RFC 0xXXL: Server-Client Communication for Multiplayer R-Type Game

Authors: Omer DEDO and Christ HOUNKANRIN

Date: 01/11/ 2024

RFC Number: 0xXXL

Category: Game Networking Standardization

Table

1. Introduction

This RFC defines a **multiplayer communication protocol** for an R-Type styled multiplayer game. The protocol is designed to handle real-time communication between a server and multiple clients using **UDP** as the transport layer and **Boost Asio** for asynchronous I/O operations. The primary focus is on the exchange of player positions, actions such as movement and shooting, and game status updates.

The protocol uses a **binary message format** and incorporates basic security measures via encryption to protect in-game communication. This RFC will outline the necessary steps and message formats required to implement both server and client functionalities in the game.

2. Protocol Overview

The system consists of a central **UDP server** and multiple **clients**. The server manages the overall game state and communicates with each client in real-time to broadcast player actions and updates. Each client sends requests to the server and receives responses, ensuring that all connected players are synchronized.

- **Server Responsibilities**: Manage player connections, game state, and send updates to all clients.
- **Client Responsibilities**: Send movement and action requests, and update its state based on server responses.

The server operates in a non-blocking manner using **asynchronous UDP sockets** to handle communication with clients.

3. Communication Protocol

3.1 Message Structure

All communication between the client and server follows a **message structure** designed to encapsulate the necessary game data.

This structure is transmitted using a binary protocol for efficiency and encrypted for security.

Field	Type	Description
header	uint16_t	Identifies the type of message
id	uint16_t	Unique ID assigned to each player
message	std::array<char,10>	Any additional message payload (if applicable)
pos_x	uint16_t	Player's X-coordinate
pos_y	uint16_t	Player's Y-coordinate

3.2 Message Headers

The message headers are crucial in defining the type of action or response involved in client-server communication. Below is a table that outlines each possible header and its meaning:

Header	Description
0x000	Error Message (Invalid message received)
0x01	Connection Request
0x02	Login Request / Username Prompt
0x03	Position and ID Assignment
0x04	Start Game Command
0x05A	Move Up(key up)
0x05B	Move Down(key down)
0x05C	Move Left(key left)

0x05D	Move Right(key right)
0x051	Broadcast Player Position to All Clients
0x06	Player Shooting Action(key A)
0x07	Death Notification
0x09(A, B, C, D, F)	Enemy Spawn Notification
0x010	Deconnexion Player Notification
0x011	Server shutdown Notification
0x015	Game Over
0x020	Next level

Error Handling (Header `0x000`):
If the server or client receives a malformed or invalid message, it will respond with an error message using the `0x000` header. This message will contain the following information:

- `header = 0x000`: Error message indicator.
- `message`: Contains the original erroneous message or a brief description of the error.

The error message ensures that both client and server can handle invalid data gracefully and notify the user or developer about the issue.

4. Client-Server Workflow

4.1 Connection and Login

The interaction between the client and server begins with a connection request and a subsequent login process:

Client Connection:

- The client sends a message with the `header = 0x01` to request a connection.
- The server responds with `header = 0x02`, asking the client for its username.

Login Process:

- The client responds with its username in a message with `header = 0x02`.
- The server generates a unique `id` for the client and assigns initial coordinates (`pos_x`, `pos_y`).

4.2 Starting the Game

- Once the server detects that the minimum number of clients is connected, it sends a **start message** (`header = 0x04`) to all clients. This signals that the game has officially begun.
- Clients wait for the start signal before taking any game actions.

4.3 Player Movements

- When a client wishes to move, it sends a **movement request** to the server using one of the sub-headers of `0x05`:
  - **0x05A**: Move Up
  - **0x05B**: Move Down
  - **0x05C**: Move Left
  - **0x05D**: Move Right
- The server updates the player's position accordingly and broadcasts this information to all connected clients using `header = 0x051`.

4.4 Shooting

- A client initiates a **shooting action** by sending a message with `header = 0x06`. This includes the player's current position (`pos_x`, `pos_y`) and their unique ID (`id`).
- The server processes the shooting action and informs all clients about the shot, ensuring the state remains synchronized.

4.5 Player and Enemy Updates

- **Player Death**: When a player is killed, the server sends a message with `header = 0x07` to notify all other clients about the event.
- **Enemy Spawn and Attack**:
  - Enemies are introduced to the game via the server, which sends a message with `header = 0x09` to all clients, indicating the spawn of an enemy.
  - The enemy `header's:`

```
0x09A => Battleship

0x09B => Crevette

0x09C => Boul

0x09D => Commandant

0x09F => First Boss
```

5. Network Protocol and Security

5.1 UDP as Transport Layer

The game uses **UDP (User Datagram Protocol)** for communication due to its low-latency, connectionless nature. UDP is chosen over TCP for real-time applications like multiplayer games, where minimizing delay is critical, and occasional packet loss can be tolerated.

- **Advantages**:
    - Fast transmission of packets.
    - No handshake mechanism, reducing latency.
    - Suitable for high-frequency updates, such as player positions.
- **Disadvantages**:
    - Lack of delivery guarantee. Some messages may be lost.
    - No built-in congestion control, which can lead to dropped packets.

To mitigate these disadvantages, the server frequently sends updates to keep the clients synchronized, even if some packets are lost.

5.2 Binary Protocol, Encryption and Compression for Efficiency

All messages are encoded in **binary format** for efficient transmission and then **encrypted** using an internal protocol to prevent unauthorized access or tampering. The encryption ensures that:

- Only authenticated clients and the server can interpret the data.
- The game's critical information (like player positions or actions) is protected from potential attackers.

This encryption step is essential, especially in competitive multiplayer environments, where data integrity and confidentiality are vital.

Data Compression : To optimize bandwidth, messages are compressed using zlib. Compression reduces the message size, mitigating the risk of packet drops and minimizing latency.

6. Conclusion

This RFC outlines the communication protocol for a real-time, multiplayer R-Type styled game. By utilizing **UDP** and **Boost Asio**, the game ensures fast, non-blocking communication between the server and clients. The message structure, headers, and the binary encryption protocol provide a secure and efficient way to exchange data. Additionally, the inclusion of an **error handling mechanism** with the `0x000` header ensures that both the server and clients can gracefully manage any incorrect

or malformed messages. This document serves as a guide for implementing both the client and server-side components, with the flexibility to extend the protocol as the game evolves.

7. References

- Boost Asio Documentation: *https://www.boost.org/doc/libs/1_75_0/doc/html/boost_asio.html*
- UDP Protocol: *https://tools.ietf.org/html/rfc768*