



FACULTAD DE INGENIERIA

Universidad de Buenos Aires

Documentación Técnica.

[1. Descripción General](#)

[2. Estructura General](#)

[2.1 Servidor](#)

[2.2 Cliente](#)

[2.3 Common](#)

[2.4 Libs.](#)

[2.4.1 BOX2D](#)

[2.4.2 SDL](#)

[2.5 Assets](#)

[2.6 Music](#)

[2.7 Tests](#)

[2.8 UI](#)

[3. Protocolo](#)

[4. Travis CI](#)

[5. Diagramas de Clases](#)

1. Descripción General

Rocket League es un juego multijugador cliente-servidor que consta de un partido de fútbol, reemplazando a los tradicionales jugadores por autos. Con ellos se puede realizar todo tipo de piruetas, ya sea un salto, doble salto, aplicar turbo, realizar flips y tiros especiales, con el fin de empujar la pelota dentro del arco contrario.

Este será multijugador en línea, donde cada jugador podrá controlar uno de los autos del equipo. Se podrán formar partidas de formato 1v1 o 2v2. El equipo ganador será quien haya marcado más goles en el arco rival al finalizar el cronómetro.

2. Estructura General

Los archivos generados para el programa se pueden clasificar de la siguiente manera:

2.1 Servidor

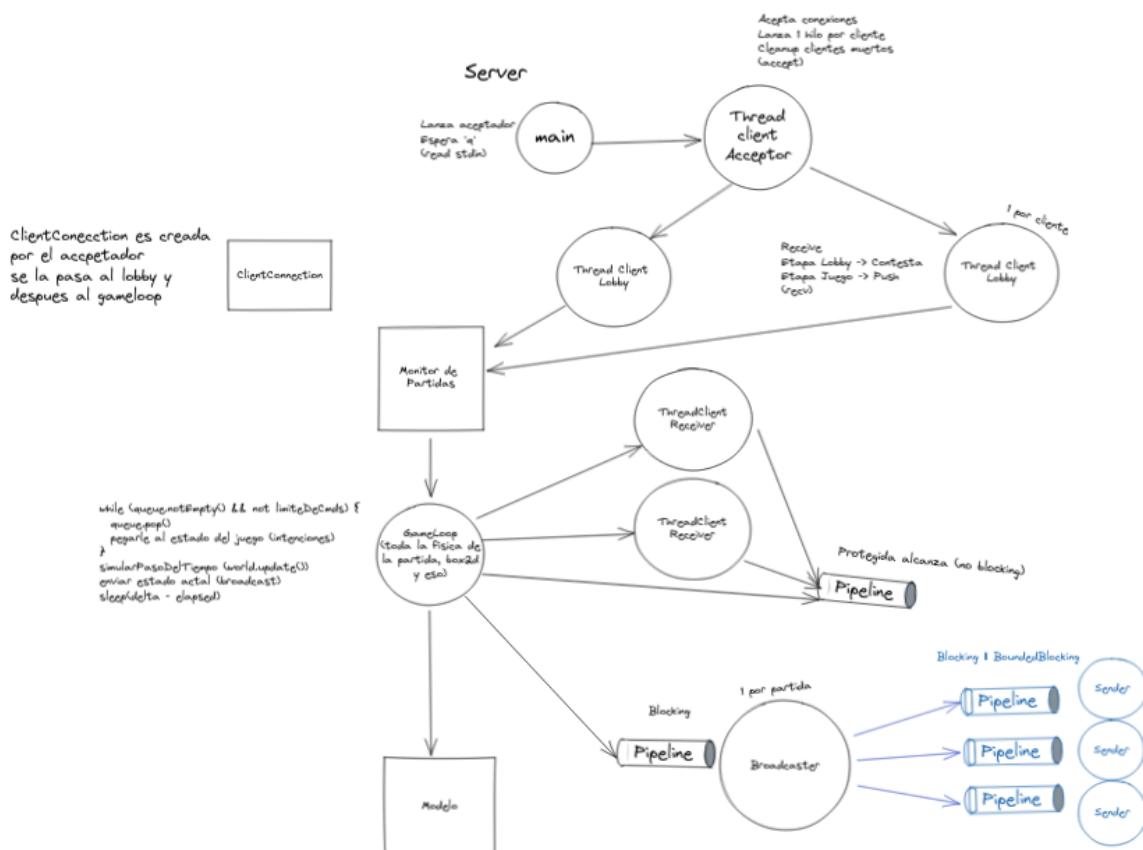
Posee un hilo main encargado de lanzar un hilo Aceptador, y luego se queda esperando por entrada estándar una letra “q” para cerrar el server.

El hilo Aceptador se encarga de aceptar nuevas conexiones para luego instanciar todas las estructuras necesarias. Por cada conexión nueva, lanza un hilo de Lobby.

El hilo Lobby se encarga de crear partidas y unir al jugador a la partida que este le indique. Una vez que la partida se llene, el hilo Lobby del ultimo jugador que se metió en la partida lanza el hilo Game.

El hilo Game es el encargado de manejar las físicas de la partida. Además, se encarga de lanzar los siguientes hilos:

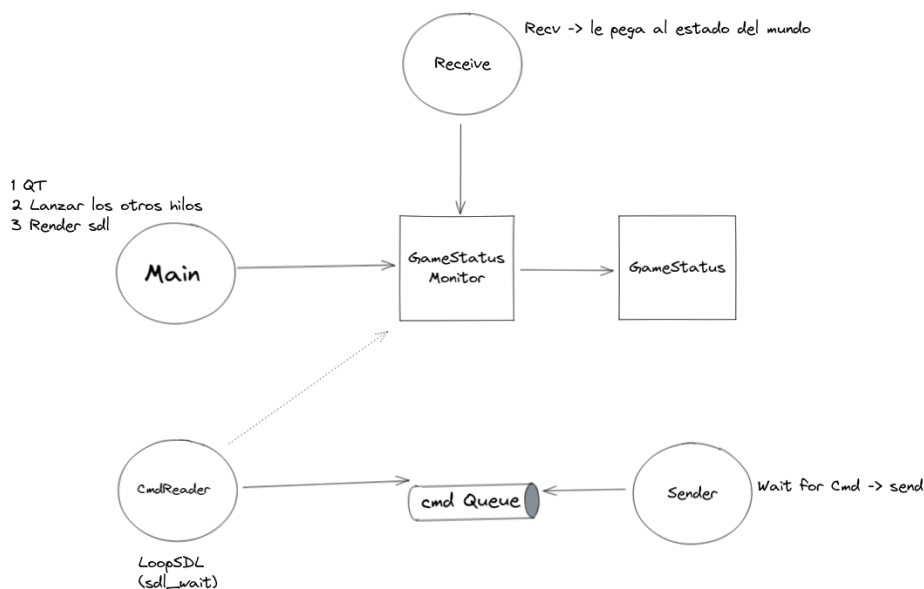
- Receivers: uno por cliente, reciben los comandos que los clientes les envían mediante sockets, crean una instancia de Acción del Cliente (ActionClient) y la pushean a la cola de acciones de cliente.
- Broadcaster: agarra el estado actual del juego (GameStatus) y se lo envía a los hilos Senders.
- Senders: uno por cliente, recibe el estado actual de juego del Broadcaster y se lo envía al cliente mediante sockets.



2.2 Cliente

Posee un hilo main encargado de lanzar a los siguientes hilos:

- CmdReader: encargado de catchear los comandos del teclado y mouse y generar las instancias de Comando.
- Sender: toma los Comandos creados por el hilo CmdReader y los envia al servidor mediante uso de sockets.
- Receiver: recibe por parte del servidor los estados actuales de la partida (GameStatus) y actualiza este estado local.
- Renderer: este es el mismo hilo main, luego de haber lanzado a los demas hilos. Se encarga de imprimir de manera gráfica el estado actual del juego.



2.3 Common

Archivos en común utilizados por el servidor y por el cliente. Ellos son:

- BlockingQueue
- Thread
- Protocol
- GameStatusSerializer
- GameStatus
- Models

2.4 Libs.

Librerías externas utilizadas en el proyecto.

2.4.1 BOX2D

Es una biblioteca libre que implementa un motor físico en dos dimensiones, encargándose de la implementación física del juego.

2.4.2 SDL

Librería que proporciona funciones básicas para realizar operaciones de dibujo en dos dimensiones, gestión de efectos de sonido y música, además de carga y gestión de imágenes.

2.5 Assets

Carpeta que contiene todas las texturas utilizadas por la librería de SDL para renderizar las mismas en pantalla.

2.6 Music

Carpeta que contiene todos los sonidos utilizados por la librería de SDL.

2.7 Tests

Carpeta que contiene todas las tests implementadas con el framework gtest. Como regla general definimos que la nomenclatura para nombrar nuestros archivos de test deben ser {NombreDeLaClase}Test.cpp, por ejemplo “*ProtocoloTest.cpp*”.

2.8 UI

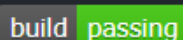
Carpeta que contiene todos los archivos generados por la aplicación QTCreator, optamos por incluir esta carpeta dentro del proyecto para tener todo integrado. Todo lo que se modifica de la UI debe estar mapeado aquí.

3. Protocolo

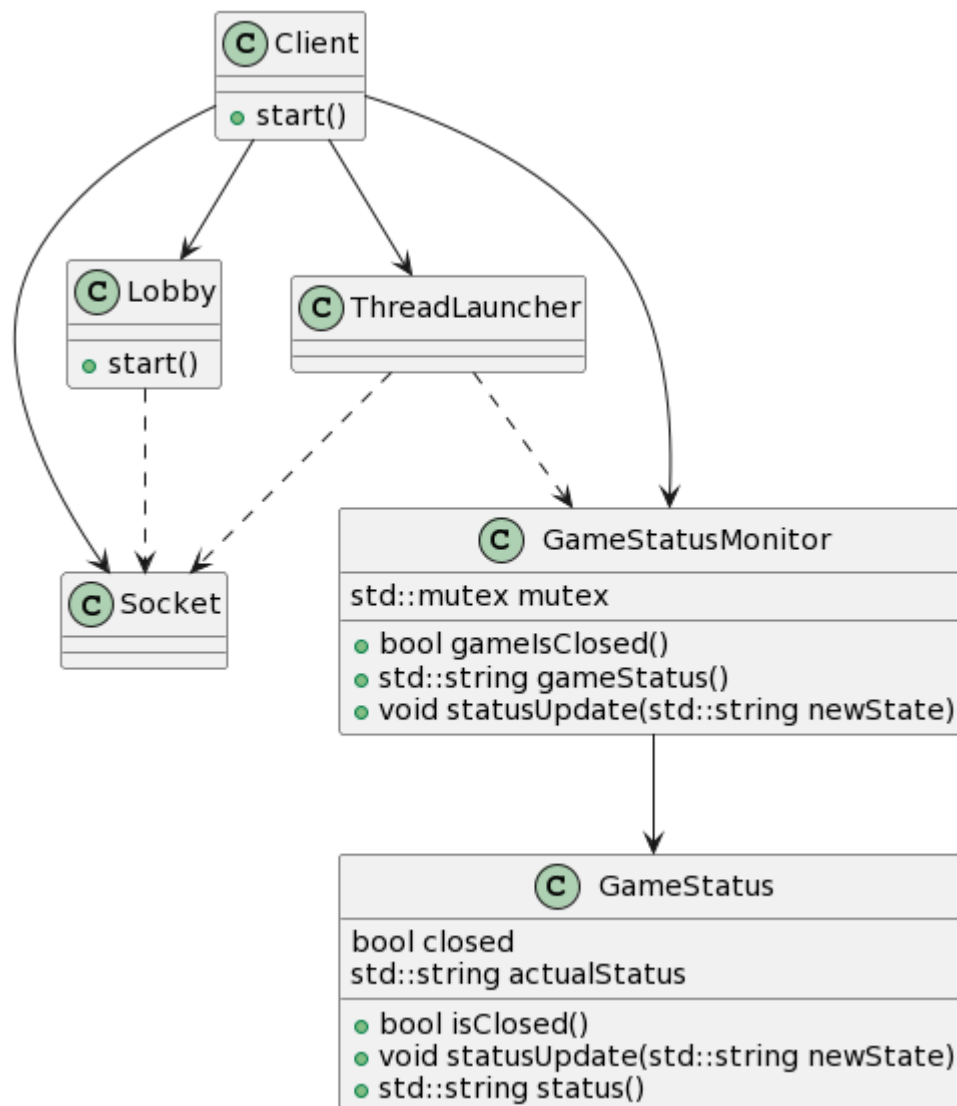
Para la comunicación entre el cliente y el servidor utilizamos un protocolo de texto. El mismo consiste en enviar mensajes separados por un doble salto de linea (“\n\n”). Una vez que la partida comienza, dentro del mensaje se encuentran todos los valores del estado actual del juego necesarios para la correcta impresion por pantalla del lado del cliente. Estos valores se encuentran separados por espacios, pero este procesamiento del mensaje no es responsabilidad del Protocolo, sino de la clase GameStateSerializer.

4. Travis CI

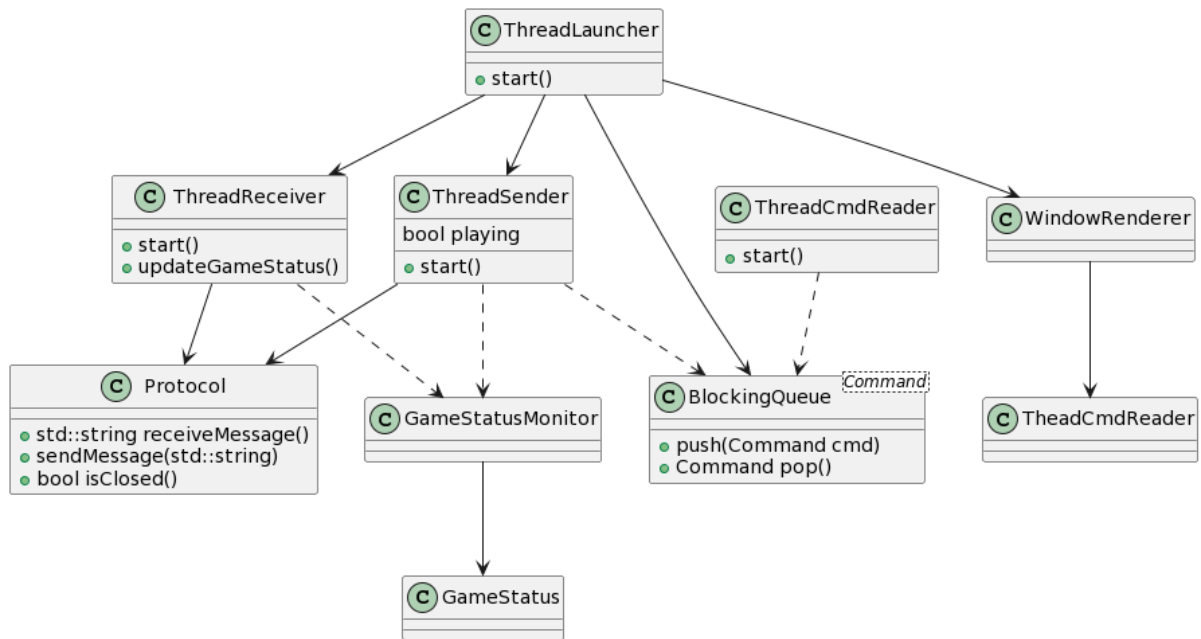
En la raíz del proyecto se encuentra el archivo **.travis.yml** , aquí se configuran todos los packages necesarios para correr la aplicación y además podemos definir que scripts correr luego de cada commit. Luego de cada commit, la aplicación se buildea y luego corre todas las pruebas ubicadas dentro de la carpeta Tests. Esto nos permite tener una visión clara de cómo esta nuestra rama master.

A dark grey rectangular box containing the text 'build passing'. The word 'build' is in a light grey box and 'passing' is in a green box.

4. Diagramas de Clases



El cliente encapsula el comportamiento general del cliente. Mientras el mismo siga en la aplicacion, se ejecutara el hilo Lobby. En el se unira el cliente a una partida, ya sea una existente o una creada en el momento. Una vez el cliente se encuentre en la partida, el ThreadLauncher se encarga de lanzar los distintos hilos que necesita el cliente durante la partida. Ademas, se encarga de que todos los hilos conozcan al objeto GameStatusMonitor, el cual tiene el estado actual del juego. Justamente como es accedido por distintos hilos, este estado de juego se encuentra dentro de un monitor.



Los distintos hilos que se ejecutan mientras el cliente se encuentra en la partida son:

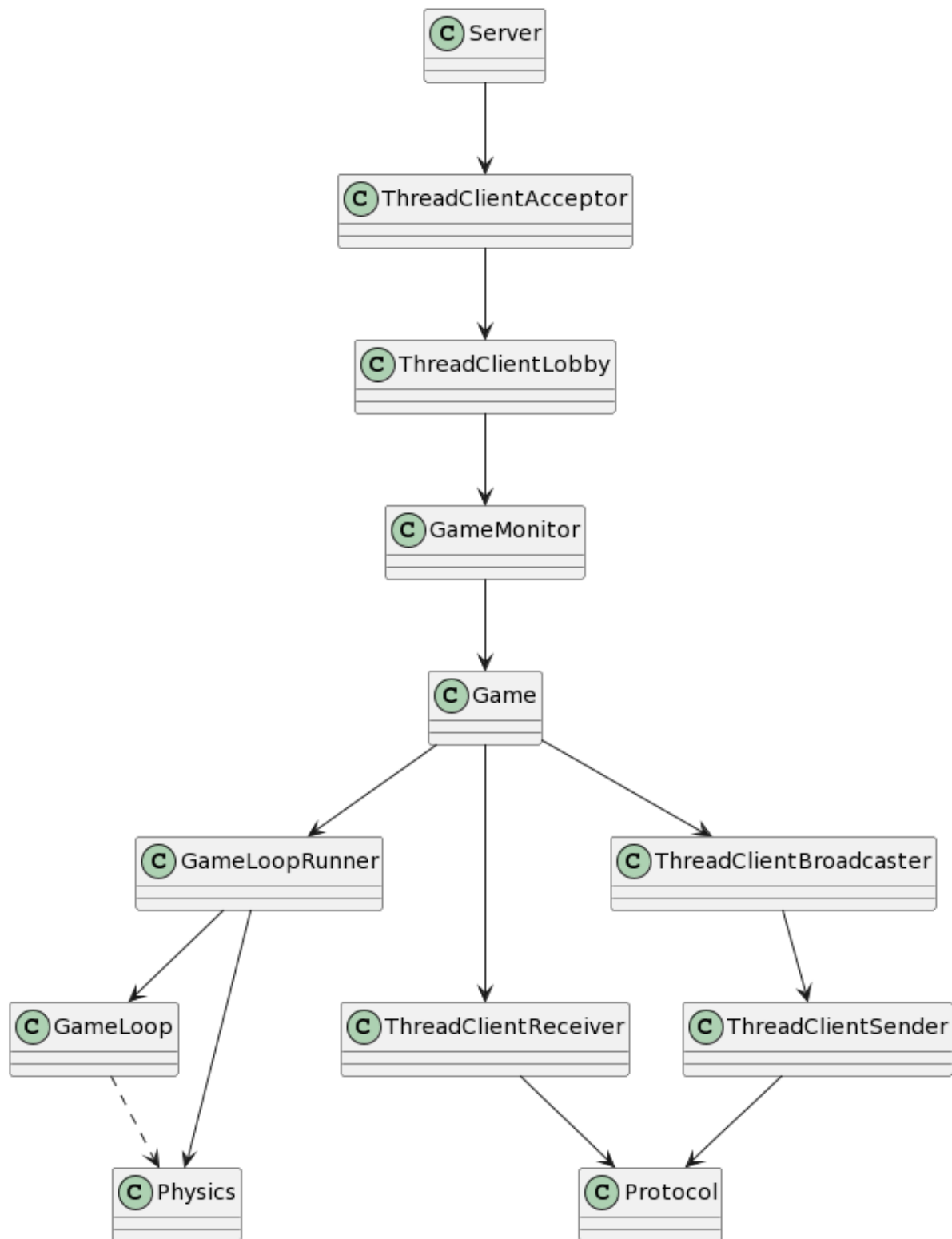
ThreadCmdReader: mientras la partida continúe, recibe comandos de la persona que utiliza la aplicación, en este caso solo del teclado, procesa el tipo de comando que corresponde a la tecla ingresada, y se lo envía al ThreadSender mediante una cola.

ThreadSender: mientras reciba un comando y mientras la partida continúe, envía el comando al server mediante el Protocolo.

ThreadReceiver: mientras la partida continúe, recibe el estado actual del juego calculado en el server, y actualiza el estado actual del juego almacenado localmente en el GameStatusMonitor.

WindowRenderrer: si bien no es un nuevo hilo lanzado, es el mismo hilo de ejecución del ThreadLauncher (el hilo main), por lo que se ejecuta en paralelo a los ya mencionados.

Mientras la partida continúe, imprime mediante SDL el estado actual del juego.



El Server encapsula el comportamiento general del server. El hilo principal lanza al hilo ThreadClientAcceptor y luego se queda en un ciclo esperando una “q” por entrada estandar, y cuando la reciba, corta el server. El hilo ThreadClientAcceptor se encarga de aceptar nuevos clientes al servidor. Una vez que llega un nuevo cliente, el ThreadClientAcceptor le lanza un hilo de ThreadClientLobby, y sigue esperando a mas clientes. El ThreadClientLobby se encarga de crear la partida o unirse a la partida que el usuario le

indique. Una vez que el ultimo jugador se une a la partida actual, el ThreadClientLobby comienza el Game, el cual a su vez lanza varios hilos:

ThreadClientReceiver: uno por cliente, encargados de recibir las acciones del cliente mediante el Protocolo. Todos pushean a la misma cola de comandos.

GameLoopRunner: posee las fisicas dl juego. Se encarga de popear las distintas acciones de los clientes (de la unica cola de comandos) y aplicar estas acciones al estado actual del mundo en las fisicas. Luego simula el paso del tiempo, obtiene el nuevo estado actual del juego, y este se pushea a una cola de estados del juego.

ThreadClientBroadcaster: popea los estados actuales del juego (los que pushea el GameLoop) y se los envia a los distintos ThreadClientSender, pusheandolos en colas de estados del juego (una por hilo ThreadClientSender).

ThreadClientSender: popea el estado actual del juego y lo envia al cliente mediante el Protocolo.