSHOT**: 20-30 Hz (reduced from server tick rate)
- **HEARTBEAT**: 0.5-1 Hz
- **PING/PONG**: 1 Hz

14.3 Packet Batching

Multiple small packets MAY be combined into a single UDP datagram:

```
struct BatchedPacket {
    uint16_t magic;            // 0x5254
    uint16_t packet_count;     // Number of packets in batch
    // Followed by multiple packets with size prefix
    // [uint16_t size][packet data][uint16_t size][packet data]...
};
```

Listing 36: Packet Batching

15 Error Handling

15.1 Malformed Packets

Receivers MUST handle malformed packets gracefully:

- Invalid magic number: Discard silently
- Invalid packet type: Discard and log warning
- Invalid size: Discard and log warning
- Corrupted data: Discard and log error

15.2 Connection Loss Detection

```
void check_connection_timeout() {
    uint32_t current_time = get_current_time_ms();
    uint32_t time_since_last_packet = current_time - last_packet_time;

    if (time_since_last_packet > CONNECTION_TIMEOUT_MS) {
        // No packets received for too long
        handle_connection_lost();
    }
}

const uint32_t CONNECTION_TIMEOUT_MS = 10000;  // 10 seconds
```

Listing 37: Connection Loss Detection

15.3 Network Congestion Handling

When network congestion is detected (high packet loss, increased latency):

- Reduce WORLD_SNAPSHOT frequency
- Increase client-side prediction
- Prioritize critical packets (player death, spawns)
- Reduce entity count in snapshots

16 Implementation Guidelines

16.1 Creating New Packet Types

When adding new packet types, follow these guidelines:

1. Choose an appropriate type code in the correct range
2. Define the structure with proper alignment
3. Document all fields and their valid ranges
4. Implement serialization/deserialization functions
5. Add validation logic
6. Determine if reliability is needed
7. Update protocol version if breaking change

```
// 1. Define the packet structure
struct NewPacketType {
    PacketHeader header;          // Always include header
    uint32_t field1;              // Document each field
    uint16_t field2;
    uint8_t field3;
    // ... additional fields
};

// 2. Implement serialization
void serialize_new_packet(const NewPacketType* packet,
                          uint8_t* buffer) {
    // Serialize header
    serialize_header(&packet->header, buffer);

    // Serialize payload in network byte order
    uint32_t* buf32 = (uint32_t*)(buffer + 12);
    uint16_t* buf16 = (uint16_t*)(buffer + 16);

    buf32[0] = htonl(packet->field1);
    buf16[0] = htons(packet->field2);
    buffer[18] = packet->field3;
}

// 3. Implement deserialization
void deserialize_new_packet(const uint8_t* buffer,
                            NewPacketType* packet) {
    // Deserialize header
    deserialize_header(buffer, &packet->header);

    // Deserialize payload from network byte order
    const uint32_t* buf32 = (const uint32_t*)(buffer + 12);
    const uint16_t* buf16 = (const uint16_t*)(buffer + 16);

    packet->field1 = ntohl(buf32[0]);
    packet->field2 = ntohs(buf16[0]);
    packet->field3 = buffer[18];
}

```

```
// 4. Implement validation
bool validate_new_packet(const NewPacketType* packet) {
    // Validate field ranges
    if (packet->field3 > MAX_VALID_VALUE) {
        return false;
    }
    return true;
}
```

Listing 38: New Packet Type Template

16.2 Serialization Best Practices

- Always use network byte order (big-endian)
- Pack structures tightly (no padding)
- Use fixed-size integer types (uint8_t, uint16_t, uint32_t)
- Avoid floating-point types in wire format
- Document byte offsets for all fields
- Consider alignment requirements

16.3 Testing Recommendations

- Test packet serialization/deserialization round-trips
- Simulate packet loss (random drop)
- Simulate packet reordering
- Simulate packet duplication
- Test with high latency (250+ ms)
- Test with bandwidth constraints
- Fuzz test with malformed packets
- Test sequence number wraparound

17 Appendix A: Quick Reference

17.1 Packet Type Summary

Code	Name	Reliable	Direction
0x01	CLIENT_CONNECT	No	C→S
0x02	SERVER_ACCEPT	No	S→C
0x03	SERVER_REJECT	No	S→C
0x04	CLIENT_DISCONNECT	Yes	C↔S
0x05	HEARTBEAT	No	C→S
0x10	PLAYER_INPUT	No	C→S

Continued on next page

Table 3 – continued from previous page

Code	Name	Reliable	Direction
0x20	WORLD_SNAPSHOT	No	S→C
0x21	ENTITY_SPAWN	Yes	S→C
0x22	ENTITY_DESTROY	Yes	S→C
0x23	ENTITY_UPDATE	No	S→C
0x40	PLAYER_HIT	Yes	S→C
0x41	PLAYER_DEATH	Yes	S→C
0x42	SCORE_UPDATE	No	S→C
0x43	POWERUP_PICKUP	Yes	S→C
0x44	WEAPON_FIRE	No	S→C
0x60	GAME_START	Yes	S→C
0x61	GAME_END	Yes	S→C
0x62	LEVEL_COMPLETE	Yes	S→C
0x63	LEVEL_START	Yes	S→C
0x70	ACK	No	C↔S
0x71	PING	No	C→S
0x72	PONG	No	S→C

17.2 Common Values

- **Magic Number**: 0x5254 ('RT')
- **Protocol Version**: 1
- **Default Port**: 4242 (UDP)
- **Max Packet Size**: 1200 bytes
- **Connection Timeout**: 10000 ms
- **Heartbeat Interval**: 1000 ms
- **ACK Timeout**: 500-1000 ms
- **Max Retries**: 5

17.3 Entity Type Codes

- 0x00: Player
- 0x01-0x0F: Enemies (various types)
- 0x10-0x1F: Projectiles
- 0x20-0x2F: Powerups
- 0x30-0x3F: Obstacles
- 0x40-0x4F: Background elements

18 Appendix B: Example Implementation

18.1 Complete Client Connection Example

```cpp
#include <required.h>
#include <required.h>
#include <required.h>
#include <required.h>

class RTypeClient {
private:
    int sockfd;
    struct sockaddr_in server_addr;
    uint32_t sequence_number;
    uint32_t player_id;
    bool connected;

public:
    RTypeClient() : sockfd(-1), sequence_number(0),
                    player_id(0), connected(false) {}

    bool connect(const char* server_ip, uint16_t port) {
        // Create UDP socket
        sockfd = socket(AF_INET, SOCK_DGRAM, 0);
        if (sockfd < 0) {
            return false;
        }

        // Setup server address
        memset(&server_addr, 0, sizeof(server_addr));
        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(port);
        inet_pton(AF_INET, server_ip, &server_addr.sin_addr);

        // Send CLIENT_CONNECT
        ClientConnect connect_packet;
        connect_packet.header.magic = 0x5254;
        connect_packet.header.packet_type = 0x01;
        connect_packet.header.flags = 0;
        connect_packet.header.sequence_number = sequence_number++;
        connect_packet.header.timestamp = get_current_time_ms();
        connect_packet.protocol_version = 1;
        strncpy(connect_packet.player_name, "Player1", 31);
        connect_packet.client_id = generate_client_id();

        // Serialize and send
        uint8_t buffer[256];
        serialize_client_connect(&connect_packet, buffer);
        sendto(sockfd, buffer, sizeof(ClientConnect), 0,
               (struct sockaddr*)&server_addr, sizeof(server_addr));

        // Wait for SERVER_ACCEPT
        if (wait_for_accept()) {
            connected = true;
            return true;
        }

        return false;
    }

    void send_input(uint16_t input_flags) {
        if (!connected) return;

        PlayerInput input_packet;
        input_packet.header.magic = 0x5254;
        input_packet.header.packet_type = 0x10;
        input_packet.header.flags = 0;
        input_packet.header.sequence_number = sequence_number++;
        input_packet.header.timestamp = get_current_time_ms();
        input_packet.player_id = player_id;
        input_packet.input_flags = input_flags;
        input_packet.aim_x = 0;
        input_packet.aim_y = 0;

        uint8_t buffer[256];
        serialize_player_input(&input_packet, buffer);
        sendto(sockfd, buffer, sizeof(PlayerInput), 0,
               (struct sockaddr*)&server_addr, sizeof(server_addr));
    }

    void disconnect() {
        if (!connected) return;

        ClientDisconnect disconnect_packet;
        disconnect_packet.header.magic = 0x5254;
        disconnect_packet.header.packet_type = 0x04;
        disconnect_packet.header.flags = FLAG_RELIABLE;
        disconnect_packet.header.sequence_number = sequence_number++;
        disconnect_packet.header.timestamp = get_current_time_ms();
        disconnect_packet.player_id = player_id;
        disconnect_packet.reason = 0x00; // Normal disconnect

        uint8_t buffer[256];
        serialize_client_disconnect(&disconnect_packet, buffer);
        sendto(sockfd, buffer, sizeof(ClientDisconnect), 0,
               (struct sockaddr*)&server_addr, sizeof(server_addr));

        connected = false;
        close(sockfd);
    }
};
```

Listing 39: Client Connection Implementation

18.2 Complete Server Example

```cpp
#include <required.h>
#include <required.h>
#include <required.h>

class RTypeServer {
private:
    int sockfd;
    std::map<uint32_t, ClientInfo> clients;
    uint32_t next_player_id;
    uint32_t sequence_number;

public:
    RTypeServer() : sockfd(-1), next_player_id(1),
                    sequence_number(0) {}

    bool start(uint16_t port) {
        sockfd = socket(AF_INET, SOCK_DGRAM, 0);
        if (sockfd < 0) return false;

        struct sockaddr_in server_addr;
        memset(&server_addr, 0, sizeof(server_addr));
        server_addr.sin_family = AF_INET;
        server_addr.sin_addr.s_addr = INADDR_ANY;
        server_addr.sin_port = htons(port);

        if (bind(sockfd, (struct sockaddr*)&server_addr,
                 sizeof(server_addr)) < 0) {
            return false;
        }

        return true;
    }

    void run() {
        uint8_t buffer[1200];
        struct sockaddr_in client_addr;
        socklen_t addr_len = sizeof(client_addr);

        while (true) {
            ssize_t received = recvfrom(sockfd, buffer, sizeof(buffer),
                                        0, (struct sockaddr*)&
                                            client_addr,
                                        &addr_len);

            if (received < 0) continue;

            // Validate packet
            if (!validate_packet(buffer, received)) continue;

            // Process packet
            PacketHeader header;
            deserialize_header(buffer, &header);

            switch (header.packet_type) {
                case 0x01: // CLIENT_CONNECT
                    handle_client_connect(buffer, &client_addr);
                    break;
                case 0x10: // PLAYER_INPUT
                    handle_player_input(buffer);
                    break;
                case 0x04: // CLIENT_DISCONNECT
                    handle_client_disconnect(buffer);
                    break;
                // ... other packet types
            }
        }
    }

    void handle_client_connect(const uint8_t* buffer,
                               struct sockaddr_in* addr) {
        ClientConnect packet;
        deserialize_client_connect(buffer, &packet);

        // Validate and accept
        ServerAccept accept;
        accept.header.magic = 0x5254;
        accept.header.packet_type = 0x02;
        accept.header.flags = 0;
        accept.header.sequence_number = sequence_number++;
        accept.header.timestamp = get_current_time_ms();
        accept.assigned_player_id = next_player_id++;
        accept.max_players = 4;
        accept.game_instance_id = 1;
        accept.server_tick_rate = 60;

        uint8_t send_buffer[256];
        serialize_server_accept(&accept, send_buffer);
        sendto(sockfd, send_buffer, sizeof(ServerAccept), 0,
               (struct sockaddr*)addr, sizeof(*addr));
    }
};
```

Listing 40: Server Implementation Skeleton

19 Appendix C: Packet Size Reference

Packet Type	**Size (bytes)**	**Notes**
CLIENT_CONNECT	49	Fixed size
SERVER_ACCEPT	23	Fixed size
SERVER_REJECT	77	Fixed size
CLIENT_DISCONNECT	17	Fixed size
HEARTBEAT	16	Fixed size
PLAYER_INPUT	24	Fixed size
WORLD_SNAPSHOT	18 + 16n	Variable, n = entity count
ENTITY_SPAWN	26	Fixed size
ENTITY_DESTROY	21	Fixed size
ENTITY_UPDATE	26	Fixed size
PLAYER_HIT	29	Fixed size
PLAYER_DEATH	30	Fixed size
SCORE_UPDATE	23	Fixed size
POWERUP_PICKUP	22	Fixed size
WEAPON_FIRE	31	Fixed size
GAME_START	36	Fixed size
GAME_END	34	Fixed size
LEVEL_COMPLETE	22	Fixed size
LEVEL_START	47	Fixed size
ACK	20	Fixed size
PING	16	Fixed size
PONG	20	Fixed size

20 Appendix D: Bandwidth Calculations

20.1 Typical Bandwidth Usage

Assuming a 4-player game with typical update frequencies:

Client Upload (per client):

- PLAYER_INPUT: 24 bytes × 60 Hz = 1,440 bytes/s
- HEARTBEAT: 16 bytes × 1 Hz = 16 bytes/s
- PING: 16 bytes × 1 Hz = 16 bytes/s
- **Total Upload:** ∼1.5 KB/s = 12 Kbps

Server Broadcast (per client):

- WORLD_SNAPSHOT (40 entities): (18 + 16×40) bytes × 30 Hz = 19,740 bytes/s
- ENTITY_SPAWN: 26 bytes × 10/s = 260 bytes/s (average)
- ENTITY_DESTROY: 21 bytes × 10/s = 210 bytes/s (average)
- WEAPON_FIRE: 31 bytes × 20/s = 620 bytes/s (average)
- PONG: 20 bytes × 1 Hz = 20 bytes/s
- ACKs: Variable, ∼200 bytes/s
- **Total Download:** ∼21 KB/s = 168 Kbps

Server Total Bandwidth (4 clients):

- Upload: 21 KB/s × 4 = 84 KB/s = 672 Kbps
- Download: 1.5 KB/s × 4 = 6 KB/s = 48 Kbps
- **Total Server:** ∼90 KB/s = 720 Kbps

20.2 Optimization Strategies

To reduce bandwidth when needed:

1. **Reduce snapshot frequency:** 30 Hz → 20 Hz saves 33% on WORLD_SNAPSHOT
2. **Send only visible entities:** Culling off-screen entities can reduce entity count by 50-70%
3. **Delta compression:** Only send changed entities, potentially reducing by 80%
4. **Priority system:** Send closer/important entities more frequently
5. **Quantization:** Already implemented with int16 positions

21 Appendix E: Error Codes Reference

21.1 Disconnect Reasons (0x04)

Code	Description
0x00	Normal disconnect (user quit)
0x01	Connection timeout (no heartbeat)
0x02	Kicked by server (admin action)
0x03	Client error (crash/exception)

21.2 Server Reject Reasons (0x03)

Code	Description
0x00	Server full (max players reached)
0x01	Incompatible protocol version
0x02	Invalid player name (empty, too long, invalid chars)
0x03	Banned client (IP or client_id banned)
0xFF	Generic error (server internal error)

21.3 Entity Destroy Reasons (0x22)

Code	Description
0x00	Killed by player projectile
0x01	Killed by enemy projectile
0x02	Out of bounds (left play area)
0x03	Timeout/despawn (lifetime expired)
0x04	Level transition (new level loading)

21.4 Score Update Reasons (0x42)

Code	Description
0x00	Enemy killed (standard)
0x01	Boss killed (major bonus)
0x02	Powerup collected
0x03	Level completed (completion bonus)
0x04	Bonus points (combo, time bonus, etc.)

22 Appendix F: Debugging and Logging

22.1 Recommended Log Format

For debugging network issues, implement structured logging:

```
void log_packet(const char* direction, const PacketHeader* header) {
    printf("[%s] Type=0x%02X Seq=%u Flags=0x%02X Time=%u\n",
           direction,
           header->packet_type,
           header->sequence_number,
           header->flags,
           header->timestamp);
}

// Usage:
log_packet("SEND", &packet.header);  // \rightarrow
log_packet("RECV", &packet.header);  // \leftarrow
```

Listing 41: Packet Logging Format

22.2 Network Statistics

Track these metrics for debugging and monitoring:

- **Packets sent/received:** Total count per type
- **Bytes sent/received:** Bandwidth usage
- **Packet loss rate:** Percentage of lost packets
- **Round-trip time (RTT):** Average and peak latency
- **Retransmissions:** Count of reliable packet retries
- **Out-of-order packets:** Sequence number gaps
- **Duplicate packets:** Same sequence received multiple times

```
struct NetworkStats {
    uint64_t packets_sent;
    uint64_t packets_received;
    uint64_t bytes_sent;
    uint64_t bytes_received;
    uint32_t packets_lost;
    uint32_t packets_retransmitted;
    uint32_t duplicates_received;
    uint32_t out_of_order_received;
    float avg_rtt_ms;
    float peak_rtt_ms;

    void print_stats() {
        printf("=== Network Statistics ===\n");
        printf("Packets: Sent=%llu Recv=%llu Lost=%u (%.2f%%)\n",
               packets_sent, packets_received, packets_lost,
               (float)packets_lost / packets_sent * 100.0f);
        printf("Bytes: Sent=%llu Recv=%llu\n",
               bytes_sent, bytes_received);
        printf("RTT: Avg=%.2fms Peak=%.2fms\n",
               avg_rtt_ms, peak_rtt_ms);
        printf("Retransmissions: %u\n", packets_retransmitted);
    }
};
```

Listing 42: Network Statistics Structure

23 Appendix G: Future Extensions

23.1 Potential Protocol Extensions

The protocol can be extended in future versions to support:

- **Voice chat:** New packet types for audio streams (0x80-0x8F range)

- **Lobby system:** Room management packets (0x90-0x9F range)
- **Replay system:** Recorded game data packets
- **Server browser:** Server info query packets
- **Anti-cheat:** Validation and integrity check packets
- **Spectator mode:** Observe-only connections
- **NAT traversal:** Peer-to-peer connection establishment

23.2 Version Negotiation

When introducing breaking changes, update the protocol version:

```
// In CLIENT_CONNECT:
protocol_version = 2;  // New version

// Server checks:
if (packet.protocol_version != SUPPORTED_VERSION) {
    send_rejection(SERVER_REJECT_INCOMPATIBLE_VERSION);
}
```

Listing 43: Version Negotiation

For backward compatibility, the server MAY support multiple protocol versions simultaneously by maintaining version-specific packet handlers.

24 Appendix H: Testing Checklist

24.1 Unit Tests

- ☐ Packet serialization/deserialization for all types
- ☐ Byte order conversion (endianness)
- ☐ Quantization/dequantization accuracy
- ☐ Header validation (magic, type, size)
- ☐ Sequence number handling and wraparound
- ☐ String safety (buffer overflow prevention)

24.2 Integration Tests

- ☐ Client connection and disconnection
- ☐ Multiple clients simultaneously
- ☐ Reliable packet delivery and ACK
- ☐ Packet loss handling
- ☐ Out-of-order packet handling
- ☐ Duplicate packet detection
- ☐ Connection timeout and recovery
- ☐ High latency scenarios (250+ ms)

24.3 Performance Tests

☐ Bandwidth usage under normal load

☐ CPU usage for packet processing

☐ Memory usage for packet queues

☐ Maximum concurrent clients

☐ Packet processing throughput

☐ Network saturation recovery

24.4 Security Tests

☐ Malformed packet rejection

☐ Oversized packet handling

☐ Rate limiting effectiveness

☐ Invalid magic number handling

☐ Sequence number spoofing prevention

☐ Buffer overflow protection

25 Conclusion

This protocol specification provides a solid foundation for implementing the R-Type multiplayer game. The binary format ensures efficient bandwidth usage, while the reliability mechanism guarantees critical game events are delivered. The modular design allows for easy extension and maintenance as the game evolves.

Key takeaways:

- **Efficiency:** Binary format with quantization minimizes bandwidth
- **Reliability:** Selective ACK system for critical packets
- **Scalability:** Supports multiple clients and game instances
- **Security:** Input validation and rate limiting prevent abuse
- **Extensibility:** Reserved packet ranges and flags for future features

Developers implementing this protocol should refer to the packet type reference (Appendix A), example implementations (Appendix B), and testing checklist (Appendix H) to ensure correct and robust implementation.

25.1 Version History

- **Version 1.0** (2025-01-XX): Initial protocol specification
  - Core packet types for connection, input, world state, and events
  - Reliability mechanism with selective ACK
  - Position and velocity quantization
  - Security and validation guidelines

26 Appendix I: References

- RFC 768: User Datagram Protocol
  `https://www.rfc-editor.org/rfc/rfc768`
- RFC 2119: Key words for use in RFCs to Indicate Requirement Levels
  `https://www.rfc-editor.org/rfc/rfc2119`
- Gaffer On Games: "Networked Physics"
  `https://gafferongames.com/post/networked_physics_2004/`
- Valve Developer Community: "Source Multiplayer Networking"
  `https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking`
- Gabriel Gambetta: "Fast-Paced Multiplayer" (Client-Side Prediction)
  `https://www.gabrielgambetta.com/client-side-prediction-server-reconciliation.html`
- Glenn Fiedler: "UDP vs TCP"
  `https://gafferongames.com/post/udp_vs_tcp/`
- "Quake 3 Network Model"
  `https://fabiensanglard.net/quake3/network.php`

27 Appendix J: Glossary

ACK (Acknowledgment) A confirmation message sent to indicate successful receipt of a packet.

Big-endian Byte ordering where the most significant byte is stored first (network byte order).

Datagram A self-contained, independent packet of data sent over UDP.

Dequantization Converting compressed integer representation back to floating-point values.

Entity Any game object (player, enemy, projectile, powerup, etc.).

Latency The time delay between sending and receiving a packet (also called "lag").

Magic Number A constant value used to validate packet format (0x5254 for R-Type).

Packet Loss When network packets fail to reach their destination.

Quantization Converting floating-point values to smaller integer representation to save bandwidth.

RTT (Round-Trip Time) The time for a packet to travel from sender to receiver and back.

Sequence Number A monotonically increasing counter used to detect packet loss and duplication.

Snapshot A complete state update of the game world at a specific point in time.

TCP (Transmission Control Protocol) A reliable, connection-oriented transport protocol.

UDP (User Datagram Protocol) An unreliable, connectionless transport protocol optimized for speed.

World Tick A discrete time step in the server's game simulation (typically 60 Hz = 16.67ms per tick).

End of Document

For questions, issues, or contributions to this protocol specification,
please contact our R-Type development team.