Networked Game Class Project

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79. This document may not be modified, and derivative works of it may not be created, except to publish it as an RFC and to translate it into languages other than English.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

This Internet-Draft will expire on Fail 1, 0000.

Copyright Notice

Abstract

This memo describes the communication protocol for a client/server based multiplayer maze racing game for the Internetworking Protocols class at Portland State University.

Table of Contents

1. Introduction

This specification describes a simple multiplayer protocol to support the Maze Racer 2D game online. The protocol is called Maze Racer Multiplayer Protocol (MRMP), and it is a protocol by which clients can participate in a game session with 1 other player at a time and race to see who can reach the exit of a randomly generated maze first.

2. Conventions used in this document

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 [RFC2119].

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying significance described in RFC 2119.

In this document, the characters ">>" preceding an indented line(s) indicates a statement using the key words listed above. This convention aids reviewers in quickly identifying or finding the portions of this RFC covered by these keywords.

In this document, the abbreviation “MRMP” will stand for “Maze Racer Multiplayer Protocol”. The words “client” and “player” will be used interchangeably.

The term “session” used in this document will refer to a separate thread of execution created on the server process, that tracks 2 different client connections, a generated maze in order to conduct a race between them, and the client positions within the maze, as well as the winning position that either client must move onto first in order to win.

3. Basic information

All communication described in this protocol takes place over TCP/IP, with the server listening for connections on port 9898 by default. Clients connect to this port and maintain this persistent connection to the server. The client can send messages and requests to the server over this open channel, and the server can reply via the same. This messaging protocol is synchronous in the setup phase, where there is an expected order of message between client and server (hello, hello acknowledgement, join, join response, ready, start),and then asynchronous during the game session phase of the connection - the client is free to send messages (move, leave) to the server at any time, and the server may asynchronously send messages back to the client (bad move, opponent move, error, timeout).

Both the server and client MAY terminate the connection at any time for any reason. They MAY choose to send an error message to the other party informing them of the reason for connection termination.

The server MAY choose to allow only a finite number of users and sessions, depending on the implementation and resources of the host system. Users MUST be referenced through their connected TCP sockets at all times. Error codes are available to notify connecting clients that there is currently a high volume of users or groups accessing the server.

4. Message infrastructure

4.1. Generic message format

```
struct mrmp_pkt_header {
     uint8_t opcode;
     uint32_t length;
};

struct mrmp_pkt_generic {
     struct mrmp_pkt_header header;
     uint8_t payload[header.length]
};
```

4.1.1. Field definitions

- opcode - the code corresponding to the type of message the client wants to send to the server.
- length - the length of the body/payload of the packet in bytes.
- payload[header.length] - the actual data that will be sent as the packets payload, may not be used by some messages.

4.1.2. Operation codes (opcodes)

MRMP_OPCODE_ERROR	0b00000001
MRMP_OPCODE_HELLO	0b00000010
MRMP_OPCODE_JOIN	0b00000011
MRMP_OPCODE_LEAVE	0b00000100
MRMP_OPCODE_MOVE	0b00000101
MRMP_OPCODE_BAD_MOVE	0b00000110
MRMP_OPCODE_RESULT	0b00000111
MRMP_OPCODE_JOIN_RESP	0b00001000
MRMP_OPCODE_START	0b00001001
MRMP_OPCODE_READY	0b00001010
MRMP_OPCODE_TIMEOUT	0b00001011
MRMP_OPCODE_OPPONENT_MOVE	0b00001100
MRMP_OPCODE_HELLO_ACK	0b00001101

4.2. Error messages

```
struct mrmp_pkt_error {
     struct mrmp_pkt_header header =
```

```
            {.opcode = MRMP_OPCODE_ERROR, .length = 1};
        uint8_t error_code;
    };
```

4.2.1. Usage

MAY be sent by either the client or the server prior to closing the TCP connection to inform the other party why the connection is being closed. If either client or server receives this message, that entity SHOULD consider the connection as terminated.

4.2.2. Field definitions

- error_code - specifies the type of error that occurred.

4.2.3. Error codes

```
MRMP_ERR_UNKNOWN              = 0b00000000;
MRMP_ERR_ILLEGAL_OPCODE       = 0b00000001;
MRMP_ERR_VERSION_MISMATCH     = 0b00000010;
MRMP_ERR_FULL_QUEUE           = 0b00000011;
```

5. Client messages

5.1. First message sent to the server

```
struct mrmp_pkt_hello {
    struct mrmp_pkt_header header =
        {.opcode = MRMP_OPCODE_HELLO, .length = 1};
    uint8_t version = 0;
};
```

5.1.1. Usage

Before subsequent messages can be sent, a connecting client MUST identify which version of the protocol they are using, in this case 0b00000000. This message SHOULD be sent only once; if the server receives the message more than once, the server MAY either ignore the message or terminate the client's connection.

5.1.2. Field definitions

- version - is used to determine if the client is using the same version of the protocol. If the server receives any

other value than the expected value 0b00000000, the server MAY alert the client using the error code MRMP_ERR_VERSION_MISMATCH, and MUST terminate the connection.

5.2. Joining sessions

```
struct mrmp_packet_join {
     struct mrmp_pkt_header header =
          {.opcode = MRMP_OPCODE_JOIN, .length = 0};
};
```

5.2.1. Usage

Sent by a connected client in order to let the server know that the connection associated with the client should be placed in a queue with other client connections in order to be eventually put into a session with one other client connection to proceed with the game.

5.2.2. Response

Server MUST return an mrmp_pkt_join_resp as described in [6.1].

5.3. Leaving sessions

```
struct mrmp_pkt_leave {
     struct mrmp_pkt_header header =
          {.opcode = MRMP_OPCODE_LEAVE, .length = 0};
};
```

5.3.1. Usage

MAY be sent by a connected client at any stage of the protocol when closing the connection to the server to allow the server to report the stale connection as intentional, and additionally notify the other client in a session if the connected client was in a session to begin with.

5.4. Making moves

```
struct mrmp_pkt_move {
     struct mrmp_pkt_header header =
          {.opcode = MRMP_OPCODE_MOVE, .length = 2};
```

```
        uint8_t row;
        uint8_t column;
    };
```

5.4.1. Usage

Sent by a connected client to the server if they want to change their position in the maze during a session with another connected client.

5.4.2. Field definitions

- row - represents the new row in the 2D maze the client wishes to move to.
- column - represents the new column in the 2D maze the client wishes to move to.

5.4.3. Response

The server MUST respond with an mrmp_pkt_bad_move as defined in [6.3] if a bad move is made. The client is responsible for moving themselves locally for rendering purposes and sending the move to the server for validation. On a bad move, the client must rewind back to the last validated move they made, which will be included in the [6.3] bad move packet by the server. No response is expected if the move is valid, the server will silently accept and update its own record of where the client is in this case.

5.5. Declaring readiness

```
struct mrmp_pkt_ready {
    struct mrmp_pkt_header header =
        {.opcode = MRMP_OPCODE_READY, .length = 0};
};
```

5.5.1. Usage

Sent by a connected client after receiving the mrmp_pkt_join_resp from the server, to signify they are ready to begin the session (the race).

5.5.2. Response

The server MUST respond with a mrmp_pkt_start as defined in [6.6], or MAY respond with a mrmp_pkt_timeout if the client does not signify they are ready within a specific amount of time that is up to the server to decide. In the case of the timeout, the connection should be closed whether the client was notified or not.

6. Server messages

6.1. Hello acknowledgement

```
struct mrmp_pkt_hello_ack {
     struct mrmp_pkt_header header =
          {.opcode = MRMP_OPCODE_HELLO_ACK, .length = 1};
};
```

6.1.1. Usage

The server MUST send this packet in response to a received hello packet with the correct version number .

6.2. Client join response

```
struct mrmp_pkt_join_resp {
     struct mrmp_pkt_header header =
          {
               .opcode = MRMP_OPCODE_JOIN_RESP,
               .length = 2 + rows * columns
          };
     uint8_t rows;
     uint8_t columns;
     uint8_t cells[rows * columns];
};
```

6.2.1. Usage

A representation of the 2D maze sent by the server to the client for client-side rendering purposes. The server is responsible for generating the maze using whichever algorithm it sees fit as long as it complies with [8]. Both clients in a session MUST receive the same maze, and the server MUST keep the maze in memory for the duration of the session in order to validate client movements.

6.2.2. Field definitions

- rows - the number of rows in the 2D maze.
- columns - the number of columns in the 2D maze.
- cells - the actual cells making up the maze.

6.3. Move messages

```
struct mrmp_pkt_bad_move {
     struct mrmp_pkt_header header =
          {.opcode = MRMP_OPCODE_BAD_MOVE, .length = 2};
     uint8_t last_row;
     uint8_t last_column;
};



struct mrmp_pkt_opponent_move {
     struct mrmp_pkt_header header =
          {.opcode = MRMP_OPCODE_OPPONENT_MOVE, .length = 2};
     uint8_t row;
     uint8_t column;
};
```

6.3.1. Usage

Mrmp_pkt_bad_move MUST be sent to provide a notification to the client that a requested move they made using mrmp_pkt_move was invalid. An invalid move is made when a client requests to move either to a cell that is not vertically or horizontally adjacent to their current position, a cell that is further than 1 cell away from their current position, and or a cell that is beyond a wall of their current cell in the direction of the requested cell.

Mrmp_pkt_opponent_move is meant to serve as a way to replicate game state. It MUST be sent to provide a notification to a client that their opponent has made a valid move within the maze, with the intention that the client will be able to render their opponent in the correct position.

6.3.2. Field definitions

- last_row - the last valid row in the maze that the client was at, according to the server.
- last_column - same as last row, but for columns.
- row - the new row that the opponent has successfully moved to.
- column - same as row but for columns.

6.4. Win/Lose response

```
struct mrmp_pkt_result {
     struct mrmp_pkt_header header =
          {.opcode = MRMP_OPCODE_RESULT, .length = 1};
     uint8_t winner;
};
```

6.4.1. Usage

MUST be sent by the server to both clients when the server decides who won and who lost the race in any given session. The server MUST stop the thread executing the session upon a result being decided, but MAY either close both client connections, or keep them open in a kind of limbo to await another mrmp_pkt_join from either client.

6.4.2. Field definitions

winner - represents a boolean value denoting who is the winner and who is the loser. 0 denotes a loser while a non zero value represents a winner.

6.5. Client timeout

```
struct mrmp_pkt_timeout {
     struct mrmp_pkt_header header =
          {.opcode = MRMP_OPCODE_TIMEOUT, .length = 0};
};
```

6.5.1. Usage

MAY be sent by the server to a connected client in the event a client takes too long to send a mrmp_pkt_join or a mrmp_pkt_ready, where “takes too long” refers to an amount of

time that is up to the server to decide. Once that time is up, the connection MUST be terminated. A mrmp_pkt_timeout may also be sent at the start of a session, if either of the clients do not indicate their readiness within a set amount of time. The same applies to if in a session, neither client sends a message for a set amount of time. In that case, the mrmp_pkt_timeout MAY be sent, and the session MUST be terminated.

6.6. Game start

```
struct mrmp_pkt_start {
     struct mrmp_pkt_header header =
          {.opcode = MRMP_OPCODE_START, .length = 0};
};
```

6.6.1. Usage

Sent by the server to let both clients in a session know that they can start sending move commands to the server.

6.6.2. Response

Once this start packet is sent, the game session is considered started on the server's end. The only messages a client can send back that will have any effect on the server state of the session will be the mrmp_pkt_move and mrmp_pkt_leave packets. Any other message type can either be responded to with a mrmp_pkt_error with an error code of MRMP_ERR_ILLEGAL_OPCODE and nothing happens or no response at all and nothing happens.

7. Error handling

Both server and client MUST detect when the socket connection linking them is terminated. If the server detects that the client connection has been lost during a session, the server MUST disconnect the other client if they are still there and close the thread of execution running said session. If the client detects that the connection to the server has been lost, it MUST consider itself disconnected and MAY choose to reconnect. As stated previously, it is OPTIONAL for one party to notify the other in the event of an error. If a client receives an mrmp_pkt_error, they MAY choose to send an mrmp_leave_pkt and MAY choose to disconnect from the server.

8. Maze encoding/allowed interpretations.

8.1. Maze encoding

A maze MUST represent a 2D array of cells of dimensions rows * columns, indexed from (0, 0) to (rows - 1, columns - 1). Each cell

MUST consist of at minimum, an 8 bit unsigned integer where the 4 least significant bits denote whether or not a wall exists in a specific direction. The 4 least significant bits denote walls in these directions from the least to the most significant of the 4: NORTH, SOUTH, EAST, WEST. A 0 bit represents a wall existing in a given direction, while a 1 bit specifies no wall, aka free passage. An example cell: 0b00001011 says that there are no walls in the NORTH, SOUTH, and WEST directions of the cell, but there is a wall in the EAST direction.

8.2. Maze generation

The 2D maze SHALL be generated using any algorithm seen fit by the server, as long as it generates a representation described by [8.1]. It is REQUIRED that every cell is reachable starting from any other

cell, and that walls always exist in cells along the border of the maze that enclose the entire maze. Generated mazes must be greater than 5 rows and 5 columns.

8.3. Win condition

The server MUST decide the winning condition by either hard-coding a cell coordinate as the “exit” or “solution”, or by randomly selecting a cell coordinate, or by algorithmically determining the cell coordinate via whichever maze generation algorithm is used. Whichever client sends a move command for the solution cell first, will win, and the other will lose.

9. Conclusion

This specification provides a framework for reliable client game state replication and validation through a central server over the internet for the Maze Racer 2D game.

10. Security considerations

The message system defined by the Maze Racer Multiplayer Protocol in this document (this document presents version 0 of the protocol) does not concern itself with matters of security. Servers and clients implementing agreed upon security measures is OPTIONAL.

11. IANA considerations

None

12. Acknowledgements

This document was prepared by Abdurrahman Alyajouri and Ariella Marchuk.