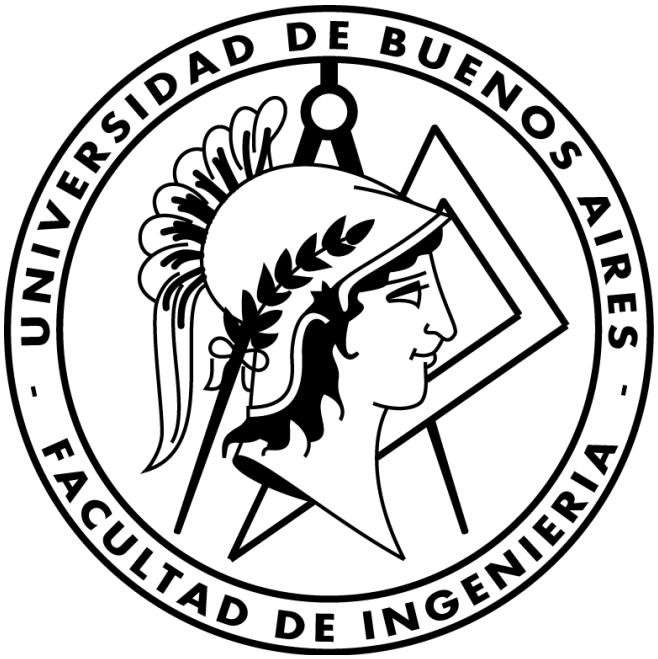
Trabajo Práctico Final

Remake de juego Worms en C++

75.42 - Taller de Programación

Documentación Técnica



Integrantes:

ALVAREZ WINDEY ARIEL JUSTO – 97893

DIZ GONZALO – 98546

ROBLES GABRIEL – 95897

Índice

1. Requerimientos de Software

<table>
<tr><td>Sistema Operativo</td><td>Linux</td></tr>
<tr><td>Bibliotecas (Necesarias para compilar, desarrollar, probar y depurar el programa)</td><td>
<table>
<tr><td>Nombre</td><td>Versión mínima</td></tr>
<tr><td>libsdl2-dev</td><td>2.0.8</td></tr>
<tr><td>libsdl2-image-dev</td><td>2.0.3</td></tr>
<tr><td>libsdl2-ttf-dev</td><td>2.0.14</td></tr>
<tr><td>libsdl2-mixer-dev</td><td>2.0.2</td></tr>
<tr><td>qt5-default</td><td>5.9.5</td></tr>
<tr><td>cmake</td><td>3.10.2</td></tr>
<tr><td>box2d</td><td>2.3.1</td></tr>
<tr><td>yaml-cpp (https://github.com/jbeder/yaml-cpp)</td><td>0.6.2</td></tr>
<tr><td>gcc</td><td>7.3.0</td></tr>
</table>
</td></tr>
</table>

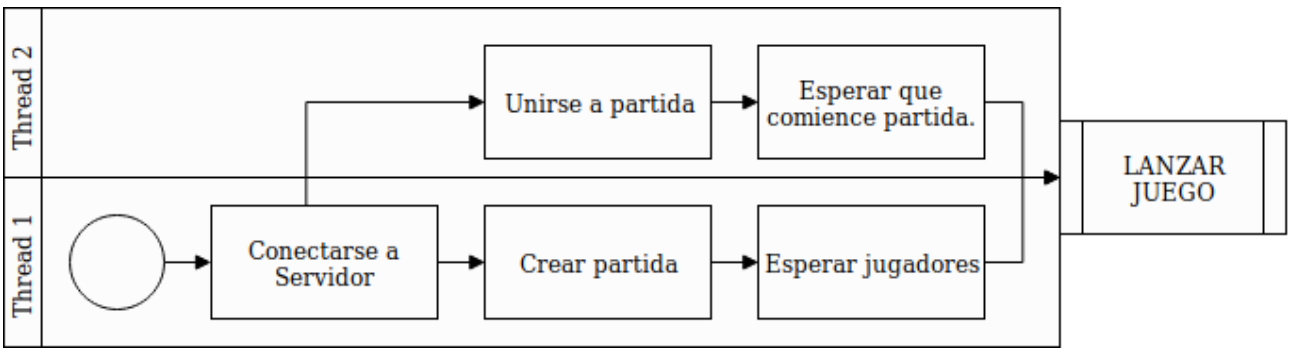
2. Descripción General

El proyecto se divide en tres aplicaciones independientes: Cliente (1), Servidor (2) y Editor (3).

2.1 Cliente

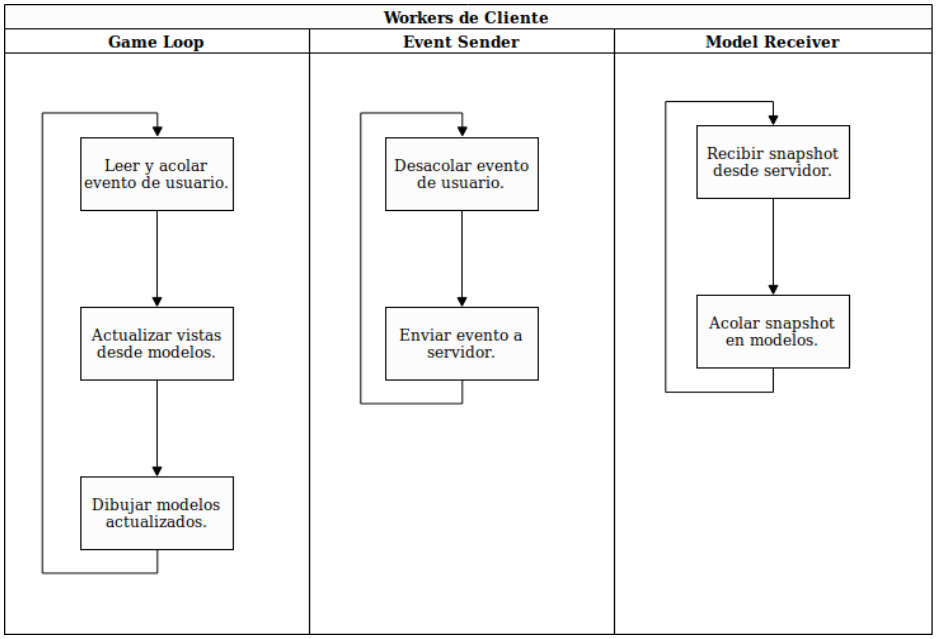
El cliente puede entenderse en dos partes independientes que conforman una única aplicación: el Lobby y el Juego.

El Lobby, mediante una interfáz gráfica desarrollada en QT5, permite conectarse a un servidor y crear o unirse a partidas que éste administre.



Esquema 1: Flujos posibles para casos exitosos en lanzar un juego desde el Lobby.

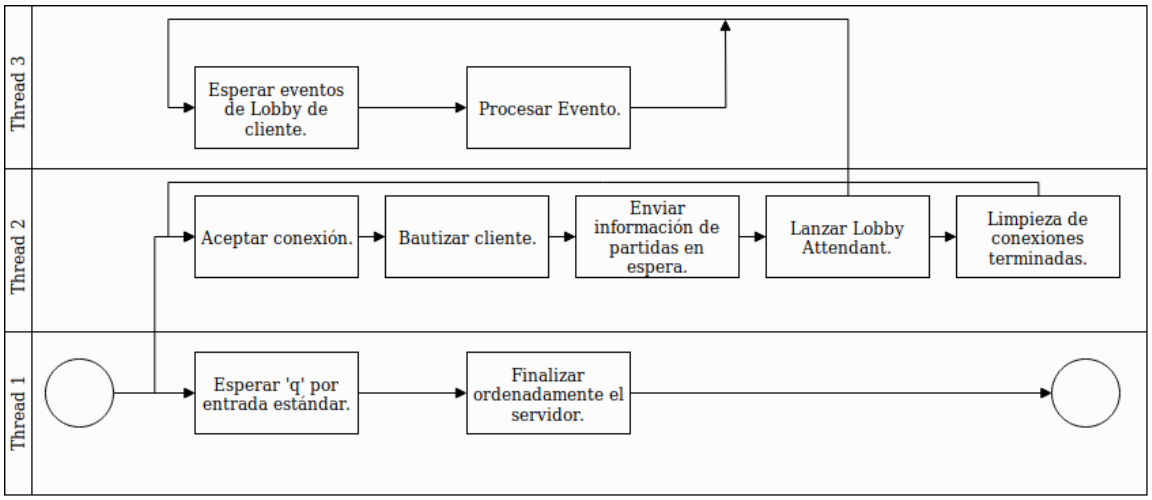
El Juego, lanzado desde el Lobby de cada cliente, puede representarse mediante 3 módulos que a su vez significan 3 hilos (threads) de ejecución diferentes: Game Loop, Event Sender y Model Receiver. Estos tres hilos trabajan en conjunto para hacer funcionar el juego.



Esquema 2: Módulos (e hilos) básicos del juego para un Cliente.

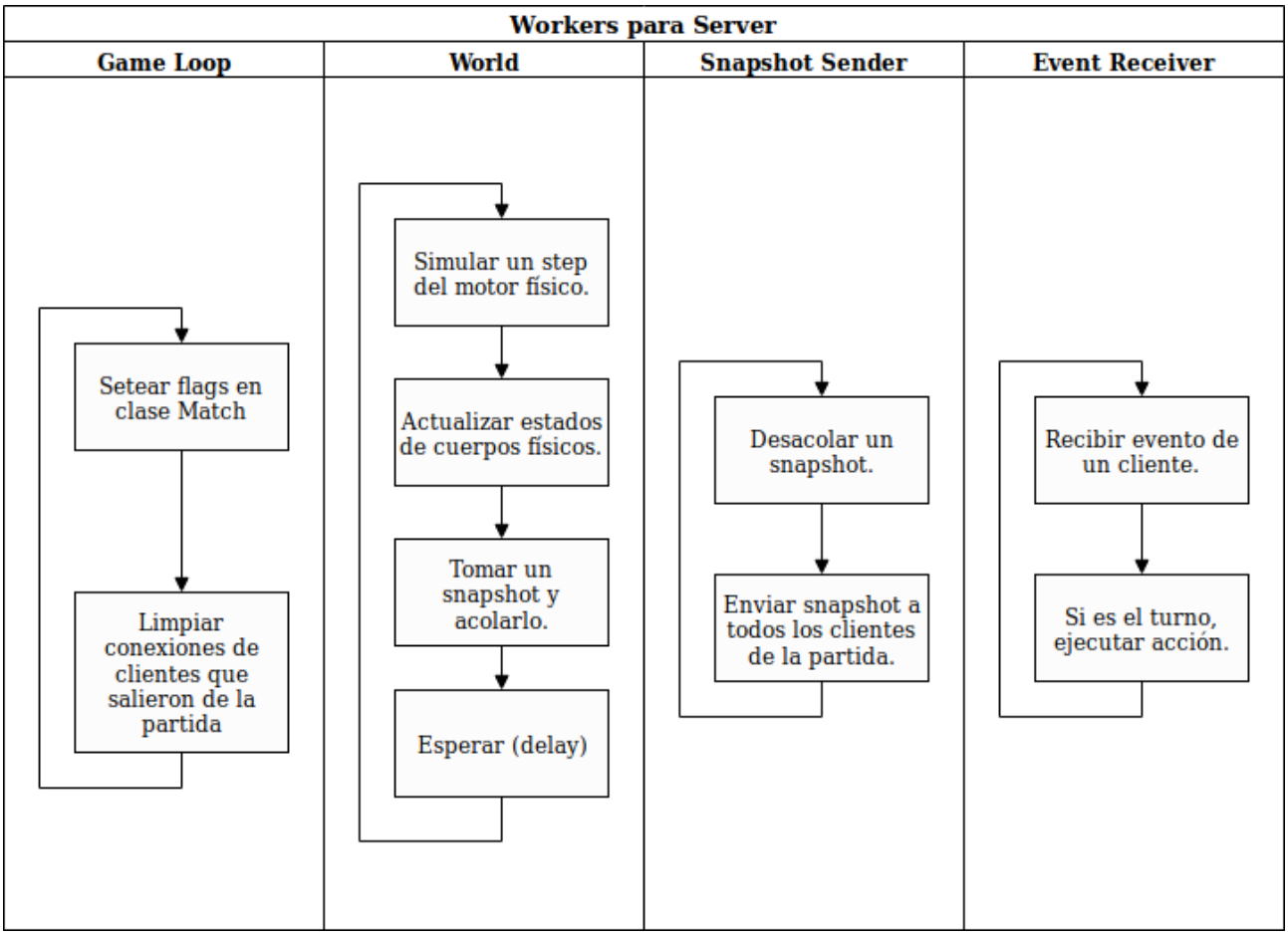
2.2 Servidor

El servidor, al igual que el cliente, puede inicialmente entenderse en dos partes independientes: el Server Lobby y el Juego.



Esquema 3: Modulos (e hilos) en el servidor para administrar el Lobby.

Como puede observarse en el Esquema 3, el servidor en su bloque de Lobby tendrá como mínimo 2 hilos, uno para esperar una ‘q’ por entrada estándar, para finalizarse, dado que no tiene interfáz gráfica y su control es por terminal. El otro hilo es el Servidor propiamente dicho, que acepta y administra conexiones de clientes. Luego, por cada cliente que se conecte se lanzará un hilo adicional, los llamados “Lobby Attendant’s”, que esperan eventos del tipo Lobby (como por ejemplo, que un cliente se va del Lobby, que crea una partida, que se une a una partida, etc.).

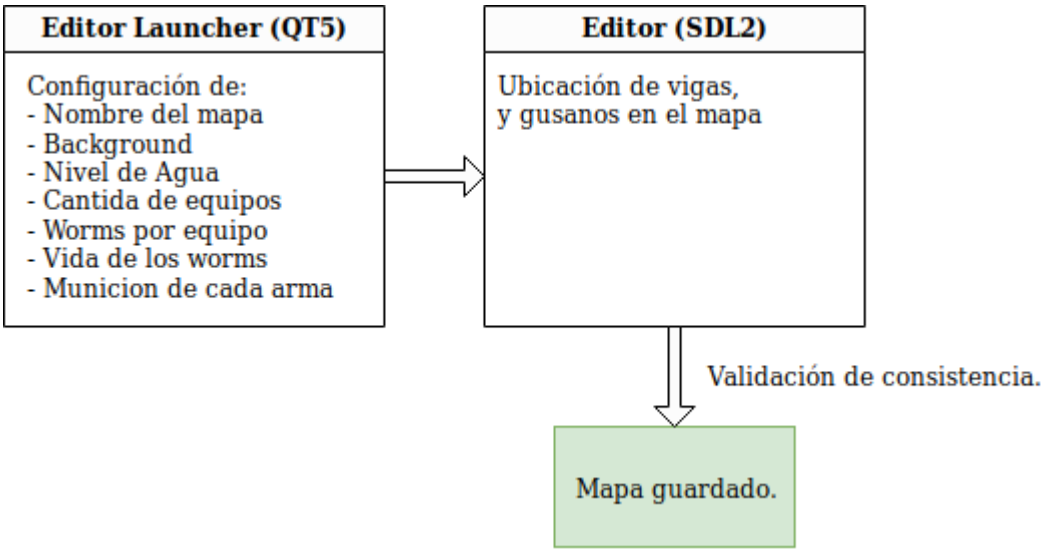


Esquema 4: Módulos (e hilos) del server para administrar un Juego.

En el Esquema 4 se pueden observar los hilos que trabajan en conjunto para crear un servicio de Juego. Habrá un mínimo de 4 hilos. Habrá tantos hilos de Event Receiver como clientes participen en la partida, pues se necesita escuchar eventos de ellos en hilos por separado ya que el receive() de socket es una función bloqueante.

2.3 Editor

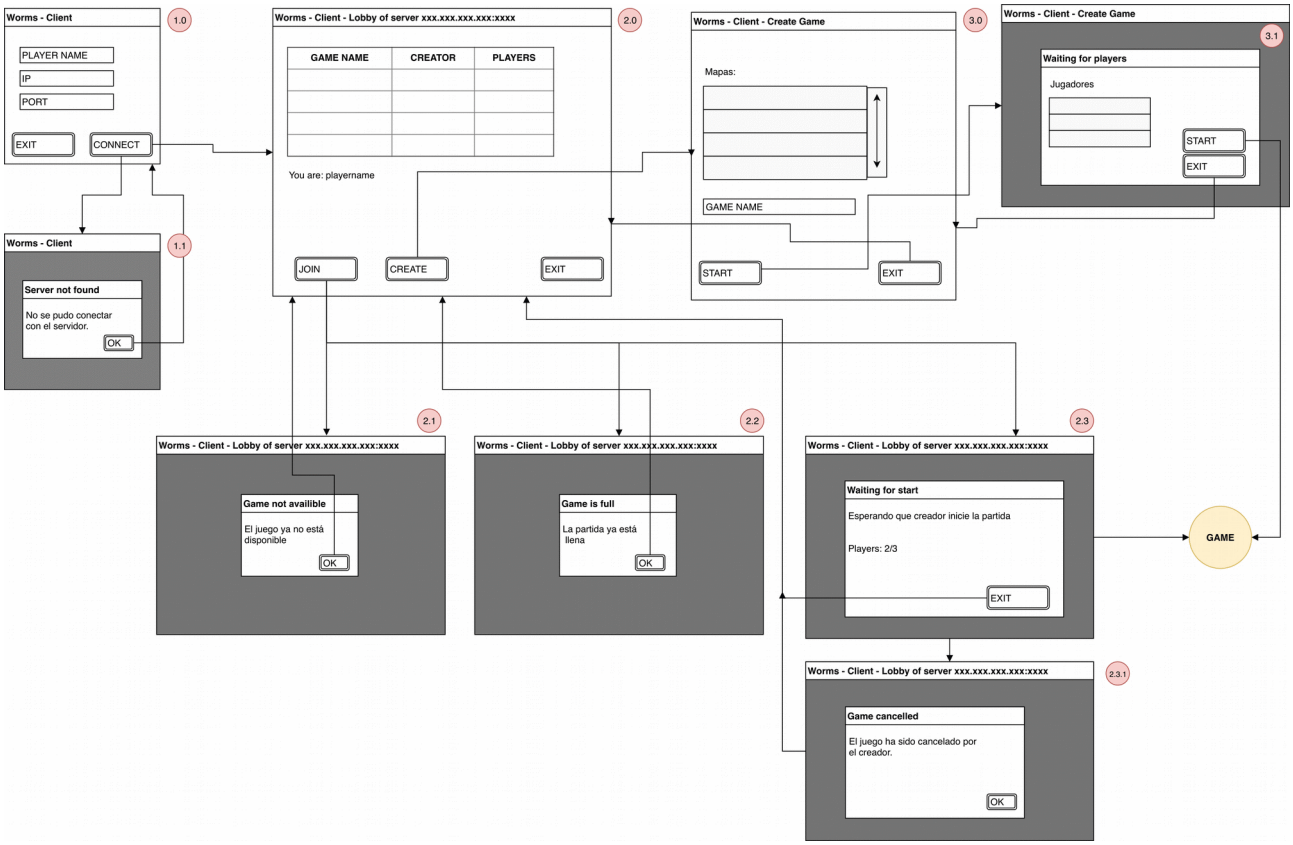
A diferencia del cliente y el servidor, el editor no tiene conectividad por socket con otra aplicación. Funciona en forma independiente. Sin embargo, comparte similitudes con el Cliente: Una parte funciona con QT (módulo llamado editor-launcher) y otra parte con SDL (módulo llamado editor). Corre en un único hilo de ejecución. En el módulo editor-launcher se configuran aspectos como el background del mapa, el nivel del agua, la cantidad de equipos y gusanos, la vida de los mismos y las municiones de las armas. Cuando todo lo anterior fué configurado, se procede al editor, que permite colocar vigas y gusanos en un mapa.



Esquema 5: Modulos del aplicativo del Editor.

3. Descripción detallada de módulos

3.1 Lobby de Cliente



Esquema 6: Diagrama de ventanas para el Lobby del Cliente.

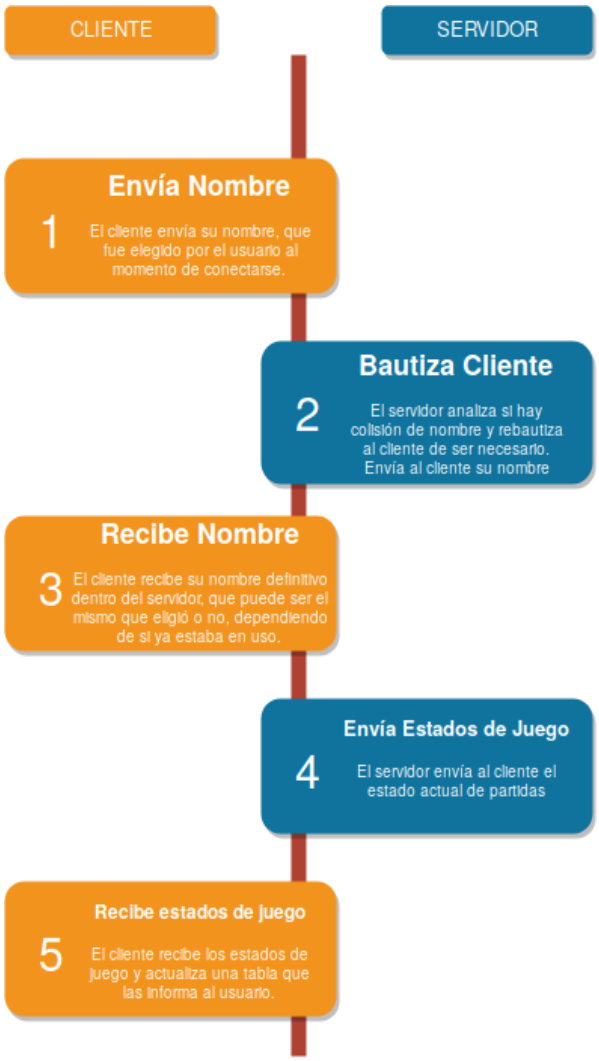
Clases intervinientes y sus responsabilidades:

Clase	*Responsabilidad*	*Métodos*
ClientLobby	Lanzar la ventana del Lobby, desarrollada con QT5. Conecta eventos de botones con métodos. Obtiene datos de cajas de texto. Permite conectarse a un servidor. Administra la interacción del usuario con el Lobby. Desde esta clase se lanza una partida de juego (sólo para el creador de la partida).	Conectar Eventos Limpiar Cajas de Texto Conectar con Servidor Crear partida en espera Unirse a partida Elegir mapa Comenzar partida
WaitingMatch	Hereda de Thread. Es un objeto vivo. Su función es esperar un evento del servidor que indique que el creador de la partida decidió comenzarla o bien cancelarla. Desde esta clase se lanza una partida de juego (sólo para jugadores no-creadores de la partida). Ver Thread 2 de Esquema 1.	Comenzar Detenerse ¿Está corriendo?
ClientGame	Lanzar una partida de juego. Contiene el Game	Comenzar Juego

	Loop y todos los hilos apreciables en el Esquema 2. Mientras este objeto esté en el método startGame(), el cliente verá una ventana de SDL con la partida en curso.	Loop de Juego Remover archivos temporales
ClientSettings	Clase global de configuración del Cliente. Permite configurar aspectos como la resolución de la pantalla de juego, activar o desactivar los efectos de sonido, cambiar las teclas de mando o setear el juego en modo ventana o full screen.	Setters

Archivos y Protocolos:

En el momento que el cliente se conecta con el servidor (Esquema 6, pasaje de ventana 1.0 a 2.0) se ejecuta el siguiente protocolo de mensajes:



Esquema 7: Protocolo de mensajes tras una conexión de cliente al servidor.

En el paso 4 del Esquema 7, los estados del juego se envían a partir de un objeto YAML, con el siguiente formato:

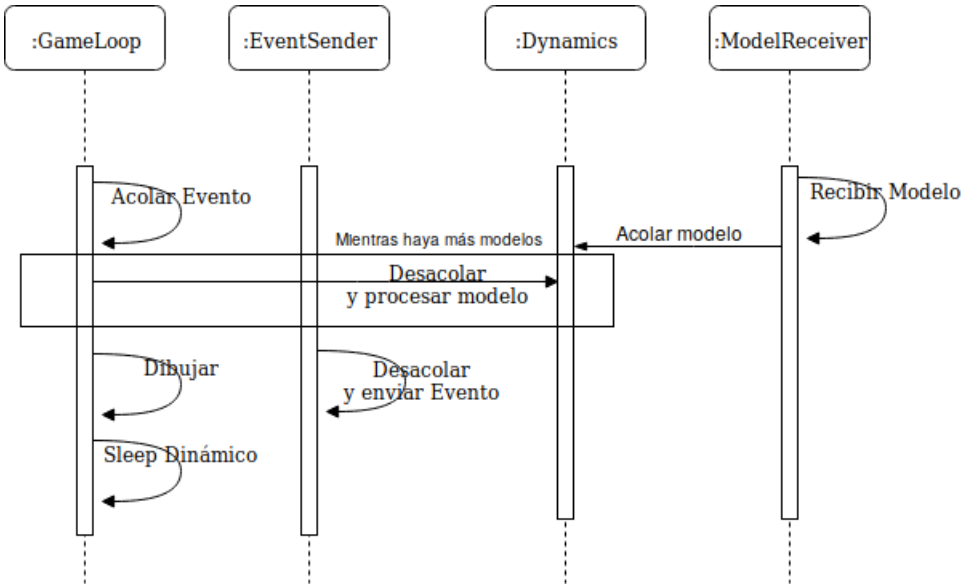
```
waiting_games:
    - match_name: juego1
      creator: gabriel
      required_players: 3
      joined_players: 1
    - match_name: juego2
      creator: ariel
      required_players: 2
      joined_players: 1
```

3.2 Juego de Cliente

Clases intervinientes:

Clase	*Responsabilidad*	*Métodos*
ClientGame	Lanzar una partida de juego. Contiene el Game Loop y todos los hilos apreciables en el Esquema 2. Mientras este objeto esté en el método startGame(), el cliente verá una ventana de SDL con la partida en curso.	Comenzar Juego Loop de Juego Remover archivos temporales
ClientConfiguration	Administra los parámetros configurables de una partida propio de cada cliente. Los mismos son: qué información mostrar de los gusanos (nombre, vida), tiempo de detonación de las bombas, vista de las vidas totales de los equipos,etc.	Handler de eventos Obtener arma seleccionada Obtener angulo de mira Obtener potencia de tiro
EventSender	Hereda de Thread. Desacola eventos registrados del usuario y los envía por socket al servidor para que ejecute la acción si así lo decide.	Comenzar Detenerse ¿Está corriendo?
ModelReceiver	Hereda de Thread. Recibe modelos por socket (son los snapshots que toma el cliente) y los acola en la clase Dynamics. Es el primero en detectar que la partida terminó.	Comenzar Detenerse ¿Está corriendo?
Dynamics	Desacola modelos (snapshots) que fueron acolados por el ModelReceiver y mantiene actualizada la última foto del estado de los objetos dinámicos. Conoce la posición y el estado actualizado de los worms y proyectiles. También conoce datos del estado de juego (cómo quién tiene el turno, el cronómetro, la vida total	Agregar modelo Obtener worms Obtener proyectiles Obtener estado de juego Obtener tiempo de turno Obtener worm protagónico Obtener último modelo

	de los equipos, etc.).	¿Terminó el juego? ¿Mi equipo perdió? Obtener equipo ganador



Esquema 8: Comunicación entre workers de un Juego de Cliente.

El Game Loop y Event Sender se comunican mediante una cola bloqueante en la cual el primero acola eventos percibidos del usuario (una tecla o mouse) y el segundo lo desacola y lo envía al servidor.

El Game Loop, antes de dibujar, realiza un update de los estados de los cuerpos dinámicos (worms y proyectiles), como así también del estado de juego (cronómetro, vida de equipos, etc). Este update lo hace en un ciclo que dura hasta que no haya más modelos acolados en la clase Dynamics. Luego de este ciclo, se procede a dibujar los cuerpos (que estan actualizados al último snapshot recibido).

Por otra parte, los modelos (snapshots) se acolan en Dynamics desde el hilo ModelReceiver. Se asegura que la velocidad en que se desacolan y procesan los modelos desde el Game Loop es mayor que la velocidad en que dichos modelos ingresan a la cola de Dynamics.

Este sistema permite que el cliente siempre invierta su tiempo en dibujar la última snapshot recibida.

El sleep dinámico que realiza el Game Loop permite que se dibuje lo mas cercanamente posible a razón de 60 veces por segundo.

La cola de modelos que maneja la clase Dynamics es estándar. No necesita ser bloqueante, y de serlo, podría llegar a trabjar el hilo del Game Loop, arruinando la experiencia del usuario. Si llegase a ocurrir que la conexión de socket sea lenta, entonces el GameLoop dibujará el mismo modelo una y otra vez, hasta que se acole uno nuevo en Dynamics.

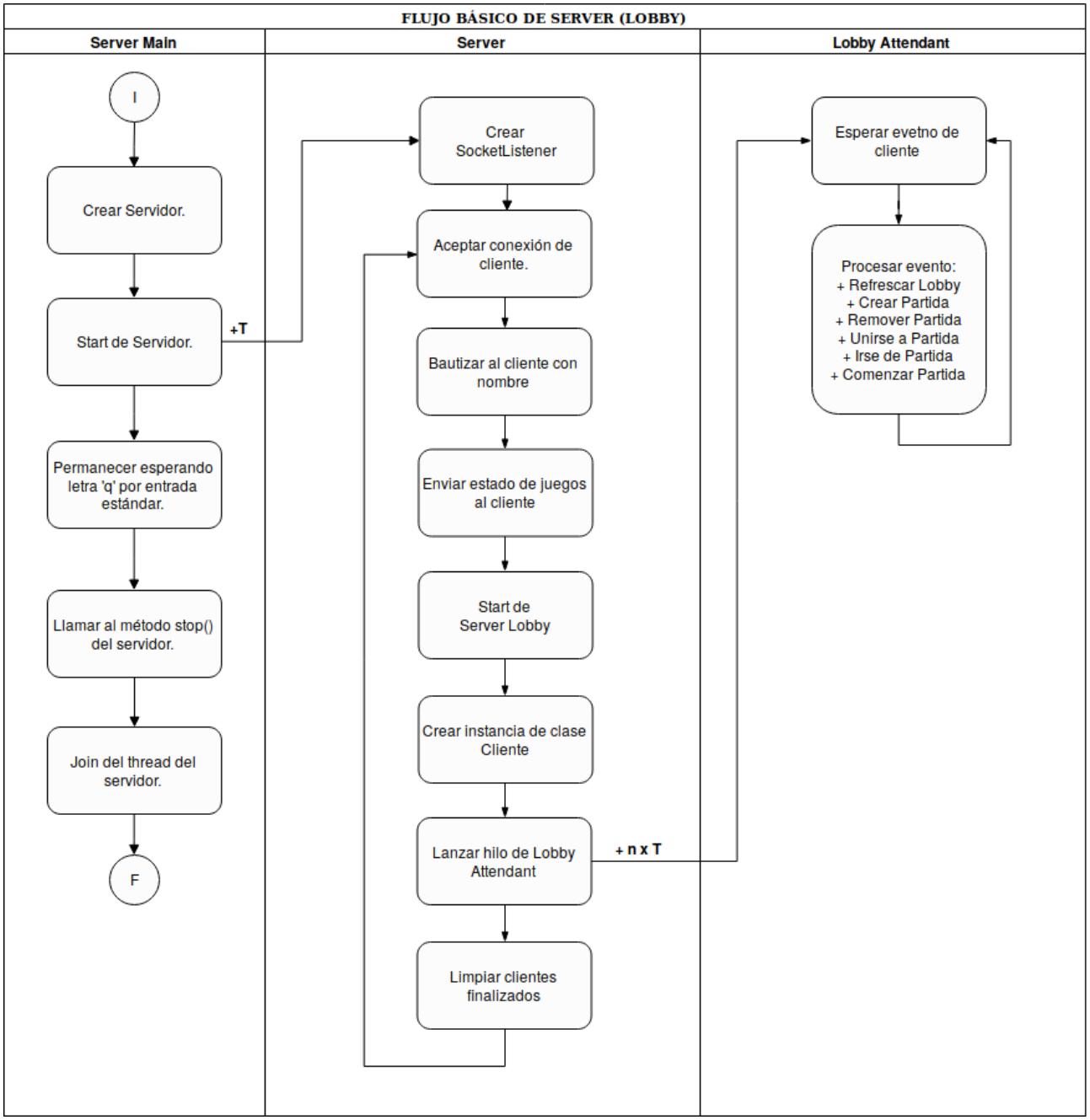
Al finalizar el juego, el cliente imprimirá por salida estándar valores de renderCount y de updateCount. El primero contó la cantidad de veces que se dibujaron modelos. El segundo contó la cantidad de veces que se desacolaron y procesaron modelos.

3.3 Lobby de Servidor

Clases intervinientes:

Clase	*Responsabilidad*	*Métodos*
Server	Hereda de Thread. Aceptar conexiones de clientes y ejecutar el protocolo del Esquema 7. Lanzar un thread de LobbyAttendant por cada cliente conectado. Limpiar conexiones de clientes finalizados.	Comenzar Detener Limpiar Clientes Buscar nombre libre ¿Está corriendo?
LobbyAttendant	Hereda de Thread. Espera eventos de un cliente en el Lobby (actualizar estado de juegos, crear partida, salir de lobby, etc.).	Comenzar Detener Procesar Evento Refrescar Lobby Crear Partida Remover Partida Unirse a Partida Comenzar Partida
ProtectedWaitingGames	Monitor de las partidas en espera, que significan una race condition ya que son accedidas y modificadas desde mas de un hilo de ejecución. Aloja las partidas en espera en un mapa, y permite interactuar con ellas como así removerlas o agregarlas.	Agregar partida Remover partida Obtener nombre de partida Agregar jugador a partida Remover jugador de partida Notificar a todos comienzo Notificar a todos cancelación Comenzar partida Esperar a que termine partida
Client	Encapsula el socket de conexión con un cliente. Permite interactuar con el cliente a través del protocolo. Guarda estados del mismo dentro del servidor (si está en una partida, su nombre, etc.)	Recibir evento Obtener nombre Enviar respuesta Enviar mapa de juego Recibir mapa de juego Enviar estado de juegos ¿Salió del server?

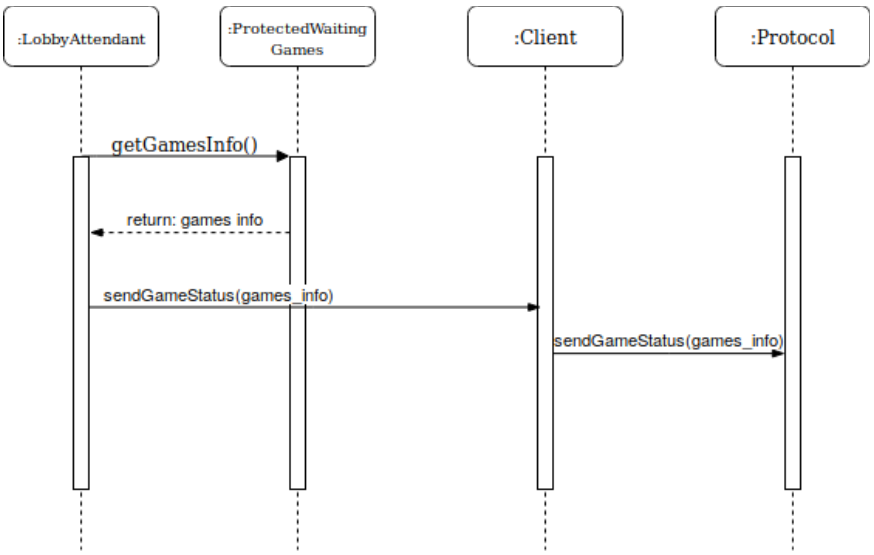
ServerGame	Encapsula la lógica de servidor de Juego para una partida. Contiene el game loop y ejecutará el motor físico y todos los threads workers del Esquema 4.	Comenzar juego Detener Game Loop Limpiar clientes finalizados Remover archivos temporales



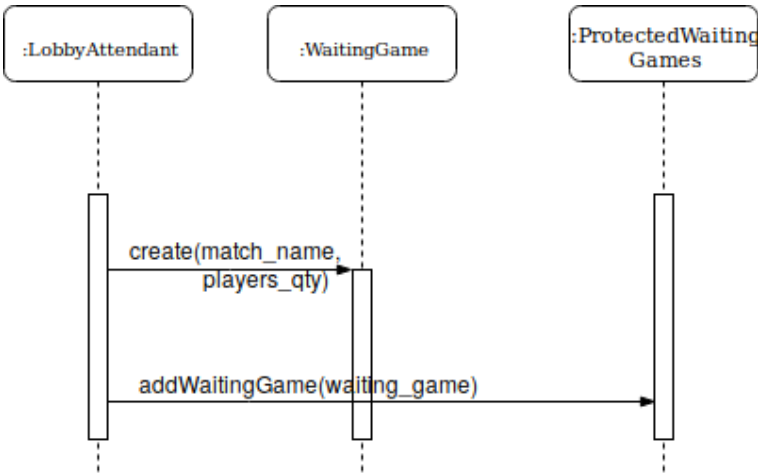
Esquema 9: Flujo básico del lobby del servidor.

Como puede observarse, el lobby del servidor tiene un mínimo de 2 Threads: el hilo principal o main y el hilo de servidor. Luego, por cada cliente conectado, se lanza un hilo de Lobby Attendant.

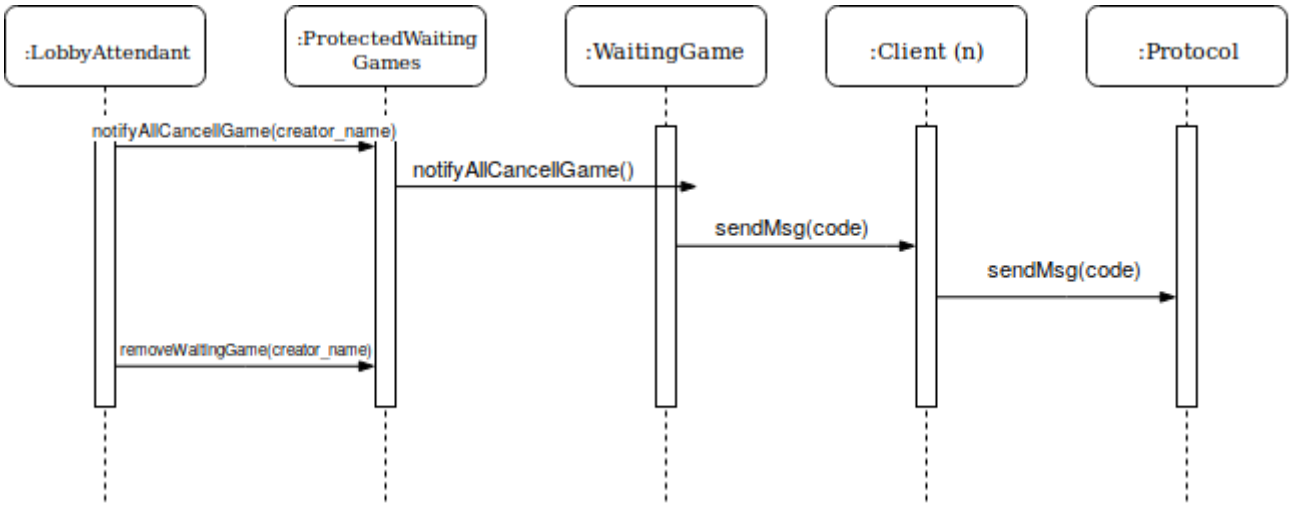
El detalle de los flujos disparados por cada evento recibido en Lobby Attendant puede observarse en los Esquemas 10, 11, 12, 13 y 14.



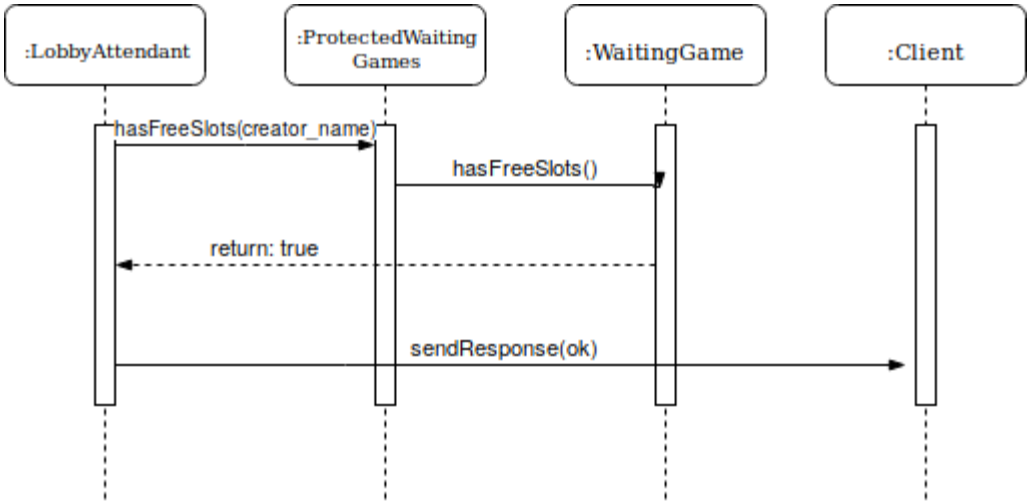
Esquema 10: Secuencia en server lobby tras evento de refresh lobby.



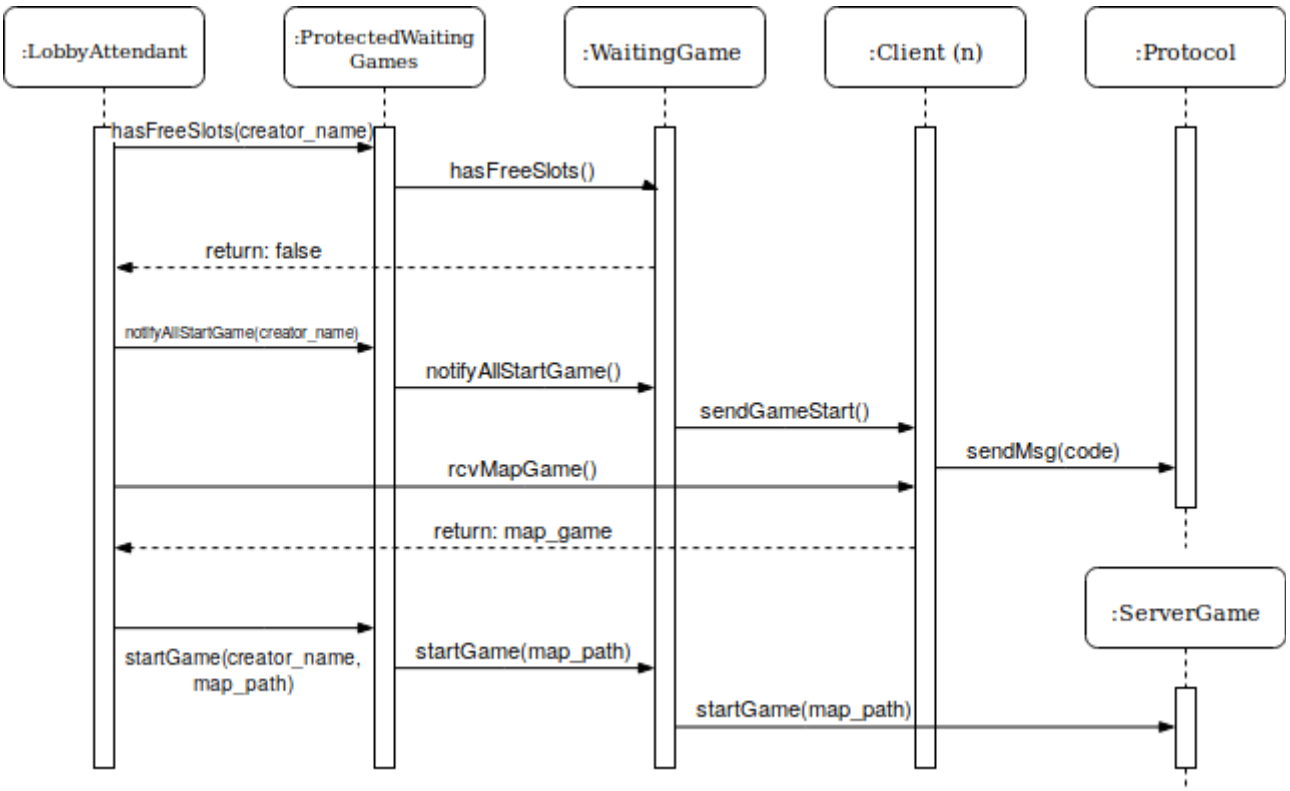
Esquema 11: Secuencia para evento create match.



Esquema 12: Secuencia para evento remove match.



Esquema 13: Secuencia para evento join match exitoso.



Esquema 14: Secuencia de evento start match, enviado por el creador de una partida.

3.4 Juego de Servidor

Clases intervinientes

Clase	*Responsabilidad*	*Métodos*
ServerGame	Encapsula la lógica de servidor de Juego para una partida. Contiene el game loop y ejecutará el motor físico y todos los threads workers del Esquema 4.	Comenzar juego Detener Game Loop Limpiar clientes finalizados Remover archivos temporales
EventReceiver	Hereda de Thread. Recibe eventos de un cliente por socket, y si es el turno de ese cliente en la partida, ejecuta la acción correspondiente.	Comenzar Detener ¿Es evento de salir del juego? Obtener id de cliente
SnapshotSender	Hereda de Thread. Desacola un snapshot (modelo) de una cola bloqueante y se lo envía a	Comenzar Detener

	cada uno de los clientes de la partida.	Enviar snapshot Enviar última snapshot
Snapshot	Encapsula lo que denominamos snapshot, que no es mas que una “foto” que guarda el estado actual de los objetos dinámicos del mundo (worms, proyectiles y estado de juego).	Actualizar equipos Actualizar proyectiles Actualizar estado de juego Obtener snapshot
Team	Encapsula a un equipo, jugador de una partida. Contiene referencias a los Worms y a su inventario.	Agregar miembro Inicializar inventario Obtener inventario Obtener ID de equipo ¿Tiene algun miembro vivo? Obtener vida total Matar a todos los worms ¿Tiene municiones de … ? Descontar munición de ...
Match	Encapsula los datos y la lógica de una partida: Equipos, Worms, administra los turnos, y sabe cuando la partida termina y si hay un ganador o es un empate.	Comenzar partida Avanzar turno Otener equipo en turno Obtener worm protagónico Remover equipos muertos Remover worms muertos ¿Terminó la partida? Remover equipo Obtener vida total de un equipo Obtener equipo ganador Actualizar partida dado reloj Obtener tiempo restante de turno
World	Hereda de Thread. Encapsula los objetos físicos que son afectados por la simulación del motor: gusanos, vigas, proyectiles y agua. Tiene un reloj que cuenta en segundos. Su loop principal genera setps de la simulación física a razón de 60 steps por segundo.	Comenzar Detener Inicializar mundo Actualizar cuerpos Obtener Worms Obtener Equipos Obtener viento Ejecutar acción Obtener tiempo en segundos ¿Hay gusanos moviéndose? ¿Hay proyectiles vivos? ¿Hay gusanos afectados por explosión? ¿El gusano … sufrió daño? ¿El gusano … efectuó un disparo?

WeaponManager	Administras las armas durante el juego. Controla los disparos efectuados por los gusanos y la actualización de los estados de las distintas armas/proyectiles que están en el juego.	Atender un evento de disparo de un determinado gusano. Actualizar el estado de las armas en cada step.

El siguiente listado corresponde a clases del Servidor utilizadas durante una partida que sirven para las simulaciones físicas:

WorldPhysic
ContactListener
ExplosionManager
QueryCallback
RayCastClosestCallBack

El siguiente listado corresponde a clases del Servidor utilizadas durante una partida que encapsulan la lógica de la física de diversas entidades:

Girder
Wall
Wind
Water
Worm

El siguiente listado corresponde a nombres de clases utilizadas para modelar las armas del juego:

Weapon
AirStrike
Bat
Bazooka
Dynamite
Fragment
Grenade
Missil

Mortar
RedGrenade
Teleportation

3.5 Editor

Clases intervinientes:

Clase	*Responsabilidad*	*Métodos*
EditorLauncher	Encapsula la ventana hecha en QT5 del launcher de editor. Alberga la lógica de interacción con los botones, text boxes y demás, necesarios para configurar el nombre del mapa, el background, las municiones, etc.	Conectar eventos Elegir background Lanzar editor Cargar y editar mapa
Editor	Dispara la ventana SDL donde el usuario puede editar el mapa, insertando vigas y worms. Posee un “game loop” similar al del Cliente de Juego. Encapsula el renderer y la cámara.	Comenzar
MapState	Permite guardar distintos estados del mapa, necesario para las funcionalidades de ctrl+z y ctrl+y (undo y redo) durante la edición del mapa.	Agregar viga Agregar worm Obtener vigas Obtener worms Dibujar
MapGame	Encapsula el mapa actual que se esta dibujando en pantalla. Hace uso de MapState.	Agregar inventario a equipos Agregar viga corta Agregar viga larga Agregar worm Setear estado previo Setear proximo estado Validar mapa Guardar

4. Módulos de uso general

4.1 Descripción de Snapshots – YAML

Los snapshots, que no son otra cosa que fotos con el estado actualizado de las cosas dinámicas, no son otra cosa que objetos YAML, que se parsean y/o se emiten tanto en client-side como en server-side con la biblioteca yaml-cpp. Los mismos, tienen la siguiente estructura:

```
worms_teams:
  1:
    worms:
      - id: 1
        health: 0
        x: 1293
        y: 260
        status:
          grounded: 1
          falling: 0
          mirrored: 0
          walking: 0
          inclination: 0
          affected_by_explosion: 0
          angle_direction: 0
    inventory:
      0:
        supplies: 10
      1:
        supplies: 10
      2:
        supplies: 10
      3:
        supplies: 10
      4:
        supplies: 10
      5:
        supplies: 10
      6:
        supplies: 10
      7:
        supplies: 10
      8:
        supplies: 10
      9:
        supplies: 10
  2:
    worms:
      - id: 2
        health: 200
        x: 977
        y: 497
        status:
          grounded: 1
          falling: 0
          mirrored: 0
          walking: 0
          inclination: 0
          affected_by_explosion: 0
          angle_direction: 180
    inventory:
      0:
        supplies: 10
      1:
        supplies: 10
      2:
        supplies: 10
      3:
        supplies: 10
      4:
        supplies: 10
      5:
        supplies: 10
      6:
        supplies: 10
      7:
        supplies: 10
      8:
        supplies: 10
      9:
        supplies: 10
projectiles: []
game_status:
  teams_health:
    1: 0
    2: 1000
  wind_force: 3
  protagonic_worm: 3
  turn_timeleft: 46
  finished: 1
```

El nodo **worms_teams** contiene nodos con los id de los equipos que participan de la partida, que a su vez, contienen nodos llamados ‘worms’, que listan la información de todos los gusanos, e ‘inventory’, que lista la munición de cada weapon.

El nodo **proyectiles** listará todos los proyectiles que haya en el mapa, brindando información como id, tipo, coordenadas x e y, cuenta regresiva de explosión, radio de explosión, si se está moviendo y en qué ángulo lo háce.

El nodo **game_status** brinda información general del juego, como la vida total de los equipos, la fuerza del viento, el id del worm protagónico del turno, el tiempo restante del turno y un booleano que indica si la partida finalizó.

4.2 Descripción de los mapas de juego – YAML

Los mapas de juego, generados por el editor, también son archivos de texto YAML. Los mismos tienen la siguiente estructura:

```
static:
  background:
    file: background.png
    display: expanded
  water_level: 300
  teams_amount: 2
  worms_health: 200
  init_inventory:
    0:
      item_name: Bazooka
      supplies: 10
    1:
      item_name: Mortar
      supplies: 10
    2:
      item_name: Cluster
      supplies: 10
    4:
      item_name: Banana
      supplies: 10
    3:
      item_name: Grenade
      supplies: 10
    5:
      item_name: Holy bomb
      supplies: 10
    7:
      item_name: Dynamite
      supplies: 10
    6:
      item_name: Air Strike
      supplies: 10
    9:
      item_name: Teleport
      supplies: 10
    8:
      item_name: Bat
      supplies: 10
  short_girders:
    - id: 1
      x: 1028
      y: 573
      angle: 0
    - id: 2
      x: 1302
      y: 711
      angle: 0
  long_girders:
    - id: 1
      x: 836
      y: 721
      angle: 0
    - id: 2
      x: 1676
      y: 712
      angle: 0
  max_worms: 7
```

```
dynamic:
  worms_teams:
    1:
      worms:
        - id: 1
          name: Worm 1
          health: 200
          x: 1020
          y: 549
          sight_angle: 0
          status:
            grounded: 0
            falling: 1
            mirrored: 0
            walking: 0
        - id: 2
          name: Worm 2
          health: 200
          x: 1348
          y: 411
          sight_angle: 0
          status:
            grounded: 0
            falling: 1
            mirrored: 0
            walking: 0
      inventory:
        0:
          item_name: Bazooka
          supplies: 10
        1:
          item_name: Mortar
          supplies: 10
        2:
          item_name: Cluster
          supplies: 10
        4:
          item_name: Banana
          supplies: 10
        3:
          item_name: Grenade
          supplies: 10
        5:
          item_name: Holy bomb
          supplies: 10
        7:
          item_name: Dynamite
          supplies: 10
        6:
          item_name: Air Strike
          supplies: 10
        9:
          item_name: Teleport
          supplies: 10
        8:
          item_name: Bat
          supplies: 10
```

```
    2:
      worms:
        - id: 3
          name: Worm 1
          health: 200
          x: 1144
          y: 416
          sight_angle: 0
          status:
            grounded: 0
            falling: 1
            mirrored: 0
            walking: 0
        - id: 4
          name: Worm 2
          health: 200
          x: 1506
          y: 540
          sight_angle: 0
          status:
            grounded: 0
            falling: 1
            mirrored: 0
            walking: 0

      inventory:
        0:
          item_name: Bazooka
          supplies: 10
        1:
          item_name: Mortar
          supplies: 10
        2:
          item_name: Cluster
          supplies: 10
        4:
          item_name: Banana
          supplies: 10
        3:
          item_name: Grenade
          supplies: 10
        5:
          item_name: Holy bomb
          supplies: 10
        7:
          item_name: Dynamite
          supplies: 10
        6:
          item_name: Air Strike
          supplies: 10
        9:
          item_name: Teleport
          supplies: 10
        8:
          item_name: Bat
          supplies: 10
```

Un mapa de juego, generado por el editor de mapas, es un archivo comprimido de formato *tar.gz* con dos archivos en su interior: *map.yml* y *background.png*.

El archivo map.yml es un archivo de texto, cuyo contenido es la estructura básica de un mapa, como puede ser el ejemplo anterior. El background.png es la imágen de fondo que se utilizará para el mapa, elegida por el usuario creador en el editor-launcher.

4.3 Clase Protocolo

La clase protocolo es utilizada para encapsular los sockets y el mensajeo entre cliente-servidor y es compartida por ambas aplicaciones. A continuación, se hará un análisis del flujo del programa desde que se conecta al servidor hasta que se juega una partida y se hará mención de cómo se utiliza la clase Protocolo para establecer la comunicación.

Desde cliente, nos conectamos al servidor y le informamos nuestro nombre. Para eso se utiliza el método **sendName()** de protocolo. Primero envía el largo del nombre en un bloque de 4 bytes fijos. Luego envía el nombre.

El servidor, recibe el nombre con el método análogo **getPlayerName()**, que obedece el protocolo anterior.

Luego, el servidor envía al cliente el estado actual de las partidas en espera, para que alimente la tabla del lobby. Para eso, hace uso del método **sendGameStatus()** que primero envía el tamaño en un bloque de 4 bytes de la información que se intenta enviar y luego envía la información (que es un nodo YAML serializado).

El cliente, por su parte, recibe el estado actual de las partidas en espera con el método **rcvGameStatus()** que obedece el protocolo anteriormente mencionado.

```
waiting_games:
    - match_name: juego1
      creator: gabriel
      required_players: 3
      joined_players: 1
    - match_name: juego2
      creator: ariel
      required_players: 2
      joined_players: 1
```

El cliente decide crear una partida, entonces envía un evento de tipo lobby al servidor con el siguiente formato (usa el método **sendEvent()**)

```
event:
   action: a_createMatch
   match_name: juego1
   map_players_qty: 3
```

Un segundo cliente, decide unirse a la partida anterior, entonces envía el siguiente mensaje al servidor:

```
event:
   action: a_joinMatch
   creator_name: player1
```

El servidor analiza si el cliente puede ingresar a la partida, y le da una respuesta con el siguiente formato (utilizando el método **sndMsg()** de protocolo):

code: 1 msg: ok	code: 0 msg: Partida llena

El mensaje de la izquierda corresponde con que el servidor da el OK al cliente a ingresar a la partida. El de la izquierda, lo impide y explica el motivo en el nodo ‘msg’.

El cliente recibe el mensaje con el método **rcvMsg()**.

Antes de iniciar la partida, el player creador le envía el mapa al servidor con el método **sendGameMap()** de protocolo. Primero envía, en un bloque fijo de 4 bytes, el tamaño en bytes del mapa. Luego envía el mapa. El servidor lo recibe con el método **rcvGameMap()** que obedece al protocolo anterior.

Una vez que el servidor tiene el mapa de la partida, se lo envía a todos los jugadores que no sean el creador. Para eso se utilizan nuevamente los métodos sendGameMap() y rcvGameMap().

Aquí, ya comienzan los loops de juego tanto en el servidor como en los clientes. El servidor toma snapshots del estado del juego y se los envía a todos los clientes. Para eso utiliza el método **sendGameMap()** de protoclo, preparado para enviar estructuras YAML. La estructura de los snapshots puede verse en la tabla de la sección 4.1.