**Network Game Development**

Report

**Architecture**

The application is a multiplayer 2D fight game developed using both client-server and peer-to-peer network architectures.

The first architecture has been implemented for the matchmaking system, which allows the players to connect to a known server lobby to wait for second opponent.  
When two players are connected to the lobby, these establish a peer-to-peer connection between them.

The matchmaking client-server network structure has been chosen taking into account the number of clients expected to connect simultaneously: when two player connect to the lobby, the server sends them their reciprocal information to establish their own peer-to-peer connection and, after this has been successfully done, the players automatically disconnect from the lobby. This way it’s unlikely to have more than two players in the lobby. Moreover, since the players have to establish a peer-to-peer connection, they both need to know their opponent information, such as IP and port to connect to. The lobby is therefore used to send relevant information to establish the direct connection between players.

The peer-to-peer networking structure has been chosen for the number of players involved: as there are only two players in the game, a peer-to-peer architecture is ideal as players can directly communicate their information without having to pass through a server, which would increase the communication latency. For this reason, even considering peer-to-peer’s poor scalability in terms of number of players involved, this type of architecture is ideal for the type of application developed.

**Protocols**

The application-layer has been split in two parts, which are the Server-lobby application and the peer-to-peer player application.

The Lobby has been designed to automatically detect new players connected. The player and the lobby exchange packets every frame, making sure they are both online. Few information are sent in these packets, such as player IP and port, player ID, character sprite chosen and a string used as match trigger. The string value is changed when there are two plyers in the lobby: when both players are ready to fight, the lobby sends them both a “clientready” string, which makes the game start.

When the game starts, the two players both sends their own data to each other every frame to detect disconnected players. Packets are structured as following:

|  |
| --- |
|  |

struct PlayerInfo

{

float timestamp;

int health;

sf::Vector2f position;

bool isAttacking;

bool isMovingLeft;

bool isMovingRight;

bool jumping;

bool isDamaging;

};

These packets are used to send the current state of player to opponent, for it to update their copy according to data received. States will be “converted” to simulated corresponding input, to perform a movement, attack or jump. States have been preferred over other type of data as these allow to avoid code repetition.

The protocol chosen for Transport-layer is TCP/IP.   
TCP has been preferred over UDP for few reasons:

1. Packet order: as the type of game relies on the order in which the actions are performed, TCP has been preferred as it guarantees correct order of packet delivery
2. Performances: even though UDP has lower overhead, when there is no packet loss they both perform well [1]
3. Implementation simplicity: TCP is simpler to implement than UDP, as it doesn’t require custom handling of packet loss recovery.

**API**

SFML 2.5 has been used to develop the game. This API has been chosen for its simplicity of use, detailed documentation and flexible capabilities.

SFML has a built-in Network module, which is user-friendly and guarantees flexible implementation of protocols. Moreover, it is ideal for 2D games as the Sprite and Input management are straightforward and well documented.

**Integration**

As previously discussed, the application is split in two networking architecture, each dedicated to a specific feature.

Matchmaking Server Integration

The server-lobby is a simple application that detects new incoming connections and checks how many clients are currently connected.  
This is done using SFML *Selector* to accept incoming connection and switch between clients to send and receive packets from them and check these are still connected to lobby.

|  |
| --- |
| if (selector.wait())  {  if (selector.isReady(listener))  {  clients.emplace\_back();  if (listener.accept(clients.back().socket) != sf::Socket::Done)  {  clients.pop\_back();  }  else  {  clientCounterID++;  clients.back().clientID = clientCounterID;  selector.add(clients.back().socket);  }  }  } |

The player has to manually connect to the lobby by inserting the IP and port of the dedicated server that needs to be open. If the player can’t connect to the server, a “Server not found” message will be shown and the player can type again the information needed to establish the connection.   
When the player successfully connects to lobby, they exchange packet with information such as player ID, sprite ID, IP and port

|  |
| --- |
| packet >> clientID >> s >> opponentIp >> opponentPort >> myPort >> opponentSpriteID; |

The value “s” sent in the packet is a state string. This value is changed to “clientready” when two clients are both connected to the lobby.   
When this happens, the lobby also sends opponent’s information to both players, to make them possible to establish a peer-to-peer connection.

Now the players disconnect from lobby and the game starts.

Game Peer-to-Peer Integration

When players initialise their respective game instances, Player 1 listens first while Player 2 tries to connect. When this first connection has been established, Player 1 will then try to connect to Player 2, which is now listening to its port.

After the connection has been successfully established, the two players both update their own world copies based on the data they exchange. This operation is done in the main Update game loop method, where the animation and movement of the owning player are updated along with opponent’s information.

Similarly to connection establishment step, players send and receive packets based on their IDs: Player 1 sends the its packet first and Player 2 receives it, then Player 2 sends its packet and Player 2 receives it.

**Player 1 Player 2**

**Player 1 Player 2**

After packets have been received, their timestamps are compared, with a 100ms accuracy, to determine whether players are desynchronised. If this happens, the *Prediction* algorithm will be called and sockets will be set in *non-blocking* mode to avoid the application to completely block while waiting for incoming delayed packets. Players then update their copy of the opponent player based on data received.

Packets are also used to detect if opponent disconnected at any point in the game. If this occurs, the player will go back to lobby screen to connect to server lobby again.

**Prediction**

A Linear Model prediction has been implemented, taking into account the last two position (x and y) and timestamp messages received and stored in appropriate *std::vector*.

The formula used is the following:

This model has been chosen as the most appropriate in this context as it provides a fairly accurate prediction with low computational cost. The type of movement of the players themselves are linear most of the time, combined with jumps when required, and the linear prediction keeps the player movement plausible.

No interpolation technique has been implemented.

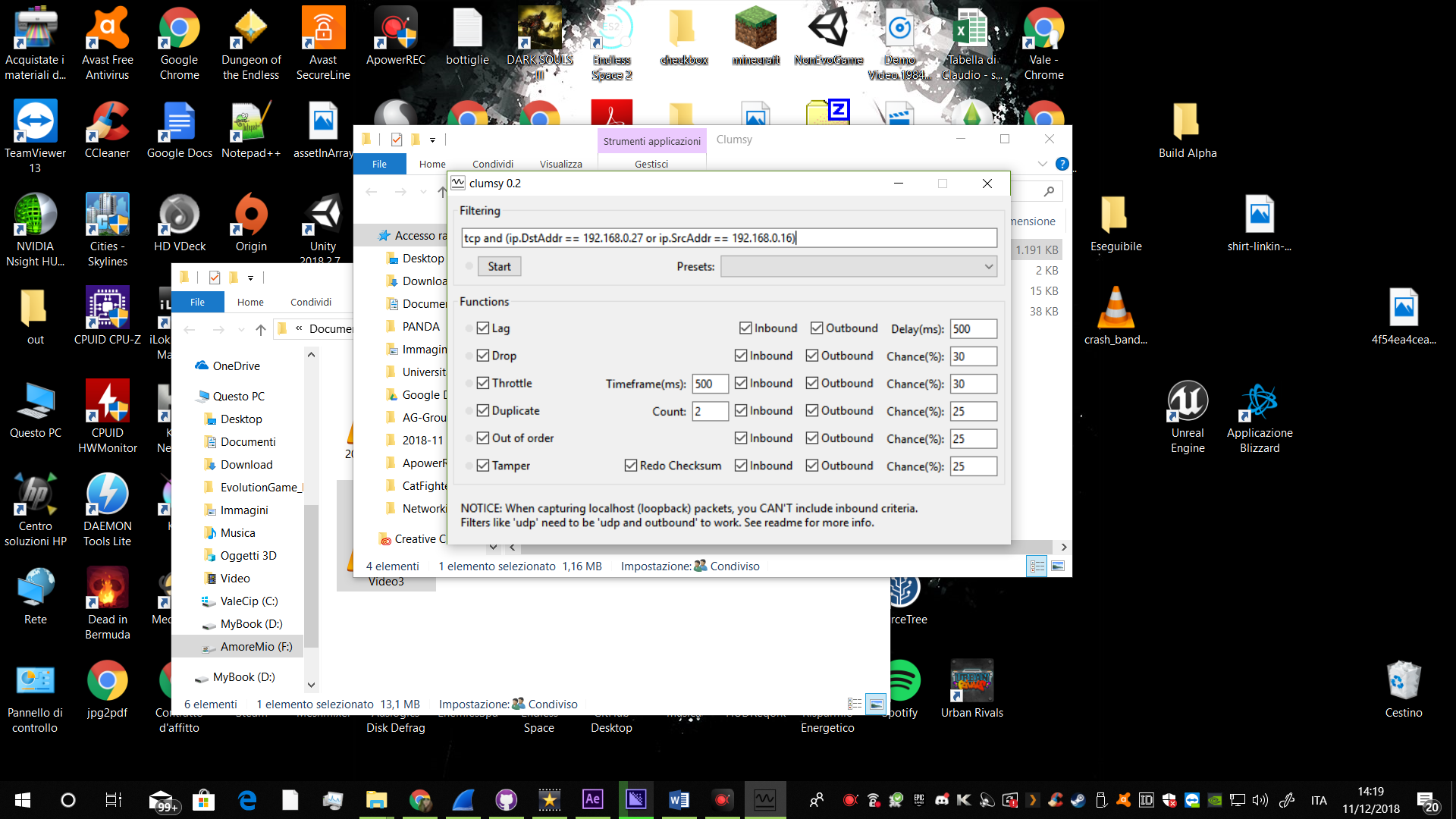
**Testing**

The game has been tested simulating networking common problems using Clumsy.

Different settings have been tested, both on peer-to-peer players connected in LAN on different computers, and on the client-server lobby system, tested locally on the same PC.

Peer-to-Peer

The following setups have been tested, one by one, on the peer-to-peer connection over LAN:



The results of these testing have shown that the application seems to perform fairly smoothly when *Lag* is on and set at 500ms. However, players will start to desynchronise and the latency is visible.

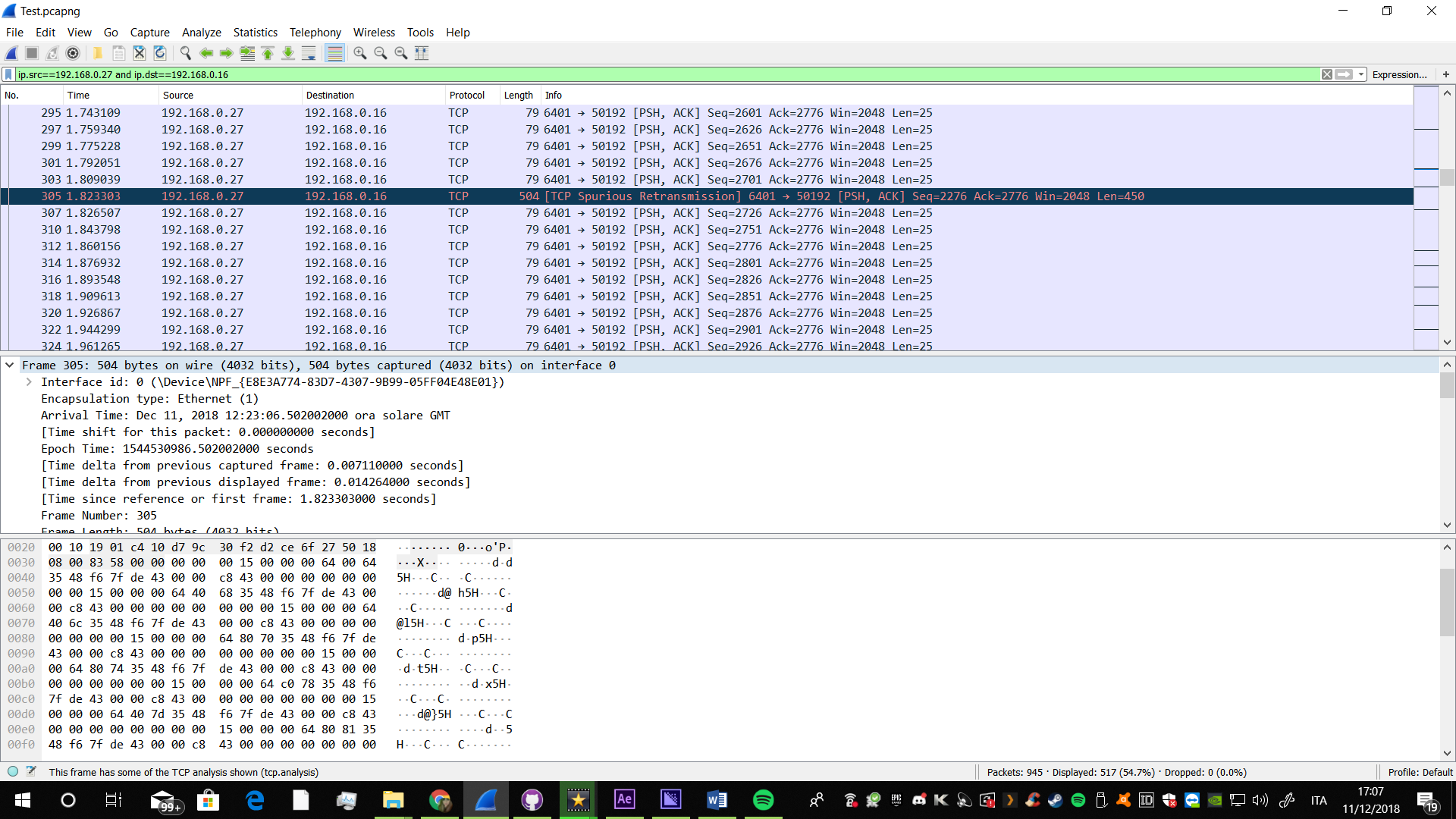
When the *Drop* is active at 30% of chance it happened that the opponent player disappeared from the screen for a couple of seconds, reappearing where it was before.

On *Throttle*, *Duplicate* and *Out of order* settings the game still looked smooth as before, with only some occasional “blinking” of the opponent player. But since the desynchronization becomes too big, even if the game is still playable, when a player tries to attack the opponent it looks like the player hit it, but its life doesn’t go down. This is caused by the high latency between the players, as player 1 will see itself hitting player 2, but player 2 has actually moved away.

However, the game seems to manage the paradox fairly well, as the real position of each player is the one that the corresponding user sees on its own screen and not the one seen by their opponent.

When *Tamper* is activated, the games looks no longer playable, as the opponent player starts to randomly move across the game area because of corrupted packets.

During some of these tests, Wireshark has been used to capture packets transmission between the two machines.



As it is noticeable, in a small window of time (5 seconds of Wireshark testing), many “TCP Retransmission” errors occurred. This is because the ACK has not been received back, therefore the packet was actually lost.

Client-Server lobby

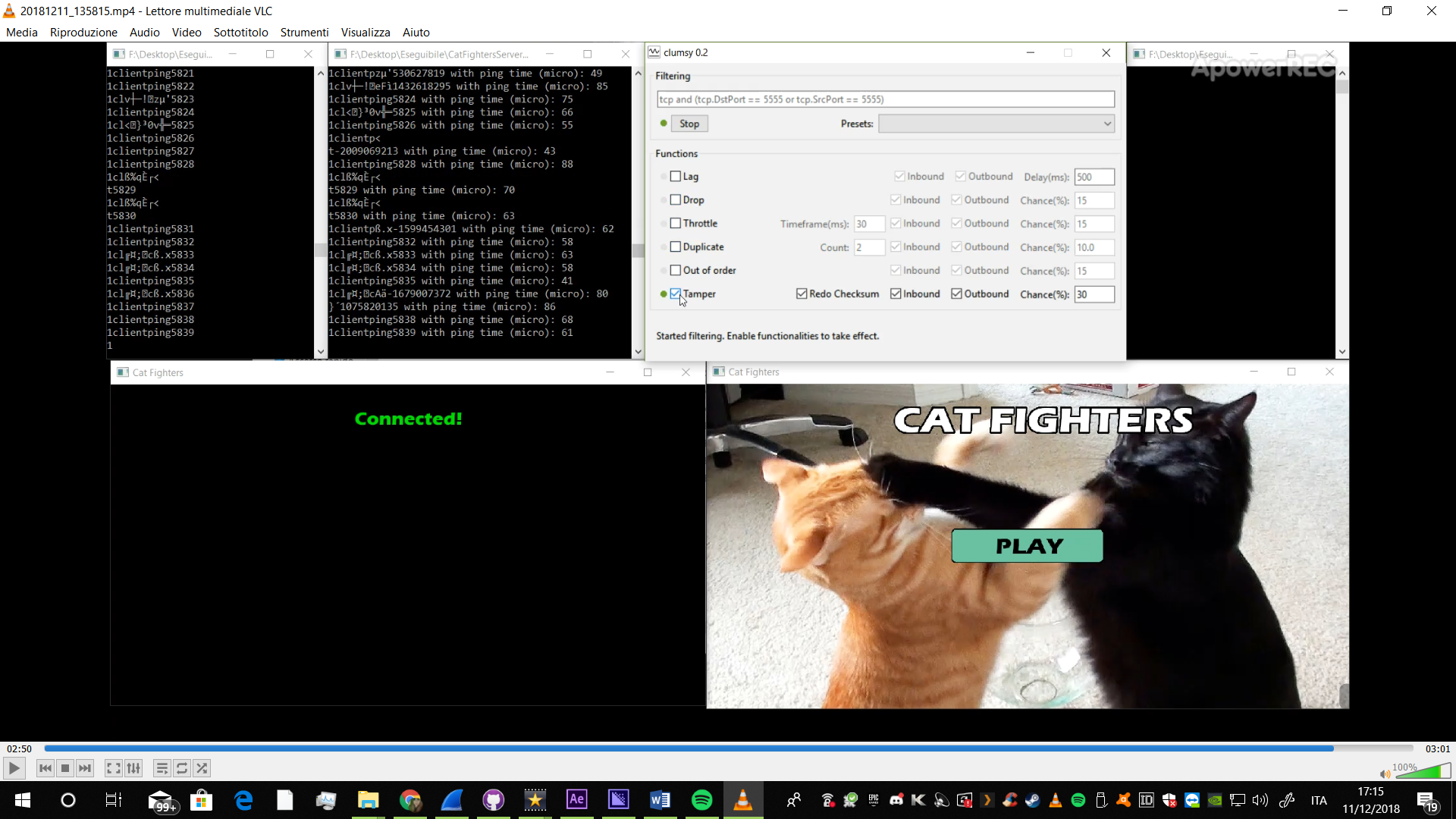
The second part of the performance testing includes the client-server connection. This test has been carried out on the same machine, therefore the values for Clumsy settings have been also divided by 2 because of the packets’ loopback on the local host [2].

The *Lag* option didn’t seem to badly affect the client-server communication: even though some lag was noticeable, packets have been delivered correctly with no issues.

*Duplicate*, *Throttle* and *Out of order* didn’t seem to have much of a bad influence on the communication, as just some occasional lag was caused.

On the other hand, *Duplicate* and *Tamper* caused some interesting outcomes.  
The first one seemed to have a great negative influence on the client-server communication between the connected player and the lobby, as packets were delivered with great delay causing a big latency that froze both server and client application for few seconds.

*Tamper* visibly corrupted packets information, as the output in both consoles were compromised and unreadable.



For this reason, since server sends its clients a trigger string “clientready” when two players have joined the lobby, it has been considered useful to test whether this message was going to be delivered correctly to both players, even considering the *Tamper*. As a result, the clients successfully received the message, even though a small delay was noticeable.

Critical Commentary

As it has been understood from these tests carried out in different networking conditions, the performances of the game can be considered playable in most of the occasions and settings tried. However, the game suffered from noticeable delays and packets corruption that it wasn’t able to recover from, even when the adverse networking conditions ended. In fact, after Clumsy is switched off, the players will stay desynchronised and they will never catch up again. This may affect the playability, as the players will see a not matching version of the game world in terms of time.

Few solutions have been tried to overcome this problem, including trying to stop sending current player packets until opponent is desynchronised and it is behind with packet updates. However, none of these implementations obtained a satisfactory resolution of the problem.

Moreover, more efficiency in terms of send and receive calls could have been implemented, such as using threads to manage sockets operation while the main thread could only have cared about the game mechanics and updates.

As an overall observation, the application performs very well in optimal networking conditions, fairly well and playable when occasional network issues occur, and almost not playable when bigger network errors start to affect the connection between players.

**References**

1. <https://thoughtstreams.io/glyph/your-game-doesnt-need-udp-yet/>
2. https://jagt.github.io/clumsy/manual.html