Design of a multiplayer game

Suvi Sipilä
University of Helsinki
suvi.sipila@helsinki.fi

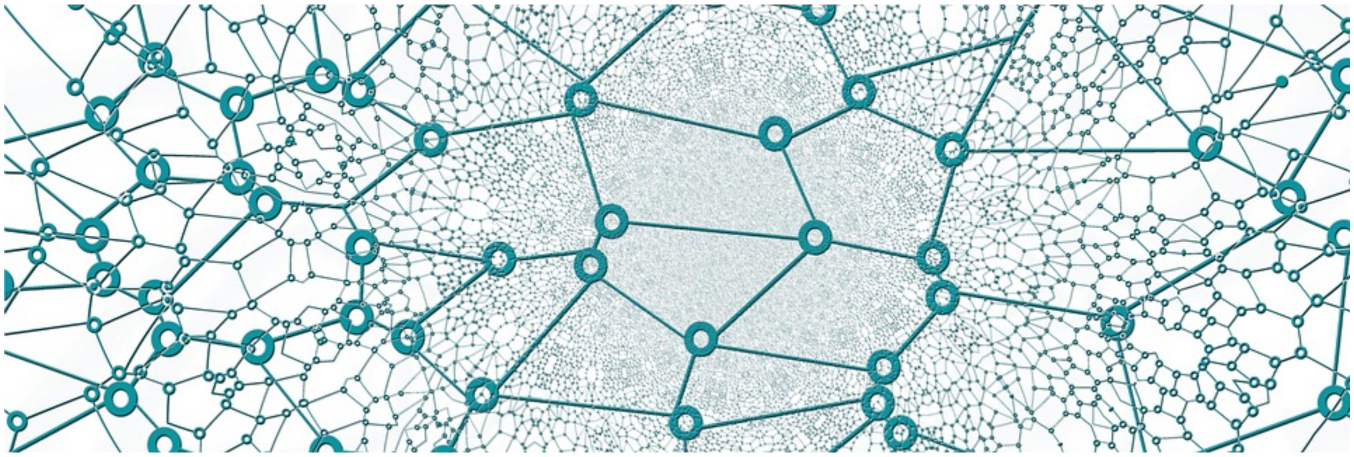

Figure 1: Network from https://pixabay.com/fi/illustrations/verkko-sosiaalisen-abstrakti-3139214/

1 INTRODUCTION

This document covers design of a multiplayer game that focuses on communication between players. The design uses a client-server architecture where all players talk via the server.

When a player wants to join the game, the player informs the server of its wish to join the game by sending a simple Hello message. When the server receives this message, it adds the player to the player array list and sends the player a response message with the game settings information. When the player receives this message, it can download the settings and start listening to the messages from the server.

When the player makes a move, it sends a move message to the server. The server receives the message and forwards it to the other players. When the game is over, the server sends an END-message to all players and the players can then close the connection to the server.

2 DESIGN

Multiplayer game is designed using a client – server architecture. The server is the host that maintains the game information, settings and is responsible for sending messages to all clients. The client contacts the server and only communicates with the server. The server is talking to N number of clients. Figure 2 shows the conversation between clients and server.

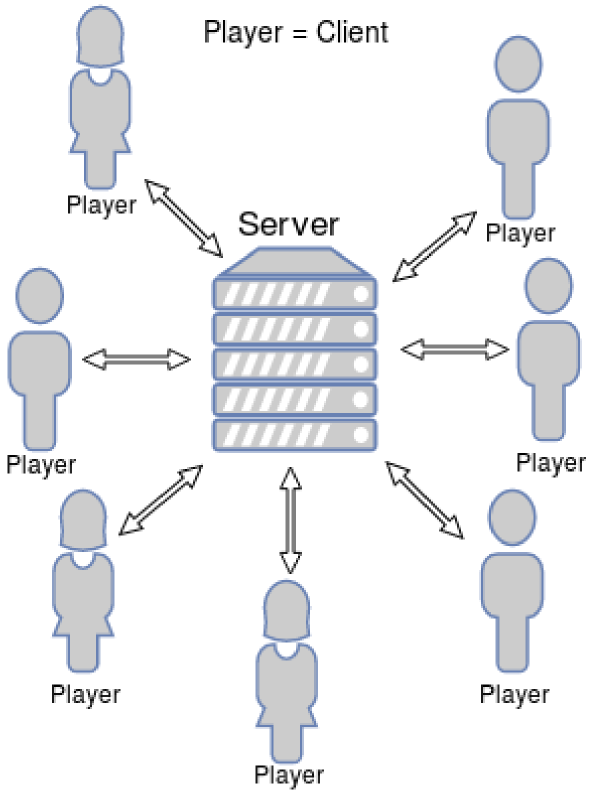

Figure 2: Client - Server conversation

2.1 Transport protocol

The client and the server transport layer is created using UDP protocol which reduces the latency of the game because the UDP does not require data packets to be received. This allows the quick response times because the server can send the packets to all players without waiting for a response from each player that the packet has been received. If we were using TCP, it would be more reliable that all the packets get through in the right order but in real-time gaming it can increase the delay. As the number of players increase, UDP does not slow down performance as much as using TCP. TCP requires an ACK message for the sent packet and will not send the next packet until it has received the ACK message from the previous packet. These ACK messages generate a lot of traffic and slow down performance.

2.2 Forward Error Correction

The loss detection is done by using Forward Error Correction. The game message sent by the player is chopped and packetized into k packet of x bytes. Then a block of k data packets generates and creates the additional redundant n packets so that there is one redundant packet for two k packets, see figure 3. The redundant packet is done and decoded by using XOR.

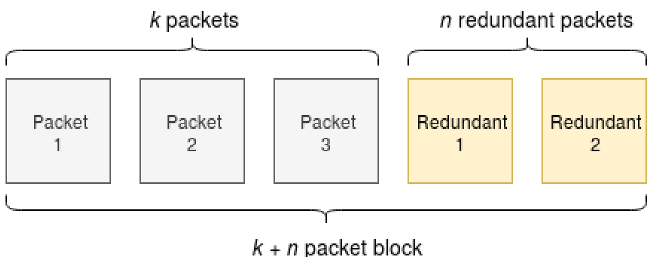

Figure 3: Packet block

The loss rate can be calculated by using formula

$$Rate = \frac{k}{k+n}$$

Thus, the loss rate for the FEC is

- Rate for even number of packets

$$= \frac{2}{3}$$

- Rate for odd number of packets

$$= \frac{3}{5}$$

This $k + n$ packet block is then sent each packet at a time to the server. The server determines whether some packet is lost and, if necessary, reconstruct the lost packet with the redundant packet (figure 4). So there is no need to retransmission the lost packet, which would cause huge delays for the game. Once the server has received all packets, it forwards them to the other players with the same way using FEC.

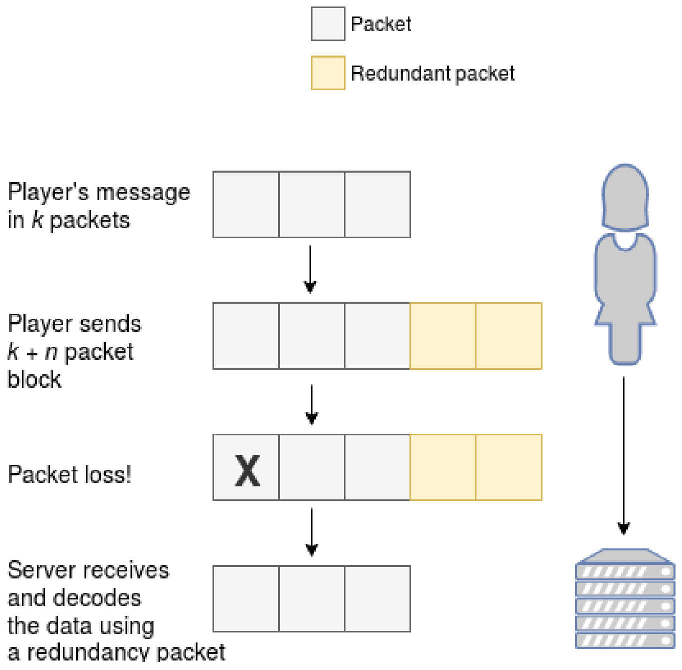
Packet
Redundant packet
Player's message in k packets
Player sends k + n packet block
Packet loss!
X
Server receives and decodes the data using a redundancy packet

Figure 4: Sending a group of packet

3 IMPLEMENTATION

The implementation is done so that all the sent packets contain a variant, that tells the receiver what the packet is and what action must be taken to decode the packet.

3.1 Client

The client makes a connection to the server by sending a simple Hello-message. The client's message includes a variant that tells the server that this is the first message and request to join the game. The variant can be, for example, an integer number. The client creates a packet with the Hello-message and variant and sends it to the server. The client then listens to the response from the server. The first message from the server is the game settings file that the client must decode with using XOR.

When the client sends the game move message to the server, it need first create a UDP packet for this move. It creates a k packet of x bytes and then sends them with using FEC. The client adds a variant number to the UDP packet which includes the information of which packet it is and helps the server to determine which packet it has received. Once the client has sent the group of packets A, B and redundant packet C, it sends the Sent-message with the variant to the server and the server then knows that it has received all packets of the group and can start decoding them.

The implementation can also implement a data loss error message that can be displayed in the client's terminal window. For example, we can add a key to all packets of the group A, B and C and if some of the packets is lost, it can be displayed in the terminal window. This key will help if we ever want to upgrade this implementation to include retransmission to the lost packets.

The client listens to all messages from the server and decodes them with using XOR. When the client receives a game over END-message, the client can then close the connection to the server.

3.2 Server

The server maintains a array list of all connected clients. When the server receives the first message from a new client, it adds the client's address (IP and port) to the list. The server then sends the game settings message to the client and this message informs the client that it is in the game.

The server listens all its clients and waits for messages to arrive on the server. When the server receives messages, it waits for a Sent variant message that tells the server that it has received all packets from this one group. The server then decodes this packet group and saves the movement of the game on the game board. If some packet is lost, it uses the redundant packet to reconstruct the message. The server then encodes the message with using XOR and sends the packets to all other clients.

While sending messages to all its clients, the server can receive an error message stating that one of its client's connection is closed. If this happens, the server removes this client from its client list. This error listening helps to prevent the server's list from being corrupted and that the list does not contain incorrect information about active players.

The server sends an END-message with a variant to all its clients when the game has ended. The server then clears the client list and is ready to start a new game.

4 TRADE-OFFS

The trade-off for using UDP is that if the packets are lost so that the FEC cannot reconstruct them, the packet then is lost and not retransmitted. If this happens when a player sends a game move message to the server, the player cannot make the move it want and someone else can do it. In the worst case scenario, the movement will result in a win for the other player.

Another case is that if one of the players does not receive the latest game move message, the player's game board is outdated compared to the others and can lead to the wrong game move. But if this player does not make a move right away, it can receive a new game move message that includes the lost move, so losing the packet was not that harmful for the game.

If the first message on the server is lost what contains the game settings, the player then will not be able to participate in the game. However, this can be corrected by setting a timestamp on the first Hello-message, allowing the player to conclude that the server either has not received the message or the player has not received the message sent by the server, and thus the player can send a new Hello-message and wait again for the game setting message.

5 IMPLEMENTED GAME

The client and the server are made in the same way as described in the section 3 Implementation. The implementation relies on variants that tell the client and the server what packet has arrived and when all the packets have arrived.

The variants are:

- 0 = Game over message
- 1 = First Hello-message (client sends only)
- 2 = Client deliberately closes the connection (client sends only)
- 4 = Sender sends XOR packet A
- 5 = Sender sends XOR packet B
- 6 = Sender sends XOR packet C
- 7 = One packet group A, B, C has been sent
- 8 = Entire message has been sent
- 9 = Entire game instructions has been sent (server sends only)

The client has the boolean parameter *received_instruction* which is true when the client has received the instruction from the server about the game. The client cannot make a move before this parameter is true. The instruction of the game is now “*Lorem ipsum*” in a *Game_Instructions.txt*, which the server reads from the file and sends it to clients in XOR packets. The server creates 100 bytes UDP packets while the client creates 2 bytes UDP packets because the clients' messages is shorter than the server's game instruction message. In the implementation, the server does not save the progress of the game anywhere.

There is one problem with the client side: it cannot listen to the server and the command line at the same time. This has been resolved with a 10 second wait. When the game asks the player "*Press x to send move*", it waits 10 seconds for a response and if no response is received, the client begins to listen if the server is sending packets. The client must receive at least one message from the server so that it can invoke the command line input again. So if every client listens to the server, the game can get stuck because no one is sends messages. I think it should be done by invoking own threads for listening the command line and for listening the incoming messages from the server. I did not know how to implement this in my code.

The client listens to the command line, and when the player wants to make a move, it have to press *x* and type what movement it wants to make. Because the implementation focuses on communication between clients and server, there is no real game involved. So any of the clients can write "*win*" instead of "*x*" and then this client is the winner of the game. The client can also write "*out*" instead of "*x*" and the connection between the client and the server is closed. This "*out*" input is mainly for testing purposes only.