Argentum Online
Documentación Técnica

Taller de Programación I
Primer Cuatrimestre 2020

Mauro Parafati - 102749
Santiago Klein - 102192
Yuhong Huang - 102146

Índice

1. Requerimientos de software

1.1. Sistema Operativo

Se necesita un sistema operativo basado en **Debian**, como por ejemplo, *Ubuntu*.

1.2. Dependencias

Se listan las distintas dependencias necesarias:

- Son necesarias las librerías de **SDL2** *(SDL2, SDL_ image, SDL_ ttf, SDL_ mixer)*, utilizadas para la interfaz gráfica del cliente.
- Se utiliza la herramienta **CMake** para la compilación e instalación de los aplicativos.
- Es necesario contar con **gcc** para la compilación.
- Si se desea depurar el programa, se puede utilizar **gdb**.
- La librería de **json** que se utiliza se incluye en los archivos del programa, por lo que no hay que realizar ninguna instalación extra.

Todas las dependencias pueden ser instaladas mediante el *script de instalación* (**installer.sh**), seleccionando la opción 'd' o realizando una instalación completa con la opción 'a'.

2. Descripción general

El proyecto consta de dos grandes módulos: el **servidor** y el **cliente**.

El **cliente** desarrolla el *front-end* del aplicativo, manejando la interacción con el usuario y proveyendo un sistema de comunicación con el servidor de envío de comandos y recepción de notificaciones

El **servidor**, por su parte, implementa el *back-end* del aplicativo, manejando toda la lógica del juego y de sus complejos componentes, y respondiendo adecuadamente ante comandos del usuario a través de un sistema de comunicación cómodo y eficiente.

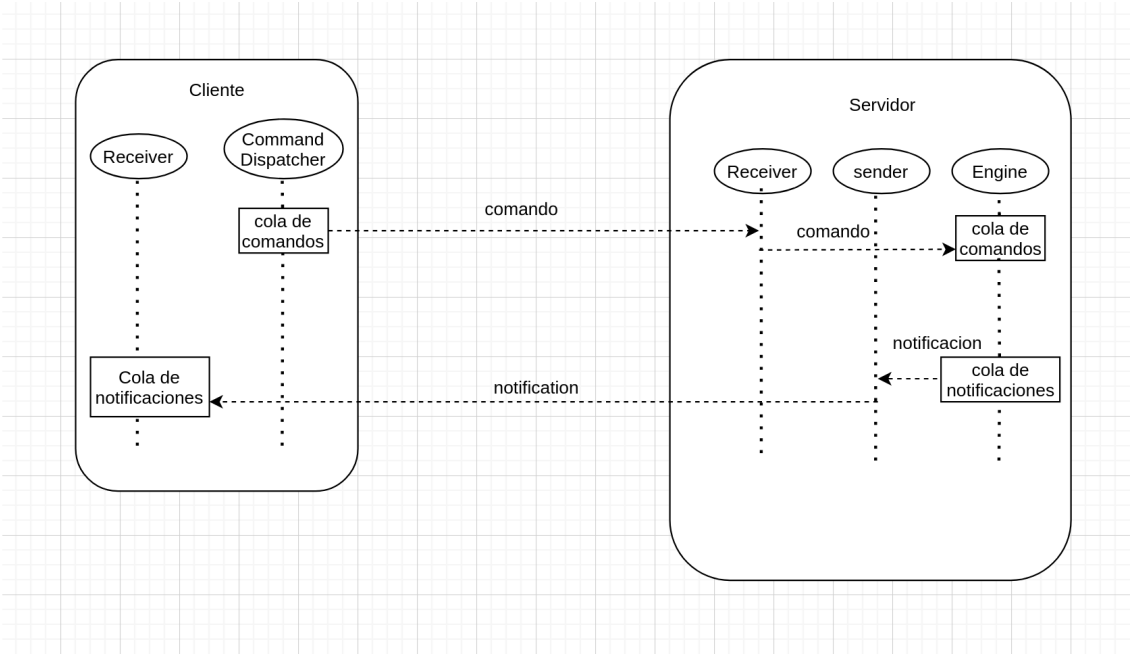

Figura 1: Flujo de información entre módulos.

A grandes rasgos, la disposición física del proyecto se organizó, de manera jerárquica, en los siguientes módulos y submódulos [todos ellos separados por *includes* —headers—, y *src* —archivos fuente—]:

- **Common:** contiene los archivos comunes al cliente y al servidor. Entre ellos, se encuentran los mapas [*assets*], el protocolo *Protocol*, la bilbioteca de json empleada, las estructuras comunes de comunicación de información, APIs esqueléticas predefinidas, y muchos otros.
- **Client:** contiene los archivos del cliente, sus principales submódulos son:
  - **GameModel:** maneja, principalmente, el envío y la recepción de información *(comandos, notificaciones)*.
  - **GameView:** desarrollo del contenido audiovisual gráfico del cliente.
  - **SDL:** abstracciones de la librería **SDL2** de renderizado y reproducción multimedia.
- **Server:** contiene los archivos del servidor, separados en *Control* y *Model*.

- **Control:** modela el funcionamiento del motor del juego y el manejo de conexiones con los clientes.
- **Model:** contiene toda la lógica del juego y sus componentes.

3. Servidor

3.1. Descripción general

En lo que respecta al servidor, existe una división primaria entre lo que es **Control** y **Model**.

Por un lado, el **control** consta de un *Engine* que, en líneas generales, se encarga de inicializar el modelo, aceptar nuevos clientes, validarlos, y comunicarse con ellos para el desenvolvimiento del juego, persistiendo periódicamente la información.

Por otra parte, el **modelo** concentra la lógica de juego per se, y en consecuencia, todas las clases que intervienen en ella, ejecutando los comandos enviados por los distintos clientes.

3.1.1. Inicialización y aceptación de clientes

Al comenzar su ejecución, el servidor ejecuta los threads del *Engine* y el *Accepter*.

El *Accepter*, por su parte, se bloquea ante la espera de nuevas conexiones. Una vez se tiene una nueva conexión entrante, lo que hace es instanciar un hilo de *ClientLogin*, el cual manejará la conexión con el cliente para su registro o logueo. Luego, al terminar su ejecución (habiendo iniciado sesión exitosamente el cliente), el *ClientLogin* pushea una *NewConnection* a la cola correspondiente.

En tanto, el *Engine* al comienzo de su loop permanente, vacía la cola de *NewConnections* y agrega al *Game* los jugadores nuevos.

3.1.2. Desarrollo del juego

Una iteración del loop del motor *Engine* consiste en:

1. **Procesar las nuevas conexiones** (vaciando la cola de *NewConnections*) y agregarlas al juego, creando para cada una el *Character* y la *ClientConnection* correspondientes.
2. **Ejecutar todos los comandos** que fueron recibidos por los *receivers* de las ClientConnections.
3. **Dar un turno para que actúen todos** los jugadores y las criaturas.
4. **Persistir periódicamente** la información del estado del juego.
5. **Actualizar los componentes dependientes del tiempo** (spawneo de criaturas, items droppeados, jugadores resucitando, etc).
6. **Procesar las conexiones finalizadas**, liberando los recursos ya no utilizados.

De esta manera, el juego se desarrolla de manera ordenada y sincronizada.

El *Game*, por su parte, responde adecuadamente a todas las órdenes dictadas por el Engine, articulando todas las componentes del juego y brindando las respuestas adecuadas ante las distintas acciones efectuadas por jugadores o criaturas, hacia los clientes.

3.1.3. Comunicación entre hilos

Para una comunación inter-hilos segura, libre de race conditions y thread-safe, se optó por utilizar colas protegidas por *mutex*, tanto bloqueantes como no bloqueantes.

1. **Cola no bloqueante de nuevas conexiones** [*NewConnection**], para la comunicacion entre el *Engine* y el *ClientLogin*. Como fue previamente explicado, una vez el usuario inicia sesión, se pushea una *NewConnection* a esta cola, que luego será popeada por el *Engine*, incorporando el cliente al *Game* y a *ActiveClients*.

2. **Cola no bloqueante de conexiones finalizadas**, indicando el *InstanceId* del cliente desconectado, común a *Engine* y *ClientConnection*. Cuando el jugador se desconecta y finalizan su ejecución ambos hilos *sender* y *receiver* de *ClientConnection*, se pushea el id de dicho cliente a esta cola. Luego, el *Engine* al final de su main loop popeará dicho id y lo eliminará del *Game* y de *ActiveClients*.

3. **Cola bloqueante de notificaciones** [*Notification**], común al hilo *sender* de ClientConnection, a través de la cual *Game*, llamando a los respectivos métodos de *ActiveClients*, pushea las notificacionse que deben ser enviadas al cliente.

4. **Cola no bloqueante de comandos** [*Command**], compartida por el *Engine* y el hilo *receiver* de *ClientConnection*, mediante la cual el hilo *receiver* recibe y crea los comandos enviados por el cliente, los pushea a ella, y el *Engine* luego los popea y ejecuta.

3.1.4. Comunicación con los clientes

Las *Notifications* proveen una forma cómoda de notificar a los distintos clientes de las acciones ocurridas en el juego. Existen 5 tipos de notificaciones, entre los cuales se encuentran:

- **Broadcast:** los hay tanto de entidades [*EntityBroadcast*] como de items [*ItemBroadcast*]. Los primeros sirven para enviar actualizaciones de estado de los Characters y *Creatures*, los segundos para los *Items* droppeados en el mapa.

  Una importante decisión de diseño fue la de optar por enviar broadcasts diferenciales, esto es, no estar broadcasteando todo el tiempo todos los datos, sino únicamente hacerlo ante cambios de estado (new, delete, update). Además, cada hilo *sender* en las *ClientConnections* filtra los broadcasts y envía únicamente los que corresponden al mapa del cliente, brindando una sutil eficiencia.

- **Reply:** sirve para enviar mensajes de respuesta ante las acciones en las que interviene un jugador. Los mismos pueden ser de éxito, error, o informativos.

- **Message:** para la implementación del chat privado y general.

- **Event:** su función es notificar al cliente que ocurrió un evento en el que intervino, o bien en sus alrededores, lo cual suele desencadenar efectos audiovisuales que enriquecen la jugabilidad.

- **List:** se utiliza como respuesta ante un comando */listar* en la interacción con un NPC, y consta de un mensaje base y una lista de strings.

3.2. Clases

3.2.1. Punto de entrada

La ejecución del **Servidor** comienza por la función *main* definida en *main.cpp*. Esta función recibe un parametro *puerto*, y su única responsabilidad es instanciar a la clase **Servidor** y comenzar su ejecución:

```
int main(int argc, char* argv[]) {
if (argc != EXPECTED_ARGC) {
    fprintf(stderr, "Usage: %s <port>\n", argv[NAME]);
```

```
        return USAGE_ERROR;
    }

    std::string port = argv[PORT];

    try {
        Server server(port, MAX_CLIENTS_QUEUED);
        server.run();

    } catch (const std::exception& e) {
        fprintf(stderr, "%s\n", e.what());
        return ERROR;
    } catch (...) {
        fprintf(stderr, "Unknown error.\n");
        return ERROR;
    }

    return SUCCESS;
}
```

3.2.2. Clases de control

A continuación se detallan las clases de control del servidor, que se encargan de manejar las conexiones entre servidor y cliente, persistir los datos, y llevar a cabo la ejecución del juego, manteniendo un control permanente sobre las acciones de los distintos clientes.

- **Servidor:** clase principal que orquesta la ejecución del servidor. Proporciona un sólo método en su API pública: *run*, encargado de comenzar los hilos del *Engine* y el *Accepter*, y su posterior joineo una vez el usuario ingresa el caracter **'q'**, finalizando la ejecución de manera ordenada.

- **Engine:** loop principal del juego. Se encarga de realizar las iteraciones controlando el manejo de frames. Procesa las nuevas conexiones encoladas en la cola de *NewConnections* (y si las hay, las agrega al juego), procesa los comandos recibidos por los distintos clientes y encolados en la cola de *Commands* y los ejecuta (llamando éstos a los métodos respectivos de la clase *Game*), provee un turno a todos los *Characters* y *Creatures*, persiste periódicamente los datos y actualiza todos los atributos dependientes del tiempo.

- **Accepter:** la función de este hilo es aceptar todas las conexiones entrantes por parte de clientes remotos, hasta que la variable atómica booleana keep accepting sea false. Por cada una de ellas, se instancia un nuevo hilo *ClientLogin* y se joinea a los que terminaron.

- **ClientLogin:** instanciado por el accepter, este hilo se encarga de llevar a cabo las acciones de **SignIn** y de **SignUp** del cliente, y una vez los datos ingresados son correctos, finaliza su ejecución y encola los datos correspondientes a la cola de *NewConnections*.

- **ClientConnection:** maneja la comunicación con el jugador activo, una vez este pasó satisfactoriamente el proceso de logueo. Cuenta con una cola bloqueante de notificaciones y una referencia hacia la cola principal de comandos del Engine. Consta de dos hilos, *Sender* y *Receiver*. . El primero se encarga de enviar las *Notifications* de la cola bloqueante hacia el cliente, y el segundo se encarga de recibir los comandos que envió el cliente, y pushearlos a la cola principal del *Engine*. Su ejecución finaliza una vez los dos hilos terminan, ya sea porque el socket remoto se cerró o bien el servidor termina su ejecución de manera ordenada.

- **ActiveClients:** básicamente es un contenedor de *ClientConnections*. Provee una interfaz cómoda para pushear las *Notifications* a las respectivas colas de cada **ClientConnection**.

- **Database:** objeto activo que valida los datos de registro o logueo de los distintos clientes, así como persiste los datos de los jugadores conectados (ya sea periódicamente o cuando se desconectan).
- **Command:** clase abstracta para la implementación de comandos polimórficos que se sepan ejecutar según el tipo. Se utilizó el patrón **Command** para su implementación, y el patrón **Factory** [*CommandFactory*] para su creación. Algunos comandos, a modo de ejemplo, son: *StartMovingUpCommand*, *ListCommand*, *SellItemCommand*, *UseWeaponCommand*, etc. Cada comando, al ejecutarse, llama al método correspondiente de la clase *Game*.
- **CommandFactory:** pseudo-implementación del *FactoryPattern* para recibir distintos tipos de comandos. Proporciona un único método: *newCommand*, que se encarga de realizar la comunicación necesaria con el cliente para recibir el comando que corresponde.
- **Notification:** clase abstracta para la implementación de notificaciones de manera polimórfica. Entre ellos, hay varios tipos: *EntityBroadcast*, *ItemBroadcast*, *Event*,*List*, *Message*, y *Reply*. En la sección *Protocolo* se puede obtener más información en lo que respecta a la comunicación entre el servidor y los distintos clientes.

3.2.3. Clases del modelo

A continuación, se explicará de la manera más concisa posible cómo es que fuasas las distintas componentes que conforman el juego.

- **Game:** su responsabilidad es articular la ejecución del *Engine* con la evolución del juego. Contiene los distintos objetos que componen el juego, estos son: los mapas [*LogicMaps*], los items - armas, báculos, defensas, pociones, etc. - [*ItemsContainer*], los jugadores [*Character*], las criaturas [*Creature*], los diversos NPCs, la *Database* de persistencia, y otros objetos auxiliares que aportan al desarrollo del juego. Así, todos los comandos llaman a métodos de esta clase, la cual sirve de intermediaria para la interacción con los otros objetos que componen el juego. Es por ello que por cada comando hay un método correspondiente en esta clase. Asimismo, dispone de métodos para el armado y pusheo de notificaciones (broadcasts, mensajes y eventos), y métodos auxiliares para la ejecución de los comandos.
- **LogicMaps:** modela la abstracción del contenedor de mapas del lado del servidor, proveyendo los métodos necesarios para la agregación o eliminación de metadata en los tiles, la búsqueda de tiles libres para el spawneo/dropeo de items, etc.
- **Item:** interfaz que define el comportamiento de los items del juego. Un item tiene un id, un nombre, un precio, es almacenable en el *Inventory* y puede ser equipado. En pos de utilizar dicha interfaz, los items serán allocados en memoria en el heap y así serán polimórficos. Se listan a continuación aquellos objetos que cumplen esta interfaz.
  - **Potion:** interfaz a la que obedecen las pociones de vida [*HealthPotion*] y maná [*ManaPotion*]. A diferencia de los otros objetos, las pociones, al equiparse, restablecen inmediatamente la vida o maná del jugador en *recovery points*, no respetando la interfaz *Wearable*. Para el modelado de las mismas, se utilizó el patrón **Factory** [*PotionFactory*], mediante el cual se recibe un *PotionCfg* y se crea la poción correspondiente.
  - **Wearable:** interfaz que define el comportamiento de los *Items* que pueden ser equipados en el *Equipment*. Cada *Wearable* (al ser Item también) puede ser equipado, pero en este caso se efectúa un **double dispatch** en el que se agrega al *Equipment*. Además, tienen un uso (de ataque o defensa), y se caracterizan por el **wearable type** [Weapon, Helmet, Armour, Shield], que define la ubicación del mismo cuando se porta en el cuerpo del *Character*. Los mismos son:
    - **Weapon:** las armas tienen un determinado rango, daño mínimo, daño máximo, tiempo de cooldown y un tipo (que determina los efectos audiovisuales).

    - **Wand:** los báculos se caracterizan por lanzar un hechizo [Spell].
        - **Spell:** los hechizos pueden ser de ataque [*AttackingSpell*] o de curación [*HealingSpell*]. Los primeros tienen un daño mínimo y un daño máximo, y al ser lanzados le causan un daño al target en dicho rango. Los segundos, en cambio, tienen determinados puntos de curación en los que al ser lanzados recuperan la vida. Ambos cuentan con un determinado costo de maná (para ser usados), rango, cooldown y tipo (weapon type). Para el modelado de los mismos, se utilizó el patrón **Factory** [*SpellFactory*], mediante el cual se recibe un *SpellCfg* y se crea el hechizo correspondiente.
    - **Defence:** las defensas tienen puntos de defensa mínimos y máximos. Ante un ataque, absorben una cantidad de daño delimitada por ese rango.

- **ItemsContainer:** lee los archivos de configuración de *Weapons*, *Wands*, *Spells*, *Defences* y *Potions*, y a partir de ellos crea dinámicamente los *Items* presentes en el juego, almacenándolos. Así, concentramos en un sólo lugar la reserva de memoria del heap y su destrucción, y se provee de un punto de acceso único para la utilización de los mismos, el cual conocerán las entidades que los precisen.

- **Attackable:** tanto los *Characters* como las *Creatures* implementan esta interfaz, la cual provee los métodos a los que se debería poder llamar a cualquier ente atacable. Facilita más que nada la modularización y evitar la repetición de código.

- **Creature:** modela las criaturas, que poseen, principalmente, vida, daño mínimo y máximo, posición [*Position*], rango de visibilidad de characters, velocidad de movimiento y nivel, entre otros atributos. En cada turno que se les asigna, buscan en su rango si hay algún character para atacar. Si lo hay, se mueve en su dirección y lo ataca cuerpo a cuerpo, de lo contrario, se mueve aleatoriamente cada cierto tiempo. En caso de que muera, dropea aleatoriamente algún item, oro, o nada.

- **Character:** representa a un personaje manejado por un jugador. Se caracteriza por tener, principalmente, vida y maná (limitadas), atributos [inteligencia, constitución, fuerza y agilidad], nickname, raza [*Race*] y clase [*Kind*], estado [*State*], nivel [*Level*], inventario [*Inventory*], equipamiento [*Equipment*] y posición [*Position*]. Responde a todos los comandos válidos que son recibidos, y tiene sus respectivos métodos para llevarlos a cabo. Los distintos métodos abarcan la actualización de atributos (con el paso del tiempo y la actualización del nivel), el movimiento del jugador (en el mapa y entre mapas), el manejo de items (equipar, tomar, desequipar, tirar), la recuperación del maná y la vida, el ataque a otros jugadores o criaturas y la recepción de ataques por parte de los mismos, y todas las acciones en las que pueda intervenir.

- **Kind:** las distintas clases definidas en el json de configuración (mago, clérigo, paladín y guerrero) aportan factores a tener en cuenta para el *Character* en las ecuaciones de vida, maná y meditación. Además, son capaces (o no) de meditar y/o hacer magia, y esto último se ve reflejado en si puede lanzar hechizos o no.

- **Race:** las razas definidas en el json de configuración (humano, elfo, enano y gnomo) son estructuras y, como tales, no tienen comportamiento alguno. Simplemente aportan, por un lado, factores a tener en cuenta en el cálculo de las ecuaciones de maná y vida, y por otro lado, modificadores que intervienen en el cálculo de atributos del *Character*.

- **Inventory:** el inventario es esencialmente un contenedor de *Items*, almacenados en *Slots*, cada cual con el item correspondiente y la cantidad disponible. Tiene un número de slots limitado, por lo que sólo se puede tener 18 items distintos en cualquier cantidad. Además, almacena el oro seguro y en exceso, llevando el control adecuado sobre sus cantidades máximas.

- **Equipment:** el equipamiento es esencialmente un contenedor de *Wearables*. Únicamente se puede tener un wearable de cada *WearableType*. Ante un ataque o la recepción de un ataque, se utilizan los elementos de ataque o defensa (respectivamente) que el jugador tenga equipados, causando o absorbiendo el daño correspondiente.
- **States:** define la interfaz común a los diversos estados en los que puede estar un *Character*. Estos son: vivo [*Alive*], muerto [*Dead*], o resucitando [*Resurrecting*]. Asi, en cada clase de estado se definen las acciones que se pueden realizar (o no), respetando la interfaz *States*.
- **Level:** cuenta con un nivel, experiencia actual y experiencia necesaria para subir de nivel. Lleva a cabo la actualización de nivel cuando se reciben updates de experiencia al efectuar un ataque, y determina si un jugador es newbie o no.
- **Position:** tanto los characters como las creatures tienen una posición, que contiene el id del mapa en el que se encuentran, las coordenadas x e y de su posición y su orientación. Delegan en esta clase el movimiento dentro del mapa y entre mapas.
- **Bank:** modela al banquero, siendo no más que un container de *BankAccounts*. Cada jugador, asociado a su nickname único y permanente, tiene asociado una cuenta bancaria que se persiste junto con toda su información.
- **BankAccount:** almacena los items y sus cantidades en un unordered map indexado por los respectivos id, de tamaño limitado debido a la necesidad de persistencia. Además, cuenta con una suma de oro. Provee los métodos necesarios para depositar o retirar tanto items como oro.
- **Formulas:** provee los métodos para el cálculo de las ecuaciones necesarias en el juego, por tanto, todas las clases que necesitan hacer uso de la misma tienen una referencia hacia ella. Los modificadores que intervienen en las mismas son cargados mediante un archivo json de configuración, pudiéndose modificar a gusto.

3.3. Diagramas UML

Se muestran a continuación los distintos diagramas diseñados para facilitar la comprensión delmodelo al lector. Estos fueron separados según categoría y tipo.

3.3.1. Diagramas de clases

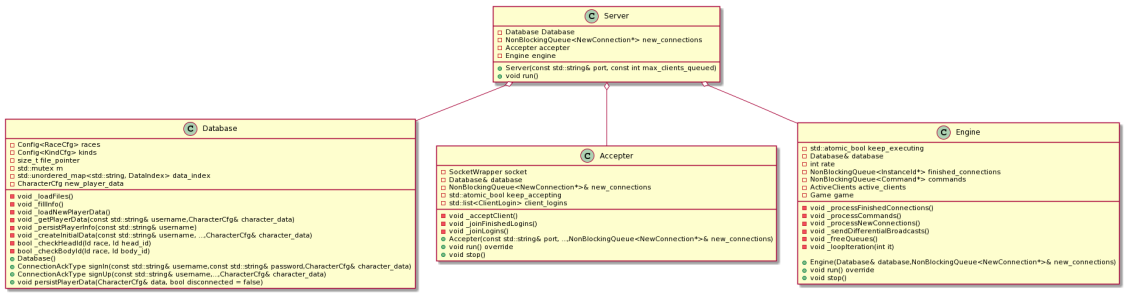

Figura 2: Clases principales de control del servidor

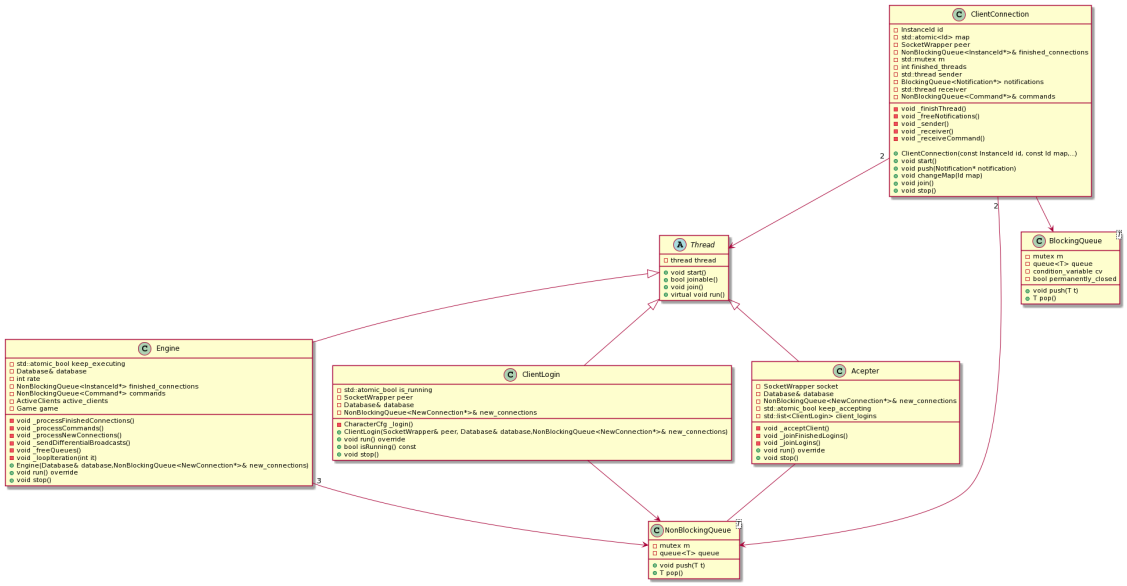

Figura 3: Hilos de ejecución del juego y su manejo de colas

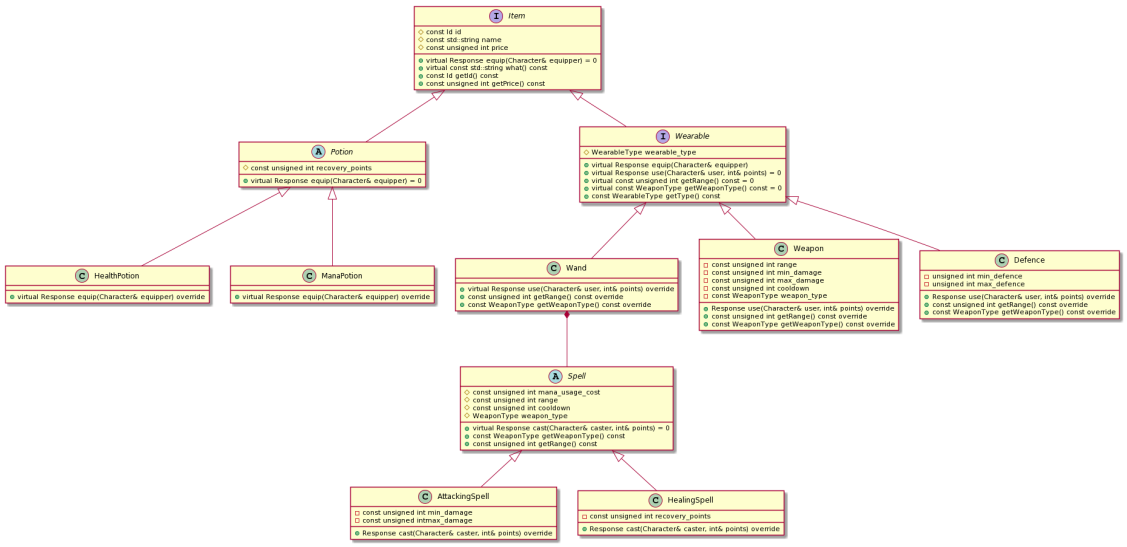

Figura 4: Modelado de Items

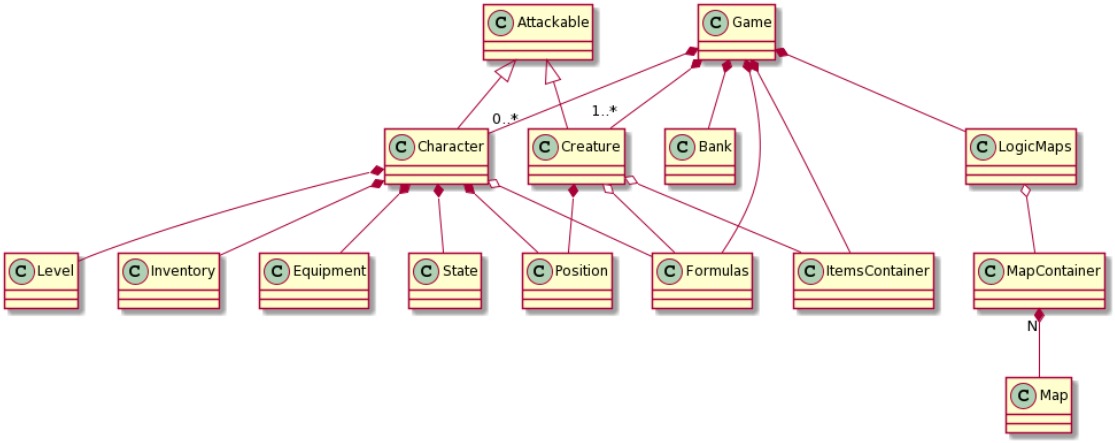

Figura 5: Diagrama general de clases del juego

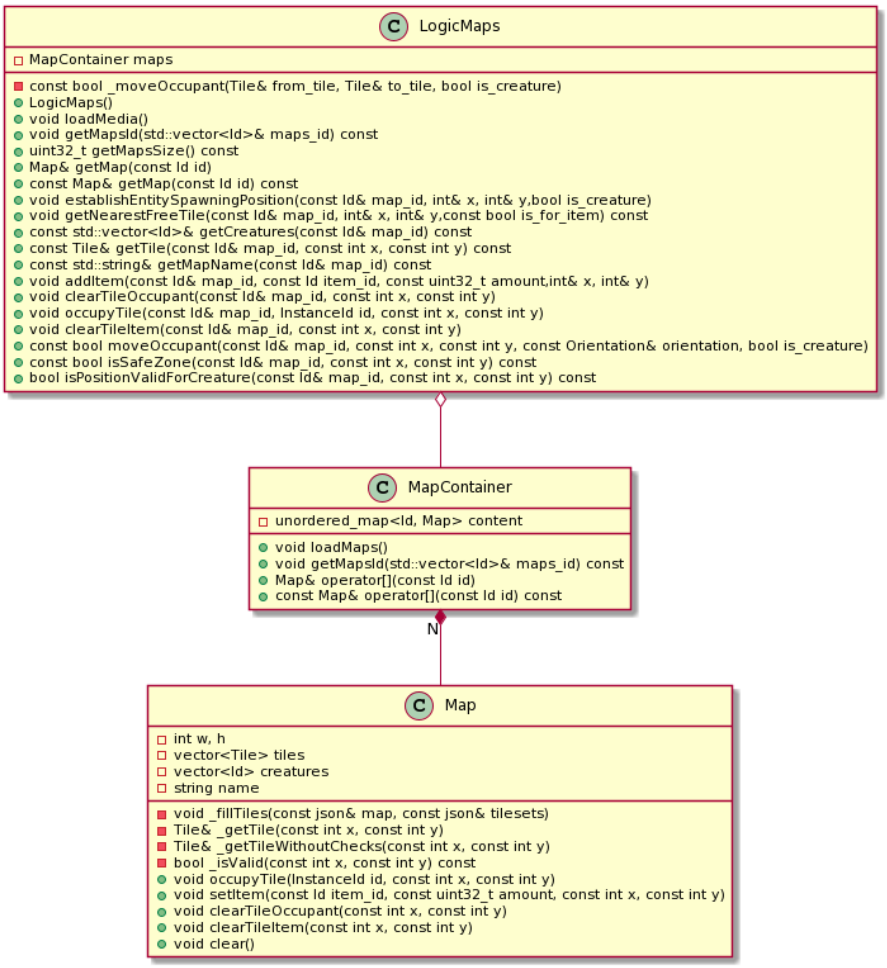

Figura 6: Abstracción de mapa del servidor

3.3.2. Diagramas de secuencias

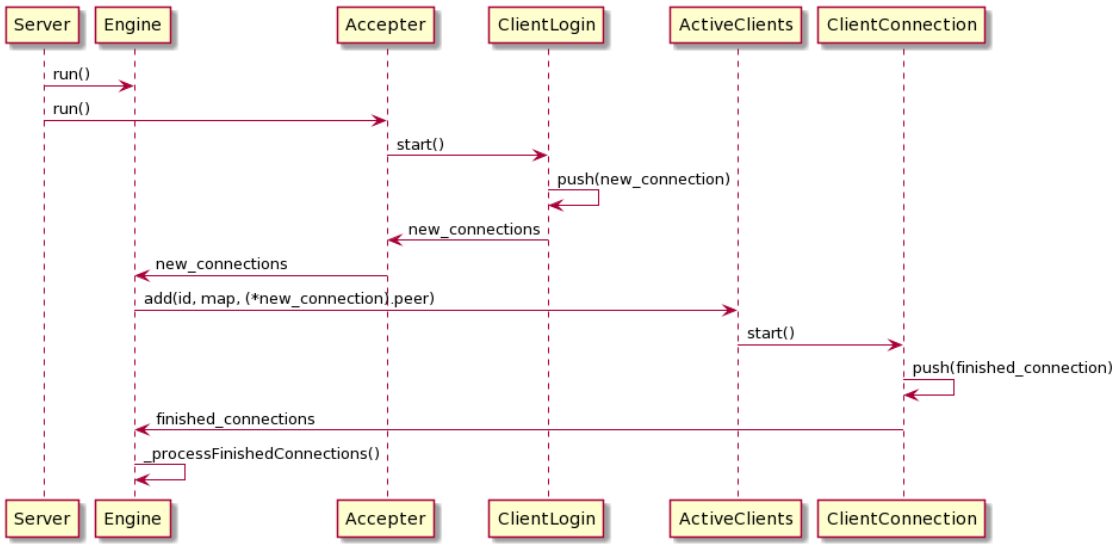

Figura 7: Secuencia de conexión y desconexión de un jugador

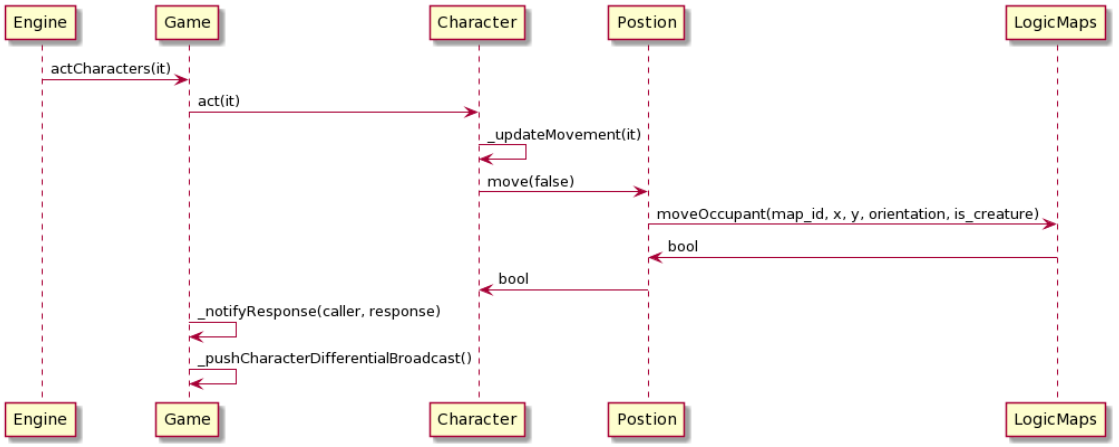

Figura 8: Secuencia de actualización de la posición de un jugador cuando se está moviendo

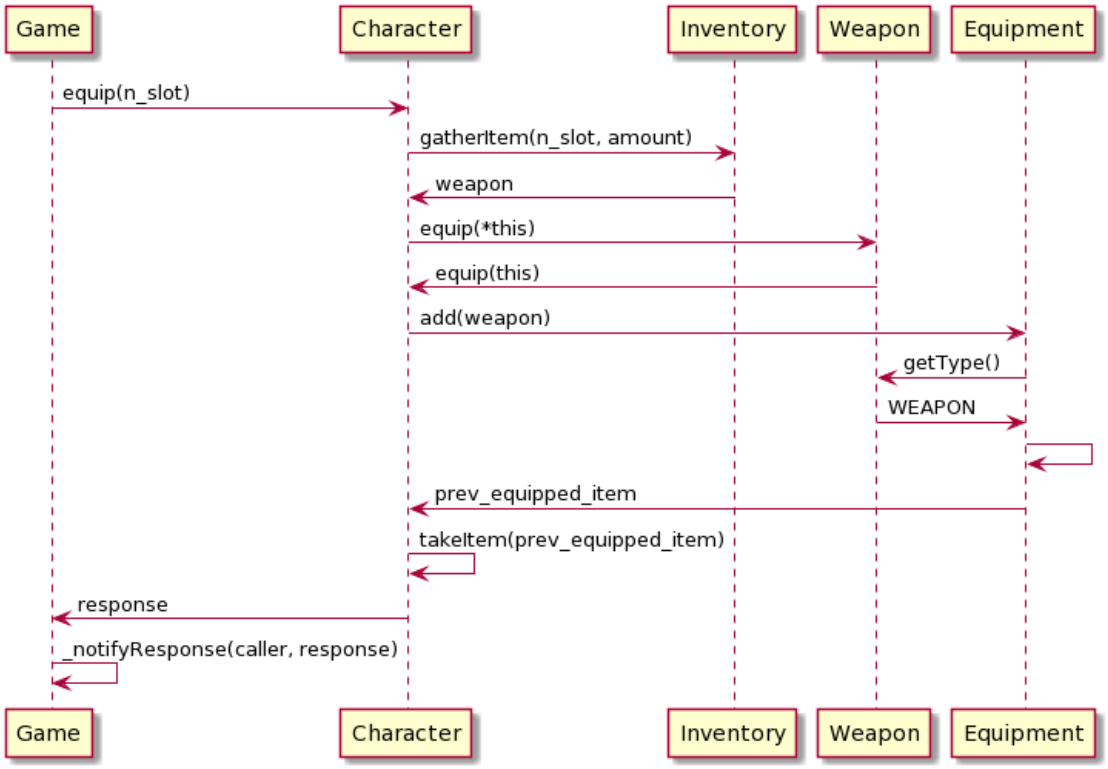

Figura 9: Secuencia del comando equipar arma [de la clase **Weapon**]

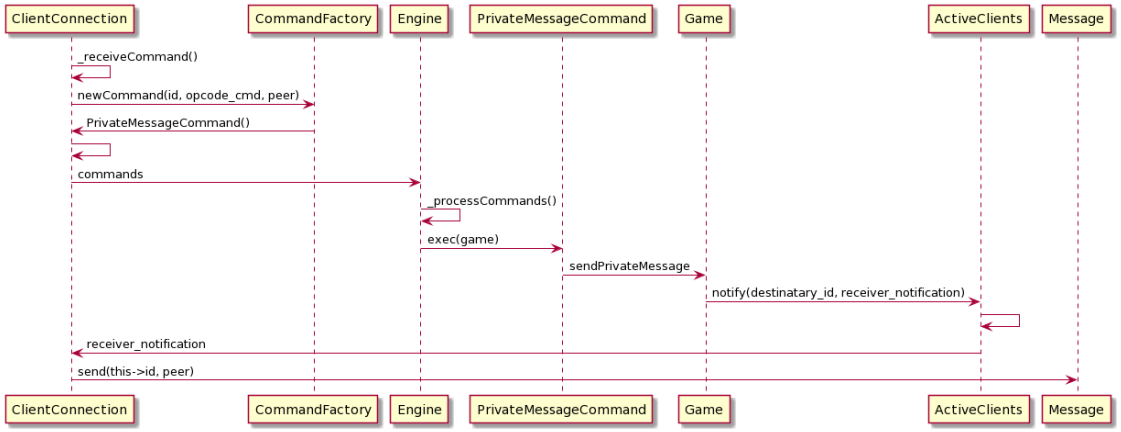

Figura 10: Secuencia del comando mandar un mensaje privado a otro jugador

4. Cliente

4.1. Descripción general

4.1.1. Máquina de estados

Debido a que toda sección del cliente tiene asociada una represetanción gráfica, y que para esto se utilizó SDL2, con el objetivo de crear la ventana y su renderer una sóla vez y reutilizarla para las distintas sub-ventanas, se decidió implementar una **máquina de estados**. En particular, en nuestro programa los estados son çontextos"de sub-ventanas, tales como una ventana de inicio, una de conexión, de creación de personajes, etc. Esta compone el loop principal del cliente y su funcionamiento es muy sencillo:

1. Se inicializan los sistemas de SDL2.
2. Se pone a correr la máquina comenzando con un estado inicial.
3. Cada estado se ejecuta en un contexto aislado, y es responsable de indicar ante su finalización cuál será el siguiente.
4. Se repite el loop hasta que algún estado indique que el usuario desea salir.

En el caso del aplicativo en cuestión, los estados fueron los siguientes: **Home, Tutorial, Connection, SignUp, Game**. Game funciona como **estado terminal**: cuando la ejecución del mismo termina, se sale ordenadamente del aplicativo.

Todas las vistas heredan de la clase abstracta **ConstantRateFunc**, que proporciona la interfaz necesaria para llamar a una función respetando un período determinado. En el caso del cliente, el período representa el tiempo durante el cual cada frame será presentado (inversa de *fps*).

4.1.2. Estados previos al juego

Antes de comenzar a jugar, será necesario conectarse a un servidor, y luego a un personaje existente en el mismo. Para esto, se proporcionan varias vistas previas al juego en sí que se explicana continuación:

- **HomeView:** vista inicial cuando se abre el cliente. Desde esta, el usuario puede enviar solicitudes de conexión a una dirección IP y puerto específicos, Si la conexión se logra establecer, se pasará a la siguiente vista: **ConnectionView**. En caso de que no se logre establecer la misma, se le informará al usuario y se le volverá a permitir ingresar una nueva solicitud. También se puede acceder desde esta ventana al **Tutorial**, mediante el botón dado.
- **TutorialView:** vista sencilla que le enseña a los usuarios a utilizar la aplicación. Se basa en un *scrolling de imágenes.*
- **ConnectionView:** una vez que se llega a esta vista, la conexión con el servidor **ya está establecida**. Lo único que falta para empezar a jugar es conectarnos con un personaje existente, proporcionando el usuario y la contraseña. Al hacerlo, se enviará la solicitud de *log-in* al servidor, y en caso de que ambos datos coincidan con alguna entrada en la base de datos, empezaremos a jugar, pasando a **GameView**. En caso de no tener ningún personaje, se puede crear uno mediante el botón dado que nos llevará a la ventana de **SignUpView**.
- **SignUpView:** vista que le proporciona al usuario un formulario a rellenar para poder crear su personaje. Una vez que se haga click en **Crear**, se enviará la solicitud de creación al servidor quien validará los datos y en caso exitoso, se creará efectivamente el personaje. Ante cualquier error, se le informará al usuario del mismo.

4.1.3. Juego

Una vez que el usuario logró conectarse al servidor con un personaje válido, iniciará la vista del juego en sí, **GameView**. Para modelar el juego, se utilizaron **tres hilos de ejecución**:

- **Receiver:** objeto activo (que corre sobre su propio hilo de ejecución) que se encarga de recibir toda la información necesaria del servidor mediante el protocolo. Deposita la información recibida en distintas *colas thread-safe no bloqueantes* de las cuales el loop principal la leera.
- **CommandDispatcher:** objeto activo que loopea tomando comandos polimórficos de una *cola thread-safe bloqueante* para luego enviarlos al servidor mediante delegación. Estos comandos nacen en el loop principal, quien los agrega a la cola.
- El **hilo principal**, donde corre **GameView**, encargado del procesamiento y del renderizado, donde se utiliza SDL para representar gráficamente la información del juego.

Ambos hilos adicionales *(el Receiver y el CommandDispatcher)* se ponen a correr justo antes de poner a correr el loop principal del juego.

4.1.4. Comunicación entre hilos

Como se mencionó previamente, para lograr una comunicación segura entre hilos que evite todo tipo de *race-conditions* y que sea eficiente, se decidió utilizar **colas thread-safe**, ya sean bloqueantes o no, protegidas mediante **mutex**. Para implementar colas bloqueantes, se utilizó **condition variables**, mientras que para las no bloqueantes simplemente se devolvió *NULL* cuando no habían más objetos. Las cuatro colas utilizadas fueron diseñadas para manejar objetos polimórficos, permitiendo delegar la lógica en los mismos. Estas fueron:

- Cola bloqueante de **comandos** (común a GameView y al CommandDispatcher): cada comando se sabe enviar al servidor mediante el socket de manera polimórfica. Más adelante se detallarán los distintos comandos utilizados.
- Cola no bloqueante de **broadcasts** (común a GameView y al Receiver): un broadcast es una actualización del estado global del juego. Polimórficamente, cada broadcast sabe actualizar las estructuras locales del cliente para mantener en sintonía al mismo con el servidor.
- Cola no bloqueante de **mensajes** (común a GameView y al Receiver): al igual que los broadcasts, los mensajes representan de manera polimórfica información que debe ser mostrada al usuario, ya sean mensajes por consola (de error, de información, de respuesta, etc.), mensajes privados, o mensajes generales.
- Cola no bloqueante de **eventos** (común a GameView y al Receiver): estos eventos también polimórficos representan justamente sucesos en el juego que al cliente pueden interesarle para mostrarle información relevante al usuario. En este caso, los efectos se utilizaron para reproducir sondios y para mostrar animaciones especiales (explosiones, ataques, daño, curación, y muchas más).

4.1.5. Iteración del juego

Como se mencionó previamente, todas las vistas (incluyendo la del juego) heredan de la clase abstracta **ConstantRateFunc**, que provee una interfaz que realiza un loop a período constante y controlado, ejecutando en el mismo una función que tenemos que implementar. Se procede entonces a detallar en líneas generales cómo se compone esta función para nuestro juego:

1. **Se procesan los eventos de SDL**, de dónde obtendremos información sobre el accionar del jugador.

2. **Se vacían las colas compartidas:** primero se procesan los mensajes, luego los broadcasts, y por último los eventos del juego, aunque el órden es indistinto y arbitrario ya que son independientes entre sí.
3. **Se limpia la pantalla** realizando un llamado a SDL_RenderClear *(método de SDL que sirve para borrar el contenido actual de la misma).*
4. Se informa a los distintos componentes del juego que **han transcurrido *it* iteraciones**, y que deben actuar en consecuencia (avanzar animaciones, realizar sonidos, centrar la cámara, etcétera).
5. **Se renderizan los componentes del juego:** el escenario, los efectos especiales, y la interfaz *(hud).*
6. Por último, **se presenta la pantalla** utilizando el método de SDL, SDL_Render Present.

Como se puede observar, es un loop muy sencillo similar al de cualquier aplicación que implemente una interfaz gráfica de usuario: **procesar eventos, actuar en consecuencia, y renderizar el nuevo estado**. Dada la simplicidad de esta función, se adjunta a continuación el código fuente:

```
void GameView::_func(const int it) {
    /* Vaciamos las colas a procesar*/
    _processSDLEvents();
    _processMessages();
    _processBroadcasts();
    _processEvents();

    /* Limpiamos la pantalla */
    renderer.clearScreen();

    /* Acciones previas al renderizado*/
    player.act(it);
    characters.act(it);
    creatures.act(it);
    effects.act(it);
    camera.center(player.getBox(), map.widthInPx(), map.heightInPx());
    map.setRenderArea();
    hud.update(it);

    /* Renderizamos y presentamos la pantalla */
    stage.render();
    effects.render();
    hud.render();

    renderer.presentScreen();
}
```

Esta función se ejecutará en el loop descripto anteriormente hasta que el usuario cierre el juego, momento en el cuál se liberarán los recursos, se detendrá la ejecución de los sub-sistemas de SDL y se procederá a terminar ordenadamente.

4.2. Clases

Con el objetivo de facilitarle la comprensión al lector, se mostrará la información sobre las clases en distintos grupos y secciones.

4.2.1. Punto de entrada

La ejecución del **Cliente** comienza por la función *main* definida en *main.cpp*. Esta función simplemente funciona como **punto de entrada** a nuestro aplicativo, y su única responsabilidad es instanciar a la clase **Client** y comenzar su ejecución:

```
    int main(int argc, char* argv[]) {
    if (argc != EXPECTED_ARGC) {
        fprintf(stderr, "Usage: %s\n", argv[NAME]);
        return USAGE_ERROR;
    }

    try {
        Client client;
        client.launch();

    } catch (const std::exception& e) {
        fprintf(stderr, "%s\n", e.what());
        return ERROR;
    } catch (...) {
        fprintf(stderr, "Unknown error.\n");
        return ERROR;
    }

    return SUCCESS;
}
```

4.2.2. Máquina de estados

A continuación se detallan las clases más importantes del aplicativo, que implementan el loop de la máquina de estados descripto anteriormente:

- **Client:** clase principal que orquestra la ejecución de la máquina de estados del aplicativo. Proporciona un sólo método en su API pública: *launch*, encargado de comenzar el ciclo.
- **ConstantRateFunc:** clase abstracta que proporciona una interfaz para ejecutar bajo un loop a período de tiempo controlado, una función virtual pura. Utilizada para renderizar a frame-rate constante las vistas que se presentan en la pantalla. Provee de un único método en su API pública: *run*, comenzando la ejecución del loop.
- **HomeView:** vista principal que vemos al iniciar el aplicativo. Responde a las acciones del usuario, intentando conectarse al servidor indicado. Cuando lo logra, cambia el próximo estado a ConnectionView. También puede accederse al tutorial desde este. Implementa ConstantRateFunc, por lo que su único método es *run*.
- **TutorialView:** vista que muestra distintas imágenes para enseñarle al usuario a utilizar la aplicación. Su siguiente estado siempre será HomeView cuando se clickee el botón "Volver". Implementa ConstantRateFunc, por lo que su único método es *run*.
- **ConnectionView:** vista en la que el usuario podrá conectarse a su personaje o acceder al formulario para crear uno nuevo mediante comunicación con el servidor. En el primer caso, el siguiente estado será GameView, mientras que en el segundo será SignUpView. Implementa ConstantRateFunc, por lo que su único método es *run*.
- **SignUpView:** vista que proporciona al usuario de un formulario para crear un nuevo personaje. Su siguiente estado siempre será ConnectionView, cuando se presione "Volver". Implementa ConstantRateFunc, por lo que su único método es *run*.

- **GameView:** vista principal del juego, contiene todos los componentes del mismo y es quien orquestra su ejecución mediante la comunicación con el servidor mediante distintos hilos de ejecución. Implementa ConstantRateFunc, por lo que su único método es *run*.

4.2.3. Clases comunes utilizadas

Se listan a continuación las clases más importantes comunes al cliente y servidor que son utilizadas en el aplicativo:

- **Socket:** abstracción diseñada para la utilización de los sockets UNIX vistos en la materia. Sus principales métodos son *send, recv, shutdown y close*.
- **SocketWrapper:** wrapper para el Socket que sobrecarga operadores para proporcionar usabilidad (tales como el operador «<"para el envío de bytes a través del socket, y el operador »>"para recibirlos).
- **Thread:** clase abstracta para definir objetos activos. Quienes hereden de esta, deberán implementar el método *run*, que será la función que se ejecutará en un nuevo hilo de ejecución. Se proporciona de los métodos de *start, joinable y join*.
- **NonBlockingQueue<>:** Template de cola no bloqueante thread-safe que protege su estructura de datos con un **mutex**. Devuelve NULL en caso de estar vacía.
- **BlockingQueue<>:** Template de cola bloqueante thread-safe que protege su estructura de datos con un **mutex**. En caso de estar vacía, duerme al thread solicitante utilizando una **condition variable**. Si la misma está definitivamente cerrada, devuelve NULL.
- **Map:** abstracción para el mapa. Se inicializa llenando sus estructuras desde un archivo json. Compuesto por un vector de **Tiles** (estructura que guarda la información de un tile, como el id del ocupante, el id gráfico del piso, etc.). Se proporcionan distintos métodos de lectura como *getTile, getWidthTiles, getHeightTiles, getCreatures, getMapName*, y otros de escritura como *occupyTile, setItem, clearTileOccupant, clearTileItem, clear,* entre otros.
- **MapContainer:** clase que contiene todos los mapas del juego actual en un map de la forma <Id, Map>. Permite un acceso constante ya sea para lectura o escritura a cualquier mapa mediante operadores.

4.2.4. Clases lógicas del juego

Se listan las clases que componen el modelo del juego en el cliente:

- **Command:** clase abstracta para la implementación de comandos polimórficos que se sepan enviar al servidor. Entre ellos, están *StartMovingCommand, StopMovingCommand, GrabItemCommand, UseMainWeaponCommand,* y muchos más.
- **CommandDispatcher:** objeto activo que saca comandos de una cola no bloqueante y los envía al servidor según el protocolo.
- **Broadcast:** clase abstracta para la implementación de actualizaciones de estado polimórficas que saben modificar las estructuras de datos del juego. Algunas de ellas son: *NewCharacterBroadcast, UpdateCharacterBroadcast, DeleteCreatureBroadcast,* etcétera.
- **BroadcastFactory:** pseudo-implementación del *FactoryPattern* para recibir distintos tipos de broadcasts. Proporciona un único método: *newBroadcast*, que se encarga de realizar la comunicación necesaria con el servidor para recibir el broadcast que corresponda.

- **Message:** clase abstracta para la implementación de mensajes polimórficos que son responsables de actualizar la información relevante en la pantalla, como la consola de texto, o el mensaje flotante en los personajes, entre otros. Los tipos de mensajes son: *PlainMessage, ListMessage, y SignedMessage.*
- **MessageFactory:** al igual que BroadcastFactory, es una pseudo-implementación del *FactoryPattern* para recibir distintos tipos de mensajes. Su API pública también está constituida por un único método: *newMessage*, que recibirá el mensaje del servidor.
- **GameEvent:** clase abstracta para la implementación de eventos del juego polimórficos que saben manifestarse en el cliente, reproduciendo un sonido y/o mostrando una animación, según corresponda. *NOTA: como en la actualidad del proyecto los eventos utilizados hacen todos lo mismo (reproducir un sonido y una animación) se decidió hacer que la clase sea concreta en vez de abstracta. Sin embargo, para futuras extensiones, se mantiene la estructura polimórfica planteada anteriormente.*
- **Receiver:** objeto activo que recibe mensajes, broadcasts o eventos del servidor y los almacena en las colas correspondientes.
- **EventHandler:** objeto que se encarga de procesar los eventos de SDL que lleguen al cliente. Para los eventos que el usuario ingrese por consola, utiliza un *InputParser*. Su método principal es *handleEvent*.
- **InputParser:** objeto que se encarga de parsear un input ingresado por el usuario en la consola para identificar mensajes generales, privados, comandos, o inputs desconocidos según corresponda. Su método principal es *parse*.

4.2.5. Abstracciones de SDL diseñadas

Para utilizar SDL y RAII al mismo tiempo, se diseñaron distintas abstracciones muy utilizadas por todo el cliente. Estas son:

- **Window:** clase fundamental que encapsula la ventana de SDL. La misma ventana se mantiene desde el comienzo hasta el final de la ejecución, siendo la máquina de estados quien varía las pantallas que se muestran sobre esta ventana.
- **Renderer:** la clase más importante de todo el cliente, encargada de manipular la pantalla, ya sea para borrar su contenido, renderizar nuevos objetos o para presentar la misma al usuario. También se encarga de cargar texturas desde una *Surface* de SDL. Los métodos más utilizados a lo largo del juego son: *clearScreen, render, fillQuad y presentScreen.*
- **Texture:** la segunda clase más importante de todo el cliente. Si bien el Renderer es quien renderiza las imagenes, sin texturas no tendría nada que renderizar. Se construyen desde *SDL_Surfaces* y utilizan (si está disponible) aceleración por hardware.
- **Mixer:** implementación del patrón *Singleton*, se encarga de manejar todo el sistema de audio del cliente: tiene una **lista round-robin** de canciones que suenan de fondo, así como distintos efectos cortos de sonido denominados ***chunks*** que ofrece para ser reproducidos en respuesta a ciertos eventos.

A su vez, dado que se decidió usar SDL puro para todo el cliente, se diseñaron algunos "widgets"para facilitar el armado de las distintas vistas: **Button**, **SelectionInputBox** (un cuadro que permite scrollear entre distintas opciones a seleccionar), y **TextBox**.

4.2.6. Clases gráficas del juego

Por último, se listan las clases más importantes de la parte gráfica del juego:

- **Camera:** clase encargada de mostrarle al usuario los tiles que corresponden, centrando al jugador principal en ellos.
- **EffectPlayer:** se encarga de reproducir todo tipo de efectos especiales, como explosiones, bolas de fuego, rayos, sangre en la pantalla al recibir daño, etcétera.
- **Unit:** clase abstracta que proporciona métodos para el movimiento de las unidades, y su renderizado. Quienes hereden de Unit deberán implementar *render* (para renderizar los sprites que correspondan), pero utilizando la el método protegido *_render* que renderiza un sprite dado.
- **Player:** clase que representa al personaje del jugador principal. Hereda de Unit. Comparte muchos métodos con la clase Character, pero contiene mayor información, como su inventario, su equipamiento, y muchas cosas más.
- **Character:** clase que representa a los personajes de los demás jugadores. Hereda de Unit. Al igual que Player, sabe renderizarse, así como mostrar cierta información útil (nickname, nivel) al resto. Se ofrecen también mensajes flotantes que se mostrarán en su cabeza cuando corresponda.
- **Creature:** clase que representa a los monstruos. Están compuestos por un único sprite, a diferencia de los characters que se forman "de a partes"(un Character tiene un sprite para la cabeza, otro para el cuerpo, otro para la armadura, etc.).
- **Contenedores de unidades:** se diseñaron las clases **CharacterContainer** y **CreatureContainer** con el fin de encapsular comportamiento. Ambas utilizan un **unordered_map** para poder acceder en tiempo constante a cada unidad.
- **Contenedores de texturas:** con el objetivo de centralizar la carga de recursos gráficos, se generaron contenedores de sprites compartidos entre las distintas unidades que los requieran: **UnitSpriteContainer**, **ItemSpriteContainer** y **TileContainer**. Utilizan **unordered_map**s para almacenarlas y su acceso se realiza por **Id**.
- **MapView:** abstracción del cliente para el Mapa, que contiene el **MapContainer** común a ambos aplicativos. Proporciona métodos para renderizar sus distintos componentes, así como otros para modificar el contenido de un tile.
- **Stage:** clase que encapsula el la presentación del mapa con sus unidades y objetos. Se encarga de que el renderizado sea inteligente, realizando el mismo por filas para dar ilusión de una vista de aguila desde el punto de vista del jugador.
- **HUD:** objeto que encapsula toda la información que el usuario puede ver además de la ventana del juego en sí, como el inventario, su nombre de usuario, su nivel, sus barras de vida y mana, y hasta la consola. Se actualiza en base a los broadcasts recibidos del servidor sobre el player (**UpdatePlayerData**). Contiene distintos componentes: **Console**, **UserInfo**, **UserInventory** y **UserStats**
- **Console:** clase con dos funciones independientes y muy importantes: obtener texto del usuario, y mostrarle texto al usuario. Permite el ingreso de mensajes, comandos, así como el loggeo de errores, warnings, o informaciño que el jugador mismo haya solicitado.

4.3. Diagramas UML

Se muestran a continuación los distintos diagramas diseñados para facilitar la comprensión del modelo al lector. Estos fueron separados según categoría y tipo.

4.3.1. Máquina de estados

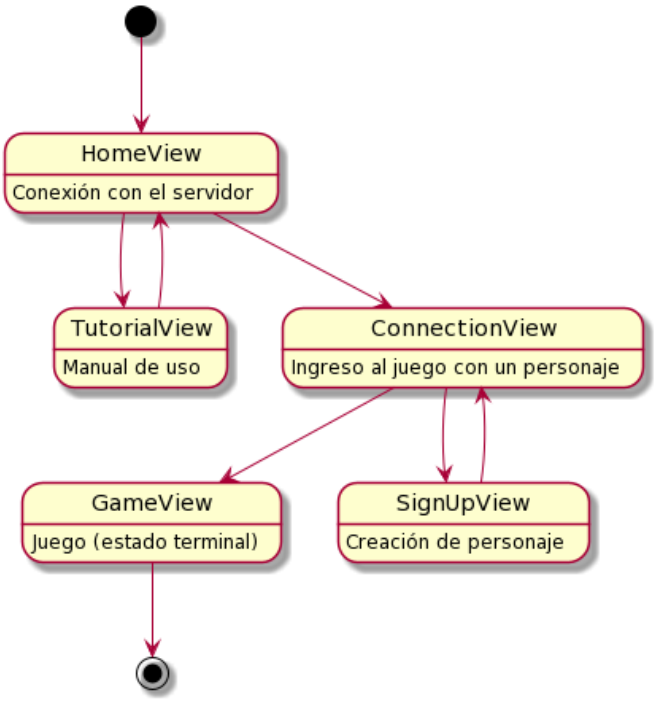

Figura 11: Máquina de estados

4.3.2. Diagramas de clases

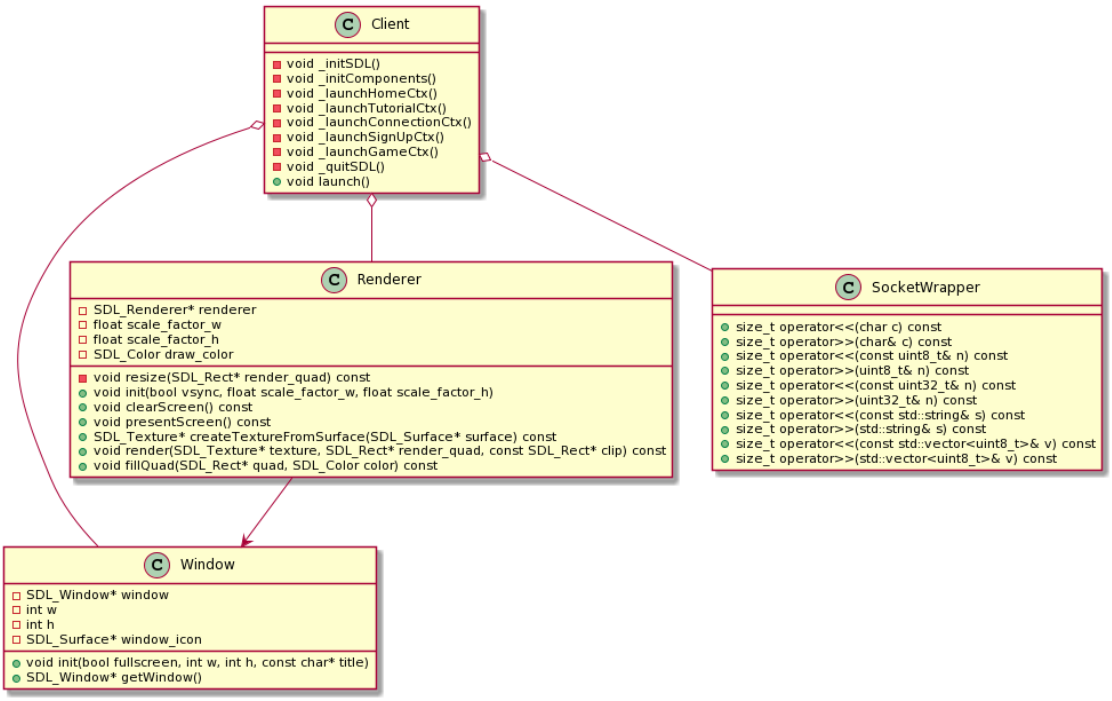

Figura 12: Clases principales del cliente

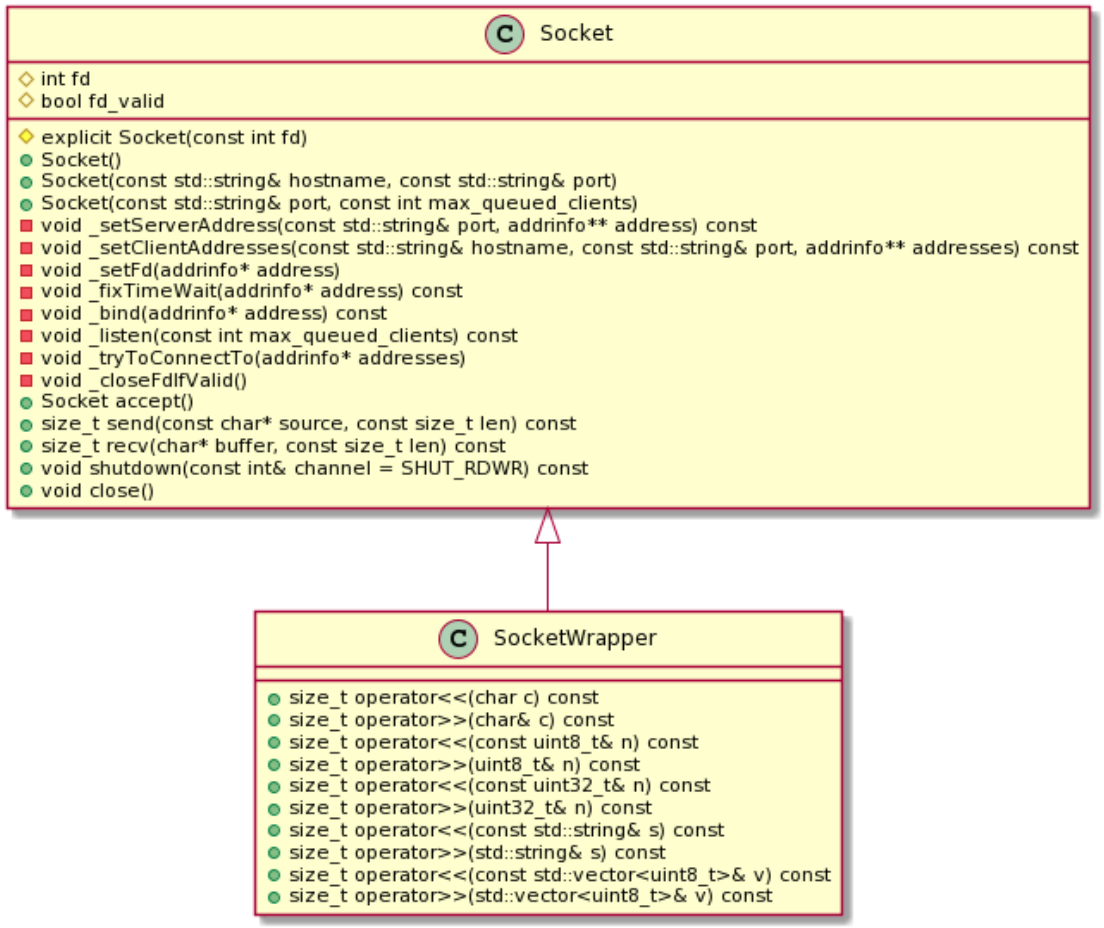

Figura 13: Abstracciones para comunicación

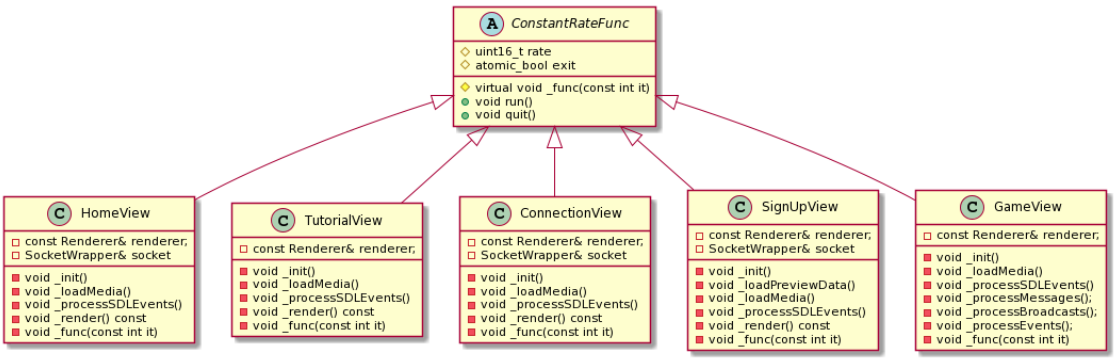

Figura 14: Vistas

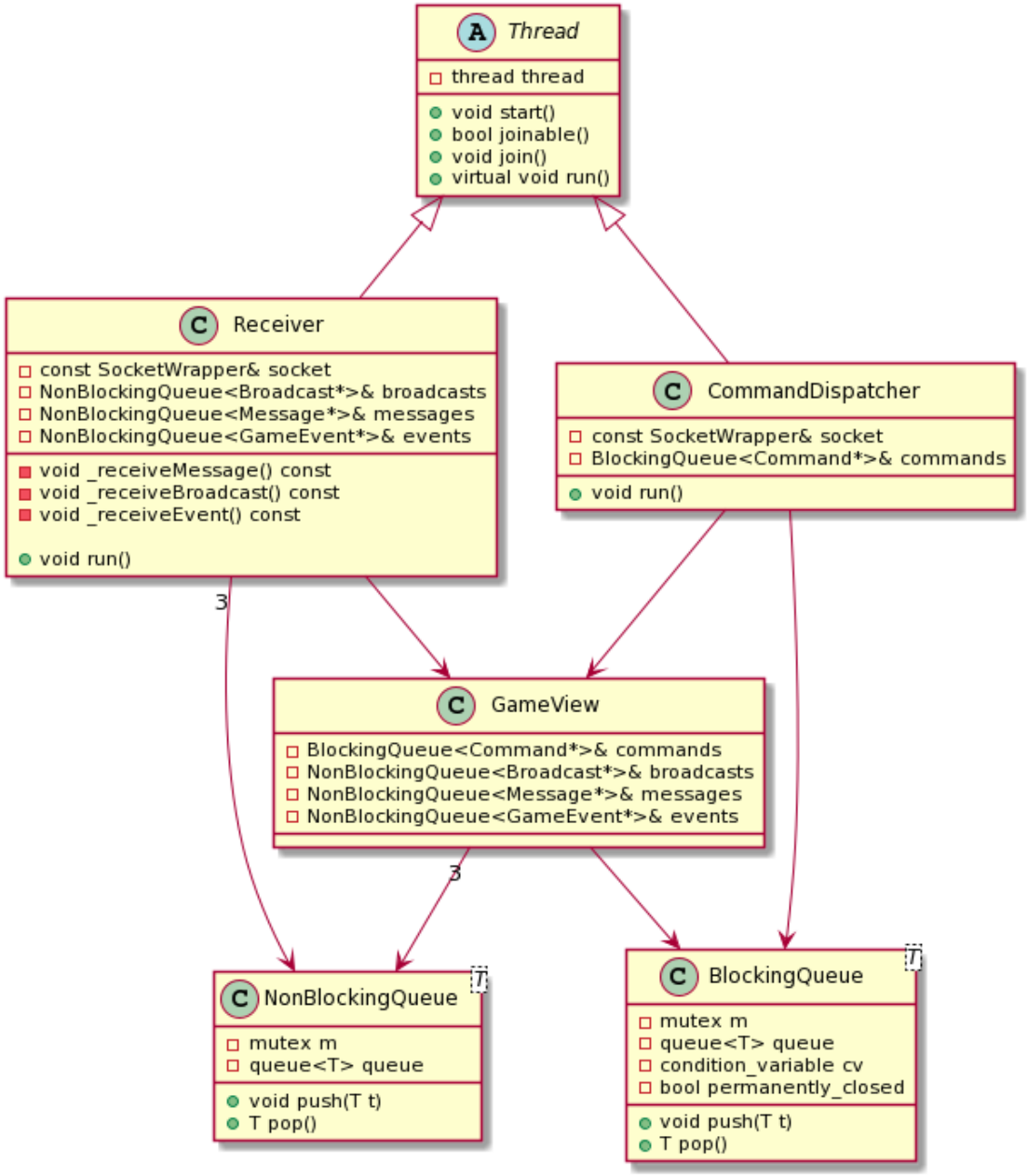

Figura 15: Hilos de ejecución del juego

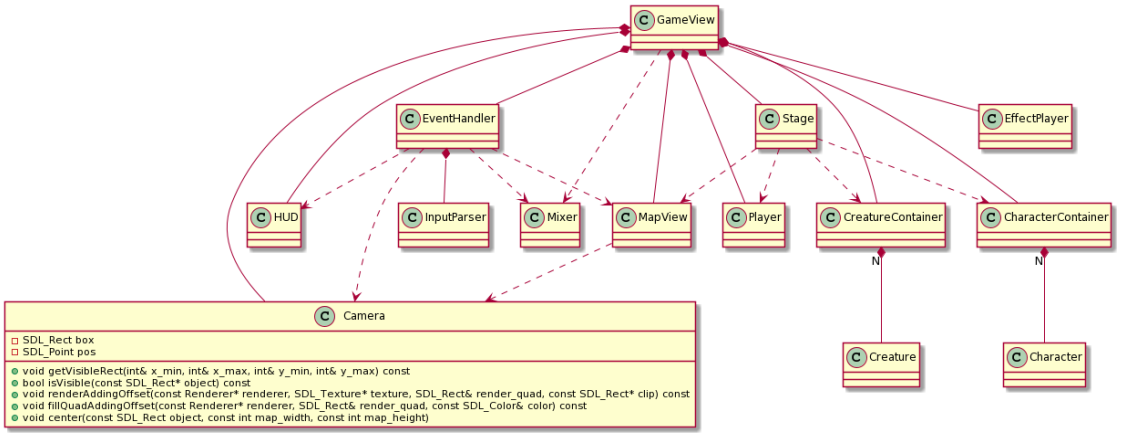

Figura 16: Diagrama general de clases del juego

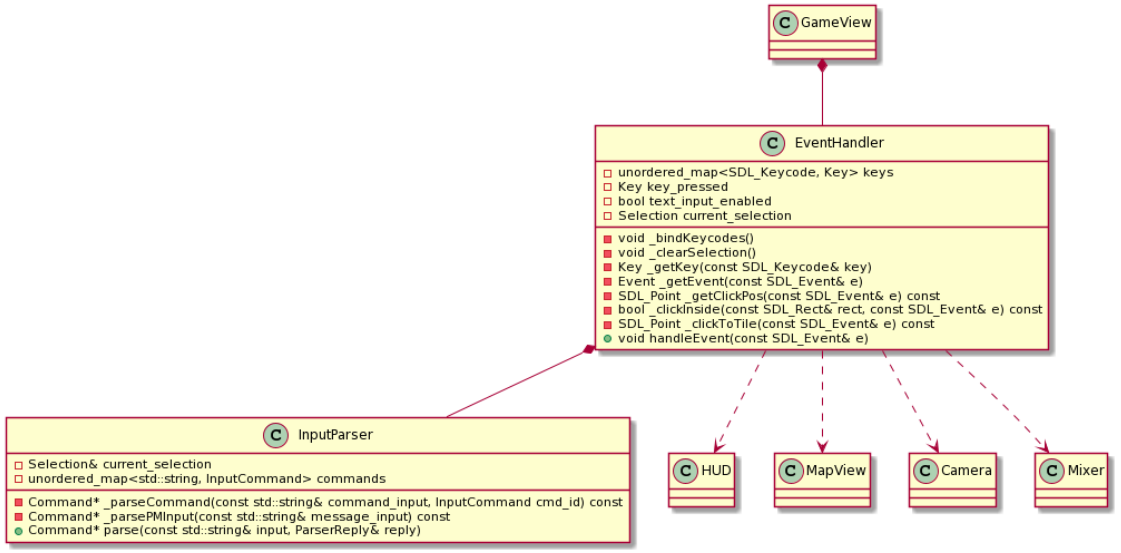

Figura 17: Handler de eventos y parser de comandos

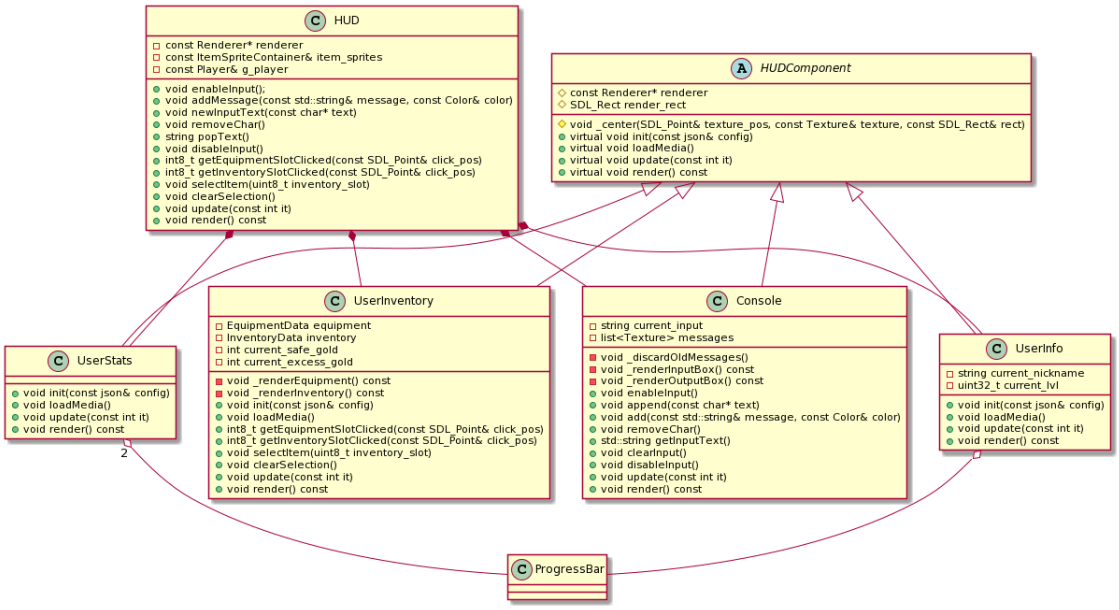

Figura 18: Diagrama general de la interfaz

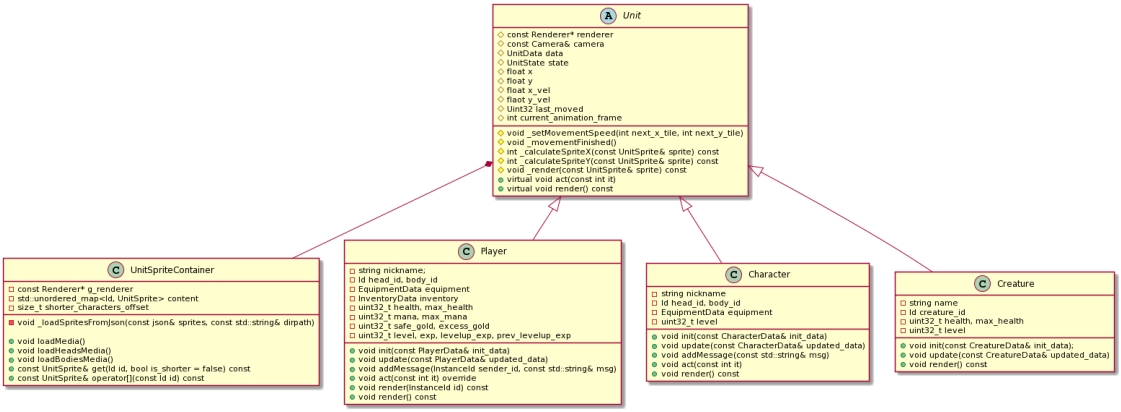

Figura 19: Unidades móviles

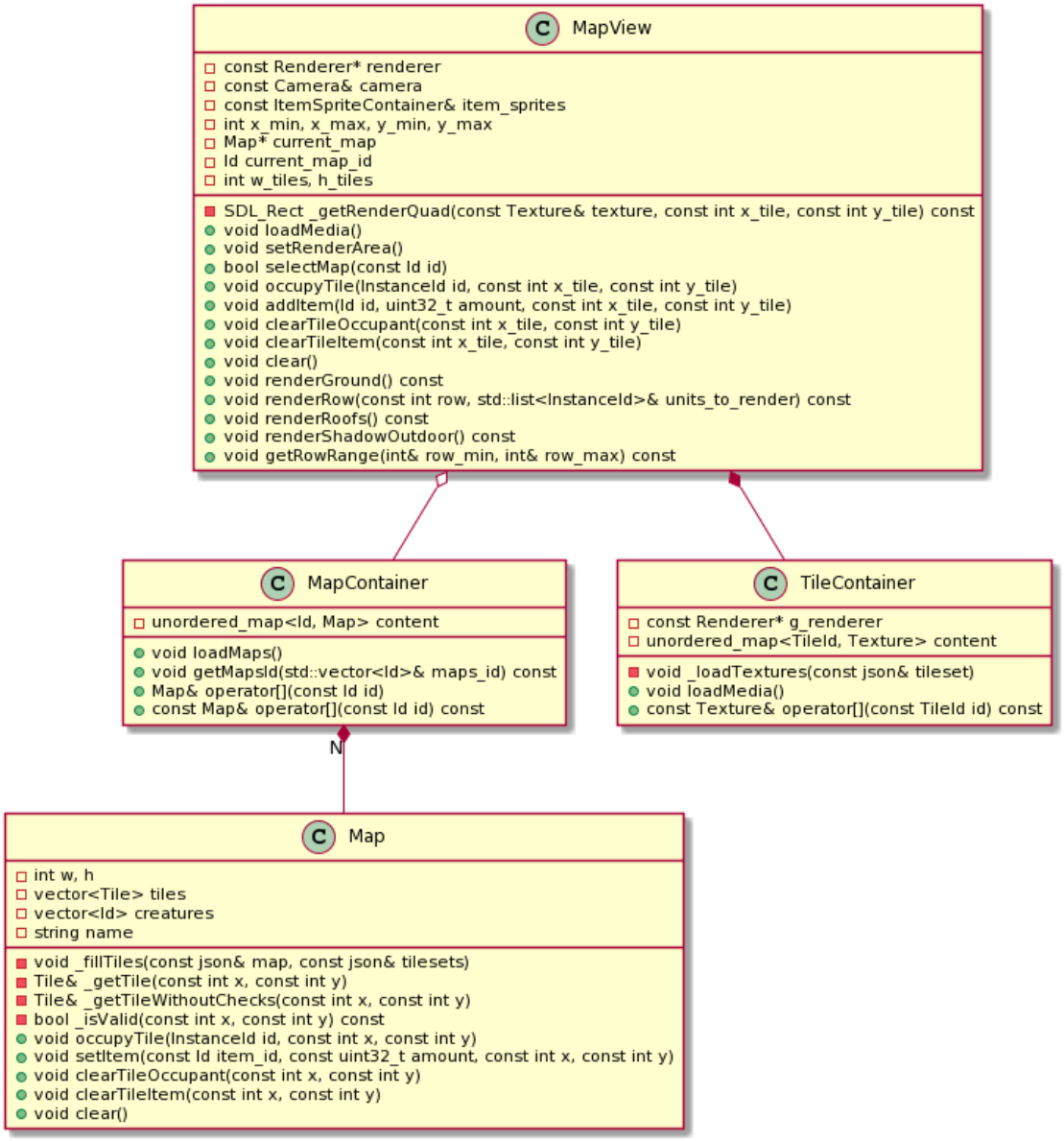

Figura 20: Abstracción de Mapa

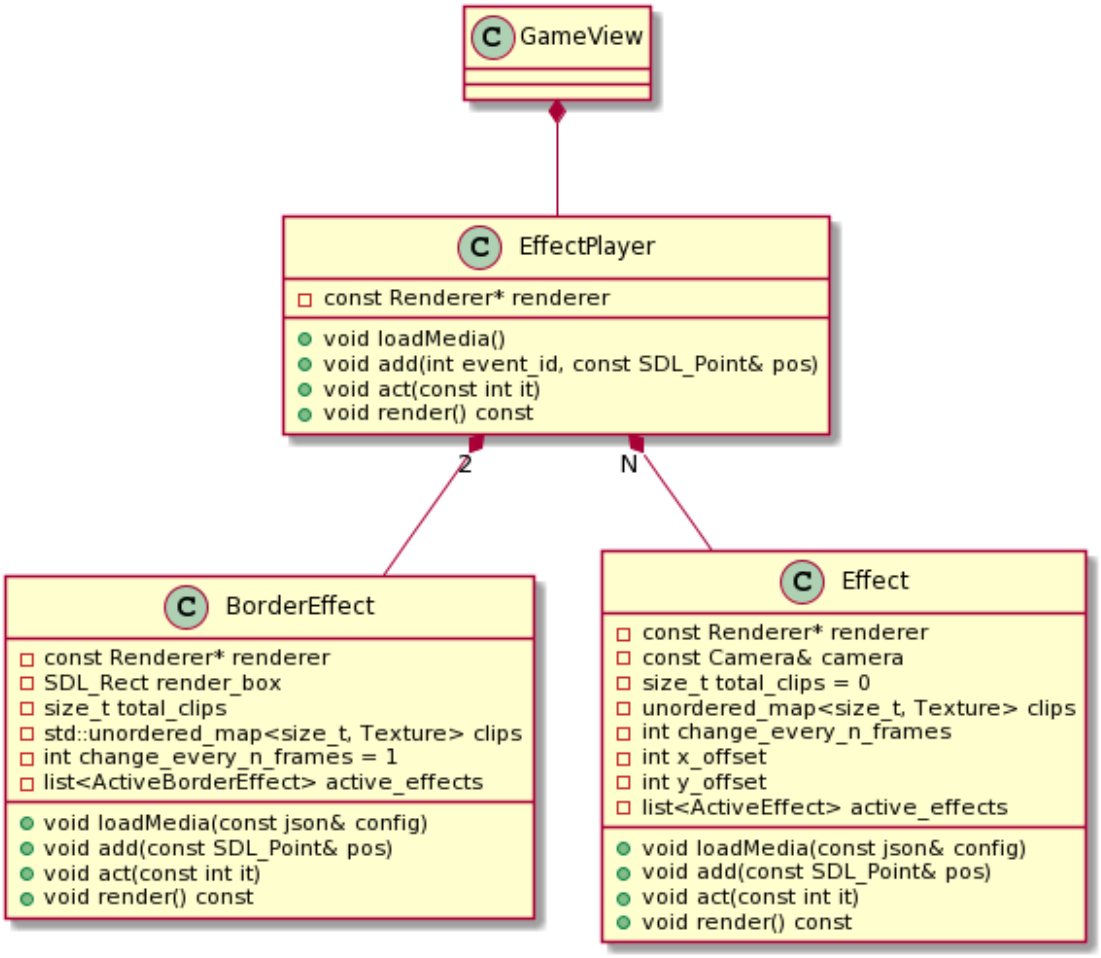

Figura 21: Reproductor de efectos especiales

5. Archivos

5.1. Archivos de configuración

Es necesario definir un formato para los archivos de configuración a utilizar, a fin de facilitar su parseo y procesamiento. Se manejaron dos opciones: **JSON** y **YAML**. Para decidir cual utilizar, se realizaron pruebas pequeñas con ambos formatos para finalmente decidirse por **JSON**, principalmente porque la librería utilizada fue muy fácil de comprender y utilizar, y porque **Tiled** (el editor de mapas utilizado) exportaba sus mapas en este formato, permitiéndonos unificar el mismo para todos los archivos utilizados.

Para encapsular su uso, definimos un *namespace* **JSON**, que contaría con la función *loadJsonFromFile*, y que se encargaría de realizar verificaciones pertinentes a la hora de manipular los distintos archivos. Este haría uso del método *loadJsonFile* de la librería utilizada.

Para su utilización, fue necesario implementar métodos de conversión para estructuras que no pertenecen a la STL, conversiones definidas en *json_container.h*.

5.2. Base de datos

Es necesario implementar un sistema de persistencia que garantice a los usuarios que su progreso en el juego no se perderá ante una desconexión, parte fundamental en un juego de este género. Con el fin de evitar tener en memoria los datos de todos los jugadores, inclusive aquellos que se encuentren desconectados, se decidió utilizar una estructura de dos archivos:

- *playerdata.cfg*: el primer archivo contiene los **datos de cada jugador**, donde estos son structs de tamaño fijo y constante.
- *playerinfo.cfg*: el segundo es un diccionario (map) que indexa al primero. Básicamente, asocia el nombre de un jugador con la posición (offset) de los datos del mismo en el primer archivo, a fin de (gracias al tamaño constante de los structs) poder acceder sólo a los datos necesarios. Como detalle de implementación, también guarda las contraseñas del usuario.

Para encapsular el manejo de los mismos, se diseñó la clase **Database**. En la inicialización de la misma, se chequea la existencia de ambos archivos: si existen, se prosigue con al ejecución normalmente. Si no existen, se crean los archivos binarios vacíos. El siguiente paso consiste en generar un diccionario en base al segundo archivo, cuya clave es el **nombre de usuario** del jugador y los valores son la contraseña y la posición (offset) de los datos del jugador en el primer archivo.

5.2.1. Estructura de datos del primer archivo

Para guardar los datos del jugador, se definió la estructura **CharacterCfg** con sus valores por defecto:

```
struct CharacterCfg {
Id map = 0;
int x_tile = 0, y_tile = 0;
char nickname[MAX_USERNAME_SIZE + 1] = {0};

Id race = 0;
Id kind = 0;
Id head_id = 0;
Id body_id = 0;

StateType state = ALIVE;
```

```
    EquipmentData equipment = {0};
    InventoryData inventory = {0};
    unsigned int bank_gold = 0;
    BankAccountData bank_account = {0};

    uint32_t health = 0;
    uint32_t mana = 0;
    uint32_t safe_gold = 0;
    uint32_t excess_gold = 0;
    uint32_t level = 0;
    uint32_t exp = 0;
    bool new_created = true;
};
```

Como se mencionó anteriormente, esta estructura tiene tamaño fijo y constante, que nos permite utilizar el segundo archivo como índice.

5.2.2. Estructura de datos del segundo archivo

Para el segundo archivo, se definió la estructura **PlayerInfo** donde se almacenan los siguientes datos:

```
    struct PlayerInfo {
    char username[MAX_USERNAME_SIZE + 1] = {0};
    char password[MAX_PASSWORD_SIZE + 1] = {0};
    size_t index;
}
```

5.2.3. Detalles de implementación

Para persistir los datos en ambos archivos, se utiliza **reinterpret_cast<char*>(&data)** para castear la estructura a **char ***, ya que la esctrutura que se definió es un *PDS(passive data structure)*, es decir que todos los miembros de la estructura tienen el mismo tamaño.

6. Protocolo

Para establecer cualquier comunicación, es necesario definir previamente un protocolo a respetar por ambas partes, con el objetivo de la correcta transmisión y recepción de información. En este caso, se decidió utilizar un protocolo binario muy sencillo que se detallará a continuación.

El primer byte de todo mensaje siempre será el **OPCODE**. Este opcode nos dirá que tipo de mensaje se está transmitiendo, para saber como proseguir enviando o recibiendo el mismo. Sus posibles valores son los siguientes, según sea el servidor o el cliente quien los use:

Valor	Descripción
0	Respuesta a solicitud
1	Mensaje
2	Broadcast
3	Evento

Cuadro 1: Valores del byte de opcode que envía el **servidor**

Valor	Descripción
128	Comando
129	Solicitud de sign-in
130	Solicitud de sign-up

Cuadro 2: Valores del byte de opcode que envía el **cliente**

Una vez que se recibe el opcode, cada tipo de mensaje proseguirá de forma distinta. Se detalla a continuación este formato para cada uno.

6.1. Respuesta a solicitud ($opcode = 0$)

Luego del opcode, sólo se agrega un byte más que dice que tipo de respuesta es. Este puede tomar los siguientes valores:

Valor	Descripción
0	Éxito
1	Nombre de usuario incorrecto
2	Contraseña incorrecta
3	Usuario ya conectado
4	Nombre de usuario existente

Cuadro 3: Tipos de respuesta

6.2. Mensaje ($opcode = 1$)

Luego del opcode, se agrega un byte de tipo, y dependiendo de este tipo se agregan más o menos bytes. Se detallan a continuación los tipos con sus formatos:

Valor	Descripción	Formato
0	Error	Longitud (4) + Mensaje (Longitud)
1	Información	Longitud (4) + Mensaje (Longitud)
2	Éxito	Longitud (4) + Mensaje (Longitud)
3	Lista de items	Longitud (4) + Lista (Longitud)
128	Mensaje privado	Longitud (4) + Transmisor (Longitud) + Longitud (4) + Mensaje (Longitud)
129	Mensaje general	Id del transmisor (1) + Longitud (4) + Transmisor (Longitud) + Longitud (4) + Mensaje (Longitud)

Cuadro 4: Tipos de mensaje y sus formatos

6.3. Broadcast (*opcode* = 2)

Con el broadcast se repite lo mismo: un siguiente byte de tipo, y distintos formatos dependiendo de este:

Valor	Descripción	Formato
0	Nueva entidad	Tipo de entidad (1) + Longitud (4) + Paquete (Longitud)
1	Actualización	Tipo de entidad (1) + Longitud (4) + Paquete (Longitud)
2	Eliminar entidad	Tipo de entidad (1) + ...

Cuadro 5: Tipos de broadcast y sus formatos

Como se puede ver, se envía un tipo de entidad y en base a este valor, el último broadcast *(Eliminar entidad)* varía su formato:

Entidad	Descripción	Formato
0	Player	Combinación inválida. No debería transmitirse este mensaje.
1	Character	Id (4)
2	Creature	Id (4)
3	Item	X (4) + Y (4)

Cuadro 6: Formatos para el broadcast *Eliminar entidad*

6.4. Evento (*opcode* = 3)

Para los eventos, luego del opcode se agrega un byte que especifica de que evento se trata, y luego se envía la posición donde ocurrió el mismo como X (4) + Y (4). Los tipos de evento son:

Valor	Descripción
0	Movimiento
1	Tirar un objeto
2	Tomar un objeto
3	Empezar a meditar
4	Subir de nivel
5	Muerte
6	Resurrección
7	Ataque general
8	Hechizo de explosión
9	Hechizo de curación
10	Curación de un sacerdote
11	Evasión de ataque
12	Daño
13	Hechizo de fuego
14	Hechizo de rayo
128	Recibir daño (sólo jugador principal)
129	Recibir curación (sólo jugador principal)

Cuadro 7: Tipos de evento

Se puede ver que se distinguen eventos que empiezan desde 0, a eventos que empiezan desde 128. Esto se debe a que en nuestro modelo, existen eventos que son broadcasteados a todos los jugadores, mientras que otros sólamente son enviados a un jugador.

6.5. Comando ($opcode = 128$)

Los comandos son más complejos, pues además del opcode, se envía un byte que identifica el comando, y luego cada comando en particular tendrá un formato distinto. Se especifican los mismos en la siguiente tabla:

Id	Comando	Formato
0	Empezar a mover arriba	-
1	Empezar a mover abajo	-
2	Empezar a mover a la izq.	-
3	Empezar a mover a la der.	-
4	Dejar de mover	-
5	Usar arma principal	Id objetivo (4)
6	Equipar objeto	Slot Inventario (1)
7	Desequipar objeto	Slot Equipamiento (1)
8	Tomar objeto	-
9	Tirar objeto	Slot Inventario (1) + Cantidad (4)
10	Meditar	-
11	Resucitar	-
12	Help del NPC	X (4) + Y (4)
13	Resucitar (sacerdote)	X (4) + Y (4)
14	Curar (sacerdote)	X (4) + Y (4)
15	Listar (npc)	X (4) + Y (4)
16	Depositar (banquero)	X (4) + Y (4) + Slot Inventario (1) + Cantidad (4)
17	Depositar oro (banquero)	X (4) + Y (4) + Cantidad (4)
18	Retirar (banquero)	X (4) + Y (4) + Id del objeto (4) + Cantidad (4)
19	Retirar oro (banquero)	X (4) + Y (4) + Cantidad (4)
20	Comprar (comerciante)	X (4) + Y (4) + Id del objeto (4) + Cantidad (4)
21	Vender (comerciante)	X (4) + Y (4) + Slot Inventario (1) + Cantidad (4)
22	Listar jugadores conectados	-
23	Mensaje general	Longitud (4) + Mensaje (Longitud)
24	Mensaje privado	Long. nick (4) + + Nick (Long. nick) + Longitud (4) + Mensaje (Longitud)
25	Teletransportar	X (4) + Y (4) + Id del mapa (4)

Cuadro 8: Tipos de mensaje y sus formatos

6.6. Solicitud de sign-in (*opcode* = 129)

La solicitud de sign-in tiene un formato muy sencillo. Al mensaje de opcode, se le agregan: Long. nick (4) + Nombre de usuario (Long. nick) + Long. pass (4) + Contraseña (Long. pass).

6.7. Solicitud de sign-up (*opcode* = 130)

La solicitud de sign-un también tiene un formato muy sencillo, similar al de sign-in. Al mensaje de opcode, se le agregan: Long. nick (4) + Nombre de usuario (Long. nick) + Long. pass (4) + Contraseña (Long. pass) + Raza (4) + Clase (4) + Cabeza (4) + Cuerpo (4).

7. Serialización

Para serializar las estructuras de datos a enviar, se utilizó la herramienta **MsgPack**. Una facilidad que tuvimos fue que la librería utilizada para el manejo de archivos (JSON) incluía métodos para convertir un objeto json en un vector de chars serializado con MsgPack, y viceversa, por lo que este proceso se nos hizo muy sencillo y no fue necesario agregar lógica adicional.