Documentación técnica

Año: 2022
2do Cuatrimestre
Proyecto integrador: Rocket League
Integrantes: Luis Escalante, Mario Santiago Adris, Santiago Nicolás Penalva

Cliente

El cliente consta de 3 hilos. El principal, uno encargado de enviar datos al servidor y otro encargado de recibir datos de él.

El programa se inicia conectando al servidor con los datos ingresados en los argumentos del programa. Una vez conectado, se lanzan los hilos sender y receiver y el hilo principal lanza una app desarrollada en QT para que el usuario interactúe con el lobby de partidas. Una vez el usuario se una a una partida, el hilo principal cerrará la aplicación de QT y lanzará una nueva con SDL, la cual contiene el “Game Loop” que leerá las interacciones del usuario, enviará al servidor, actualizará el estado si llegó alguno nuevo y renderizará todo el desarrollo de la partida.

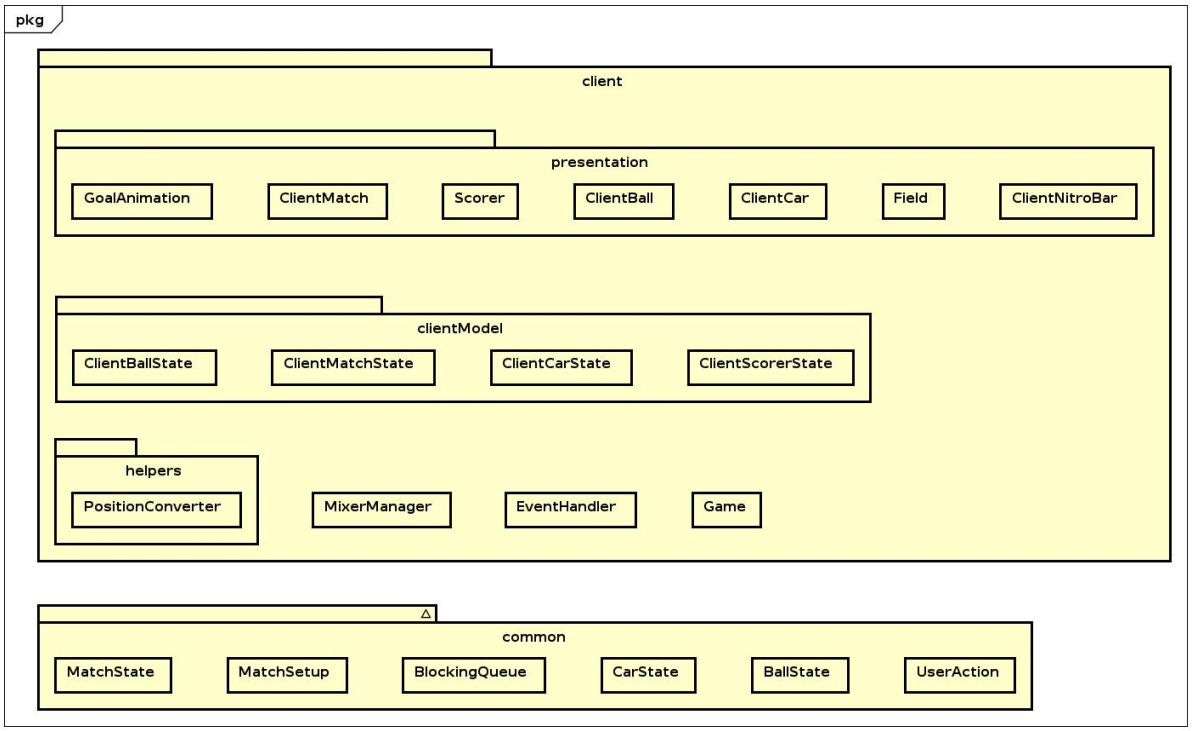
Diseño

Para está aplicación se trabajó en desacoplar el modelo con respecto al que llega del server (el common) , para ello se separó en 3 capas (directorios).

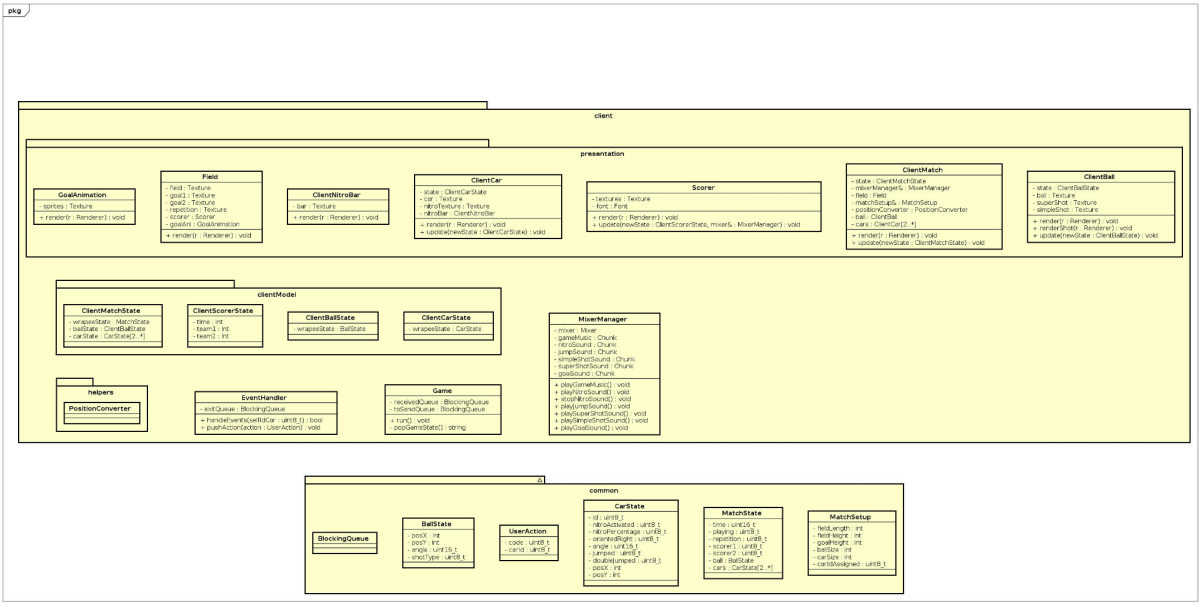
1. Presentación - (src/client/presentation): En esta capa están todas las clases que son renderizables, es decir, todas estas tienen un método render() y es donde se toca más SDL, la que maneja las texturas, sonidos, animaciones, etc. Obs: quedó como “TODO” generar una clase virtual pura con el método render (como una interfaz).
2. Modelo del cliente - (src/client/model): Esta nace con la necesidad de tener una capa de presentación más simple, con menos lógica y desacoplada de la capa 3. Y aunque en este punto del desarrollo parece no tener mucho sentido porque lo que hace es “wrappear” la capa 3 y no modifica sus datos ni les aplica alguna transformación, conforme vaya creciendo el juego irá tomando relevancia y podremos agregar las adaptaciones y transformaciones que se necesiten en esta para mantener más simple la capa 1 y desacoplada de la 3.
3. Modelo del protocolo (src/common).

Se adjuntan diagramas para verlo mejor: (Nota: en la carpeta doc del repo estarán estos mismos diagramas en .svg para una mejor apreciación):

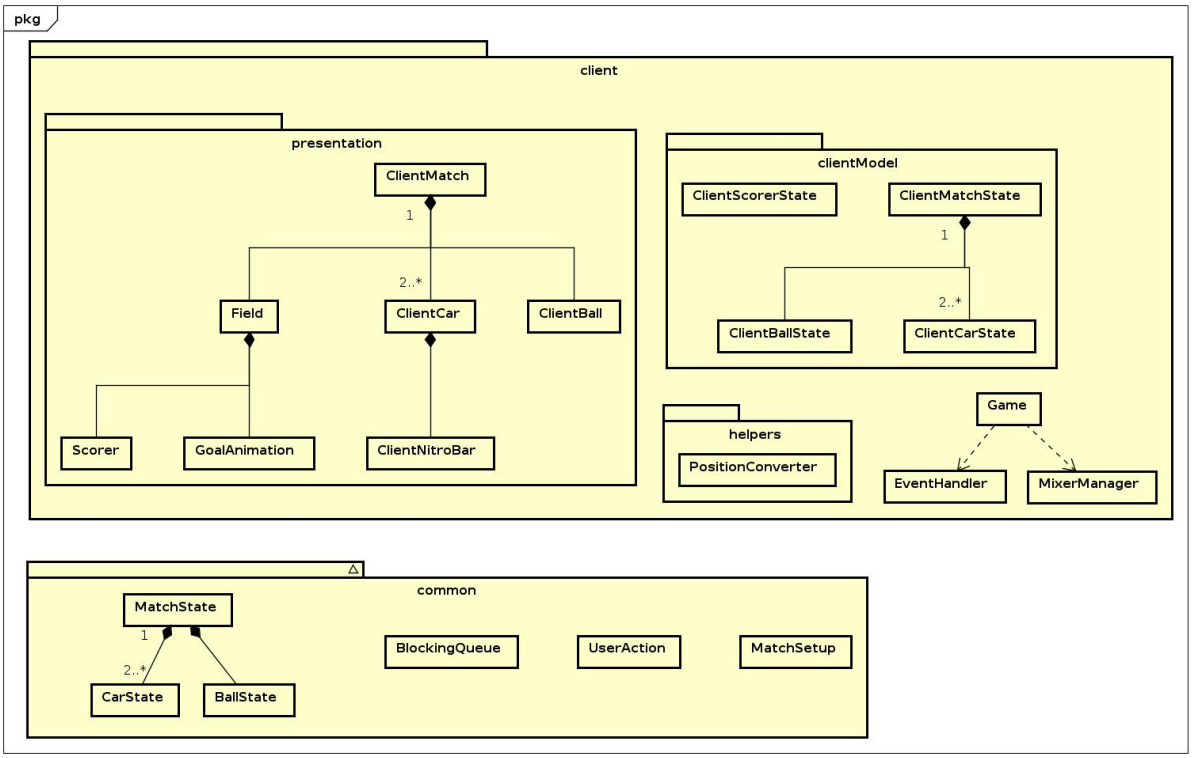
Diagrama de paquetes del cliente junto con las clases relevantes de common simplificado:



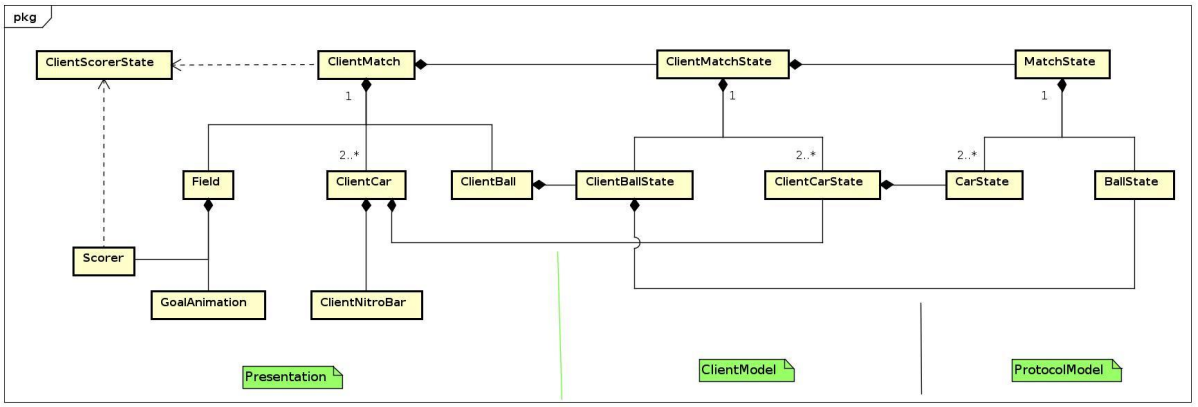
Mismo diagrama pero con los detalles relevantes de cada clase



Mismo diagrama que el primero pero con las asociaciones entre las clases de la **misma capa**



Relaciones entre las clases principales de las 3 capas

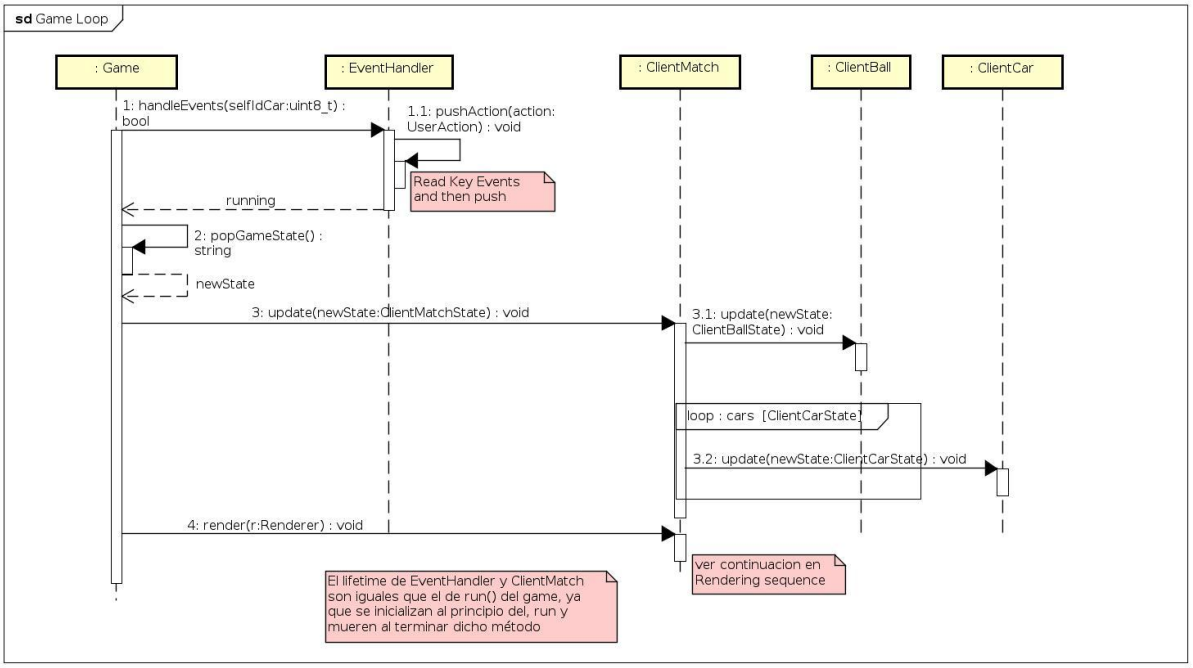


Observaciones respecto a estos diagramas mostrados:

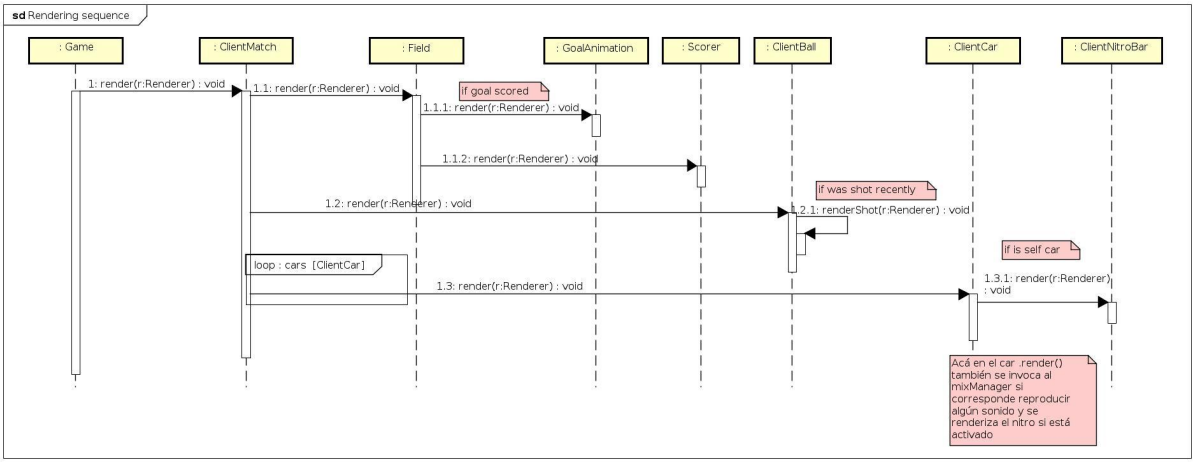
1. El handleEvents() en realidad recibe una referencia a ClientMatch pero haciendo el diagrama de clases me di cuenta de que no hace falta la clase completa, sino que solo necesita el selfCarId y por eso preferí colocarlo así en el diagrama ya que hace más sentido así.
2. Algunos parámetros en los métodos render() no fueron agregados para no hacer tan grande la firma.

Diagramas de secuencia:

Game Loop:



Rendering:



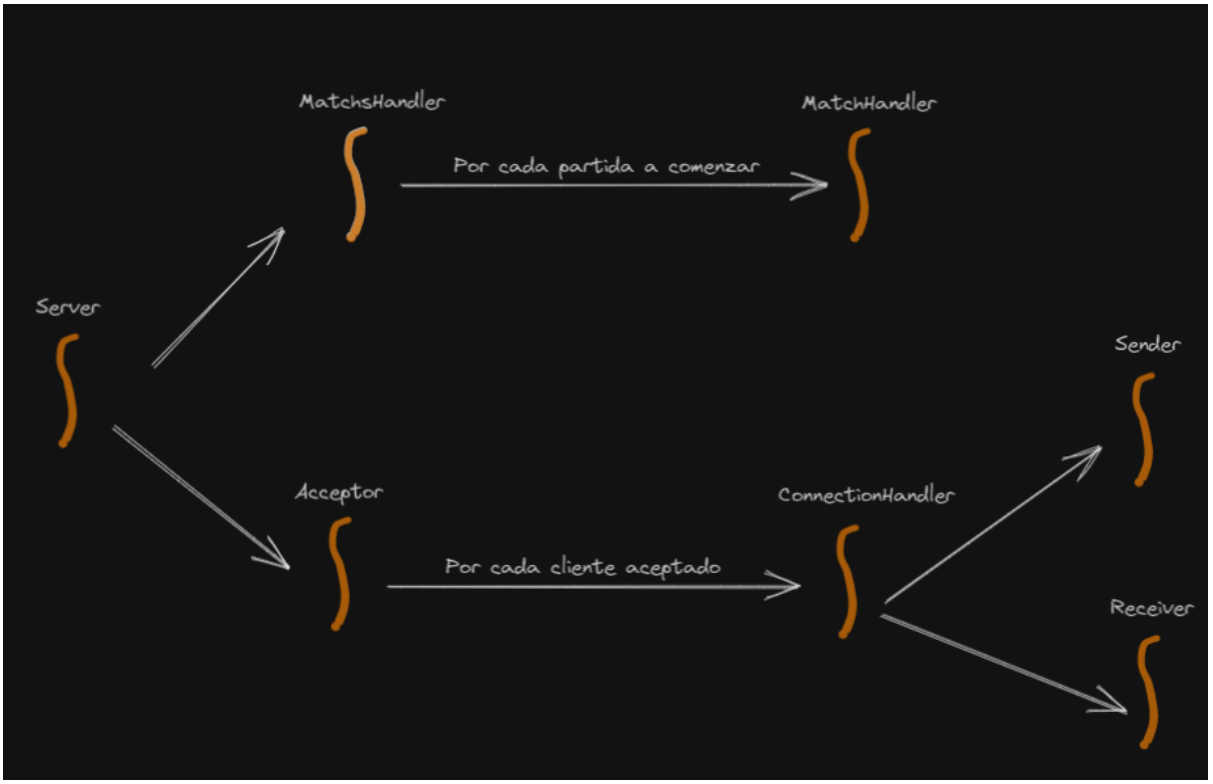
Servidor

Debido a que el servidor debía aceptar la conexión de múltiples clientes, había que diagramar una arquitectura multi threading para manejar la aceptación de dichas conexiones y la comunicación con cada una de ellas. De manera resumida, los hilos que forman parte del servidor son:

- Main: el main thread se encarga de leer la entrada estándar hasta recibir una “q” lo cual hará que se dejen de aceptar nuevas conexiones al servidor
- ThreadAcceptor: se encarga de esperar y aceptar conexiones de nuevos clientes. Por cada conexión aceptada se lanza un nuevo hilo ThreadConnectionHandler que se encargará de administrarla.

- ThreadConnectionHandler: encargado de la conexión con el cliente. Hay uno por cliente conectado. Lanza dos nuevos hilos, ThreadSender y ThreadReceiver
- ThreadSender: Encargado de enviar datos al cliente, hace “pop” de una cola bloqueante creada por el ThreadConnectionHandler para inmediatamente mandarlo al cliente
- ThreadReceiver: Queda bloqueado esperando envíos de datos por parte del cliente y procesa lo enviado para interactuar con el lobby de partidas, en caso de que el cliente todavía no se haya unido a una partida, o de pushear acciones a la cola de la partida correspondiente.
- ThreadMatchsHandler: lanzado por el main thread, hace “pop” de una cola bloqueante que contiene partidas listas para ser comenzadas (partidas que alcanzaron el límite de usuarios. Por cada partida a comenzar lanza un nuevo hilo ThreadMatchHandler pasándole dicha partida.
- ThreadMatchHandler: Contiene el game loop de una partida, haciendo pops de la cola perteneciente a la partida en ejecución (a la cual pushea cada cliente que integra la partida), actualizando el estado de la partida de Box2D, y pusheando cada nuevo estado a la cola para que cada ThreadSender mande la actualización a cada cliente

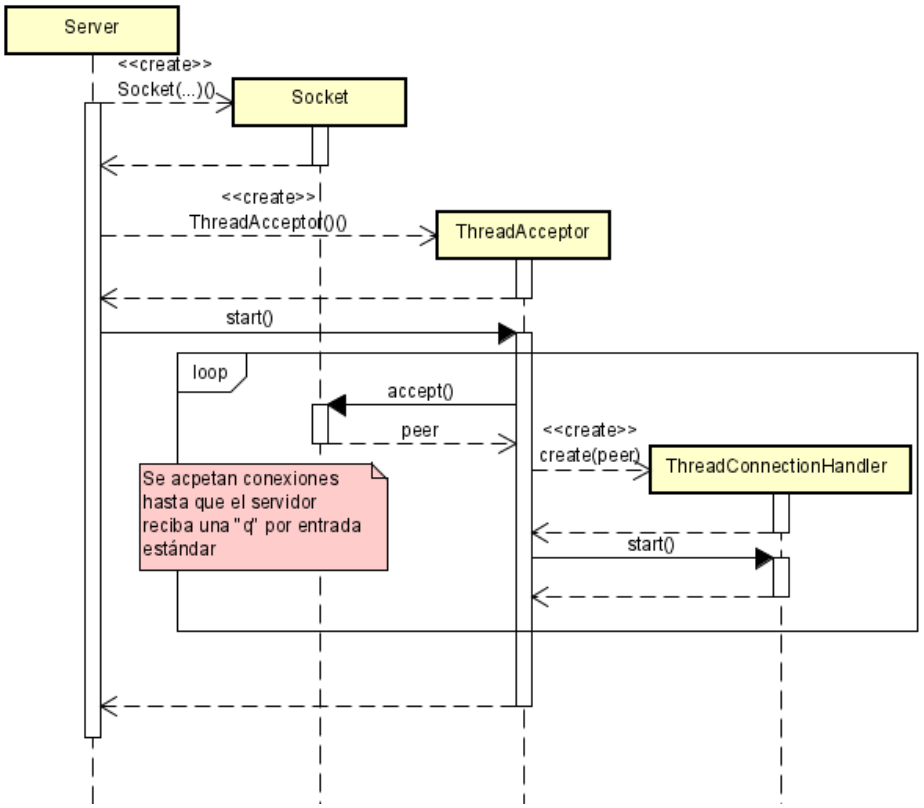
A continuación se puede ver un pequeño diagrama con los hilos detallados anteriormente



Para evitar los problemas que pueden ser generados por arquitecturas multithreading, como las race conditions por ejemplo, se hacen uso de mutex, para proteger el lobby de partidas, ya que múltiples clientes pueden estar queriendo interactuar con él (creando partidas, listándolas, uniéndose). Por otro lado, se hace uso de colas bloqueantes y no bloqueantes compartidas por los diferentes hilos para su comunicación.

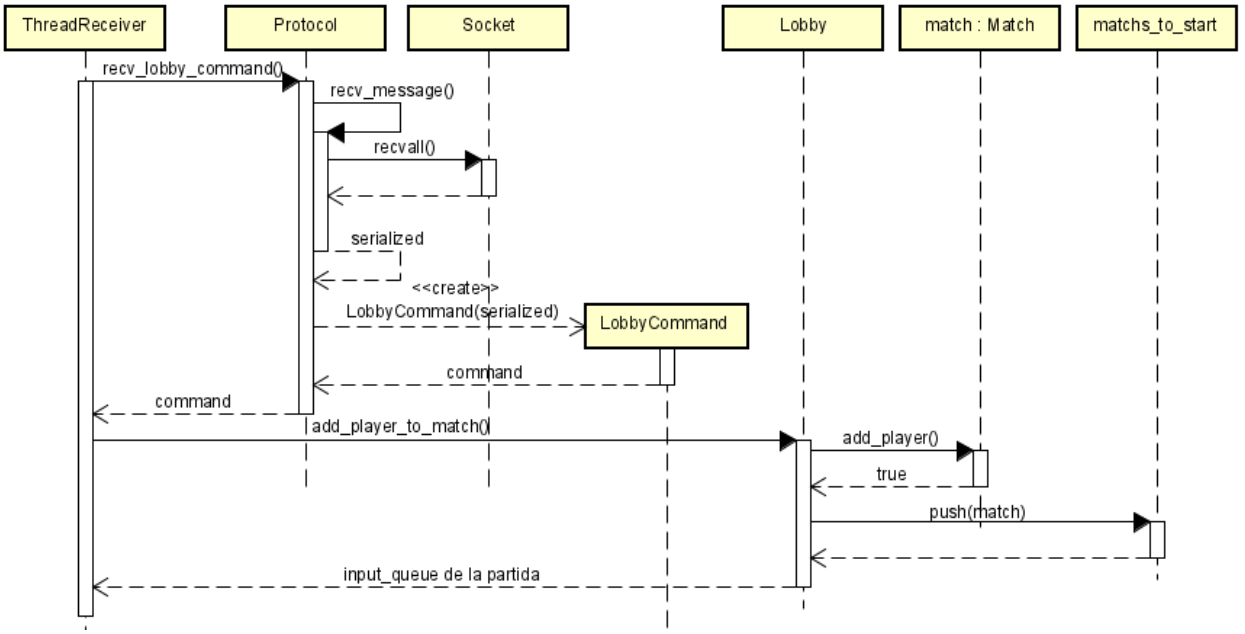
A continuación se enseñarán unos diagramas que ayudarán a visualizar el funcionamiento de unas partes del servidor

- El servidor lanza un hilo que aceptará las conexiones entrantes, lanzando un nuevo thread que manejará la conexión.

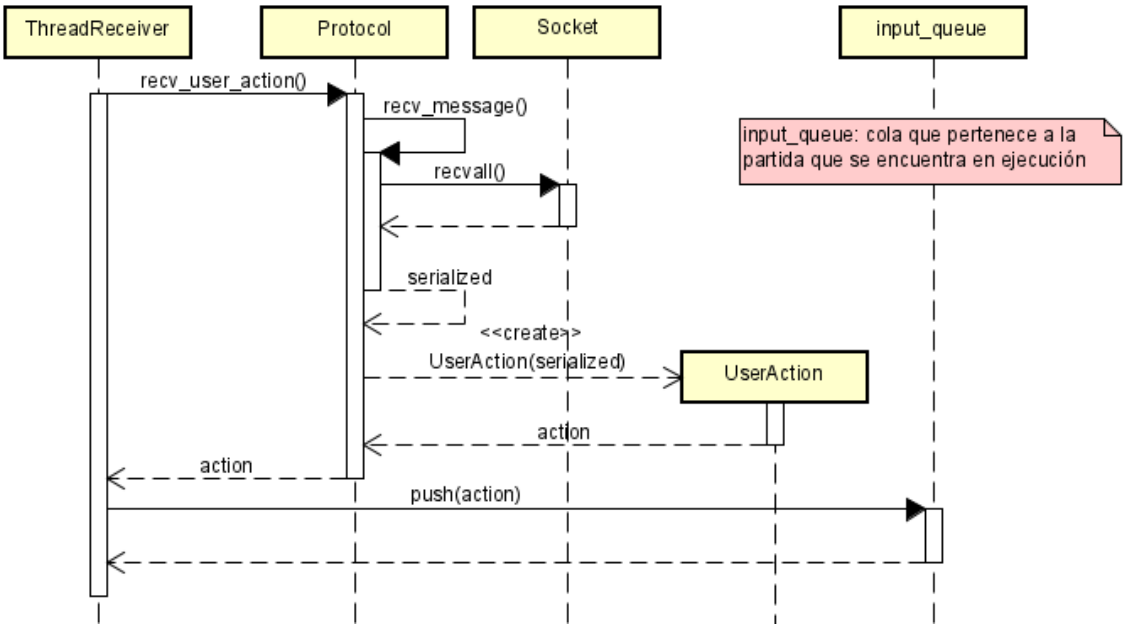


Luego ThreadConnectionHandler lanzará los hilos sender y receiver respectivos a la nueva conexión

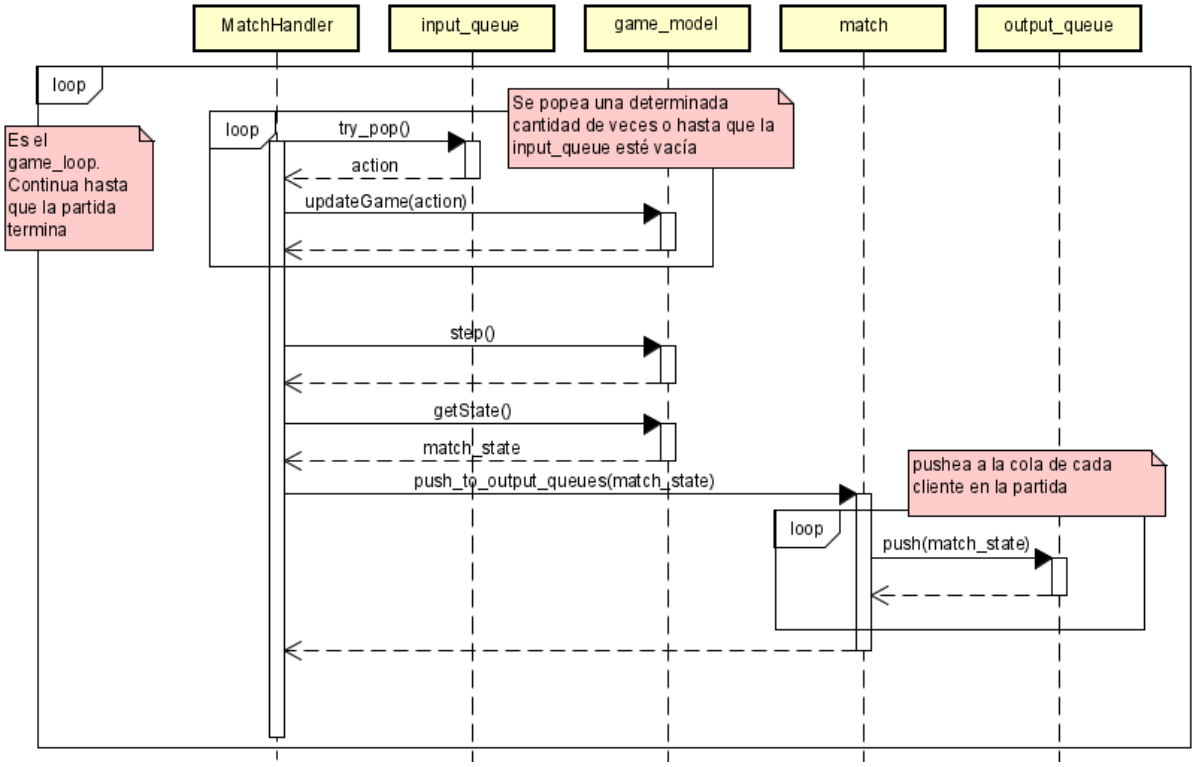
- Con el cliente todavía en el lobby de partidas el ThreadReceiver recibe un comando para unirse a una partida, la cual completa la cantidad de jugadores y tiene que ser comenzada.



- Con el cliente en un partida en ejecución, el ThreadReceiver recibe una acción de juego que es pusheada a la cola de partida correspondiente.



La acción del usuario queda pusheada en la input_queue lista para ser “popeada” por el MatchHandler



Protocolo

Para la comunicación entre cada cliente y el servidor se implementó un protocolo binario el cual enviará de cliente a servidor y viceversa distintos tipos de entidades:

- Comandos de lobby: Utilizados por el cliente para indicar una acción a realizar en el lobby de partidas. Estos pueden ser:
    - Para crear una partida, el cliente enviará:
        - 0x01 longPayload cantJugadores nombrePartida
    - Para unirse a una partida:
        - 0x02 longPayload nombrePartida
    - Para listar partidas existentes:
        - 0x00 0x00
- Acciones de partida: Durante el desarrollo de una partida, el cliente puede realizar determinados movimientos, que serán enviados al servidor mediante un byte. Los movimientos realizables son:

LEFT_PUSH	0x01
LEFT_RELEASE	0x02
RIGHT_PUSH	0x03
RIGHT_RELEASE	0x04
UP_PUSH	0x05
UP_RELEASE	0x06
DOWN_PUSH	0x07
DOWN_RELEASE	0x08
NITRO_PUSH	0x09
NITRO_RELEASE	0x10
JUMP	0x11

- Setup de partida: Al comenzar una partida, el servidor manda a cada cliente el setup de la partida, indicando cantidad de autos, tamaño de ellos, dimensiones de la cancha, del arco, de la pelota, etc
- Estado de partida: A medida de que el servidor va actualizando la física de la partida con Box2D y las acciones de cada jugador, el servidor irá enviando a cada cliente todas las actualizaciones de la partida. En cada envío, el servidor envía todos los datos de la partida: posición de la pelota, posición de cada auto, si hubo un shot, si hubo un salto, si se realizó un gol, etc

Cada una de las entidades mencionadas está representada con una clase (MatchState, LobbyCommand, UserAction, etc) en la que cada una cumple sí o sí con dos requisitos:

- Poseer un método “serialize” el cuál devuelva un string con todos sus datos (para luego ser enviado por el protocolo hacia el cliente o servidor)
- Un constructor que reciba un string previamente serializado y deje construido el objeto de manera correcta (por ejemplo, el servidor serializa un MatchState, lo manda al cliente, y éste utiliza el constructor de string para acceder a los datos)

Test

Para los tests sobre el protocolo usamos el framework GoogleTest.

Nuestro protocolo recibe un puntero al Socket (clase que encapsula el funcionamiento de un socket) el cual utilizará para enviar y recibir datos del cliente/servidor. Para la implementación de test creamos una nueva clase SocketTest que extiende a la ya mencionada clase Socket y sobreescribe los métodos de enviar y recibir para, en lugar de utilizar un socket real, simplemente guardar los datos en una cola, en los envíos, y devolverlos de la cola, en los recibos.

Para correr los test simplemente basta con ejecutar el archivo “protocol_test.cc” dentro de “build/src/test/” una vez buildeado el proyecto.

Box2D

Para implementar la física del juego se usó la api de Box2D (https://box2d.org/documentation/). El juego consta de dos tipos de objetos que interactúan entre sí (además de los cuatro límites del campo de juego), autos y pelota, ambos son cuerpos de tipo dinámico.

El auto fue implementado con tres cuerpos (b2Body): el chasis, y las dos ruedas. Para unir las ruedas al chasis se utilizó un tipo de unión llamada b2MotorJoin que, convenientemente, simula el giro de un motor. Gracias a esta unión se puede modificar el torque, velocidad angular del motor, y amortiguación, entre otras cosas. Además, si se quisiera aumentar el grip cada b2Fixture (manera de asociar una forma a un cuerpo) tiene la capacidad de setar un valor de fricción.

Los saltos fueron implementados aplicando impulsos, pero para el nitro se setea una velocidad al chasis, esta decisión se tomó para poder implementar nitro continuo (mantener apretado la tecla correspondiente acciona el boost hasta que se vacía).

Para los shots se podría haber utilizado sensores (para más detalles ver la documentación de box2d) pero por rapidez de implementación, se optó por detectarlos "a mano", es decir revisando las posiciones de auto y pelota a la hora de hacer un salto, y es que los shots se dan cuando el jugador acciona la tecla de salto, este detalle permite implementar la feature de esa manera.

SDL

Para todo el rendering del juego, interacción con el usuario se utilizó la librería SDL, para los sonidos SDL_Mixer y para los textos en el marcador SDL_ttf, todas estas a través del wrapper para C++ con clases RAII SDL2pp.

Para el renderizado de las imágenes se usa la clase SDL2pp::Texture que recibe como atributo un SDL2pp::Renderer y un SDL2pp::Surface.

Para los sonidos la clase que se encarga de manejar/orquestar todos los sonidos es SDL2pp::Mixer y cada sonido/musica que tenemos (game music, el sonido del salto, nitro, etc) son de tipo SDL2pp::Chunk, y dichos sonidos los reproducimos a través del Mixer

Configuración

Antes de buildear el proyecto se pueden editar algunas configuraciones del juego dentro del archivo ".r_config.yml" en la raíz del proyecto. Son configurables:

- el largo de la cancha (camp_length)
- el alto de la cancha (cmap_height)
- el alto de los arcos (scorer_height)
- el tamaño de la pelota (ball_size)
- y la duración de la partida (match_duration)

Al ejecutar el servidor, se tomarán las configuraciones de este archivo para el desarrollo de las partidas. Para la carga de estas opciones se utilizó la librería de YAML.