FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Wolfenstein 3D[1]

Documentación Técnica

[75.42] Taller de Programación I
Segundo cuatrimestre de 2020

Chiara Bauni - 102981

Damian Ganopolsky - 101168

[1]Link a repositorio https://github.com/DamianGanopolsky/TDPWolfenstein

Índice

1. Requerimientos de software

1.1. Sistema Operativo

Es necesario tener un sistema operativo linux con distribución Linux. El juego fue probado con los sistemas operativos Linux Mint 19.2 y Ubuntu 20.

1.2. Dependencias

Tanto para compilar el cliente como el servidor, es necesario instalar GCC, Make y CMake. Además es necesario instalar los paquetes de desarrollo de SDL y de YAML. Todos los paquetes nombrados anteriormente se pueden instalar con los siguientes comandos:

```
$ sudo apt-get install build-essential
$ sudo snap install cmake --classic
$ sudo apt-get update
$ sudo apt-get install libsdl2-dev libsdl2-image-dev
libsdl2-ttf-dev libsdl2-mixer-dev libyaml-cpp-dev
```

2. Descripción general

Este trabajo práctico consiste de dos entidades principales: el servidor y el cliente. El servidor es multi-cliente, es decir, que se comunica con varios clientes a la vez. Además contiene toda la lógica del juego y responde adecuadamente según los comandos que recibe de los clientes respondiendo si fuese necesario.

El cliente por su lado no contiene nada de lógica del juego sino que se actualiza constantemente según los cambios que el servidor le envía. Los eventos capturados del lado del cliente son enviados al servidor para su procesamiento. En función a la respuesta enviada por el servidor el cliente actualiza y muestra los cambios.

Por otro lado también tenemos el editor de mapas. En donde se diseñan los mapas colocando las paredes, puertas, tesoros y otros objetos, también se indica en donde arrancará cada jugador. Luego estos mapas pueden ser utilizados a la hora de comenzar la partida.

3. Servidor

3.1. Descripción general

Dentro del servidor por un lado tenemos lo que respecta al modelo y la lógica del juego y por el otro el control y comunicación con los clientes. El control y comunicación consta de comenzar una nueva partida, aceptar clientes, recibir comandos y responder, manteniendo una comunicación constante con ellos. El modelo contiene varias clases que son las que generan una respuesta frente a un nuevo comando, utilizando la lógica establecida por la partida.

3.1.1. Inicialización del server y loop

Cuando se comienza la ejecución del server, este lanza dos hilos el del Loop y el del Accepter. El Accepter acepta nuevos clientes, cuando llega una nueva conexión, la acepta lanzando un hilo Login, el cual se encarga de recibir el nickname elegido por el cliente. Al lanzar el hilo el Accepter queda a la espera de nuevas conexiones. Posteriormente, cuando el Login recibe el nickname, pushea un ConnectionElement a la cola de NewConnections, la cual es vaciada por el Loop que se encuentra a la espera de nuevas conexiones.

Por otro lado el Loop contiene en su función principal un constant rate loop en donde se ejecutan los siguientes pasos: se encarga de procesar nuevas conexiones, procesar nuevos comandos y actualizar el modelo y finalmente limpiar las conexiones que finalizaron. A continuación, detallamos los pasos enunciados anteriormente:

1. Procesar las nuevas conexiones (vaciando la cola de NewConnections) y agregarlas al PreGame y a ClientsConnected(quien a su vez lanzara un nuevo hilo ClientHandler).
2. Recibir los comandos recibidos por los receivers de los ClientHandlers y ejecutar esos comandos(cambiando el Game).
3. Se actualizan los jugadores y las variables dependiendo del tiempo(ClientsConnected notifica a todos los ClientHandlers de las actualizaciones realizadas al Game).
4. Procesar las conexiones finalizadas y liberar los recursos no utilizados.

Es así como el Loop se repite de manera continua siguiendo un rate y el Game se va actualizando según los comandos recibidos.

3.1.2. Comunicación con los clientes

Para la comunicación con los clientes se crearon dos entidades que nos permiten la comunicación ordenada entre el juego y el envío y recibo de mensajes: los Commands y las Notifications. Los Commands son creados por el método newCommand de la clase Command, donde según el tipo de opcode enviado por el cliente, se crea un comando que ejecuta la respuesta necesaria del juego frente a la llegada del mismo. Tenemos un command distinto para cada opcode que puede llegar del cliente. Por el otro lado las Notifications se utilizan para notificar los clientes de los cambios ocurridos en el game. Por eso es que hay 2 tipos de notificaciones:

- Event: Los eventos son actualizaciones de los jugadores es decir son sucesos que únicamente afectar a los Player. Por ejemplo, notifica a todos los cliente que el jugador 1 se movió o que el jugador 2 murió, etc.
- ItemChanged: Como lo dice su nombre se notifica de algún cambio en los objetos del mapa, ya sea que se agarraron balas en cierta posición del mapa o que se dejó un arma en otra posición.

3.2. Clases

3.2.1. Clases de control

A continuación se describen brevemente las clases que manejan la comunicación entre el cliente y servidor y que ejecutan el juego generando una respuesta por parte del mismo.

- Servidor: Clase principal que lanza los hilos del Loop y Accepter cuando se ejecuta el método operator(). Para finalizar la ejecución del server es necesario ingresar 'q', luego de la misma se hacen los stop y joins correspondientes.
- Loop: Es el loop principal del juego, en donde las iteraciones del loop están controladas por el manejo del rate. Como fue explicado anteriormente, se encarga de procesar nuevas conexiones, procesar nuevos comandos y actualizar el modelo y finalmente limpiar las conexiones que finalizaron.
- Accepter: El Accepter se encarga de que, a medida que llegan nuevas conexiones, aceptarlas y lanzar un hilo de Login. Esto se repite para todas las conexiones nuevas, hasta que se le haya hecho un stop(al Accepter) y no pueda aceptar mas conexiones.

- Login: Su función es recibir el nickname por parte del cliente. Una vez logrado esto se finaliza su ejecución y pushea un ConnectionElement a la cola de NewConnections.
- ClientsConnected: Se encarga de tener las conexiones activas con los clientes y permite enviarles a todos los clientes conectados notificaciones. Es decir, pushea las notificaciones a la cola de todos los ClientHandlers activos.
- ClientHandler: Una vez que se finalizó la ejecución del login, el cliente es agregado a ClientsConnected y se lanza el ClientHandler. Este objeto, se encarga de la comunicación con el cliente, es decir, lanza dos hilos un receiver y un sender y posee una cola de notificaciones y una cola de comandos(referencia a la cola del Loop). Como lo indican sus nombres, el receiver se encarga de recibir los comandos enviados por el cliente y pushearlos a la cola del Loop y el sender se encarga de popear de la cola de Notifications las notificaciones y enviárselas al cliente.
- Command: Es una clase abstracta donde uno de sus métodos es newCommand que se encarga de crear nuevos comandos según el opcode recibido y poder realizar la comunicación necesaria entre lo recibido y la respuesta del juego. Se crean comandos como: CommandStartMovingUp o CommandChangeWeapon, entre otros.
- Notification: Es una clase abstracta para la implementación de notificaciones de manera polimórfica. Existen dos tipos de notificaciones: ItemChanged y Event.

3.2.2. Diagramas de clases de control

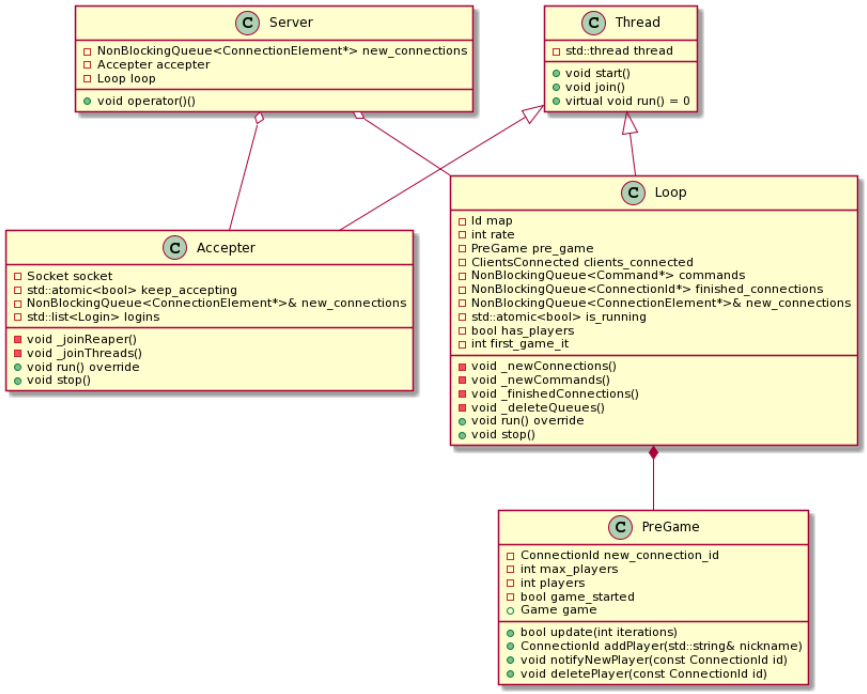

Figura 1: Clases de control principales

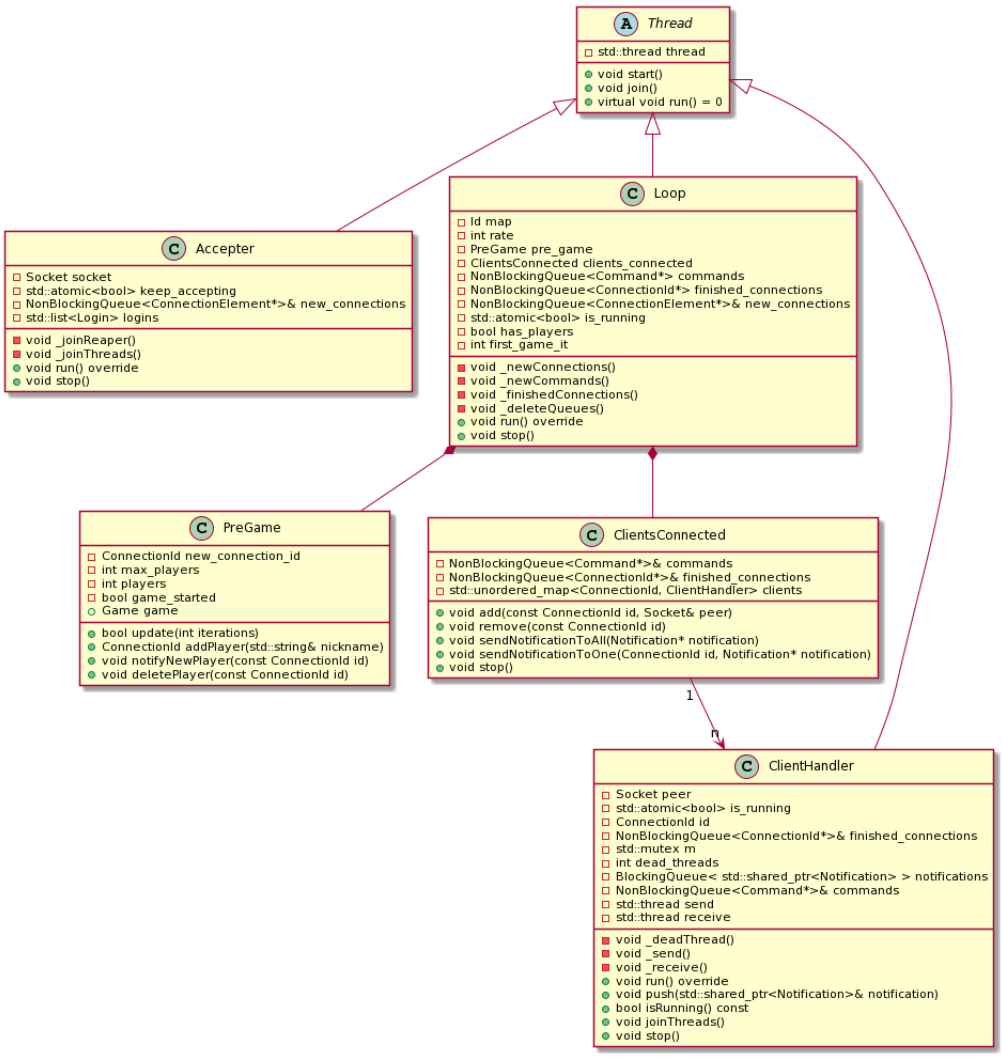

Figura 2: Clases de control principales

3.2.3. Clases del modelo

A continuación se describen brevemente las clases que modelan el juego.

- PreGame: Este objeto se encarga de esperar a que la cantidad de jugadores necesarios para cada mapa se hayan conectado, una vez que esto suceda se lanza el Game. Se encarga de agregar y borrar a los jugadores y actualiza el game cuando Loop lo requiera.
- Game: El Game contiene toda la lógica del juego, contiene una map con los Players en la partida, el Map, el ObejctMap, los ClientsConnected(para notificar cambios), y el nombre del mapa. Con estos objetos y los métodos del Game llamados por los comandos procesados se genera el desarrollo del juego.
- PostGame: El PostGame se encarga de contener a los jugadores que mueren para cuando se finalice la partida poder obtener un top 5 de los jugadores con mayor score según las variables tenidas en cuenta. En este caso tenemos en cuenta la cantidad de enemigos matados y los puntos por tesoros(a cada una de estas variables se la asigna un peso que es una constante en server_config)

- Interactor: Cuando el Player se mueve esta función se encarga de devolver una respuesta indicando si se agarró o no un objeto del mapa al cambiar de cuadrícula.
- Player: Este objeto, representa a un jugador manejado por un cliente. Tiene un id, un PlayerPosition, PlayerInfo y un State, y atributos booleanos indicando si está vivo, se está moviendo, rotando, o disparando. Responde a todos los comandos recibidos, y tiene métodos para llevarlos a cabo. Los distintos métodos responden a la actualización de atributos (con el paso del tiempo y eventos que ocurren), el movimiento del jugador, se le suma vida, balas tesoros, etc y otras acciones en las que el Player deba intervenir.
- PlayerInfo: Guarda la cantidad de vida, balas, tesoros, llaves, enemigos matados, resurrecciones que el jugador tiene. Además tiene un inventory en el que están las armas que tiene y un atributo equiped con el arma que tiene equipada en el momento.
- PlayerPosition: El PlayerPosition se encarga del movimiento y posición del Player. Tiene como atributos la posición x e y, el ángulo de visión, la dirección y rotación que tiene. Este objeto tiene la función de mover en x e y al jugador, establecer las direcciones y rotaciones y obtener la siguiente posición a la que se movería.
- State: El State es una interfaz que es implementada por dos clases Alive y Dead. Estos son los dos únicos estados que puede tener el jugador.
- Objects: Es una interfaz implementada por todos los objetos presentes en el mapa.
- Items: Es una interfaz implementada por las clases de objetos que pueden ser tomados o dejados en el mapa.
- Weapons: Weapon es una clase abstracta donde, las clases que heredan de ella implementan el método attack, según el arma que es utilizada el ataque cambia.
- Treasures: Treasure es una clase abstracta que tiene como atributo protegido los puntos que se obtienen al obtener cierto tesoro.
- Healers: Healer es una clase abstracta que implementa el método heal() y tiene como atributo protegido los puntos que se obtiene al curar.
- NonBlockingObjects: Esta clase abstracta es implementada por los objetos que no son bloqueantes, de esta manera se puede decidir si cierto jugador puede moverse a la siguiente posición o no.
- BlockingObjects: Esta clase abstracta es implementada por los objetos que son bloqueantes, de esta manera se puede decidir si cierto jugador puede moverse a la siguiente posición o no.
- Map: El Map se encarga de cargar el mapa del Yaml para tener una matriz con las cuadrículas del mapa importado del Editor. Además tiene métodos para situar objetos en cierta cuadrícula y obtener los objetos que están en una cuadrícula dada.
- ObjectMap: El ObjectMap es un objeto compuesto por un unordered map que contiene un int(codigo que representa un objeto) y un Object, asociando así el Object con su valor establecido en el mapa de cuadrículas. Su único método getObject nos permite a partir del código(int) obtener el objeto asociado al mismo.

3.2.4. Diagramas de clases del modelo

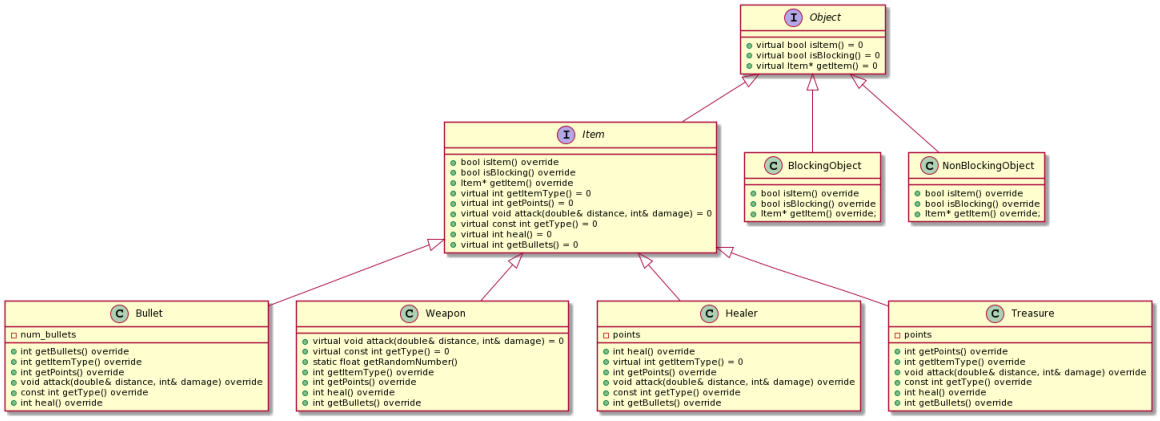

Figura 3: Clases del modelo principales

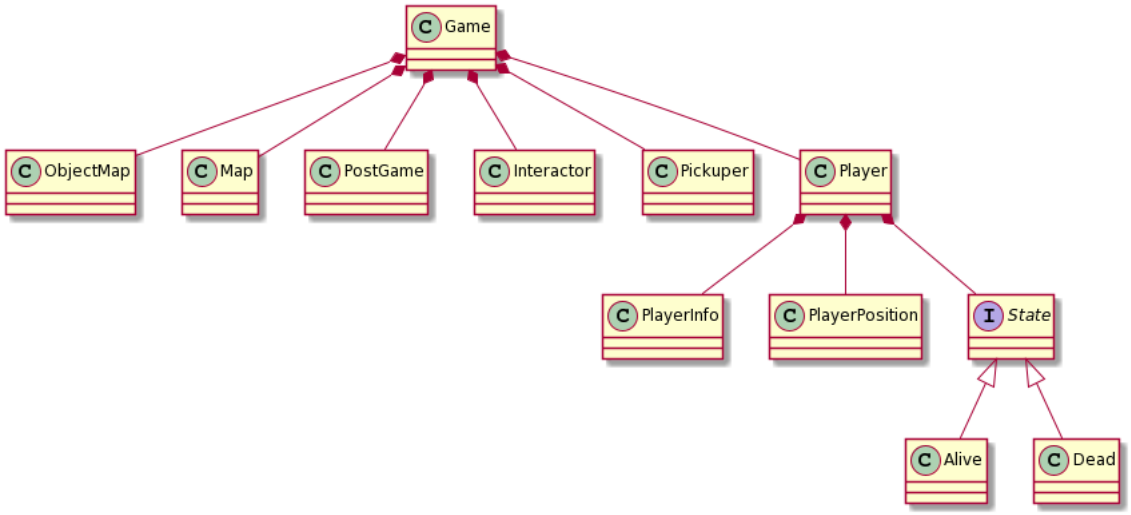

Figura 4: Clases del modelo principales

3.3. Diagramas de secuencia

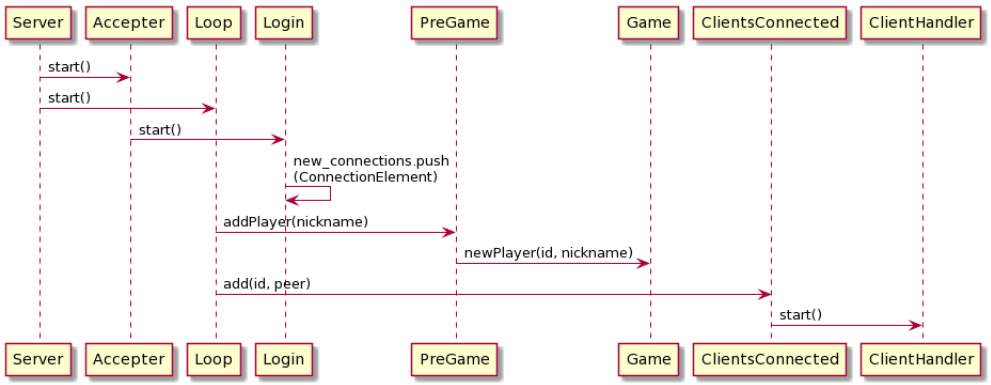

Figura 5: Llegada de una nueva conexión

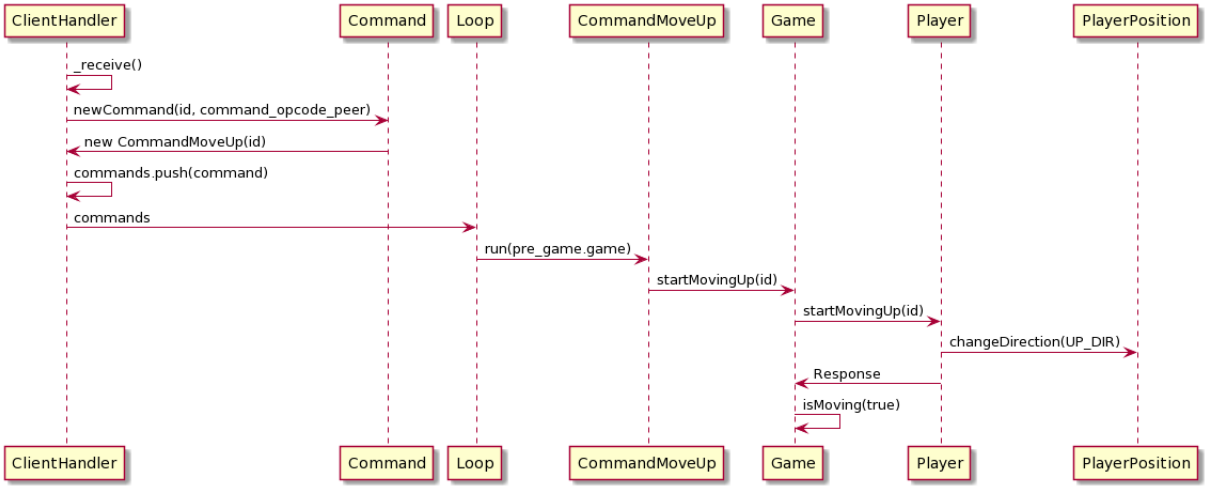

Figura 6: Llegada de el comando MoveUp

4. Cliente

4.1. Descripción general

Este módulo es el encargado de la interacción con el usuario. El modelo está dotado con la información necesaria para el renderizado de la interfaz gráfica. A grandes rasgos, el contenido de las carpetas puede ser catalogado con las siguientes funciones:

- ClientConnector: Posee los hilos send y recv, además del socket usado para la comunicación con el servidor y de las clases que son encoladas tanto en la cola bloqueante como en la no bloqueante.
- SdlClasses: Contiene clases que encapsulan los usos de SDL.
- Panel_status: Dispone de las clases principales para el cargado de imágenes, contiene la lógica del renderizado de distintos estados para una imagen dada. e.g: El renderizado del oficial estando quieto tendrá distintos "estados" que estarán dados por el ángulo en el que se visualiza el mismo.
- Yaml: Se encuentran las clases correspondientes para el parseo del archivo yaml de configuración y el de los mapas

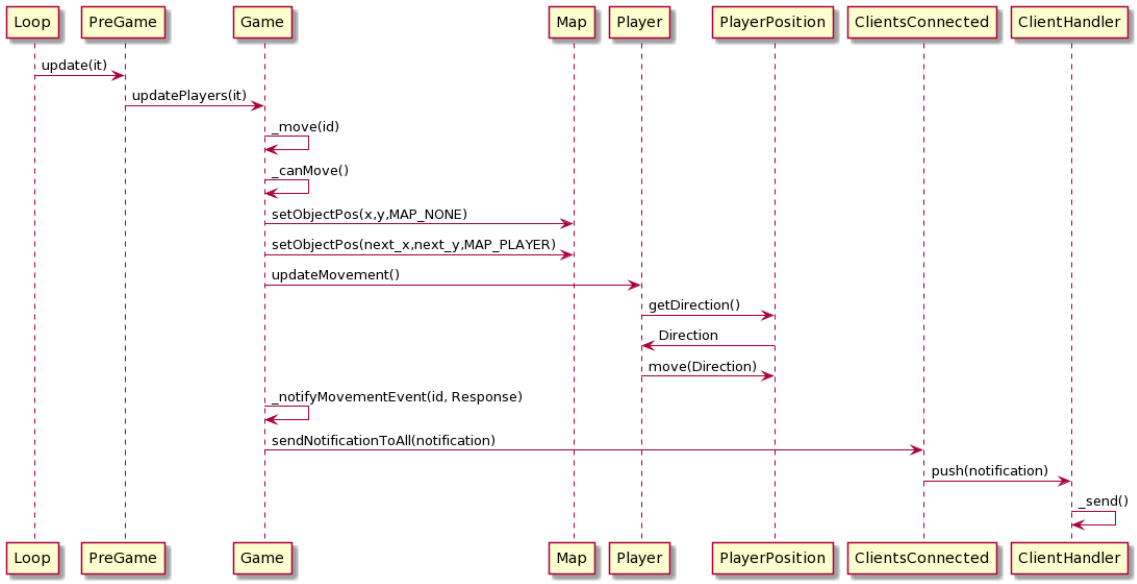

Figura 7: Procesamiento del comando MoveUp en el servidor

A su vez, en la raíz de la carpeta client hay una gran variedad de clases con distintas funcionalidades, que van desde el estado actual del mapa y del jugador, como también el renderizado usando la técnica de ray casting. A continuación se hará una breve descripción de las clases presentes en la raíz de client:

- ClientManager: Es la clase central del cliente, primero establece la conexión con el servidor y espera recibir el evento "New_player_event" para así inicializar los primeros valores del jugador y abrir el mapa inicial(recibe el nombre del archivo yaml a abrir). Luego, le envía el nickname del jugador al servidor, y posteriormente inicializa los distintos hilos que manejaran el envío y recepción de mensajes con el servidor, además de las colas usadas según lo visto en clase. Luego de dichas inicializaciones, se encarga de lanzar el constant rate loop, el cual constantemente captura eventos en el cliente para que sean enviados al servidor(mediante "Player_handler"), actualiza el modelo mediante la clase ReceiveController, y renderiza usando la clase Client.

- Player_handler: Esta clase tiene como atributo a la cola bloqueante que es usada para el envío de mensajes al servidor. Es la encargada de capturar los eventos del cliente, transformar dichos eventos en comandos cuyo formato lo provee la clase "Command", y encolarlos en la cola bloqueante.

- Client: Es la clase cuyo propósito es determinar la escena a mostrar(ya sea una pantalla, o el juego mismo) y dar la orden para que se dibuje la misma. Panel_window: Es la clase encargada del dibujado a más alto nivel, hace uso de una cola para la lógica del renderizado. Primero encola las paredes y luego encola los elementos que son visibles al jugador, luego va desencolando los mismos y los renderiza. La acción de desencolar y renderizar los elementos es el gran cuello de botella de la aplicación del cliente, debido a pocas optimizaciones en el uso del ray casting.

- Map_2d: Contiene todos los elementos del mapa, ya sean cuadrículas de paredes, las cuales estarán en el std::map booleano "boxes", como también objetos en el mapa, los cuales estarán en "elements_map" y además, contiene los estados de los jugadores enemigos según su id.

- Player: Contiene el estado del jugador referido por el cliente, a su vez, varios de sus atributos son manejados por la clase Player_info.
- Game_element: Esta clase tiene como propósito la determinación de si un dado objeto o jugador es visible por el jugador central. Además, determina que estado de textura usar para los distintos ángulos en los que se pueden ver los distintos jugadores por ejemplo.
- Elements_panel_queue: Cola usada en Panel_window para la lógica del renderizado.
- Wall_texture: Su función es el cargado de las imágenes de paredes.
- Ray: Clase encargada del trazado de rayos, sus métodos permiten obtener información sobre el punto en el que chocan los distintos rayos.
- Ray_shotter: Su función principal se encuentra en el método "Shoot", la cual devuelve un objeto de tipo Ray, según la posición y el ángulo actual del jugador. Además, tiene en cuenta el mapa de cuadrículas en las que se encuentran paredes para la obtención del Ray.

A continuación se hará una breve descripción de las clases contenidas en la carpeta ClientConnector:

- ReceiveController: Es una de las clases más importantes del cliente, esta misma en su método update, realiza un "pop" de la cola no bloqueante. En el caso de que se obtenga un UpdateMessage no nulo, actualiza el modelo del cliente(el cual está conformado por las clases Map y Player), además reproduce los sonidos característicos del evento recibido. Vale aclarar que el update del modelo se realizará dependiendo del evento recibido desde el servidor, siguiendo el protocolo de comunicación explicado al final de esta documentación.
- Command: Clase a encolar en la cola bloqueante para el envío de mensajes al servidor.
- UpdateMessage: Clase a encolar que es creada con la recepción del mensaje enviado desde el servidor
- Sender: Clase que hereda de Thread, su funcionamiento consiste en constantemente hacer un "pop" de la cola bloqueante y en dicho caso, notificar a la clase ClientSocket para que envíe el comando al servidor.
- Receiver: Clase que hereda de Thread, mientras el hilo este activo notificará a la clase ClientSocket que debe seguir recibiendo eventos enviados desde el servidor.
- ClientSocket: Es la encargada del manejo del socket usado en la comunicación con el servidor, su uso está regulado por las clases Sender y Receiver.

Breve descripción de las clases contenidas en la carpeta panel_status:

- Element_panel_status: Su propósito es la carga de secuencias de imágenes que esten numeradas en orden ascendente. Útil para el cargado de secuencias de imágenes que conformen una animación, por ejemplo, para el disparo del arma ChainCannon, al usar los nombres ChainCannon0, ChainCannon1 y ChainCannon2, se cargaran las 3 imagenes y estas estarán asociadas a dicho "elemento".
- Player_panel_status: Clase usada para el cargado de la gran parte de sprites del juego, haciendo uso de la clase anteriormente explicada, se tienen varios elementos del juego. Al usar su método get_texture, se obtiene la imagen que se debe renderizar dependiendo del ángulo con la cual tiene que ser mostrada. Su método copy_to_renderer, tiene como función el renderizado de la interfaz gráfica siempre presente del jugador, esta se dibujara dependiendo del estado, ya que, para las animaciones de los disparos como para los cambios de cara es necesario tener distintos estados.

Pequeña descripción de las clases ubicadas en la carpeta Yaml:

- MapYamlParser: Esta clase es la encargada del cargado de mapas serializados en formato yaml para obtener los mapas en el formato usado por el cliente.
- YamlConfigClient: Obtiene los valores de la configuración del cliente.

Breve descripción de las clases correspondientes a la carpeta SdlClasses:

- SdlMusic: Encargada de encapsular la reproducción de música según las librerias de SDL
- SdlSound: Encargada de encapsular la reproducción de sonidos según las librerias de SDL
- SdlText: Encargada de encapsular la obtención y renderizado de texto segun las librerías de SDL
- SoundManager: Encargada de la carga y reproducción de los distintos sonidos del juego
- MusicSoundtrack: Clase responsable de la reproducción de la música de fondo del juego

4.2. Diagramas de clases

A muy alto nivel, el loop principal del cliente es ejecutado por las siguientes clases:

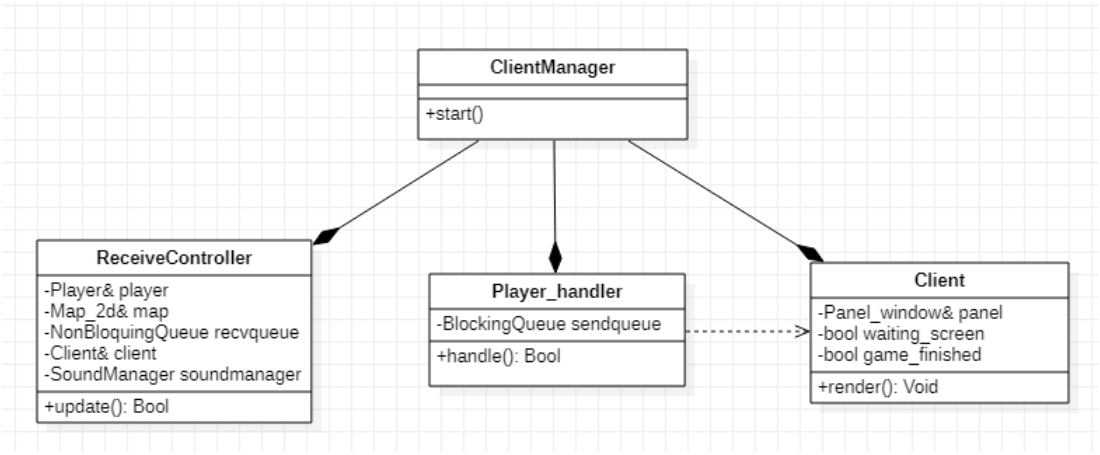

Figura 8: Diagrama de clases de alto nivel en cliente

La clase ReceiveController es la encargada del update del modelo, el cual se basa primordialmente en el estado de las clases Player y Map. En líneas generales, los eventos recibidos que afectan al jugador del cliente, afectan el estado del player. Mientras que, los eventos que modifican el mapa, o el estado de los otros jugadores alrededor del mapa, son contemplados en la clase Map_2d.

El diagrama de clases de las dependencias de dicha clase se muestra en la Figura 10.

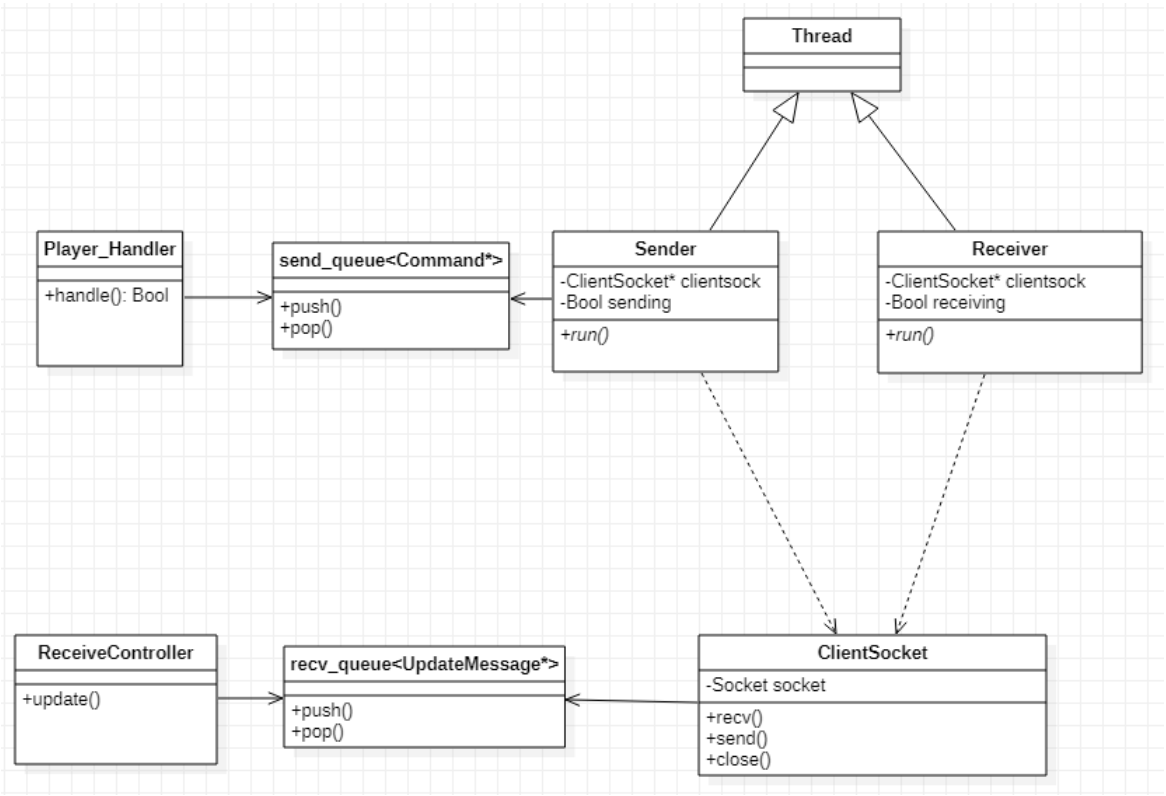

Figura 9: Diagrama de clases correspondiente a la "conexión" del cliente para respetar el protocolo de comunicación con el servidor.

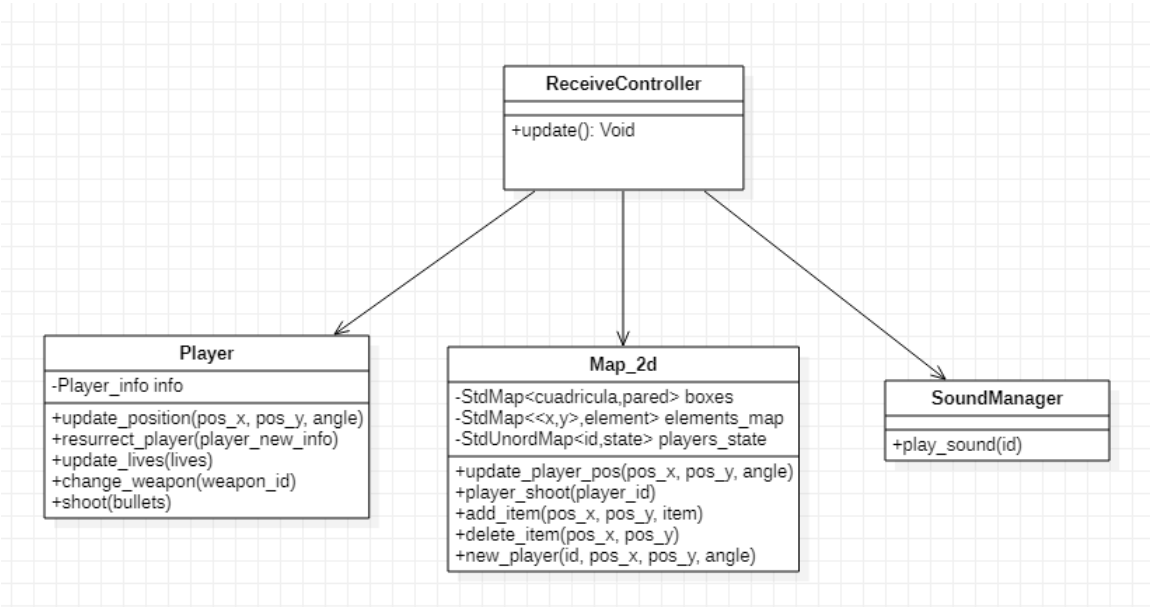

Figura 10: Clases responsables en lineas generales del update del cliente

5. Editor

5.1. Descripción general

El editor de niveles es la aplicación encargada de crear, editar y guardar mapas. Cumple un rol bastante importante, ya que, nos permite crear mapas nuevos en cuestión de minutos, mientras que si se hiciera manualmente se demoraría mucho. Además, al ver ciertos desbalances en mapas, estos se pueden modificar fácilmente.

5.2. Clases

- EditorManager: Es la encargada del manejo del cliente desde el más alto nivel. Su método start(), inicializa el renderer(encapsulado en SdlWindow), y ejecuta el loop principal de la captura de eventos y renderizado.
- Event_Handler: Es la clase encargada de capturar los distintos eventos y notificar a la clase "editor" para que se actualice dependiendo del evento capturado.
- Editor: Su propósito es determinar de qué tipo de evento se trata, ya sea, drag & drop, click & point, click & drag, etc. Además, notifica a la clase scene para que se actualice dependiendo del tipo de evento.
- Scene: Tiene la lógica de que escena mostrar, dependiendo del estado en el que se encuentre el modelo. Actualiza las distintas escenas en base al evento tratado.
- MapOptionsView: Encargada de la vista y comportamiento de la pantalla que muestra las distintas opciones de mapas a abrir. Cada opción es una entidad MapButton.
- InputTextView: Su función es renderizar el menú para poder escribir un nombre de mapa a guardar.
- Map: Es la clase encargada de mostrar el mapa a editar, actualizarlo en base a los eventos, cargarlo y hacer las acciones previas a la serialización de dicho mapa.
- YamlParser: Es responsable de la serialización del mapa para escribirlo en un archivo yaml, como también es la encargada de abrir mapas para que puedan ser usados por la clase Map.

Además, se usaron clases para encapsular las acciones relacionadas con SDL, como, SdlWindow o SdlText. Y también se utilizaron otras clases secundarias como MapButton(correspondiente a las distintas opciones de mapas a elegir), o TextBox cuyo fin fue el de encapsular el texto escrito al momento de escribir el nombre del mapa a guardar.

5.3. Diagrama de las clases más importantes

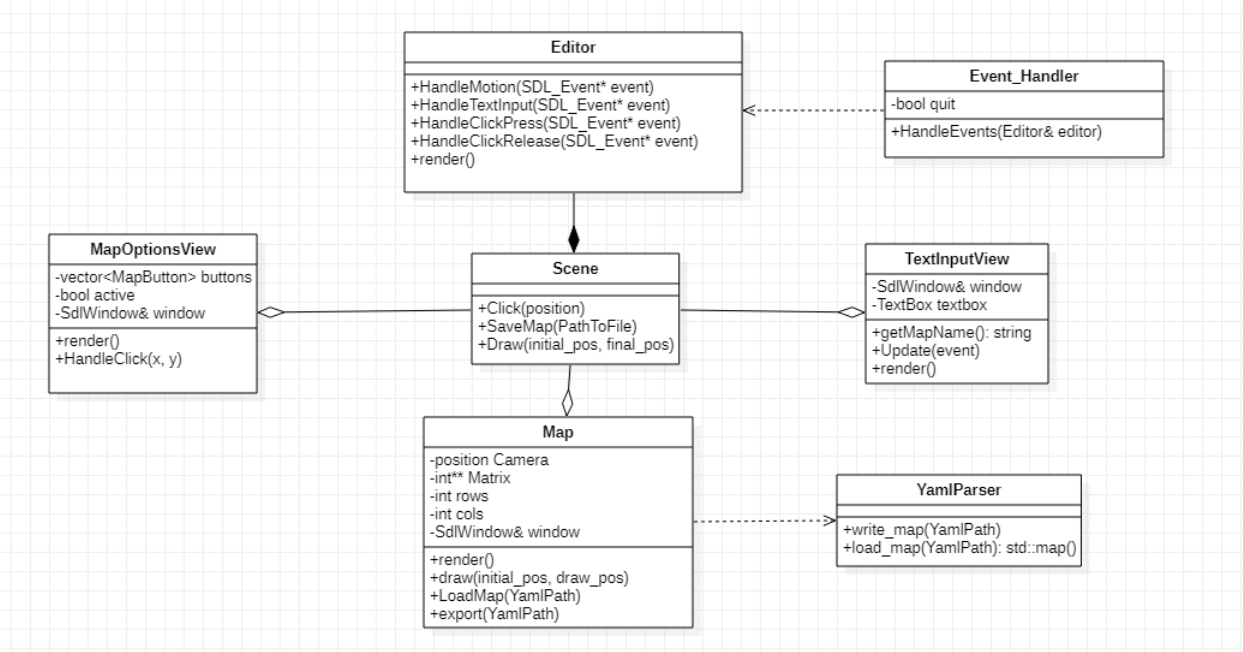

Figura 11: Agregar algo

6. Descripción del protocolo Cliente-Servidor

Para la correcta comunicación entre ambas aplicaciones es necesario saber de antemano qué mensaje esperar y el tamaño del mismo. Con este fin, se estableció que el primer Byte enviado o recibido siempre corresponderá a un Opcode, el cual determinará de qué tipo de mensaje se trata. Al saber de qué tipo de mensaje se trata, se sabrá el tamaño del mismo y a que corresponderá cada Byte.

Para el envío y recepción de variables de 4 Bytes de tamaño, antes del envío por socket se convirtieron dichos Bytes a Little Endian, usando la función “htole32” cuya documentación se encuentra en Linux man page . Mientras que, para la recepción de estas variables se cambió el endianness de little endian al endianness correspondiente del host, usando la función “le32toh”.

Los opcodes se dividieron en tres categorías, tomando los siguientes valores:

Valor	Opcode
0	Event_opcode
1	Item_changed_opcode
2	Command_opcode

A su vez, el segundo byte enviado o recibido también es usado para distinguir el tipo de mensaje que será procesado, pudiendo así distinguir más mensajes.

6.1. Valores del opcode

0 : Event_opcode.

Estos tipos de mensajes son enviados desde el servidor al cliente. Este opcode es usado cuando el servidor le envía al cliente, el cambio producido por eventos enviados desde algún cliente.

A su vez, el segundo Byte cuando se trata de un Event_opcode, tendra el siguiente significado:

Valor	Evento	Contenido
0	Movement_event	id del jugador, $\text{pos}_x, pos_y, angulo$
1	New_Player_Event	id del jugador, nombre del mapa a abrir, pos_x, pos_y, angulo, vida, vidas
2	Attack_event	id del jugador atacando, cant de balas
3	Be_attacked_event	id del jugador atacado, vida que tiene
4	Death_event	id del jugador que muere, pos_x , pos_y
5	Resurrect_event	id del jugador, pos_x, pos_y, angulo, vida, vidas
6	Change_weapon_event	id del jugador, arma cambiada 0,1,2,3
7	Scores_event	cant_de_jugadores, nickname1, puntos_finales_1, cant_asesinatos_1, tesoros_1, nickname2...
8	Start event	-

El cliente lo primero que recibirá es el New_player_event, y con la llegada de este evento inicializara los atributos del jugador y además abrira el mapa con los objetos y paredes iniciales en formato yaml.

El evento "Scores" implica el fin del juego, y el cliente mostrará la pantalla final con sus estadísticas.

El evento "Start Event" implica el inicio del juego, y el cliente terminara de mostrar la pantalla de espera y se podrá interactuar con el juego y visualizarlo.

1: ItemOpcode

Estos mensajes son enviados desde el servidor al cliente.

Valor	Tipo de item	Contenido
0	Close_Door_itm	pos_x, pos_y (de la puerta)
1	Open_Door_itm	pos_x, pos_y (de la puerta)
2	Kit_Taken_itm	id del jugador que lo agarra, pos_x(del item),pos_y(del item), vida del jugador
3	Food_taken_itm	id del jugador, pos_x, pos_y, vida del jugador
4	Blood_taken_itm	id del jugador, pos_x, pos_y, vida del jugador
5	Key_taken_itm	id del jugador, pos_x, pos_y, keys del jugador
6	Weapon_taken_itm	id del jugador, pos_x(del arma), pos_y(del arma)
7	Treasure_taken_itm	id del jugador, pos_x, pos_y, puntaje del jugador
8	Bullets_taken_itm	id del jugador, pos_x, pos_y, balas del jugador
9	Machine_gun_dropped_itm	pos_x, pos_y
10	Chain_cannon_dropped_itm	pos_x, pos_y
11	Bullets_dropped_itm	pos_x, pos_y

Debido a que el cliente primero carga el mapa inicial, al agarrar un ítem, el servidor le envía la posición en la que tiene que dejar de ser mostrado dicho ítem, y el cliente elimina el objeto que está en dicha posición. Además, en varios casos se actualiza el valor mostrado de balas, vidas, etc.

Si bien están los mensajes con respecto a la actualización del estado de las puertas, estos no se implementaron en el cliente debido a ciertas limitaciones en el ray casting.

2: Command Opcode

Valor	Tipo de comando	Contenido
0	START_MOVING_UP_CMD	-
1	START_MOVING_DOWN_CMD	-
2	START_MOVING_LEFT_CMD	-
3	START_MOVING_RIGHT_CMD	-
4	STOP_MOVING_CMD	-
5	START_ROTATING_LEFT	-
6	START_ROTATING_RIGHT	-
7	STOP_ROTATING	-
8	START_SHOOTING_CMD	-
9	STOP_SHOOTING_CMD	-
10	OPEN_DOOR_CMD	-
11	CHANGE_WEAPON_TO_KNIFE_CMD	-
12	CHANGE_WEAPON_TO_GUN_CMD	-
13	CHANGE_WEAPON_TO_AUTOMATIC_GUN_CMD	-
14	CHANGE_WEAPON_TO_CHAIN_CANNON_CMD	-

Solo son enviados 2 Bytes por cada evento, correspondientes al opcode y al tipo de comando respectivamente, no es necesario enviarle más bytes al servidor.