# Documentación Técnica

February 2021

## 1 Requerimientos de software

SO: Linux / Ubuntu

Nuestro TP hace uso de las librerias LUA, YAML y SDL con sus librerias para sonido, texto e imagenes.

Los comandos para instalar estas respectivas librerias son:

```
sudo apt-get install libsdl2-dev
sudo apt-get install libsdl2-mixer-dev
sudo apt-get install libsdl2-ttf-dev
sudo apt-get install libsdl2-image-dev
sudo apt-get install libyaml-cpp-dev
sudo apt-get install lua5.3
sudo apt-get install lua5.3-dev
```

## 2 Descripción general

El proyecto esta separado en cuatro módulos, estas son, el editor, el cliente, el servidor y un modulo común. El modulo común funciona como un modulo auxiliar para los otros tres módulos y provee el acceso a las herramientas externas que utilizamos (SDL, YAML).

El modulo editor es una aplicación, que corre localmente, esta provee una herramienta para que el usuario pueda generar mapas de juego.

El modulo cliente es una aplicación que corre en comunicación al modulo servidor. El cliente provee la interfaz de usuario, mientras que el servidor se encarga de los protocolos de comunicación y la logica de juego.

## 3 Modulo: Editor

### 3.1 Descripción general

El modulo editor es una aplicación de creación de mapas para luego correr en una partida. La aplicación permite cargar, emitir y modificar mapas que se

guardan en archivos de extensión .yaml, que luego se pueden usar como campo de juego durante una partida.

El editor tiene una clase principal editor que contiene el loop de obtención de eventos del usuario y luego delega la funcionalidad y el dibujo de la UX al resto de las clases.

## 3.2 Clases

### 3.2.1 editor

La clase editor se encarga de recibir las acciones del usuario y delegar a las distintas clases las intenciones del usuario.

### 3.2.2 map_ui

La clase map_ui se encarga de dibujar la representación 2D del mapa de juego utilizando las librerías de SDL.

### 3.2.3 menu_ui

La clase menu_ui se encarga de dibujar los elementos de interfaz del menú del editor, para esto hace uso de las librerías de SDL. Se dibujan los botones de acción del menú, la ui de selección de elementos de mapa, las flechas de navegación del menú y el tooltip de los elementos.

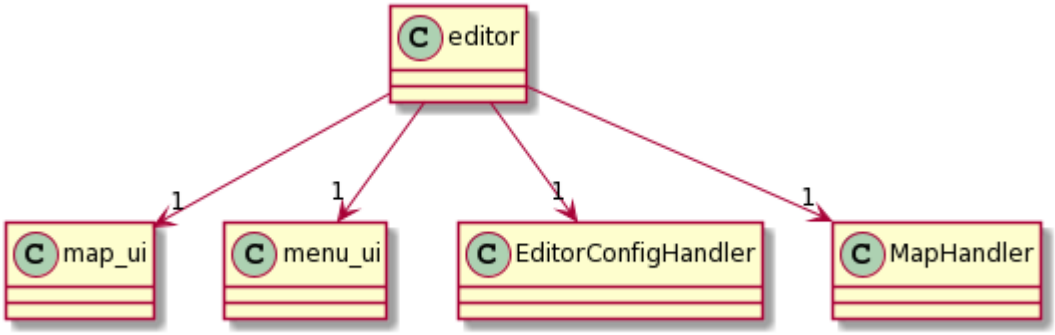
### 3.2.4 EditorConfigHandler

La clase EditorConfigHandler se encarga de levantar la configuración necesaria para el editor, esta contiene un método que lee un archivo de la carpeta de configuración y levanta del mismo los valores de configuración.

### 3.2.5 MapHandler

La clase MapHandler esta definida en el modulo common, ya que es utilizada en todos los módulos del proyecto.

### 3.3 Diagramas UML




### 3.4 Descripción de archivos y protocolos

El modulo editor hace uso de los archivos de mapa terminados en .yaml, ademas es el encargado de generar esos archivos para el uso dentro del cliente y el servidor. Tambien hace uso del archivo editorConfig.yaml que contiene la configuración del editor.

## 4 Modulo: Cliente

### 4.1 Descripción general

### 4.2 Clases

#### 4.2.1 Jugador

Se encarga de toda la lógica del jugador propio de ese cliente. Lo que le sucede a él impacta en la clase HudJugador

#### 4.2.2 HudJugador

Se encarga de dibujar en pantalla los datos de vida, balas, arma actual,

#### 4.2.3 World

La clase world se encarga de inicializar y actualizar a los jugadores,

#### 4.2.4 Menu

La clase Menu se encarga de la lógica de las pantallas de inicio del cliente, además de dibujarlas. Para esto la clase tiene dos métodos por pantalla un run[Pantalla] y un drawPantalla, donde Pantalla se reemplaza por la pantalla en cuestión.

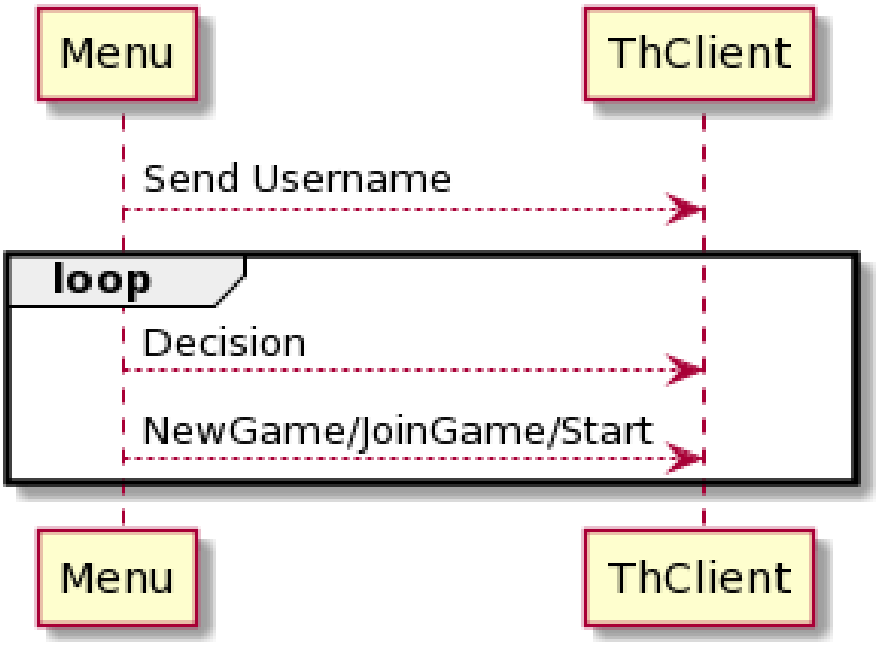
#### 4.2.5 ClientConfigHandler

La clase ClientConfigHandler se encarga de levantar la configuración necesaria para el cliente, esta contiene un método que lee un archivo de la carpeta de configuración y levanta del mismo los valores de configuración.

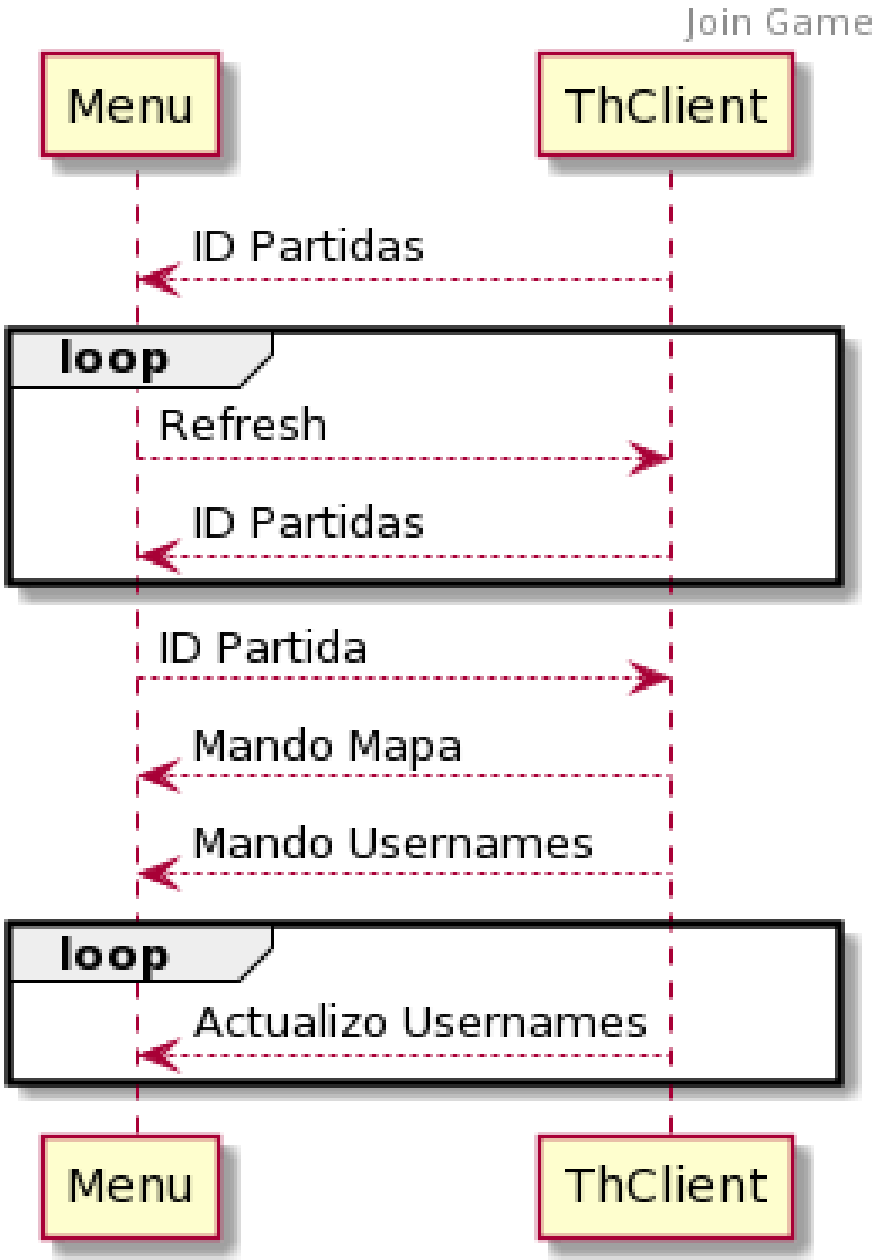
### 4.3 Interaccion del Menu con el Server

Al inicio del programa, tenemos la seccion del Menu princial, el cual permite elegir Partidas, Mapas, etc.

El siguiente grafico exhibe los primeros intercambios entre ambas partes.




En primer lugar el Menu envía el username del usuario actual, seguido de la decisión sobre si quiere empezar una nueva partida, unirse o en caso de haber creado una partida, empezarla. La decision es un ID que nos permite identificar luego entre distintas ramas de ejecucion.



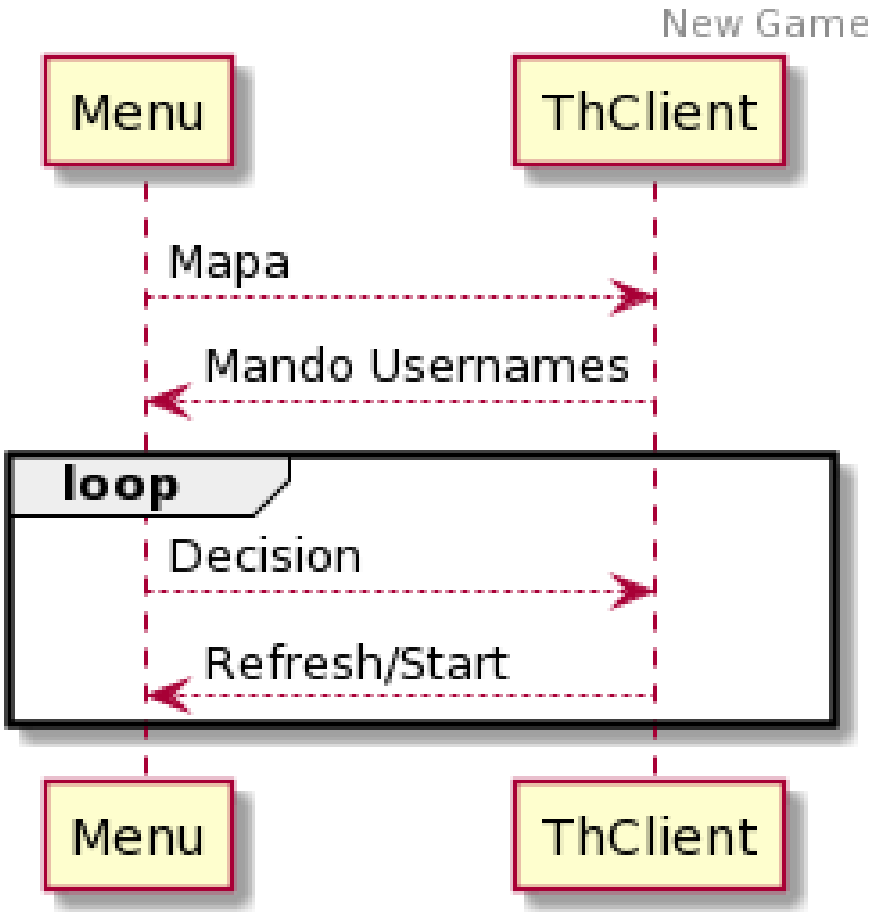

En el caso de que el jugador elija unirse a una partida, recibira un listado con los identificadores de las mismas.

Una vez recibido ese mensaje, el cliente podra elegir una, o pedir que se refresquen las opciones.

Una vez decidido, el servidor manda el mapa correspondiente de esa partida

y un listado con los usuarios de esa misma.

El usuario queda ahora, en modo de espera a que el host inicie la partida. En caso de unirse nuevos jugadores, el menu recibira el mensaje de actualizacion y los mostrara automaticamente.




El creador de la partida, el host, debe enviar el mapa que sera utilizado. De esta forma el servidor podra luego enviarles el mapa a los nuevos jugadores.

Una vez que el cliente envio el mapa, es el encargado de iniciar la partida cuando asi lo quisiera. Tambien puede pedir refrescar el listado de jugadores apuntados a su partida.

### 4.4 Descripción de archivos y protocolos

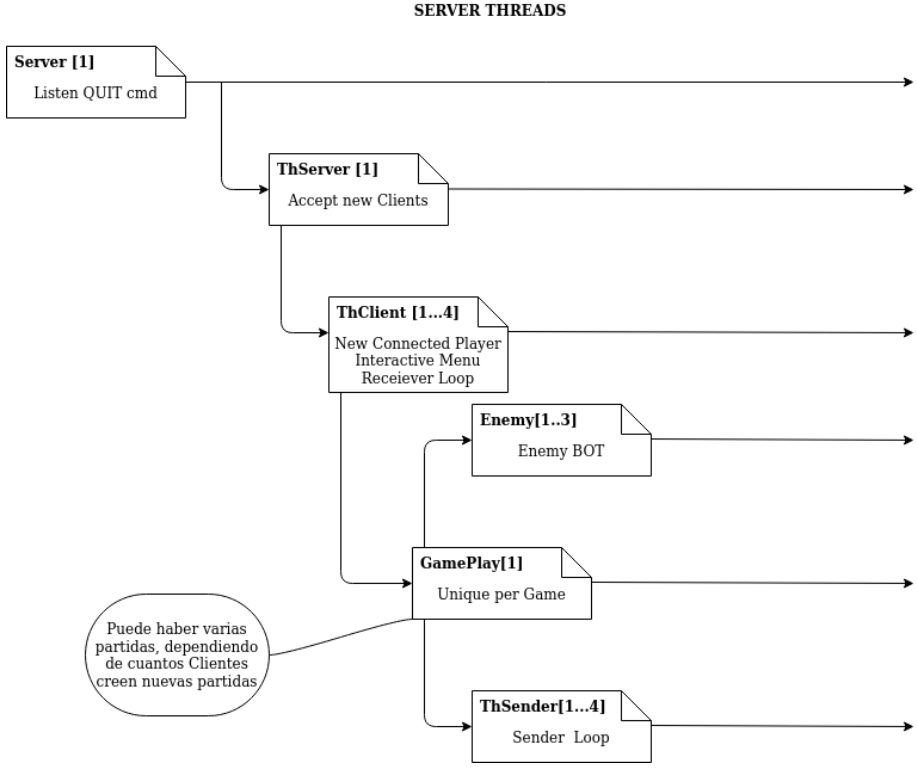
# 5 Modulo: Servidor

## 5.1 Descripción general

El servidor es el encargado de modelar los juegos. es decir, recibe intenciones del usuario y en base a eso actualiza el modelo del juego y envía a los usuarios el estado del mismo. por ejemplo, si un jugador quiere disparar, le enviará al servidor la intención de disparo. es este quien, luego de verificar que el disparo pueda ser efectuado, se encargara de actualizar el modelo con la acción pedida. El servidor permite múltiples juegos con un máximo de 4 jugadores.

## 5.2 Hilos en el Servidor

Grafico sobre la creacion de hilos en el servidor. El ejemplo muestra para el caso de una partida sola, multijugador. En caso de haber varias partidas, la cantidad de ThClients variara y en consecuencia, la cantidad de partidas



### 5.3 Clases

Como se puede apreciar en el grafico superior, distintas responsabilidades fueron repartidas en hilos para garantizar el correcto funcionamiento del juego.

En primer lugar el main del server, que instancia al ThServer, queda escuchando el comando Quit para cerrar de forma ordenada toda la secuencia de juego.

En segundo lugar, tenemos el ThServer, encargado de aceptar nuevos clientes, instanciar un Protocolo de comuniacion y distribuir el GameHandler hacia los nuevos clientes. Tambien se encarga de eliminar los hilos inactivos.

Luego tenemos el ThClient, clase principal para la interaccion con el cliente. Se encarga de la interaccion inicial con el Menu y luego finaliza su rol como Receiver Loop de las intenciones del cliente.

Cuando un ThClient elije la opcion de nueva partida, se crea una instancia de GamePlay desde el GameHandler. Este hilo comienza a correr cuando el ThClient recibe la orden de START MATCH. Sus funcionalidades incluyen el GameLoop del juego.

Cuando se inicia la partida, GamePlay asigna a cada cliente un ThSender(ThClientSender) encargado de enviar los snapshots al cliente, y al finalizar, los puntajes.

Finalmente, también es GamePlay la clase que instancia a los Enemigos(Bots) de la partida calculando la diferencia entre jugadores online y capacidad mínima de juego.

### 5.4 LUA - IA

Dentro de modulo del servidor, se encuentra la implementacion de la Inteligencia Artificial de los Bots, con scripts de LUA.

La interaccion existe entre Enemy y EnemiLogic, siendo el segundo el script de su inteligencia. El mismo recibe la informacion exclusiva de los jugadores online y la procesa. Busca al jugador mas cercano y procede de forma secuencial a: Girar el angulo hasta apuntar al jugador; Caminar hasta ubicarse dentro de determinado rango; Atacar.

Dicha secuencia, a pesar de su sencillez, tiene algunas funcionalidades adicionales. El giro lo hace en el sentido mas rápido; dependiendo del arma actual, se acerca mas, o menos. Si se queda sin municiones, pero agarra una del suelo, vuelve a portar el arma.

## 6 Logica de juego

### 6.1 Clases

#### 6.1.1 Map

El mapa o tablero es el encargado de almacenar a los jugadores, cohetes, puertas y objetos. Además del mapa real del juego que contiene los ids de cada casillero, es el encargado de pasar a los Players las intenciones del usuario y de avisarles

a las clases que pasó un loop de juego llamado tick. se verá más adelante la sección de flujo de juego

#### 6.1.2 Player

Se encarga de procesar las intenciones del usuario. es el encargado de recibir daño, conocer su vida, las vidas restantes y si está vivo. delega la lógica de posiciones a la clase position y la lógica de manejo de inventario y disparos a la clase Inventory. también almacena el score y las muertes que causó.

#### 6.1.3 Position

contiene la lógica de movimiento y posición del jugador. sabe su hitbox, se le puede preguntar si una posición es interior a esta. es la encargada de modificar la posición del jugador y de verificar que sea válida.

#### 6.1.4 Inventory

contiene las armas y llaves del jugador. esta clase conoce el arma actual del jugador, su munición y es la encargada de la lógica de cambio de armas, agregar nuevas armas y recolectar munición. Contiene un array con punteros a la clase Weapon. Esto permite realizar polimorfismo con la lógica de ataque.

#### 6.1.5 Weapon

Clase abstracta madre de todas las armas. contiene la lógica de disparo de la que hablaré en su apartado. Utiliza una clase llamada hitscan Raycast que permite obtener el punto de impacto de un arma con el primer objeto impactable en su trayectoria.

#### 6.1.6 Door

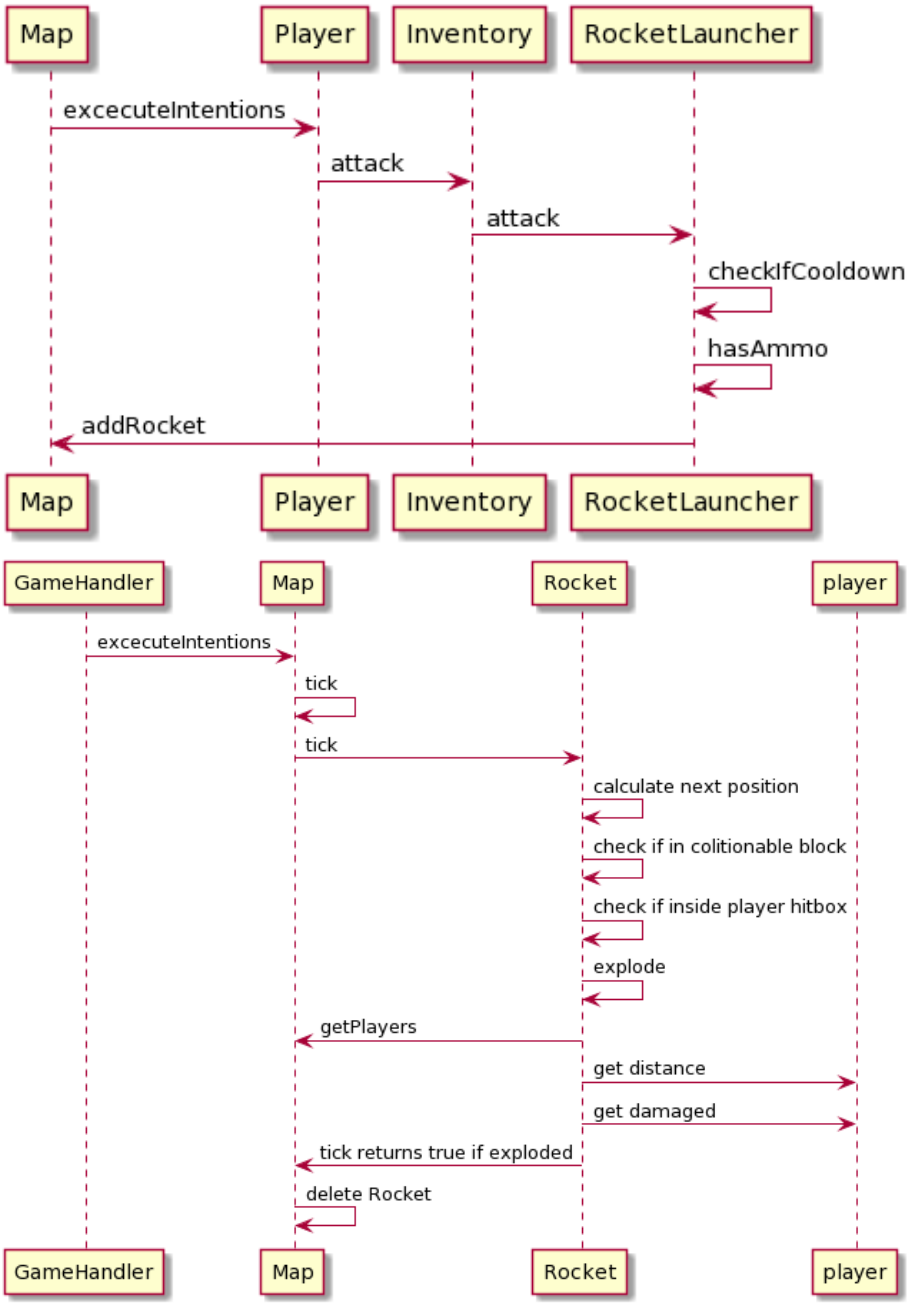
conoce su estado, si está abierta o no. y si está cerrada con llave. sabe que llave puede abrirla y permite ser abierta por esta.

#### 6.1.7 ServerConfigHandler

La clase ClientConfigHandler se encarga de levantar la configuración necesaria para el servidor, esta contiene un método que lee un archivo de la carpeta de configuración y levanta del mismo los valores de configuración.

### 6.2 Diagramas UML

Incluyo unos diagramas de secuencia de como funciona el rocket launcher


Map
Player
Inventory
RocketLauncher
excecuteIntentions
attack
attack
checkIfCooldown
hasAmmo
addRocket
Map
Player
Inventory
RocketLauncher
GameHandler
Map
Rocket
player
excecuteIntentions
tick
tick
calculate next position
check if in colitionable block
check if inside player hitbox
explode
getPlayers
get distance
get damaged
tick returns true if exploded
delete Rocket
GameHandler
Map
Rocket
player

## 6.3 Secuencia del Game Loop

Una vez que el cliente envía la señal de iniciar el partido, los jugadores ya fueron cargados por el game handler y agregados al mapa. se le envía al mapa el comando de Start. al recibir esta instrucción el mapa recorre el doble vector de is de casilleros y carga en las variables correspondientes los valores de las puertas e items. también, posiciona a los jugadores en los puntos de spawn e inicia el game timer.

Una vez iniciado, por cada loop se le pasa al mapa las intenciones de los usuarios. el mapa de forma sincrónica itera sobre estas y le pasa a cada Player las intenciones de su respectivo usuario. el mapa primero le avisa a todas las clases que lo requieran que pasó un game loop con la función tick(). Esto permite que las puertas se cierren solas luego de 10 segundos de estar abiertas, que la ametralladora dispara 5 veces con una sola intención de disparo, y que un jugador que está muerto vuelva a vivir si le corresponde. Luego uno a uno los jugadores procesan las instrucciones y modifican el modelo en función de estas.

Al acabar el procesamiento de las intenciones, el game handler le pide al mapa la información del estado actual necesaria para enviar al cliente el estado del juego. Además del estado del juego, se envían al cliente acciones particulares que al no ser un estado sino una acción puntual requieren un procesamiento distinto. en principio estas acciones son disparos o explosiones de cohetes y contienen el punto de impacto en caso de una bala, o la ubicación de la explocion en caso de un cohete. Finalizado esto, se le pregunta al mapa si termino el juego, ya sea por tiempo o porque 1 solo jugador esta con vida y se continúa o se termina y se manda al cliente la información para la pantalla del final de partida

## 6.4 Secuencias Interesantes

### 6.4.1 movimiento

para el movimiento, primero que nada implemente un sistema de hitboxes para evitar que el sprite de los jugadores no quede dentro de la pared. una vez hecho esto, simplemente se mueve al jugador, en caso de que este quede con su hitbox en una posición inválida, para que su hitbox quede "pegada" al casillero invalido. Además, como el movimiento esta separado en x e y, al caminar contra una pared, solo se anula la dirección del movimiento perpendicular al borde del casillero no permitido, esto genera una disminución de la velocidad cuando se mueve en angulo contra la pared sin frenar completamente el movimiento, esto da una sensacion de fricción que aumenta el realismo.

## 6.5 Acciones Dependientes Del Tiempo

Como ya mencionamos, tenemos una función tick que permite informar al modelo del paso del tiempo. esto al incluirle una clase Timer que permite saber el tiempo que transcurrió de un momento a otro, permite que eventos como los

disparos de la machine gun y el que las puertas se cierren solas se resuelvan de forma trivial.

### 6.5.1 Disparo

Al implementar el disparo me pareció poco interesante la propuesta del tp que planteaba un calculo de angulo y distancia. por esto me propuse implementar una lógica de impacto basada en hitscan. es decir, el arma dispara una recta y si esta colisiona con algún jugador entonces el jugador recibe daño. para esto, es necesario construir una recta entre el jugador y el punto de impacto el cual se obtiene con una función de raycast. una vez obtenido, se buscan jugadores cuya posición sea impactable. es decir, si el jugador mira hacia las x positivas, que su posición en x sea mayor a la del jugador y anterior a la del impacto. una vez obtenido estos chequeos, se tendría que buscar una colisión entre recta que el jugador con el impacto con los segmentos que componen las hitboxes cuadradas de los demas jugadores. para esto, aprovechando que las hitboxes siempre estan alineadas con los ejes. utilizo la siguiente formula

$$lambda(Impact - Player) + Player = RectaDeHitbox$$

Como cada cara de la hitbox tendrá un x o y constante, por ejemplo: recta hitbox en x seria (playerx - hitboxRadius, y ) sabiendo esto, separo la ecuación en sus coordenadas y esto me permite despejar este punto de impacto con la recta, en este caso el valor y.

la formula seria:

$$lambda = \frac{X - PlayerX}{ImpactX - PlayerX}$$

para luego remplazar y obtener el Y

$$\frac{X - PlayerX}{ImpactX - PlayerX}(ImpactY - PlayerY) + PlayerY = IntesrectionY$$

Habiendo echo esto, simplemente se tiene que verificar que este punto de impacto este en la hitbox del jugador. para esto se realiza un simple chequeo viendo que la distancia de la intersección no sea mayor a la hitbox.

$$|Y - playery| <= hitboxRadius$$

una vez detectada la colisión se le avisa al jugador que fue dañado.

# 7 Modulo: Common

## 7.1 Descripción general

En el modulo common se encuentran clases de uso general, y wrappers para herramientas externas. Se trata de clases que se llaman desde varios de los otros módulos, entre ellas la comunicación completa cliente-servidor.

## 7.2 Clases

### 7.2.1 MapHandler

La clase MapHandler provee métodos para la carga de los mapas con extensión de archivo .yaml. Provee métodos para levantar mapas, emitir mapas y para mantener un log de los mapas que se han creado localmente.

### 7.2.2 Serializer

Es la clase encargada de llenar con información valida los structs/clases que transportan la información a ser enviada, como por ejemplo: Snapshot. Se encarga tanto de cargar información como de deserializarla.

### 7.2.3 Snapshot

Esta clase es el principal transportador de información para la comunicación. Contiene distintos arreglos para cada tipo de dato en particular; players y objects. La clase esta pensada para ser desechada en cada ciclo del game-loop y solo utiliza memoria estática.

### 7.2.4 Intention

Al igual que es snapshot, es crucial para el envió de información por parte del cliente en sentido al servidor. Se opto por utilizar una cadena de chars interpretadas como intenciones. El ID en esta clase es critico para que el servidor determine de quien proviene la solicitud.

### 7.2.5 Actions

Se opto por diferenciar del resto de las intenciones a las acciones, como una medida para garantizar la consistencia de las mismas. De esta forma, cuando un cliente recibe una Action por parte del servidor, le da un tratamiento diferencial ya que garantiza su ejecución, a diferencia de los snapshots que pueden ser "pisados" por información mas reciente.

### 7.2.6 Protocol

La clase Protocol surgió como una necesidad para centralizar todas las llamadas sueltas que existían en el código, al socket. De esta forma se logro no solo eliminar código repetido, sino corregir y canalizar en una sola clase cualquier envió de información. De esta forma, los errores surgidos a lo largo del trabajo fueron mucho mas rápidos y fáciles de identificar.

### 7.2.7 Colas Protegidas y Bloqueantes

Fueron armadas a medida para garantizar la consistencia de la información una vez que llegaba tanto al cliente como al servidor. Mediante este método

se simplifico y encapsulo el acceso a los recursos que podrían generar race-conditions o busy waits.

## 7.3 SDLWrappers

Definimos wrappers para el uso de las librerías de SDL, para la parte gráfica, la visualización de texto y el uso de sonidos en el juego. Todos estos wrappers manejan la creación de los objetos de SDL para después poder usarlos desde nuestro programa.

## 7.4 Diagramas UML

Al ser recursos auxiliares del resto de los modulos, nos parecio que un diagrama de clases no tiene mucho sentido, ya que las mismas clases no siempre tienen relación entre ellas.