R-Type Scoobygang group
Version: 9
Updates: 43

L. Chloé
D. Damien
G. Vivant
S. Clovis
J. Mylo

Category: Standards RFC

Epitech Oct. 2022

RFC 6666 | RTYPE

Status of This Memo

This protocol has been created as the official RFC for the R-TYPE project. R-TYPE is a 3rd year student project in Epitech. This protocol should help you understand the logics behind our network system. Distribution of this memo is unlimited.

Copyright Notice

Abstract | Project presentation

This RFC is based on a game that is informally called a Horizontal Shmup (e.g. Shoot’em’up), and while R-Type is not the first one of its category, this one has been a huge success amongst gamers in the 90’s, and had several ports, spin-offs, and 3D remakes on modern systems. Other similar and well known games are the Gradius series and Blazing Star on Neo Geo. As you now understand, you have to make your own version of R-Type. .. but with a twist not featured in the original game (nor in the remakes by the way): Our version is a network game, where one-to-four players will be able to fight together the evil Bydos!

Please bear in mind that the technologies used to create the communication behind this protocol is based on boost::asio vocabulary. The overall syntax of this RFC is based on RFC 7990.

Table of Contents

1. Introduction

This document describes a TCP/UDP scheme, called service: TCP/UDP, which defines network access information for network services using a formal notation. In addition, it describes how to define a set of attributes to associate with a service: TCP/UDP.

These attributes will allow end users and programs to select between network services of the same type that have different capabilities.

The attributes are defined in a template document that is readable by people and machines.

The key words "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 [6].

1.1. Architecture

Our Network architecture is a basic client/server interaction protocol, it combines TCP for connection to the server, and UDP for game related interactions.

Thus our communication architecture is structured around an initial TCP connection in order to prepare the UDP connection. More on this in 2.1.

1.2. Connection specifications

The overall connection system is designed to work on machine scoped environment and local network environment. Thus one machine can run the server while other machines can connect to it as clients. Or One machine can both run servers, and initiate clients.

2. R-TYPE TCP protocol

As told just before, all the communications between the server and the client before the game start are done through TCP protocol. We made that choice because we need to have a great degree of verification going when a client connects to the server (we don't want intruders to easily connect to the server.

2.1. TCP payload (general view)

TCP is a reliable stream delivery service which guarantees that all bytes received will be identical and in the same order as those sent. Since packet transfer by many networks is not reliable, TCP achieves this using a technique known as positive acknowledgement with re-transmission. This requires the receiver to respond with an acknowledgement message as it receives the data.

Every transmission done with the RFC6666 protocol will require a payload containing these 3 elements:

```
ACTION_TYPE ACTION;
int BODY_SIZE;
void *BODY;
```

* Where ACTION is the action that is requested to process BODY data
* BODY_SIZE is the size in bytes of BODY (has to be of value 0 if no BODY)
* and BODY is the data to be processed (considering ACTION_NAME corresponds to a process on the endpoint)

To send a payload, each packet has to be written on the TCP socket in this order: ACTION_NAME, BODY_SIZE, BODY.

ACTION corresponds to an ACTION_TYPE, Which can be one of those :

```
enum ACTION_TYPE {
   OK,
   KO,
   CONNECT,
   CREATE_LOBBY,
   LIST_LOBBIES,
   JOIN_LOBBY,
   PLAYER_JOINED_LOBBY,
   START_GAME,
   GAME_STARTING
};
```

In order to receive a payload, you have to read the 4 first bytes (ACTION), then read the 4 following bytes (BODY_SIZE) and finally read the BODY_SIZE bytes long BODY.

2.2. TCP payload ORDERS AND ANSWERS

In order to play the game on the network, you have to connect to the server that maintains the lobbies.

```
   ACTION == CONNECT
BODY_SIZE == PLAYER_NAME size (in bytes)
     BODY == PLAYER_NAME
```

the server responds with a TCP payload that resumes in :

```
SUCCESS -------->  ACTION == OK
FAILURE -------->  ACTION == KO
```

Know that body may contain specific data for precision purposes (For example, a string containing information about an error), thus those PAYLOAD may or may not contain a BODY.

When you are connected to the server, you are now able to create, join or list lobbies :

The Client chooses the lobby's name, by default it is "R-TYPE SERVER"

```
   ACTION == CREATE_LOBBY
BODY_SIZE == LOBBY_NAME size (in bytes)
     BODY == LOBBY_NAME ("R-TYPE SERVER" by default)
```

the server responds with a TCP payload that resumes in :

```
SUCCESS -------->  ACTION == OK
FAILURE -------->  ACTION == KO
```

Know that body may contain specific data for precision purposes (For example, a string containing information about an error), thus those PAYLOAD may or may not contain a BODY.

The Client chooses the lobby he wants to join.

```
         ACTION == JOIN_LOBBY
      BODY_SIZE == LOBBY_NAME size (in bytes)
           BODY == LOBBY_NAME ("R-TYPE SERVER" by default)
```

the server responds with a TCP payload that resumes in :

```
            SUCCESS --------->  ACTION == OK
                               BODY_SIZE == PLAYERS_NAMES size (in bytes)
                                    BODY == PLAYERS_NAMES
```

where PLAYERS_NAMES is a vector of PLAYER_NAME which is a std::string
containing the player's nickname.

```
          SUCCESS ----ALL OTHER CLIENTS---->  ACTION == PLAYER_JOINED_LOBBY
                                             BODY_SIZE == PLAYER_NAME size
                                                  BODY == PLAYER_NAME


            FAILURE --------->  ACTION == KO
```

Know that body of KO ACTION may contain specific data for precision
purposes (For example, a string containing information about an error),
thus those PAYLOAD may or may not contain a BODY.

The Client simply sends a list of all lobbies

```
ACTION == LIST_LOBBIES
```

the server responds with a TCP payload that resumes in :

```
SUCCESS --------> ACTION == OK
                  BODY_SIZE == LOBBIES_LIST size (in bytes)
                       BODY == LOBBIES_LIST
```

LOBBIES_LIST is a vector of LOBBY_DATA, LOBBY_DATA is a structure that contains the name of the lobby, and the number of player in it :

```
struct LOBBY_DATA {
   std::string lobby_name;
   int player_count;
};
```

You can deduce the number of lobby in the table LOBBIES_LIST by dividing BODY_SIZE by sizeof(LOBBY_DATA).

```
FAILURE --------> ACTION == KO
```

Know that body may contain specific data for precision purposes (For example, a string containing information about an error), thus those PAYLOAD may or may not contain a BODY.

All clients can start the game when they want, to do so they simply send to the server this payload :

```
ACTION == START_GAME
```

the server responds with a TCP payload that resumes in :

```
SUCCESS ----ALL CLIENTS----> ACTION == GAME_STARTING
```

The clients in the lobby, receiving the GAME_STARTING ACTION, know that they now have to start receiving UDP data from the server.

```
FAILURE ------------------> ACTION == KO
```

Know that body may contain specific data for precision purposes (For example, a string containing information about an error), thus those PAYLOAD may or may not contain a BODY.

3. R-TYPE UDP protocol

After the connection between the server and the client is done, the game launches and an other protocol takes action. The UDP protocol. It permits faster transmissions which is ideal for a game like ours, but at the cost of potential lost of packets which should be resolved by interpolation.

UDP (User Datagram Protocol) is a communications protocol, used across the Internet. It is specifically used for time-sensitive transmissions such as video playback or DNS lookups. It works by creating a connectionless transmission model that requires a minimum protocol mechanism. UDP is a part of Internet Protocol (IP) suite, referred to as UDP/IP suite.

UDP enables process-to-process communication and works by using Internet Protocols to encapsulate data in a UDP packet. This packed has its header information edited to include; source, destination ports, packet length and checksum. After UDP packets are encapsulated in an Internet Protocol packet, they’re sent off to their destinations.

UDP is an alternative protocol to TCP, as it is primarily used for establishing low-latency and loss-tolerating connections between applications or softwares on the internet.

One of the main advantages that provides our architecture is the unnecessary need for sprite streaming.

We don't need to send to the client the position of every part of the game at each ticks, because both the server and the client have pretty much the same engine. So, when a client performs an action, like a Key-Press, this action is sent to all of the Client so that they can run the consequences of that event through their own game logic.

The server which is between all the clients, gets that event and tries to predict what the consequences of that event should be. It then sends the expected game data to all clients so that they can adjust.

3.1. UDP from Client (INPUT)

The client can send 3 types of events to the server as listed below, at every event the client has to send to the server a payload containing information about the event. The payload begins with the EVENT type on 4 bytes (the same way as ACTION in the TCP protocol).

The Client Generated Events can be :

```
enum EVENT {
   UP,
   RIGHT,
   DOWN,
   LEFT,
   SHOT,
   QUIT
};
```

THE UP, RIGHT, DOWN, LEFT EVENTS :

Those events are movement related, it is only 4 bytes long.

THE SHOOT EVENT :

The server is authoritative, so we don't need to send the player position in order for the shooting to happen.
this event is only 4 bytes long.

THE QUIT EVENT :

The client may choose to quit the game, this event lets the server know that this connection disruption is definitive

3.2. UDP from Server (SYNCHRONISATION)

In our game network architecture, the server is authoritative, meaning he always has the last word to what is happening in the game. When he receives the Player EVENTS (as listed before) he transmits the event to the other clients, and also sends some payload to make sure every client connected to the game on the same server has the same version of the game (sprites positions, velocity, health...).

The server has to send specific components of all the entity that possess the synced component.
Every entity that is refreshed onto the client is refreshed through a payload, this payload is always the same size for consistency reason :

```
typedef struct server_payload_s {
    int COMPONENT_NAME;
    int syncId;
    float valueA;
    float valueB;
    float valueC;
    bool shot;
} server_payload_t;
```

Having an always same size payload helps us gain time on casting our raw binary data into the refreshing value.
Thus, we only need to receive the COMPONENT_NAME, and grab the necessary values depending on the COMPONENT_NAME received.
Those COMPONENT_NAME are the one part of this enum :

```
enum COMPONENTS_SYNCED {
    POSITION,
    VELOCITY,
    ENEMY_STATS,
    PLAYER_STATS
};
```

4. Multi-Machine behaviorism

Please bear in mind that this network protocol is designed to be working on multi machine architectures and local environments. This protocol doesn’t have supplementary layers to handle high level resolve on network wide IPS.

When building with our network architecture, it is needed to deactivate general and specific firewalls. You also have to provide the public ip address of the server yourself. You can always find it with your os ip-listing binary. (ip a …)

A. Acknowledgments

Thanks to Matteo Volpi and Adrien beaussier for assisting with the design of this communication protocol. Thanks to Marius for the sharing of information about TCP.

B. References

[1] Protocol and service names, October 1994.
ftp://ftp.isi.edu/in-notes/iana/assignments/service-names.

[2] Port numbers, July 1997.
ftp://ftp.isi.edu/in-notes/iana/assignments/port-numbers.

[3] Alvestrand, H., "Tags for the Identification of Languages", RFC 1766, March 1995.

[4] ANSI. Coded Character Set -- 7-bit American Standard code for Information Interchange. X3.4-1986, 1986.

[5] Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, August 1998.

C. Authors' Addresses

Questions about this memo can be directed to:

Clovis Schneider
Epitech 3rd yr
54000 Nancy
France
EMail: clovis.schneider@epitech.eu

Damien Demontis
Epitech 3rd yr
54000 Nancy
France
EMail: damien.demontis@epitech.eu

Chloé Lere
Epitech 3rd yr
54000 Nancy
France
EMail: chloé.lere@epitech.eu

Mylo Jeandat
Epitech 3rd yr
54000 Nancy
France
EMail: mylo.jeandat@epitech.eu

Vivant Garrigues
Epitech 3rd yr
54000 Nancy
France
EMail: vivant.garrigues@epitech.eu

D. Full Copyright Statement