

Personal report

Networking game

# Petric Marcinkowski(s6073335) | COM2065 | 10/05/2018Message Structure

For this project, I decided to set up a Message base class and subclasses for each message that I want to receive from the server or send to it.

The Base class called “NetMessage” contains some virtual functions that each child need to implement, these functions are:

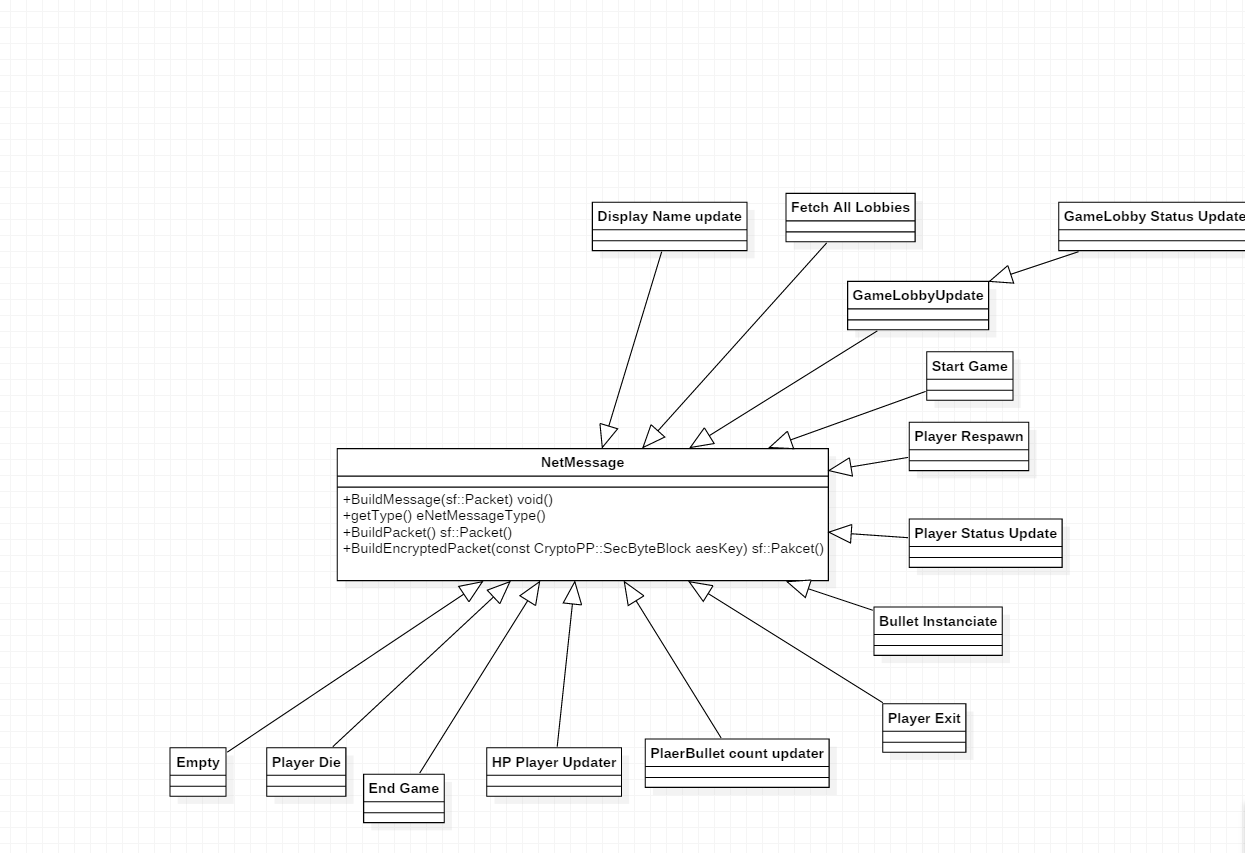
* BuildMessage: takes a SFML packet and converts it to the data structure required.
* getType: Returns the type of the current message (Useful to classify it later when we have base class pointers).
* BuildPacket: construct a SFML packet from the data inside the class
* BuildEncryptedPacket: this is not a virtual function and the base class will implement it so late one any pocket can get encrypted before been sanded off.

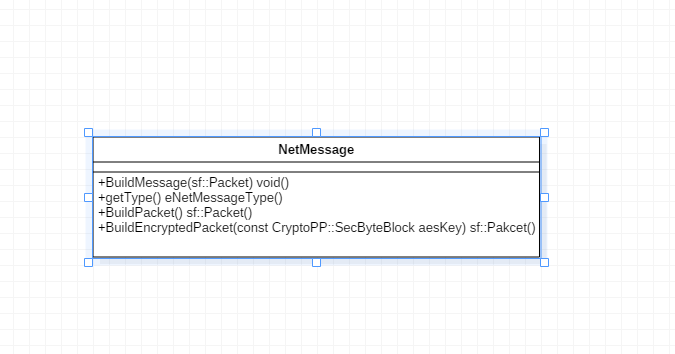
Each packet sends from and to the server have a specific format that it will always follow:

* An unsigned int of size 8: The type of the packet so we can classify it.
* The rest of the size is the data of the message. However, if the message is encrypted (Nearly all of them are), the classifier will execute a special routine over it. First, it will decode the message and treat the result as if it was a packet just received by sfml from the server and pass it again to the classifier to get the message out.

Once every packet arrives at the network thread it will be classified, and then added to a concurrent queue, and from there the currently active scene will fetch the message and handle it.

As the message structure was made using intermittence, to send them off to the server, I would not need to serialize at the point where I want to send it and potentially need to code the hole message multiple times, but instead I just create an instance of the message fill the data in (string, floats, units …) and then just pass the base pointer of the message to the function specifying if we want to encrypt it or not, and afterwards converting it in the correct format before sending it off. However, the way I implemented it is not perfect as some messages would contain the same data types but just with different names, but because of the names and nature of the packets it does not really make sense to make an HP update packet a subclass of an End Game packet.

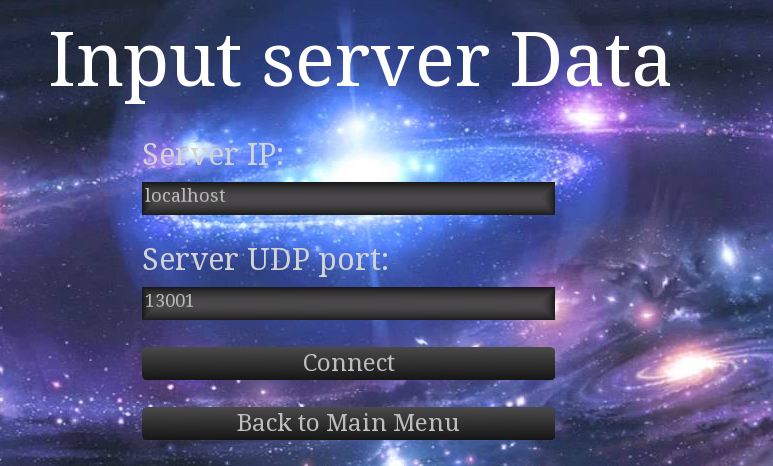




# Finding of the local server

When the user launches the application, they will have two different options to connect to the server. The most commonly used “Online Connection” where you will be required to input the server data, or LAN server.

The LAN Server will not prom pact the user to input any data but instead, it will try to find a server on the local network and start the handshake. The difference in how you connect the server does not really matter in the end as the same packages are sent over in the same order just the IP is changed.



# Initial connection and setup of encryption

Once the app starts a connection to the server, an SSH-Like handshake is performed (this is not trying to be the same way as I simplified it a lot and removed a lot of layers or verification for this project). During this “handshake”, will make the server to send over a per server lifetime public key (every time the server restarts a new public key is generated to invalidate the last one used) to the client.

After the client receiving this public key, it will generate a cryptographically secure AES-192 bits key and send it over to the server together with some other data such as IP, UDP port (which the client selects by binding to any port that is free), but this message is already encrypted with the RSA key, so that nobody can intercept this data. Or use the channel to send “fake packets” later one.

Once the server receives the key and the requested data it will register the user, generate a random identifier and username and return this information encrypted back to the user.

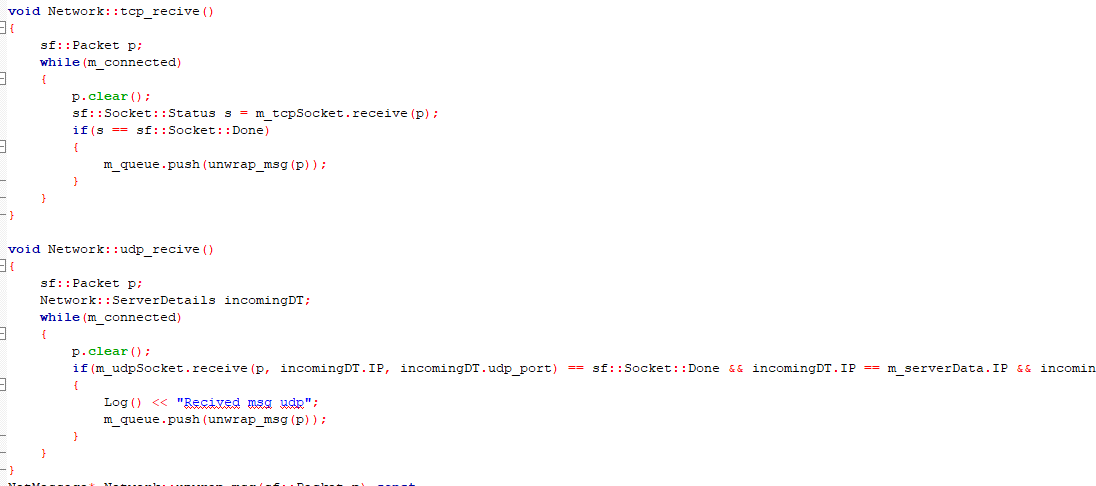
At this point, the server and the client are registered and know of the existence of each other, together with the necessary information to allow them to send all the data in an encrypted manner using the generated AES key.

# Networking server origin checking and Concurrency

As it is normal in networking, attacks could happen, to read data or send “fake data” to the server/client to get out the information they want or for other malicious, so to prevent this apart from the encryption which will make it difficult but not impossible to read the data and/or send fake data over, checks are made to make sure the information sent from the correct server/user and in case it’s no discard it. However, this is not perfect, as when finding the server locally, if another user opens a copy of the server there is a change that you connect to it, thinking it’s the genuine server.

In terms of concurrency, the network data needs to be constantly flowing between the client and the server to update the states, however, this interaction should not freeze the client or the server, as it would make the experience not smooth.

To accomplish this, I needed to create different threads on the server and client, to handle any kind of networking calls without freezing the main thread where the game is running, or the server updates its state. However, this cause issues, especially in C++ where synchronization, so to make sure the data could flow from the networking thread to the game thread, a special data type was created “Concurrent Queue”, in there the networking thread will push any data, if it is not being used by any other thread, meanwhile the game thread will try to pull data out of it if the networking code is not using it.





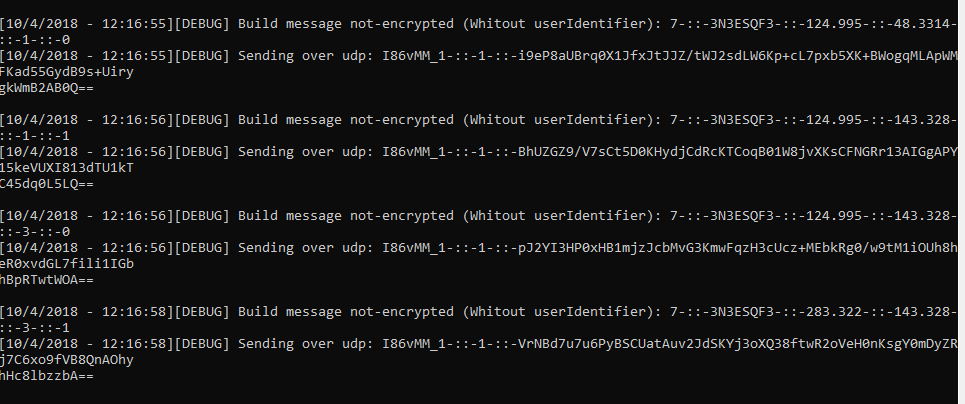
# Usage of the protocols

We were presented with 2 different protocols, UDP and TCP, each one has its drawbacks and its advantages over each one, however, I needed to make correct use of them to avoid loss of data, or possible “lag” in the game when updating the states. When thinking about which protocol to use, I followed some rules to help me decide:

* How often will it be sent?
* Does it matter if the other end does not receive the data a couple of times?
* Does the data need to arrive quick?

Depending on the answer to these questions I chose the different protocols for each packet. So, if a packet was sent multiple times in a second and with repeating data (so that the data does not rely on the old one) the protocol to use would be UDP, and if the data is important and it needs to get to the other end, the protocol would be TCP.

Using this method, I ended up using TCP for the connection setup and the entire lobby (we are not playing there so it really does not matter if it takes a bit longer to respond), and later one in the actual game, TCP was used for important messages such as character dies, character respawns, the game ends …. And UDP was left to use for ammunition count, damage done, position of the character and its status…





# Conclusion and Issues

During the development of this project I learned a lot, especially a lot of new parts of elixir that are quite powerful, also in the C++ part I learned a lot about thread and concurrency, as many things would not work the ”normal” way, such as the queue and normal std:: court, where I needed to develop specialized classed to fight against this issues (Concurrent queue, Concurrent logging system). However, during this project I did not manage my time correctly which ended up costing removing some of the features that I wanted to try implement, one of the main things was not allocating enough time to solve bugs, especially which encryption as it is a lot more difficult to debug, and it took a while to figure out what happens, one of the main problems with the encryption was to get the bytes sent by elixir out in c++, as I was using sf:: packets, if I extracted the data into a string there would be missing data, as the size of a 16 byte int was removed from it (the size for), this was mainly an implementation issue, to solve it, I needed create some helper functions that allowed me to extract the raw data from the packets, get the correct bits that I needed and then return them back to the packet with the proper headers, to do so I used a lot ArraySinks from cryptopp, as they take a pointer to byte data and extract the parts I want from it into any other container. But to get to this solution a lot of reimplementation happened and redesign of the encryption method to try to make it work with the Message system.

# References

To aid me in the development of the game some libraries where needed:

## sfml

Simple Fast Multimedia Library 2.4.2 – Used as a “Game engine” to handle textures and drawing -> <https://www.sfml-dev.org/download/sfml/2.4.2/>

## TGUI

TGUI 07.7 – Used for UI support and management -> <https://tgui.eu/download/>

## CryptoPP

CryptoPP 7.0 – Encryption library, to aid with the generation of keys, encryption-decryption and transformation of bytes (Very useful for plain byte extraction of sfml packages) -> https://www.cryptopp.com/