FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

TALLER DE PROGRAMACIÓN I

1º CUATRIMESTRE 2021

Trabajo Práctico Final - Documentación técnica

AUTORES

Cai, Ana Maria - #102 150

Giampieri Mutti, Leonardo - #102 358

Vazquez Fernandez, Francisco Manuel - #104 128

Índice

1. Requerimientos de software

Los programas enunciados corren en Linux. Fueron probados en la versión 5.4.0-77-generic. Cualquier distribución con versionado reciente debería poder ejecutar el programa, y en particular todos los programas fueron probados en Ubuntu >= 18.04.

Para poder buildear, instalar y ejecutar, se requieren las siguientes dependencias:

- build-essential (v: 12.8ubuntu1)
- qt5-default (v: 5.12.8+dfsg-0ubuntu1)
- clang-8
- libyaml-cpp-dev (v: 0.6.2-4ubuntu1)
- qtmultimedia5-dev (v: 5.12.8-0ubuntu1)
- libsdl2-dev (v: 2.0.10+dfsg1-3)
- libsdl2-image-dev (2.0.5+dfsg1-2)
- libsdl2-gfx-dev (v: 1.0.4+dfsg-3)
- libsdl2-mixer-dev (v: 3.10)
- cmake (v: 3.16.3-1ubuntu1)
- make (v: 4.2.1-1.2)
- gcc/g++ (v: 4:9.3.0-1ubuntu2)

2. Descripción general

El trabajo cuenta con 3 programas ejecutables, un cliente, un servidor y un editor de mapas. El cliente y el servidor tendrán interacción por medio de un protocolo de comunicación basado en sockets TCP/IP, y el editor, sin conectarse con otra aplicación, generará los mapas que luego se utilizaran en las partidas, de modo visual por parte del cliente y de modo lógico por parte del servidor.

3. Cliente

3.1. Descripción general

El cliente es el programa que interactua con el jugador, posee una interfaz visual que le permite conectarse a servidores a través de una dirección IP y un puerto, le permite crear partidas y seleccionar distintos mapas o también unirse a una partida ya existente. Muestra una pantalla de espera en el momento previo de comienzo de la partida, y ya dentro de la partida se muestran todos los elementos pertinentes al juego en si. Todas las animaciones, elementos visibiles, sonidos, se encuentran en el cliente.

3.2. Clases

Enumeramos algunas de las clases más importantes.

- Client: inicializa los objetos principales, como la BlockingQueue de comandos, el SoundManager y WorldView, e inicia los hilos de Receiver, Sender, Drawer y SdlLoop. Se encarga también de setupear la window según la configuración recibida por argv.
- WorldView: Fachada para despachar la lógica de updates del juego. Contiene los elementos visuales principales del juego, Map, Hud, Stencil, CharacterManager. Cada update que se realice en la vista pasa por acá, siendo ejecutado por el hilo Receiver. A su vez, se encarga de llamar a cada método render/draw, esto es ejecutado por el hilo Drawer. Como acceden múltiples hilos, funciona como un monitor.
- Camera: se encarga de decidir que y que no se debe renderizar en la pantalla, dependiendo si está en vista del jugador. Funciona como un traductor, ya que obtiene posiciones lógicas para todos los objetos y las convierte a pixeles. Sigue al centro lógico (al jugador).
- Stencil: filtro aplicable a la pantalla, dinámicamente formado mediante un fondo negro, un triangulo isóceles y un circulo concéntrico al jugador, utilizando la técnica de alpha blending, soportada por SDL. El triángulo rota en torno a la posición del mouse. Es una manera de emular el efecto de sombras del juego original.
- MapView: es el encargado de formar el mapa de juego a partir de un protocolo de texto YAML, el cual recibe del servidor cada cliente al crear o unirse a una partida. Parsea el YAML para crear el background, las paredes y todas las zonas del juego.
- CharacterManager: contiene la lógica de updates de los personajes jugables. Llama a métodos de la clase Character ante una respuesta a un Update por parte del server, ya sea que ese Character fue golpeado, que cambio su arma, que disparó, que se movió o que apuntó en alguna dirección, o cualquiera sea la acción que requiera un update sonoro o visual.
- Drawer: hilo que se encarga de dibujar todo lo que sea renderizable. Accede al método render de WorldView, el cual se despacha en los render de cada uno de los objetos que

se puedan renderizar o dibujar. Tiene además un loop de tiempo constante para evitar la desincronización de las animaciones en caso de atrasarse respecto del tiempo de loop (actualmente fijo en 60 frames por segundo), basado en la técnica drop and rest.

- SdlLoop: popea eventos de la queue de SDL, los cuales se mapean a Commands, objetos que se pueden serializar en byte stream mediante un protocolo (común al cliente y al servidor). Estos comandos son pusheados a una BlockingQueue, compartida por SdlLoop y el hilo Sender, para ser popeados por el Sender y luego ser serializados y enviados al servidor vía socket. Internamente, llama al método WaitEvent de SDL, por lo cual el hilo se pone a dormir hasta que llegue algun evento. Para mantener la prolijidad en el código, cada evento se mapea a un handler (mediante punteros a métodos de la misma clase) el cual termina creando un nuevo Command para pushear a la cola bloqueante.

- Command: interfaz a la cual se le implementa el patrón Command. Cada hija representa una acción que puede realizar el jugador, ya sea moverse en una dirección, recargar su arma, disparar. El método principal serializa mediante Protocol la acción, que luego de ser recibida por el Server, se deserializa, mapeandose a un Event, clase muy similar del lado del Server.

- Sender: hilo que popea cada Command de la cola bloqueante, llama al método de serialización, obtiene un vector con el byte stream correspondiente y lo envía al servidor. Hace esto en un loop hasta que se cierre la BlockingQueue.

- Receiver: hilo que recibe byte streams de Update del Server. Cada stream es deserializado mediante el protocolo y se mapea a un método de la WorldView. Loopea hasta que llegue un Update de juego terminado o hasta que un externo cierre el socket, ya que en ese caso, recv falla, logrando el break del loop.

- Sdlwrap: se generalizan aquí todas las clases relacionadas a SDL, como SdlWindow, SdlTexture, SdlSound, etc. Son clases encargadas de wrappear las estructuras propuestas por SDL, todas RAII, para un mejor manejo de recursos y un código en general más prolijo. Como la implementación de SDL es en C, la mayoria de las estructuras de SDL hacen manejo de raw pointers y memoria dinámica, por lo cual los wrappers RAII aseguran un buen manejo de estos recursos.

- Renderizable: clase abstracta que representa a todos los objetos que se pueden dibujar a la pantalla. Contiene dentro una textura y una posición. Junto con la cámara, se implementa el patrón Double Dispatch para evitar cuentas extra respecto a la conversion de coordenadas lógicas a pixeles dentro de las clases que la heredan.

- Character: hereda de Renderizable, representa a los personajes jugables dentro de una partida. Contiene dentro suyo animaciones.

- NonMovable: hereda de Renderizable, representa a todos los objetos que no tiene sentido reubicarlos, tal es el caso de los tiles, las armas droppeadas, las paredes, etc.

- Animation: se encarga de separar las texturas que representen animaciones en sus frames, de tal manera de poder facilmente iterar por ellos cada vez que se quiera dibujar una animación.
- Explosion: representa la explosión de la bomba. Por lo visto en el juego original, la explosión de la bomba era la misma explosión que la de una granada pero multiplicada y en posiciones random alrededor de la coordenada de explosión. Para emular esto, se contiene dentro un vector de animaciones (todas las explosiones), que se activan a medida que va pasando el tiempo.
- Hud: contiene información visual pertinente al jugador, como son la cantidad de balas, la vida, el tiempo restante de la ronda, puntajes actuales, etc. Dentro suyo convierte números a posiciones en una imagen bmp que contiene los números estilizados del juego.
- SoundManager: contenedor de todos los sonidos del juego. Controla la cantidad de sonidos que se pueden reproducir al mismo tiempo y, utilizando SDL, controla el volumen del sonido dependiendo de la distancia.

3.3. Diagramas

El siguiente diagrama muestra una vista de alto nivel de como funciona el núcleo del cliente.

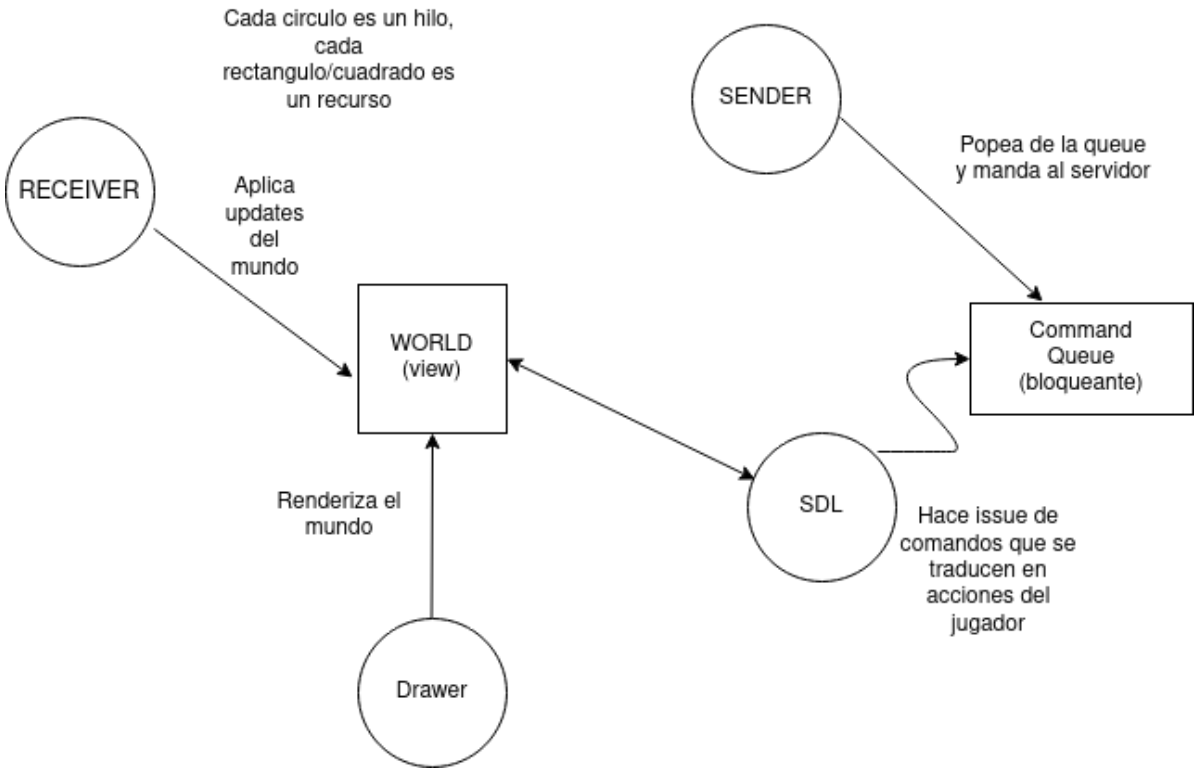

Figura 3.1: Diagrama de funcionamiento basico del cliente

Notamos que varios hilos acceden al WorldView, ya sea para hacer render (Drawer) o para hacer un update (Receiver). SdlLoop accede al WorldView para saber como despachar los comandos en ciertas instancias del juego. Por ejemplo, si nos encontramos en el lobby, no se pushea ningún comando que tenga que ver con el juego en si, sino que se ignoran los comandos que no tengan que ver con esta sección y recién se habilitan cuando se pasa al juego. Al mismo tiempo, podemos observar porqué la cola es bloqueante. Como el hilo Sender loopea constantemente buscando un nuevo comando, si no se detiene cuando no hay comandos en la queue va a seguir loopeando, quemando ciclos de cpu y llevando el proceso a una pérdida de performance innecesaria. Para esto, se implementa la BlockingQueue, que pone a dormir al hilo mientras no haya ningún comando listo para ser popeado, y SdlLoop, al pushear un nuevo comando, le avisa a Sender que ya hay un nuevo comando. Todo este sincronismo se logra utilizando Condition Variables. Vemos los diagramas de clases correspondientes.

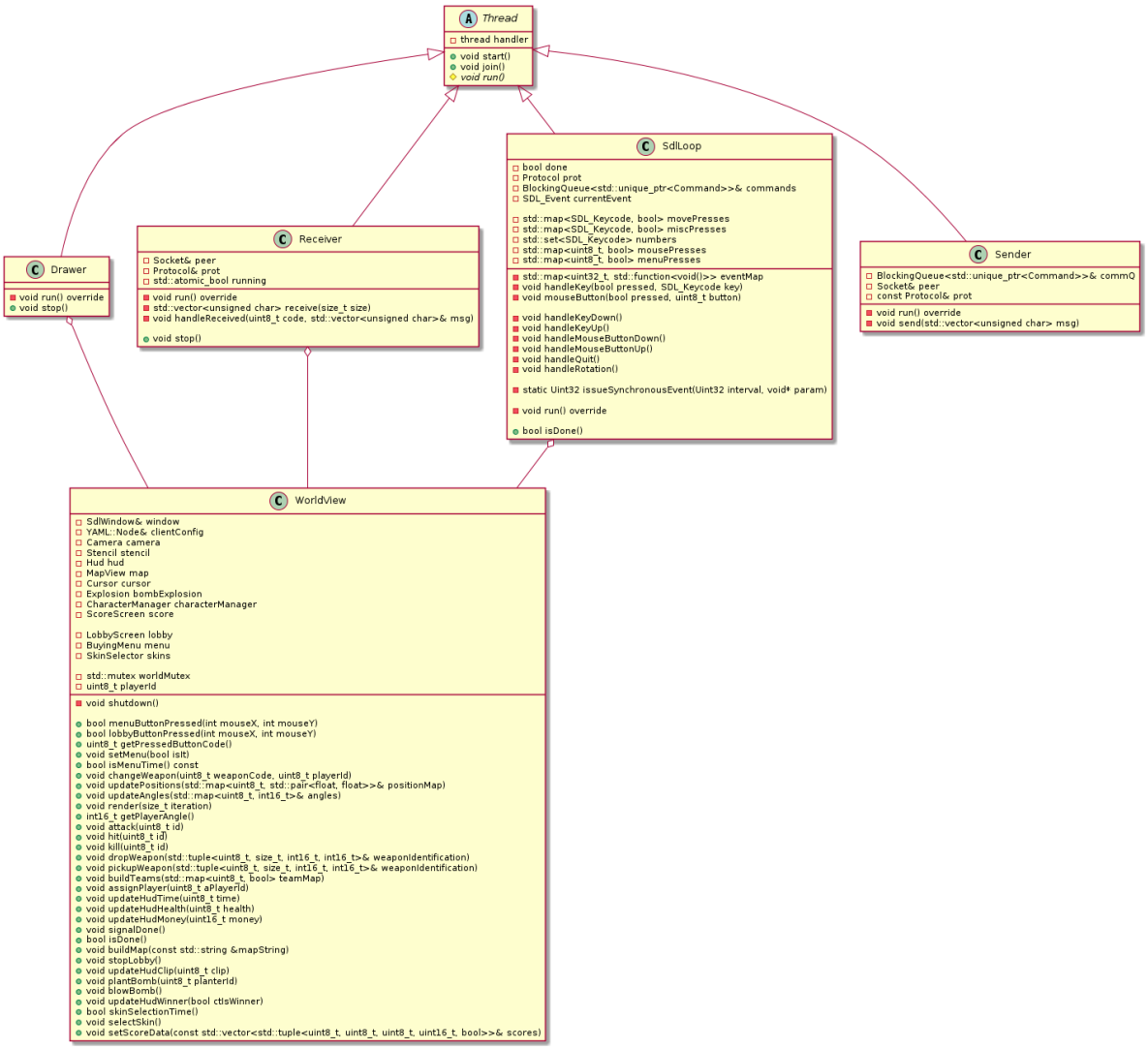

Figura 3.2: Núcleo del cliente

Notamos que WorldView es una clase muy grande. Esto se debe a que funciona como

fachada para el resto de los objetos del cliente, todos los updates se despachan desde allí.

Para dar una idea de como se relacionan estos objetos, se muestra un diagrama de secuencia de un usuario tratando de moverse ingresando la letra "w".

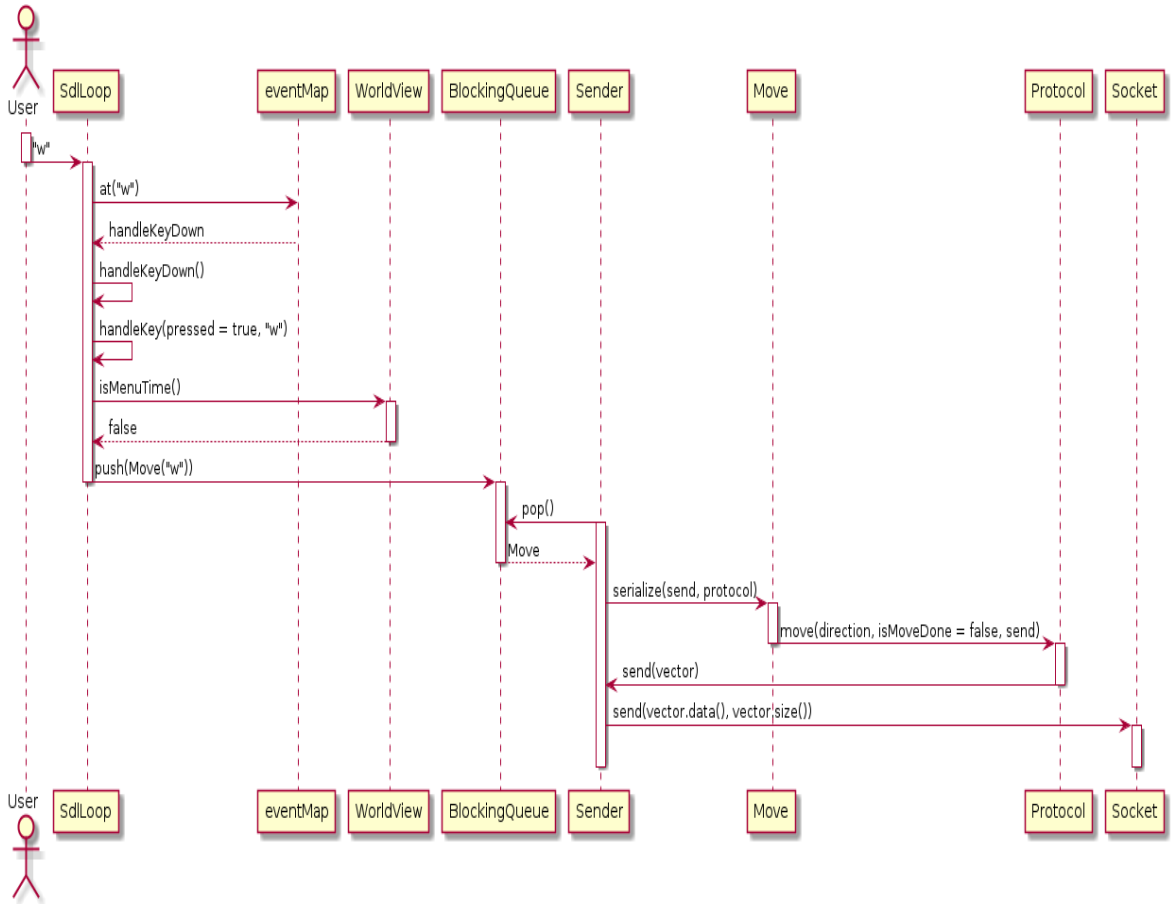

Figura 3.3: Input de la tecla W por el usuario en medio del juego

Move es una subclase de Command. Este mismo procedimiento de input y despachado según el tipo de comando sucede para todas las subclases. Mostramos su diagrama.

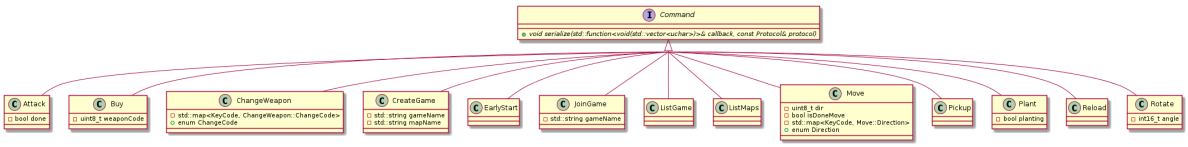

Figura 3.4: Patrón Command para la interfaz Command

Cada cierto tiempo, recibiremos Updates por parte del servidor que actualizaran elementos visuales y ciertas transiciones del juego. Por ejemplo, se puede recibir un Update de posiciones, el cual modifica las posiciones lógicas de cada Character. El siguiente diagrama de secuencia muestra como es ese proceso.

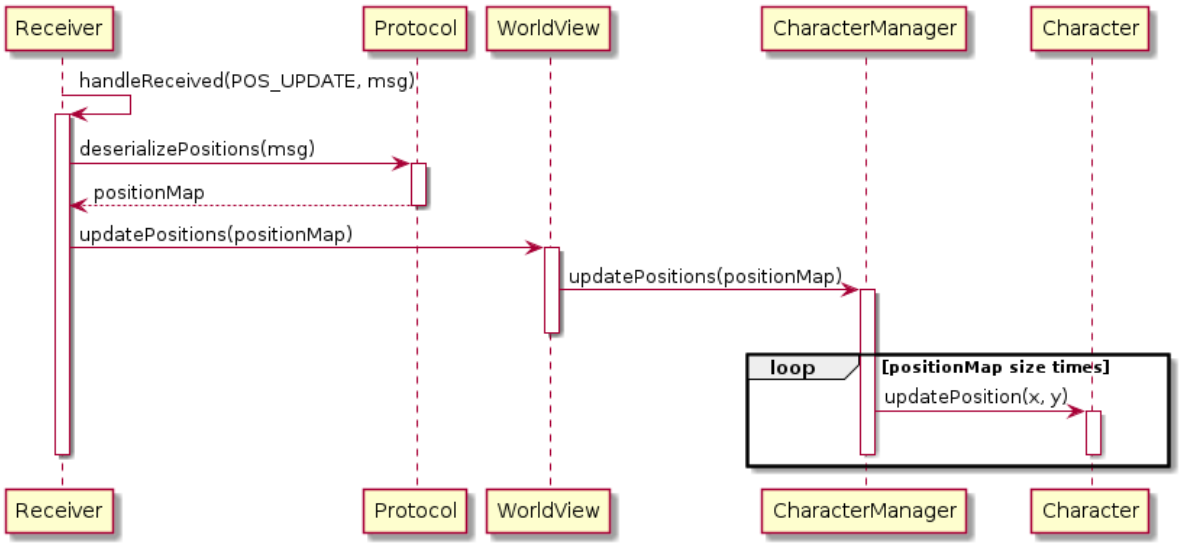

Figura 3.5: Update de posiciones para los jugadores

Cualquier tipo de Update que refiera a un Character tendrá el mismo tratamiento, pasa primero por el Receiver, que despacha en algún método de WorldView y de ahí al CharacterManager, quien identifica al Character correspondiente por su id de jugador y aplica el método correspondiente al Update recibido.

En el caso de una desconexión por parte del usuario al cliente o por un cierre del servidor, el cliente debe cerrar de una manera ordenada. Esto es, debe joinear todos los hilos que se lanzaron, vemos a continuación los dos casos.

3.3.1. Desconexión del Cliente causado por cierre del usuario

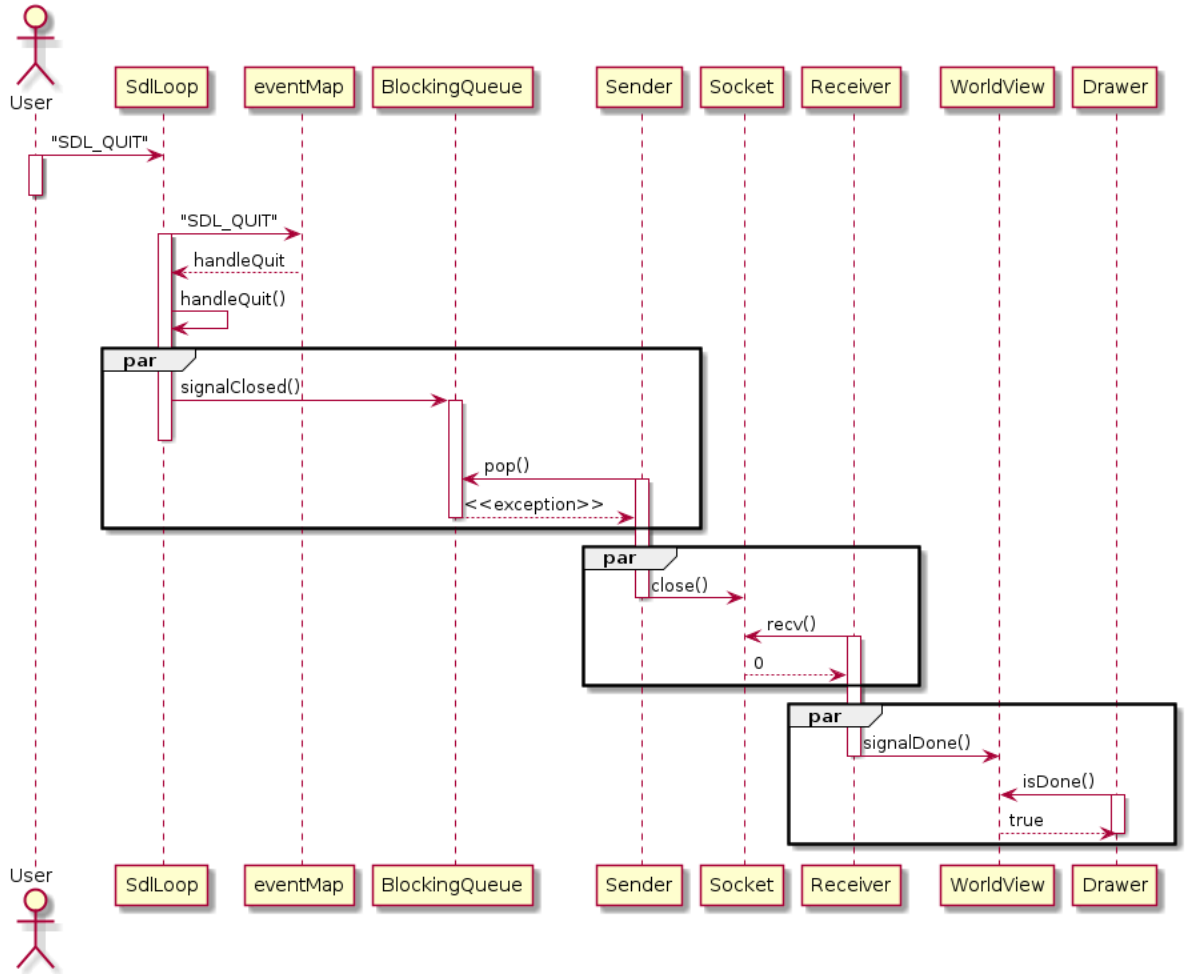

Figura 3.6: Desconexión del cliente por apretar la 'cruz'

Notamos que el usuario al tratar de cerrar al cliente, emite un evento del tipo 'SDL_QUIT', el cual es recibido por SdlLoop, encargado de cerrar la cola de comandos. El hilo Sender, bloqueado en el método pop(), recibe la señal de cierre, que hace que salga de su loop y cierre la conexión del socket. De manera concurrente, el hilo Receiver está bloqueado en recv(), y al cerrar el socket está función devuelve 0, señal que es captada por el Receiver como aviso de cierre. Sale del loop y le avisa al WorldView que ya no hay más updates para recibir. Esto triggerea al Drawer, que esta pendiente de que sigan habiendo updates para continuar dibujando. Todos los hilos hasta este punto están listos para ser joineados por main.

3.3.2. Desconexión del Cliente causado por cierre del Server

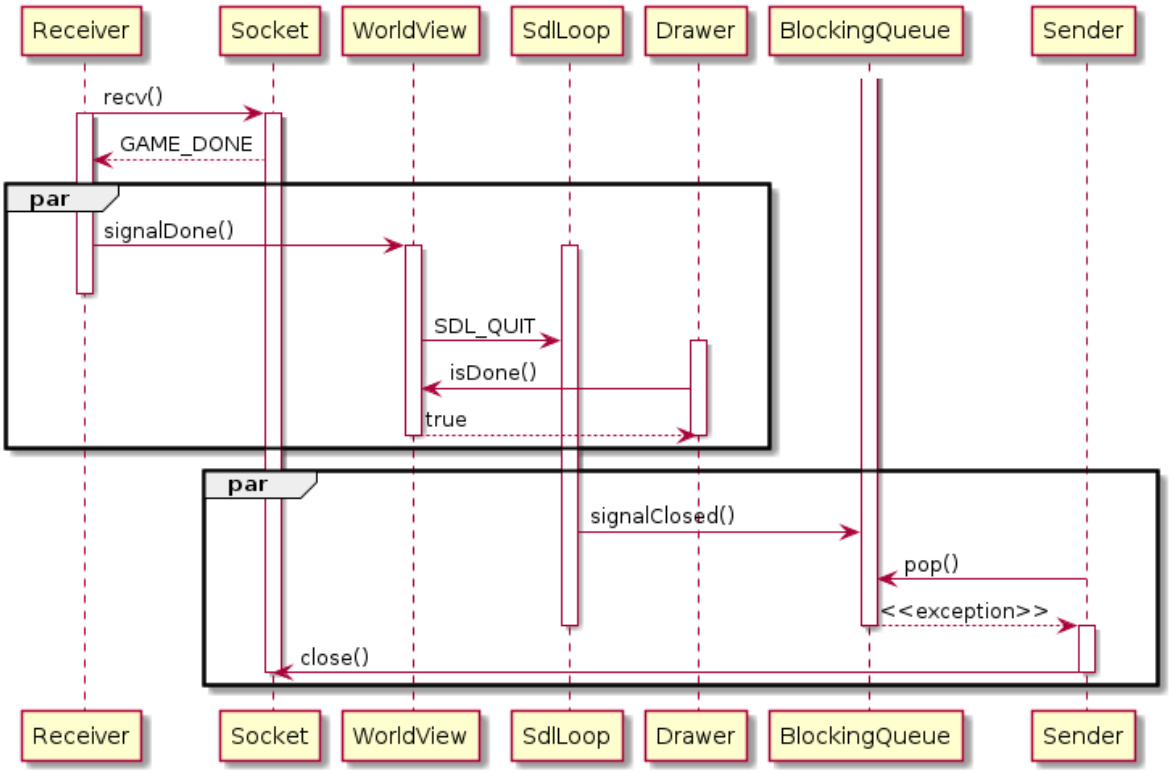

Figura 3.7: Desconexión del cliente por señal del servidor

La situación es similar, pero el primero en enterarse del cierre es el Receiver, el cual le indica a WorldView que se terminó el juego. De ahí, WorldView le manda la señal 'SDL_QUIT' a SdlLoop, encargado de realizar su cierre y causar el del Sender. De manera concurrente, el Drawer cae por la señal de WorldView. En esta instancia, todos los hilos están listos para ser joineados por el hilo main.

3.3.3. Hud

Hay información relevante para el jugador que debe ser mostrada en pantalla, como la cantidad de balas, los puntos de vida actuales, el dinero que posee. Estos elementos son traducidos de 'números lógicos' a 'números visuales' por instancias de la clase Hud. El juego original contiene la siguiente imagen de la cual se pueden extraer estos números.

Figura 3.8: Números del cliente

Viendo la imagen, se puede pensar que si todos los símbolos tienen el mismo tamaño, se podría dividir el ancho de la imagen por la cantidad de símbolos para acceder a cada número a través de un índice (el mismo número) y el offset de tamaño en anchura. No es el caso para todos los símbolos, pero los números si tienen la misma anchura. Con esto podemos acceder a cada número a partir de un índice que coincide con el número visual que queremos dibujar y el offset en su anchura. Esto es muy relevante por ejemplo para los puntos de vida. Esta conversión la logra el método loadNumberVector() de Hud, el cual itera por todos los dígitos de un número y crea areas correspondientes a esta imagen para luego mostrar en pantalla.

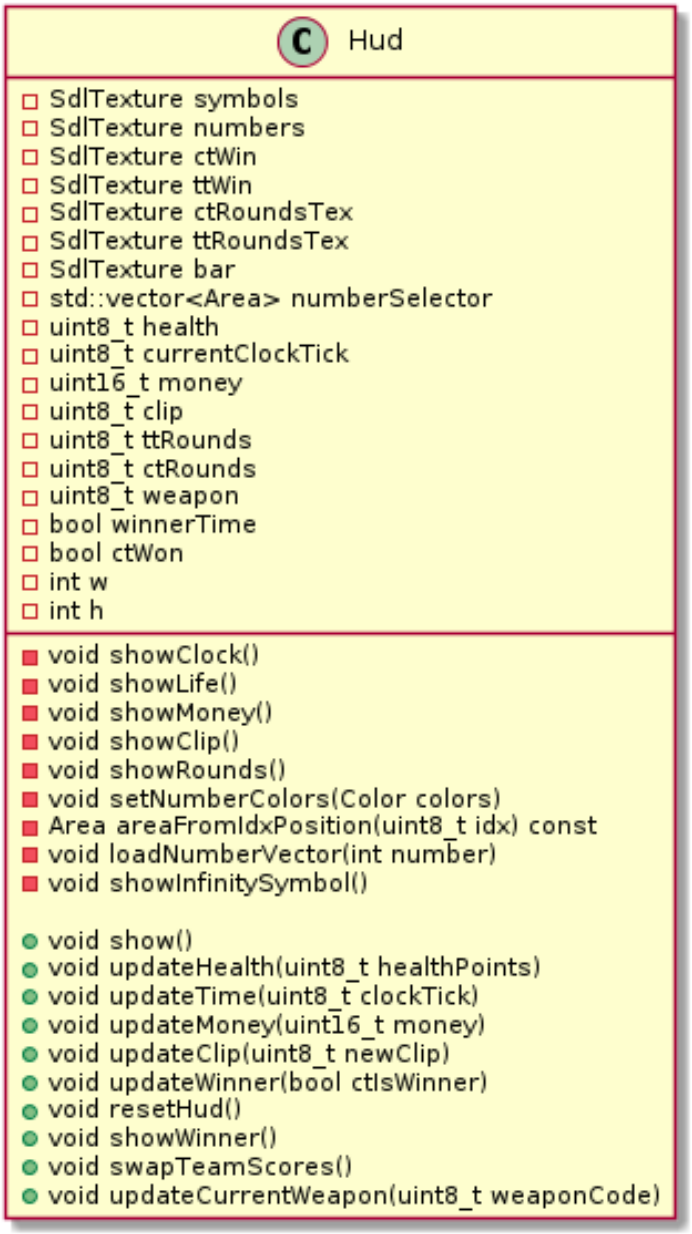

Figura 3.9: Hud class diagram

Si bien la clase es grande, el funcionamiento es fundamentalmente el mismo para todos los

métodos. Se obtiene el número correspondiente (este se recibe asincrónicamente en el cliente, enviado por el servidor), se lo mapea a una sección particular de la imagen de números y se lo dibuja en otra sección de la pantalla. La diferencia fundamental es en qué lugar de la pantalla se dibuja el número y que símbolo, si es que existe alguno, lo acompaña. La interfaz nos permite updatear los elementos visuales y mostrar al hud, escondiendo la lógica detrás.

3.4. Descripción de archivos y protocolos

Se usa para el cliente un archivo de configuración YAML en el cual se encuentran paths a archivos y texturas utilizadas en el juego, como también se encuentra la configuración del stencil. Se puede cambiar la resolución, setear el juego en fullscreen, activar o desactivar los sonidos y cambiar el tipo de cursor que uno quiere utilizar.

Para el protocolo de comunicaciones, ver la sección 6.

4. Servidor

4.1. Descripción general

El servidor, siendo una aplicación distribuida y concurrente, se encarga de recibir a nuevos clientes y mantener y resolver la lógica de todas las partidas que se estén jugando hasta ese momento. También, levanta la información sobre todos los mapas disponibles para crear el mundo de una partida y se las envía a los clientes para que estos puedan dibujarlo.

4.2. Clases

- Server: inicializa al server con el archivo de configuracion que se recibe por argv, inicializa al aceptador de clientes y al monitor de partidas.
- ThAcceptor: recibe puerto para bindear al socket, loopea aceptando a los nuevos clientes que se conectan al servidor y para cada uno, lanza un hilo ThLogin. Al cerrar el server, se encarga de ir eliminando los hilos que estén idle y los que estén muertos.
- ThLogin: escucha comandos básicos del cliente, como crear partida, unirse a una partida, listar mapas, listar partidas. Ante cualquiera de estos, manda una respuesta adecuada. Es el encargado de llamar a los métodos del monitor de partidas correspondientes para despachar los mensajes de cada cliente. Una vez que el login termina, sea porque el cliente creo una partida o se unió a una, deja de correr y es joineado por el ThAcceptor.
- GamesMonitor: es el monitor de las partidas o matches. Contiene el mapa con el nombre de la partida, crea las partidas, une a usuarios a otras partidas, las para cuando el server se cierra.
- Match: es creado por el monitor de partidas, es la entidad que contiene toda la información necesaria para la partida, los Users, el WorldModel, se encarga de iniciar las partidas.
- User: contiene la informacion relevante de cada usuario, como su ID, el socket que fue entregado por ThLogin, los hilos ThReceiver y ThSender, e inicia a los hilos cuando empieza la partida.
- ClientEvent: representa un comando del cliente del lado del servidor. Es una jerarquía que implementa un método abstracto que recibe una instacia de WorldModel. Cada vez que se llama a este método, se realiza un update sobre la lógica del juego si este es válido y no se hace nada si no lo es.
- Update: jerarquía de clases que serializan los updates de la lógica del juego. Cada Update, mediante el protocolo, se convierte en bytestream para luego ser deserializado en el cliente y actualizar elementos visuales/sonoros. UpdateManager se encarga de generar y enviar las distintas updates.

- ThReceiver: recibe todos los comandos de cliente y los despacha en eventos. Contiene una referencia a una instancia de ProtectedQueue, cola no bloqueante de eventos que le pertenece al WorldModel. Si el usuario ingresó un comando válido, este se despacha del lado del server como un evento que se pushea a la ProtectedQueue, para que luego el mundo popee de ella y lleve a cabo la acción a la cual se traduce ese evento. Como ThReceiver recibe el id de usuario, cada evento contiene el identificador de quien llevo a cabo ese comando.

- Broadcaster: Contiene BlockingQueues de Updates, una por cada usuario. Como no necesariamente todos los updates deben ser reflejados en todos los usuarios (ejemplo, la data del hud de un user solo le debe llegar a el, no al resto), implementa el metodo push, que recibe un id en particular, y el metodo pushAll, que pushea el update a cada BlockingQueue.

- ThSender: tiene una referencia a una BlockingQueue (la misma que fue creada por el broadcaster para el usuario en cuestion). Cada ThSender popea de su BlockingQueue, le ordena a los updates que se serialicen y los envía mediante socket al cliente.

- WorldModel: hilo principal que se encarga de orquestar las distintas partes necesarias para el funcionamiento de una partida. Contiene al Broadcaster; la ProtectedQueue para los eventos de usuario; el mapa con todos los PlayerModels; el mundo de Box2d y varios atributos mas. Funciona como un estilo de patrón Facade. Ejecuta los eventos recibidos por los usuarios y genera los updates que serán enviados, además de simular el flujo del juego.

- PlayerModel: modelo del jugador dentro de la partida, se encarga del comportamiento del mismo. Contiene una referencia del cuerpo Box2d que vive en el mundo, su vida, velocidad, dinero y arsenal.

- Armory: arsenal propio de cada jugador, maneja las distintas armas que tiene disponibles, las equipa y verifica si puede comprar nuevas.

- Weapons: clase abstracta que determina el comportamiento de las distintas armas. Todas comparten las mismas características generales, pero tienen ligeras diferencias en sus modo de uso.

- MapLayout: controla la creación de objetos dentro del mundo de Box2d, se guarda la posición de las paredes para poder calcular que la trayectoria de las balas no las atraviese. Determina la posición donde los jugadores aparecen y si se encuentran dentro de la zona para plantar la bomba.

- Tally: lleva cuenta de las estadísticas generales de la partida. Determina quien gano la ronda, cuantas muertes hubo, cuantas kills, el tiempo transcurrido.

4.3. Diagramas

Mostramos el diagrama de clases correspondiente a la sección de login.

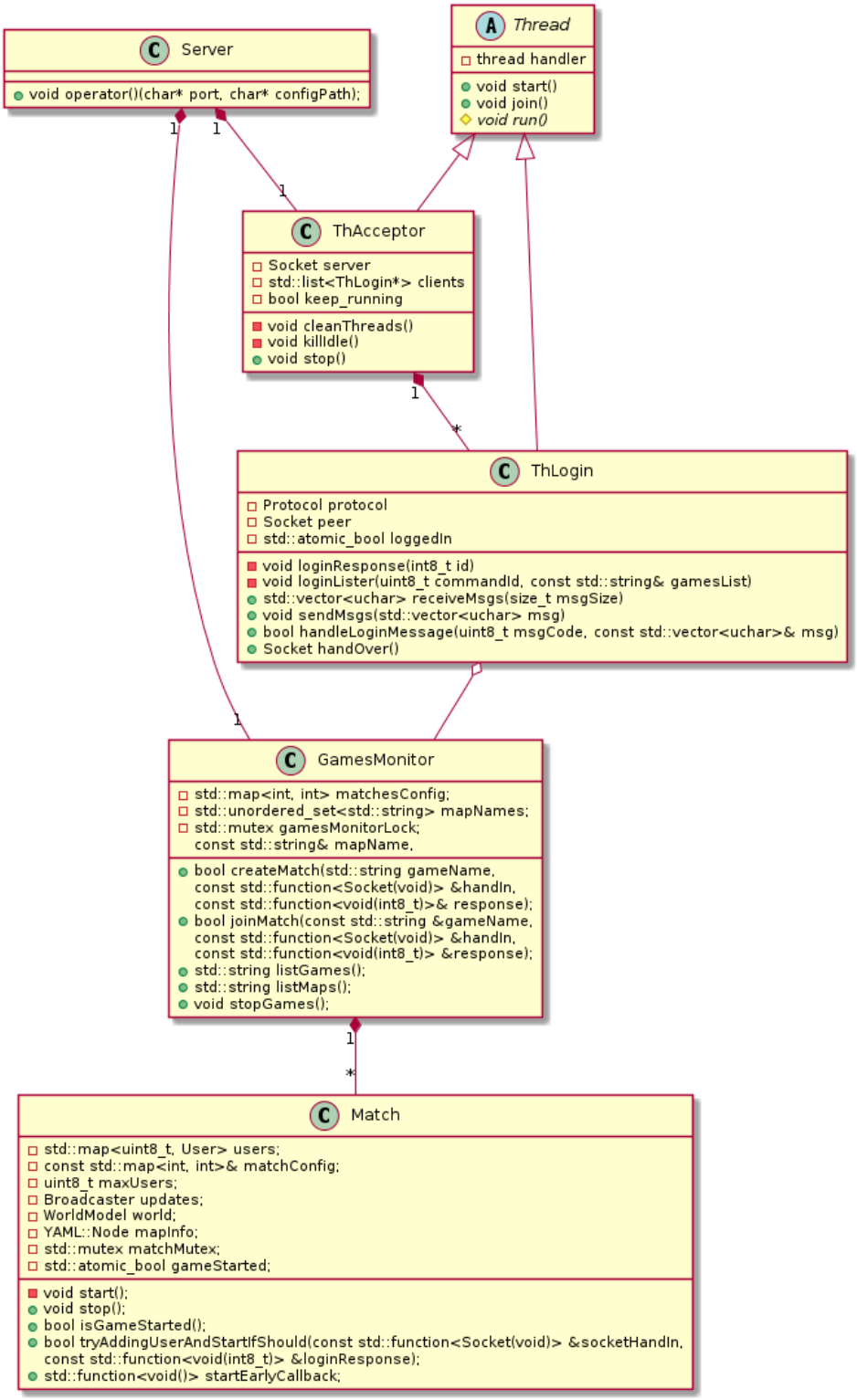

Figura 4.1: Server Login

Vemos a grandes rasgos como es el sistema. Server es dueño del aceptador y del monitor de juegos. El aceptador lanza a los hilos de login, los cuales reciben del cliente comandos que

se despachan en métodos del monitor de juegos, ya sea crear, unirse a partidas, listar mapas y nombres. Como cada ThLogin desaparece luego de crear o unirse a una partida, debe ceder el socket que usó para comunicarse con el cliente. Esto se logra a través del método handOver() de ThLogin, el cual es invocado dentro de gamesMonitor mediante un callback una vez que se logró crear la partida. Se utiliza un callback y no se pasa directamente el Socket porque puede que no se pueda crear o unirse a una partida, ya sea porque la misma existe o porque la partida esta llena. Vemos este mecanismo en el siguiente diagrama de secuencia.

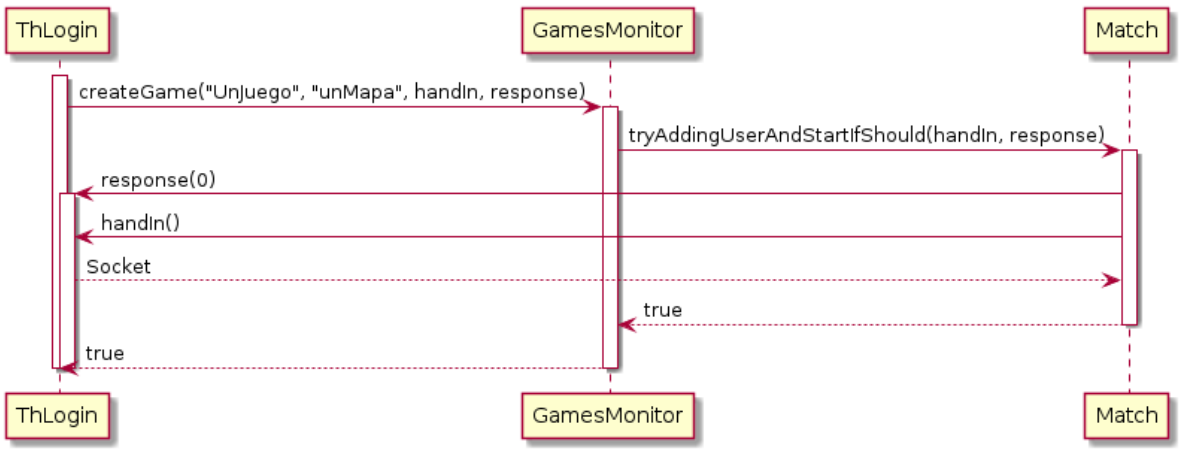

Figura 4.2: Create game sequence

ThLogin recibe un comando de creación de partida por parte del cliente, con nombre 'UnJuego' y eligiendo el mapa 'unMapa'. Ante esto, ThLogin invoca al método createGame de GamesMonitor, pasandole los datos de la partida y los callbacks necesarios para responderle al cliente y para entregar el socket. De poder crear la partida, el callback handIn mueve el socket del login hacia el Match (se lo entrega al usuario creado para este cliente) y luego devuelve true, indicándole al ThLogin que terminó la recepción y que ya esta listo para ser joineado.

Los clientes también tienen la opcion de empezar un juego con menos jugadores de los que una partida requiere, esto requiere cierto soporte por parte del Server. Como cada ThLogin es joineado luego de terminar la recepción, esto es, crear o unirse a una partida, este handle debe ser manejado en alguna estructura dentro del monitor de juegos y por otros hilos. Los encargados de este manejo son ThSender y ThReceiver, hilos que se lanzan ni bien un usuario logra crear o unirse. Si un cliente toca el boton de comienzo temprano, se debe dar comienzo a la partida, es decir, lanzar los hilos correspondientes a la partida y avisar al resto de los usuarios que se comenzó. Vemos la clase User.

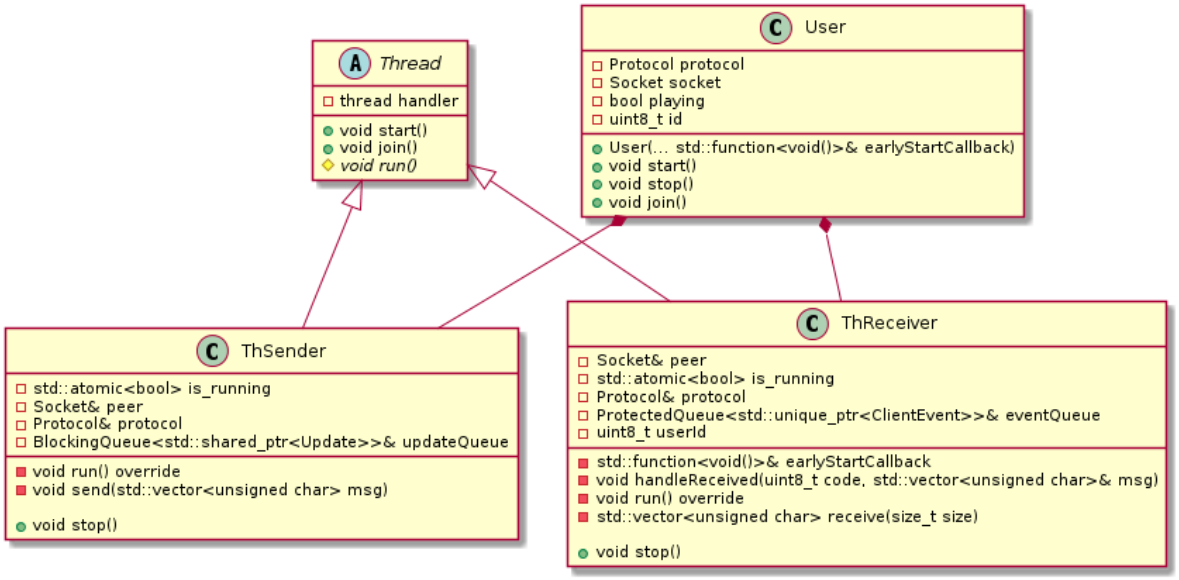

Figura 4.3: User class diagram

Cada instancia de User es creada por una instancia de Match, la cual tiene al earlyStart-Callback. Este método es forwardeado por cada User a su propio ThReceiver, el cual puede recibir por parte del cliente un mensaje de Early Start. Este mensaje luego se despacha en este callback, el cual termina por levantar el hilo WorldModel e inicia la partida. Vemos este mecanismo en un diagrama de secuencia.

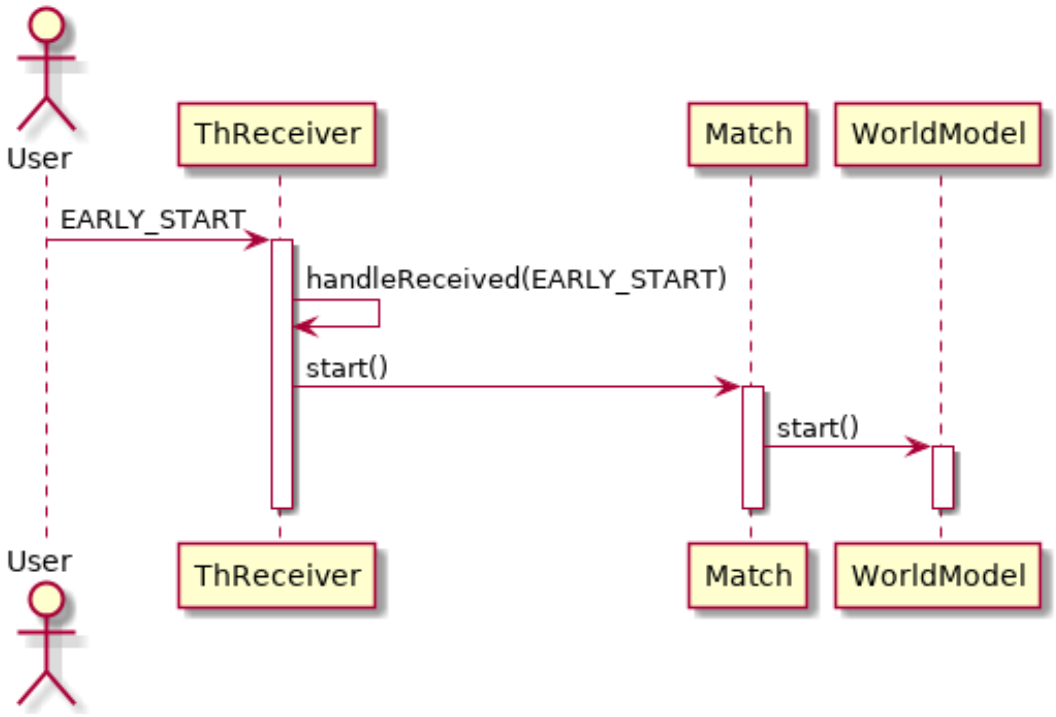

Figura 4.4: Early start

Una vez que se inicia la partida comienza el ciclo del juego. Este esta encapsulado por el hilo WorldModel que se encarga del funcionamiento de la misma. WorldModel contiene todos las partes necesarias para que podamos jugar. Tiene al mundo de Box2d, los modelos de cada jugador dentro del mundo, lleva cuenta del tiempo, entre otras cosas.

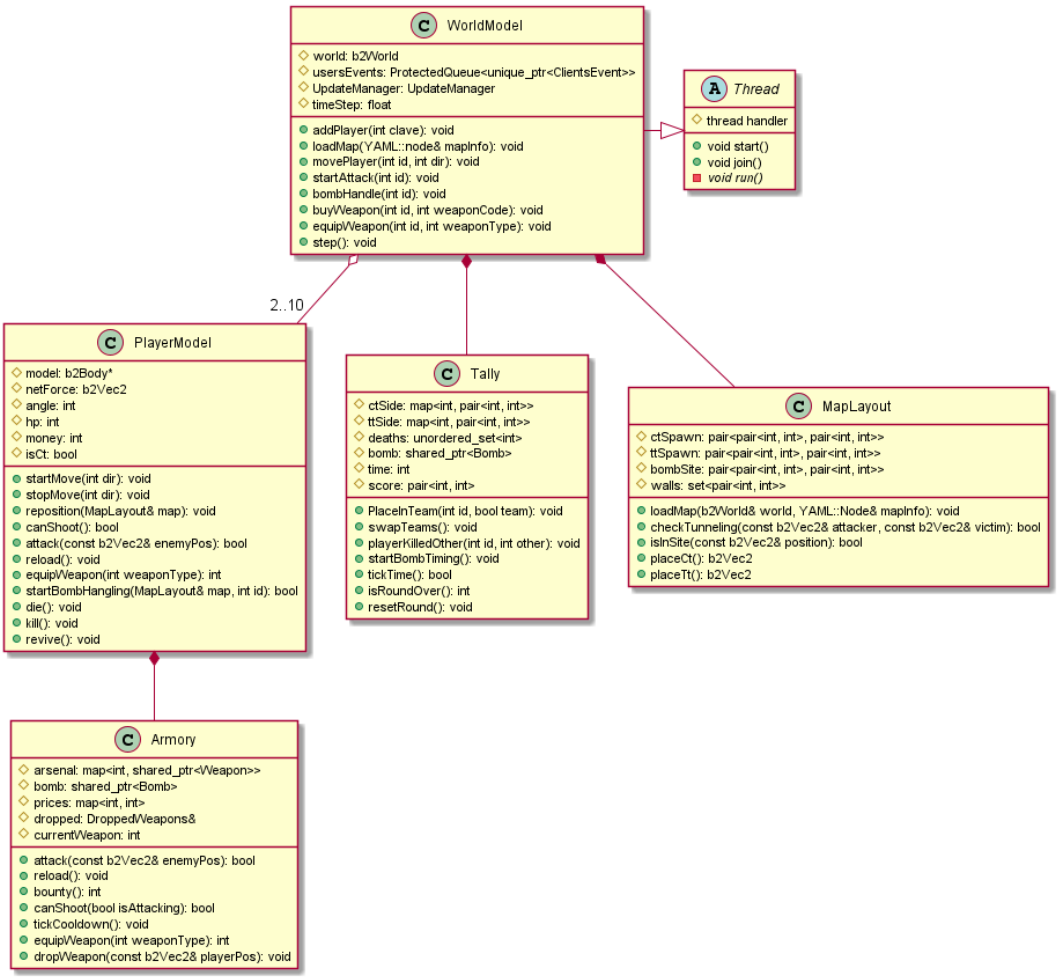

Figura 4.5: Diagrama de clases de los componentes principales de una partida

En el siguiente diagrama podemos ver como es el flujo general de una partida. Una ves que ingresan todos los participantes, o se manda un mensaje de EarlyStart, se le da inicio a la partida de WorldModel y se levanta ese hilo. Este comienza con el set-up inicial de la partida, haciendo la división de los equipos y enviando las updates necesarias para que se muestren los elementos del HUD, como el tiempo, el dinero y la vida. Luego comienza el ciclo principal de la partida, que se repite 10 veces, 1 para cada ronda. Se le da la bomba a algún terrorista al azar, comienza el periodo de compra, durante el cual los jugadores no pueden moverse, pero pueden comprar las armas que desean. Y luego de 10 segundos, comienza la ronda en si. Esta dura hasta que algún equipo gane, ya sea porque mataron al equipo enemigo o exploto/desactivaron la bomba.

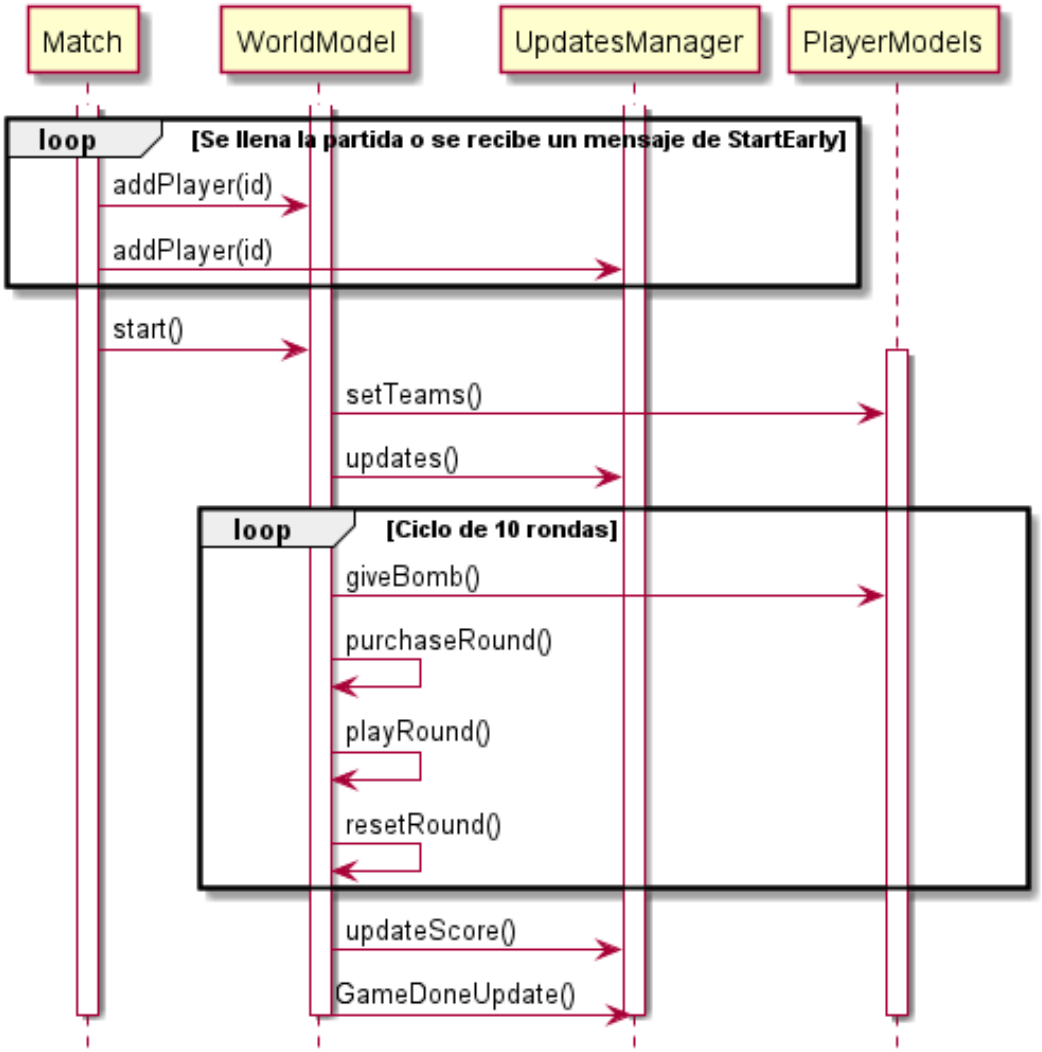

Figura 4.6: Diagrama de secuencia del ciclo principal de una partida

Una ves que termina la partida, se envía el puntaje final de cada jugador, y luego se cierran las comunicaciones con los clientes.

Una ronda consta de 3 procesos. La extracción y ejecución de los eventos de la cola de eventos, la creación de las updates pertinentes para avisarle a los jugadores que paso y de la simulación del mundo del juego. Estos procesos ser repiten hasta que la ronda termine. Cuando comienza el ciclo se extraen 50 eventos de la cola y se ejecutan 1 por 1, estos pueden ser por ejemplo el movimiento de algún jugador, que un jugador haya comenzado un ataque o que este intentando plantar la bomba. Dependiendo del resultado de estos eventos se generan las updates que se necesitan. Si algún jugador fue impactado por una bala, se envía un update de Hit, y si murió a causa del impacto, se envía un update de Death también. Al terminar cada ciclo de estos, se espera un tiempo para que se simule el transcurso del tiempo a escala real.

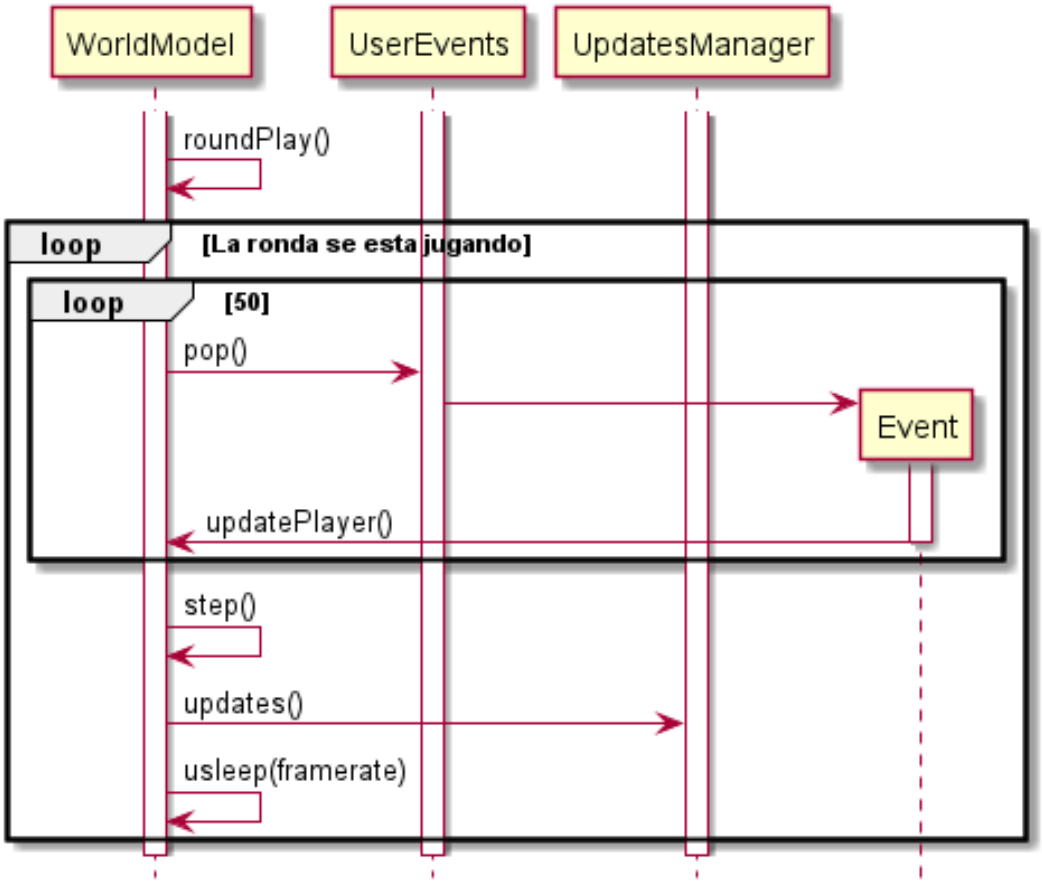

Figura 4.7: Diagrama de secuencia del ciclo de una ronda

Una ves que termina la ronda, se resetean los parámetros pertinentes para que se pueda iniciar de vuelta.

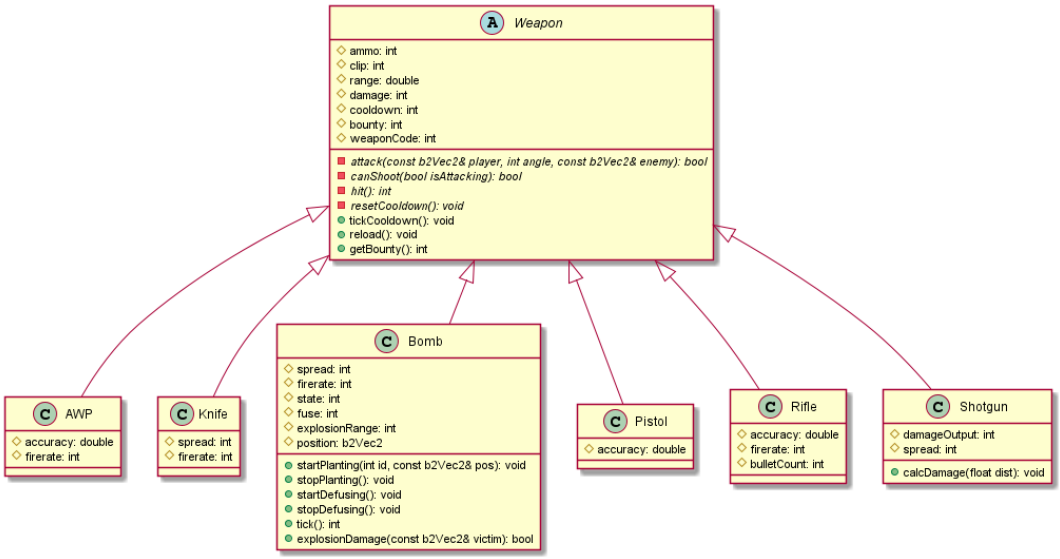

Figura 4.8: Jerarquía de las armas

Todas las armas tienen un comportamiento similar, pero cada una tiene sus particularidades. Todas tienen un cooldown que determina cada cuanto pueden disparar. En el caso del cuchillo o la pistola, estos solo pueden disparar 1 ves por click del mouse, por lo tanto el cooldown se setea a infinito después de cada disparo, y hasta que no se suelta el click no se vuelve a refrescar. El rifle dispara 3 veces por cada click, por lo tanto necesita un contador para determinar cuantas veces se disparo. La shotgun varia su daño dependiendo de la distancia que se encuentra la víctima, entonces el calculo del daño final es distinto que el resto de las armas. Y por ultimo tenemos la bomba, que tiene un comportamiento extra. Además de poder golpear con la misma, esta se puede plantar. Solamente los terroristas pueden equiparse la misma, y solo se puede plantar si se encuentran el el sitio indicado.

4.4. Descripción de archivos y protocolos

El servidor usa un archivo de configuración yaml para los valores constantes como la vida, las armas, las balas, etc. Este se describe paso por paso en el manual de usuario, donde se explicitan todas las entradas. Para el protocolo de comunicaciones, ver la sección 6.

5. Editor

5.1. Descripción general

El editor es un programa separado. No necesita conectarse con el cliente ni con el servidor. A través de éste, se pueden generar los mapas que luego el servidor va a cargar, eligiendo los elementos visuales que el cliente dibujará y objetos que serán parte de la lógica del juego, como armas, zonas de spawn de jugadores, etc, que el servidor deberá tener en cuenta para armar el mundo de una partida.

Para el desarrollo del editor se utilizó QT como herramienta para implementar la interfaz gráfica del programa.

5.2. Clases

El programa se dividió en cuatro ventanas gráficas principales con sus correspondientes clases llamadas: IntroWindow, MapConfigWindow, MapCreationWindow y MapEditor. Estas últimas clases heredan de la clase QTile, clase correspondiente a la librería QT.

5.2.1. IntroWindow

La ventana gráfica implementada en la clase IntroWindow consiste en ser una ventana que simplemente reproduce un video mp4 que contiene el nombre del juego a forma de introducción al programa.

5.2.2. MapConfigWindow

La ventana gráfica implementada en la clase MapConfigWindow es la ventana principal del editor donde se elige si se quiere editar un mapa, crear un mapa nuevo o salir del programa.

En la ventana se puede observar una lista de nombres de mapas ya creados que se pueden editar y tres botones.

Cada botón desencadena distintos eventos al ser clickeados:

- botón con etiqueta 'Edit': chequea que se haya seleccionado algún mapa de la lista de mapas y, de ser así, cierra momentáneamente la ventana actual y abre una ventana del tipo MapEditor indicándole el nombre del mapa que se desea editar.
- botón con etiqueta 'Create': cierra la momentáneamente la ventana actual y ejecuta una ventana del tipo MapCreationWindow.
- botón con etiqueta 'Quit', cierra definitivamente la ventana actual.

La lista de mapas mostrados en pantalla se implementaron haciendo uso de la clase 'QDir'. Esta herramienta provista por 'Qt' permite obtener los nombres de los archivos en ciento directorio local en formato cadena de manera dinámica. Por lo tanto, la creación de un mapa implica la creación de un archivo nuevo en un directorio local específico y la edición de un mapa implica la sobre-escritura de un archivo en un directorio local específico.

La clase de 'Qt' utilizada para representar gráficamente a la lista de mapas que se pueden editar fue 'QtListWidget'. Se creó la clase una clase hija llamada QEditorMapListWidget' tal que se le agrega las funcionalidades de agregar y actualizar nombres de mapas a la lista.

5.2.3. MapCreationWindow

La clase MapCreationWindow es representa la ventana gráfica donde se ingresa el nombre del mapa a crear y el tamaño del mapa.

Por ende, posee un campo donde ingresar el nombre del mapa, una lista de tamaños para el mapa y un botón. El botón tiene como etiqueta 'Save' y al ser clickeado chequea que se haya completado el nombre del mapa, cierra la ventana actual y abre la ventana MapEditor.

5.2.4. MapEditor

La ventana MapEditor es la encargada en crear un mapa nuevo y editar un mapa pre-existente.

Se enumeran algunas de las clases principales utilizadas para implementar MapEditor:

- QEditorItemsWidget: esta clase hereda de la clase de la librería Qt 'QTreeWidget'. Consiste en una lista desplegable con u distintos 'backgrounds' e 'items' que se pueden configurar en el mapa. Cuando se hace click sobre algún 'QTreeWidgetItem', o bien alguno de los elementos de la lista desplegable, es capaz de emitir una señal a otro objeto que herede de 'QWidget' indicando el nombre del elemento clickeado.

- QEditorMapWidget: es la clase encargada de visualizar el mapa y reaccionar de acuerdo a los eventos que ocurren al interactuar en ella como cuando se le hace un 'click' o un 'drag and drop'. Esta clase lee los archivos de los mapas y guarda las modificaciones correspondientes en estos.

  Consiste principalmente en tener un 'QGridLayout' que contiene muchas instancias de 'QTile', clase que hereda de QLabel, con una separación entre elementos nula tal que estos objetos individuales aparentan estar unidos formando un mapa.

  En la clase MapEditor, se realiza una 'conexión' entre una instancia de tipo QEditor-MapWidget y QEditorItemsWidget tal que cada vez que se realice un click sobre un item de QEditorItemsWidget, QEditorMapWidget se entere. De esta manera, sabe que elementos agregar en el posible caso que se clickee sobre el mapa en una determinada posición.

- QTile: clase hija de la clase 'QLabel' de la librería Qt que representa un 'Tile' o posición en el mapa con sus respectivas imágenes.

- ZoneValidator: Clase que valida si las tres zonas requeridas en todos los mapas han sido configurados y si las zonas colocadas cumplen en formato predeterminado (que sean zonas rectangulares).

- Elements: clase que contiene una lista de elementos posibles a agregar en el mapa y sus sub-categorías.

- MapQPixmapGenerator: clase que carga todas las imágenes requeridas en la edición y/o creación de mapas cuando es creada y al que se le puede solicitar una copia de esta imagen tan solo al solicitar el nombre de tal imagen. Hay una instancia de estapor cada instancia del MapEditor.

5.3. Diagramas

En el siguiente diagrama uml de estados se observa las diferentes interacciones entre las cuatro ventanas del editor.

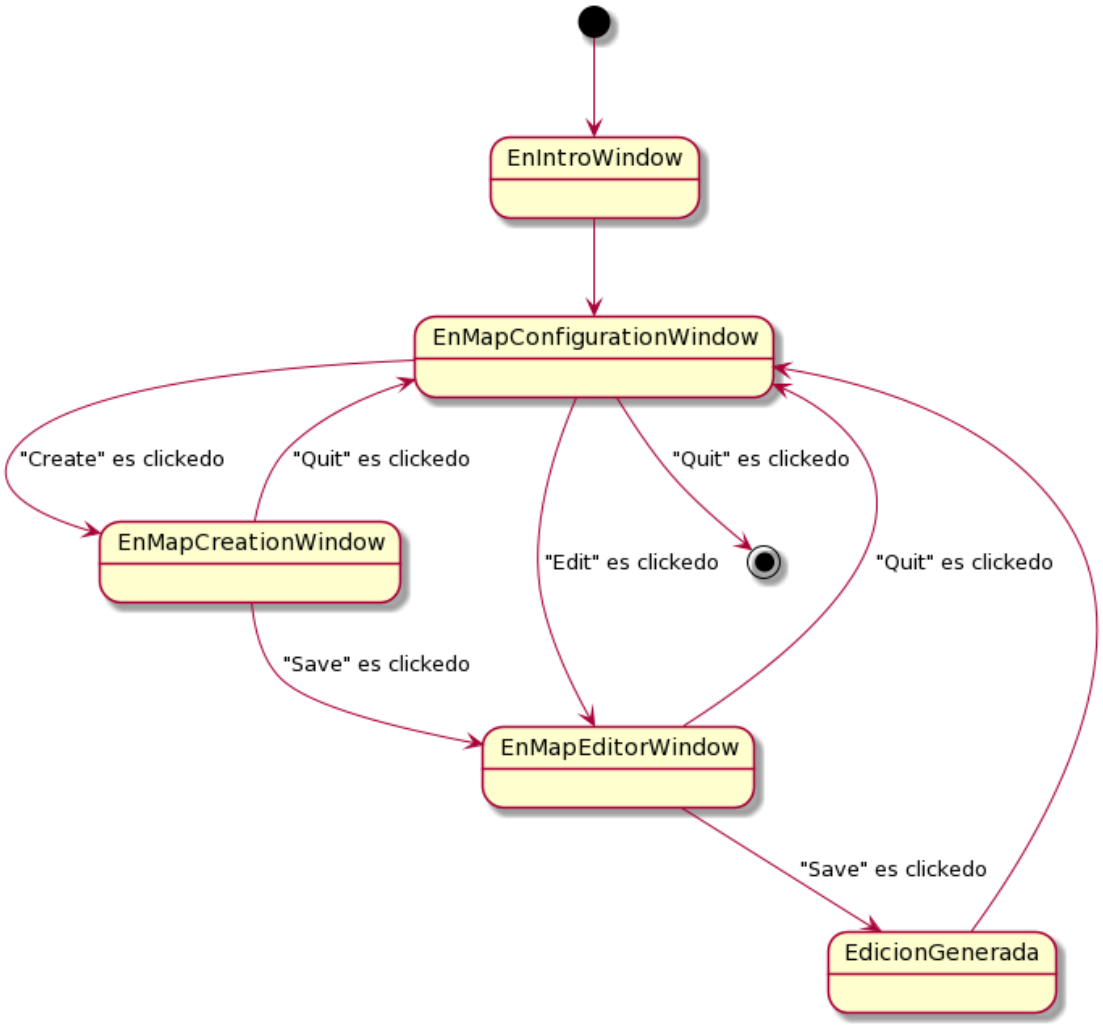

Figura 5.1: Interacción entre las ventanas del editor

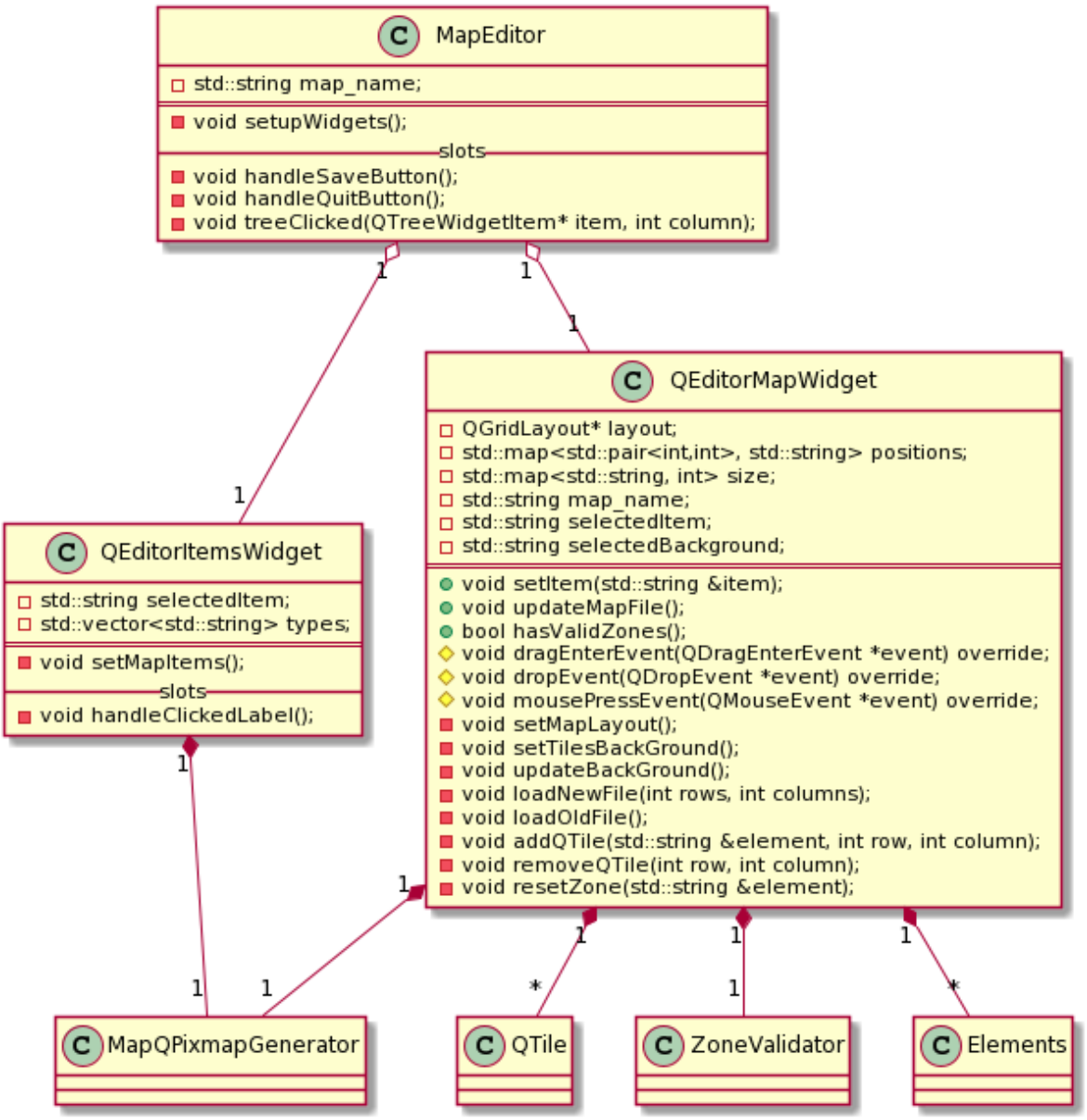

Figura 5.2: Diagrama de clases que ilustra las clases de las cuales depende la clase MapEditor

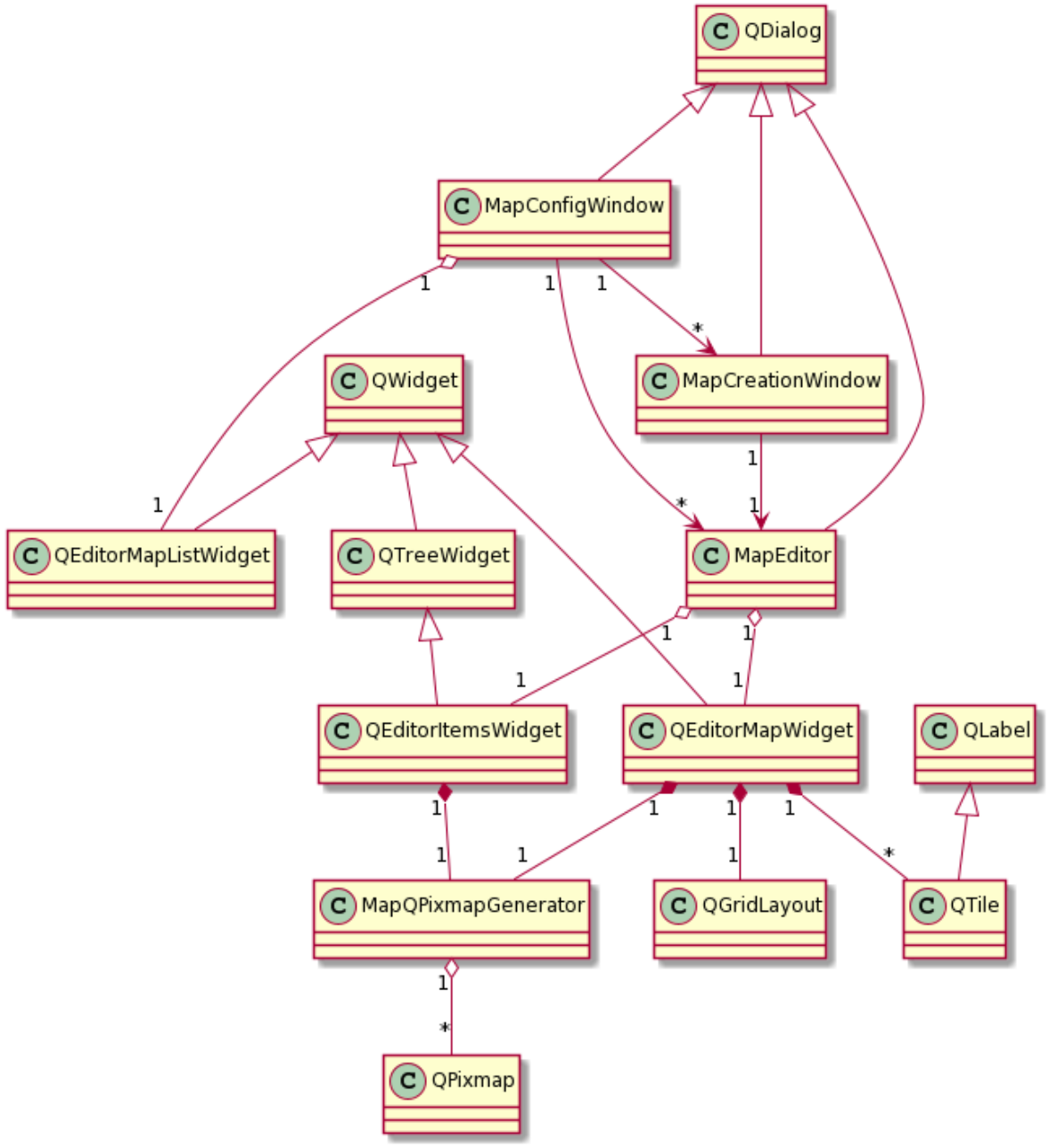

Figura 5.3: Diagrama de clases que ilustra las principales dependencias de las clases del editor con clases de Qt

6. Protocolo

El protocolo es una sección de código que utilizan el servidor y el cliente para llevar a cabo su comunicación. La manera en la cual se independiza al protocolo del resto del código es mediante el uso de callbacks. Éste no necesita conocer al cliente o al servidor, más bien necesita un método al cual llamar para enviar o recibir mensajes, que puede pertenecer a cualquiera de los dos programas. Este método es algún wrapper de las funciones send y recv. El protocolo provee una interfaz para despachar cualquier mensaje que se quiera enviar o recibir, ya sean comandos, updates o eventos, y además provee una manera de serializar y deserializar mensajes para que el cliente y el servidor no tengan que manejar la raw data que viaja por los sockets para entender el mensaje a recibir. Se vio que la manera que tienen ambos programas de enviar mensajes o recibirlos es mediante los comandos, los eventos y los updates. Las instancias de estas clases son las cuales se comunican con el protocolo, saben en que método despachar, y además forwardean el callback necesario dependiendo de lo que se quiera hacer. Por ejemplo, los eventos son respuestas ante comandos, pero para saber cual es el evento en el que hay que despachar, primero se debe saber cual es el mensaje, por lo cual se debe forwardear al protocolo el método wrapper de recv para que este módulo se encargue de esa lógica. Esto desacopla de la lógica del manejo de bits y bytes, lógica complicada y de muy bajo nivel para mezclar con la lógica del juego o la del dibujado, del servidor y del cliente. Ya que esta clase contiene muchos métodos, se recorta para hacer un seguimiento de su diagrama de clase.

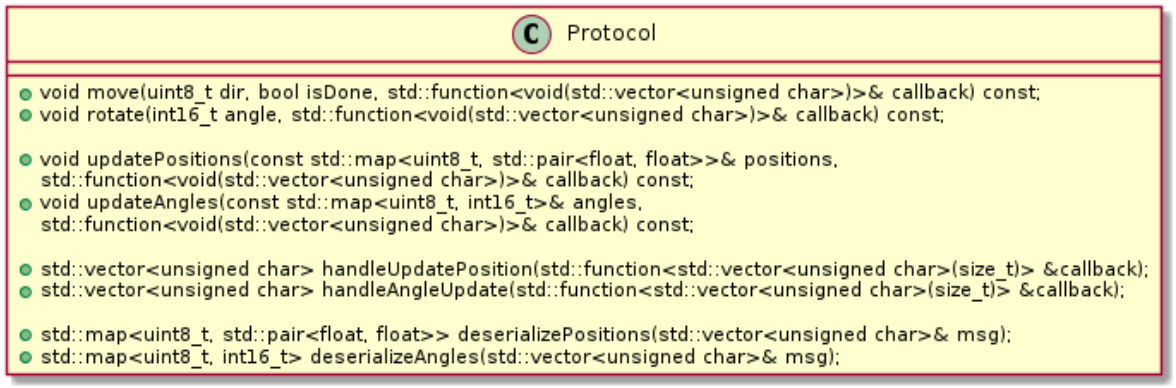

Figura 6.1: Diagrama del protocolo redux

Primero que nada, vemos el uso de std::function como wrapper de los objetos callables, en este caso, métodos de instancias. Hay dos tipos de callback en el protocolo:

```
std::function<void(std::vector<unsigned char>)>&
std::function<std::vector<unsigned char>(size_t)>&
```

El primero corresponde al wrapper de send() y el segundo al de recv(). El servidor y el cliente pueden mandar los wrappers correspondientes al protocolo para que este los llame y se logre enviar o recibir los datos que correspondan. En el diagrama hay 4 pares de métodos, correspondientes a las acciones de moverse o de rotar. El primer par es llamado por cada cliente, recibe una direccion (mapeada a una de las teclas de movimiento direccional), un boo-

leano indicando si el movimiento empezó o terminó, y un callback wrapper de send. Vemos su código:

```
void Protocol::move(uint8_t dir,
                    bool isDone,
                    std::function<
                    void(std::vector<uint8_t>)
                    >& callback)
                    const {
    std::vector<unsigned char> msg;
    msg.push_back(isDone ? STOP_MOVE : MOVE);
    msg.push_back(dir);
    callback(std::move(msg));
}
```

Lo que se hace es crear un vector de bytes al cual se le van insertando los códigos del mensaje y el payload correspondiente. En este caso, dependiendo de si el movimiento empieza o termina se pushea alguna de las macros en mayúscula, correspondientes al código del mensaje. Luego se pushea la dirección y finalmente se llama al callback para enviar el mensaje. Este mismo mecanismo sucede en mayor o menor medida con el resto de los comandos del cliente.

Cada cierto tiempo, el servidor hará un issue de un update del tipo UpdatePositions para enviar las posiciones de todos los jugadores a todos los clientes, de tal manera de poder dibujarlos. Para esto, el update llamará al método updatePositions() del protocolo.

```
void Protocol::updatePositions(const std::map<...>& positions) const {
    std::vector<unsigned char> msg;
    msg.push_back(POS_UPDATE);
    uint16_t msgSize = positions.size() * 9;
    msgSize = htons(msgSize);
    serializeMsgLenShort(msg, msgSize);
    for (auto& pair : positions){
        msg.push_back(pair.first);
        serializePosition(msg, std::get<0>(pair.second));
        serializePosition(msg, std::get<1>(pair.second));
    }
    callback(std::move(msg));
}
```

Notamos que el código ahora tiene que levantar un mapa que le otorga el servidor, con identificadores y posiciones (en float) y transformar a una secuencia de bytes. Primero, como siempre, se pushea el código del update. Luego, como el mensaje es de tamaño variable, se pushea el largo del mismo, el cual es convertido al endianess del network (big endian). Luego, se empieza a pushear el verdadero payload. Aca se incluyen los ids (bytes simples) y luego se insertan las posiciones, las cuales primero se serializan utilizando punto fijo, esto es, se multiplica por un número constante para mantener una precisión fija. Luego de haber formado el mensaje, se llama al callback wrapper de send.

A cada recibidor de cliente le llegara el código de update de posiciones y entonces despa-

chará en el método que deserializa las posiciones. Pero primero, debemos obtener el tamaño del mensaje y luego recibir el payload.

```
std::vector<unsigned char>
Protocol::handleUpdatePosition(... &recvCallback) {
    std::vector<unsigned char> msg = recvCallback(2);
    uint16_t size = deserializeMsgLenShort(msg);
    msg = recvCallback(size);
    return msg;
}
```

Dentro de deserializeMsgLenShort se encuentra la reconversión del endianess del network al host. Se devuelve el payload que luego es forwardeado al método deserializador, el cual hace exactamente el inverso del serializador de posiciones, devolviendo un mapa similar (similar debido a la conversión de float a entero) al que inicialmente salió del server.

La idea es similar para el resto de los mensajes, ya que la lógica de los métodos depende del mensaje en si y no de quien llame.

7. Programas intermedios y de prueba

Para probar al protocolo, se creo un programa que se puede encontrar en el source del juego, 'CommandTester.cpp'. En este, fuimos creando diferentes comandos para verificar efectivamente que el payload que se generaba correspondía con lo que queríamos enviar o recibir. No utilizamos herramientas especificas de testing.

8. Código fuente

El código fuente se puede encontrar acá.