FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Rocket league

Documentación Técnica

[95.08] Taller de Programación 1

Curso Veiga

Segundo cuatrimestre de 2022

Alumno	Padrón	Mail
Amaya , Santos Emanuel	96891	samaya@fi.uba.ar
Romero Gonzales, Alan Xochtiel	108316	alromero@fi.uba.ar
Bragantini, Franco Julian	97190	fbragantini@fi.uba.ar

Índice

1.Introducción

El trabajo práctico consistió en desarrollar dos programas, uno que cumpla el rol de servidor y otro de cliente, para poder jugar al Rocket League en distintas computadoras a través de concesión de puerto y protocolo TCP/IP. En el siguiente documento se detalla brevemente las herramientas empleadas en el desarrollo, requerimientos de librerías externas,la arquitectura del Servidor, idea base de implementación de Cliente, protocolos de comunicación entre cliente y servidor, detalles y consejos para futuros Sprints.

2.Herramientas y Librerías Externas

Lenguaje y herramienta de test

- Para el desarrollo utilizamos en lenguaje C++ 17
- Para los test utilizamos cxxtest

Librerías Externas

- **Qt5** para el menú del Cliente, ya que permitía crear una visual interactiva con botones de forma rápida.

- **SQL2pp** para la interfaz de juego y la emisión de sonido en el Cliente.

- **Box2d** como motor de físicas en el Servidor.

3.Arquitectura del Servidor

Sin rodeos lo más difícil de todo el desarrollo radica en entender los siguientes gráficos, notar que están divididos en dos partes que se pueden considerar casi separadas; se aconseja iniciar el recorrido de las entidades desde Server hasta el momento de tener un RunGame.

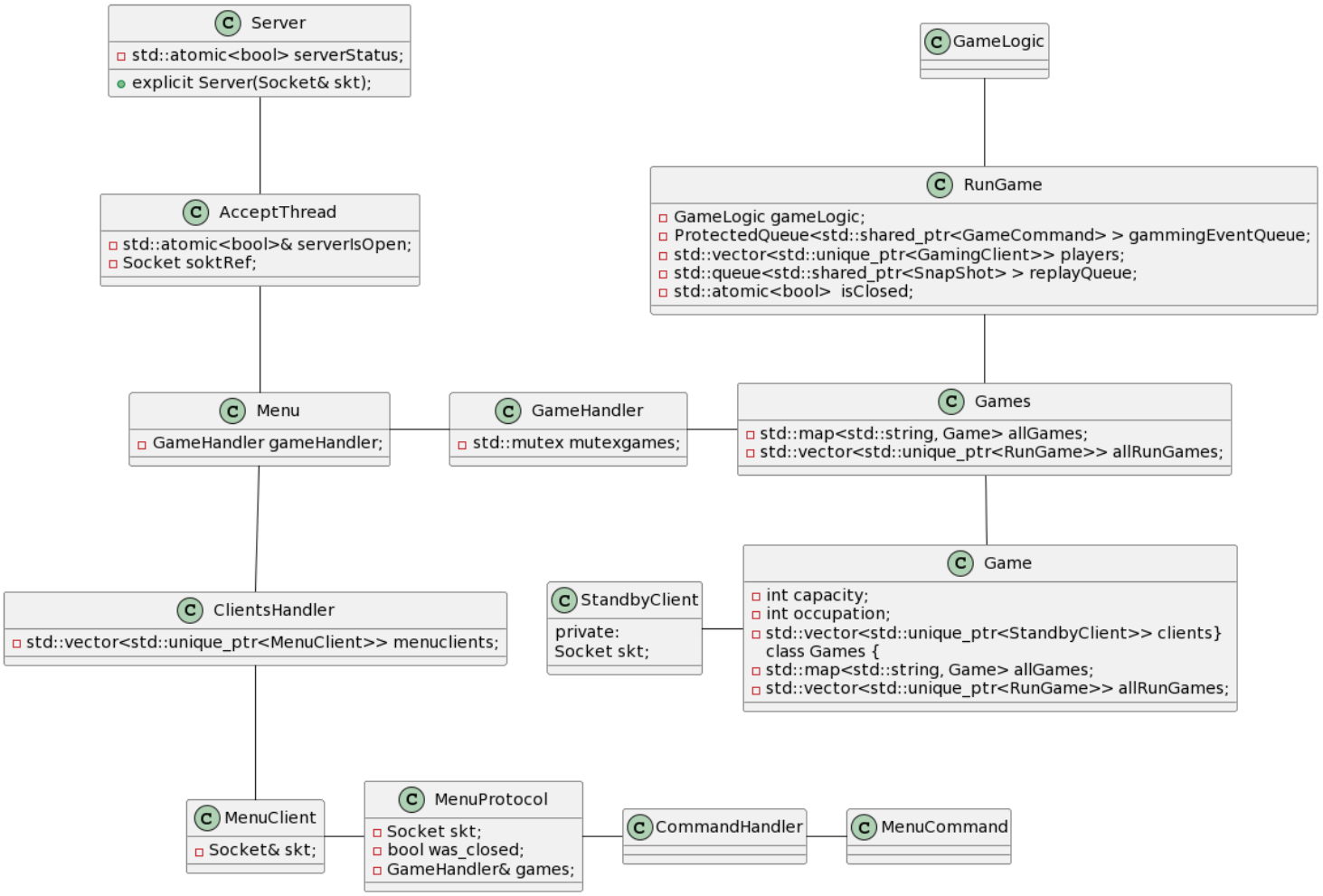


Fig.1

Los puntos claves para esta parte del servidor es entender que una vez que se conecta un Cliente a través del socket en **AcceptThread,** éste se convierte en un **MenuClient.** Estos MenuClients pueden interactuar con **GameHandeler** a través de **MenuCommad,** para obtener estos el Cliente debe mandar mensajes a través de socket siguiendo un protocolo de menú, que son parseados a MenuCommand por el **CommandHandler**.

Estos MenuCommand pueden agregar partidas (**CommandAdd**), crear partidas (**CommandCreate**) y listarlas (**CommandList**); todos estos commands heredan de una clase Command que está en la carpeta.

Cuando un Cliente crea o elige una partida pasa a ser **StandByClient** en el caso del que comando de ejecute con éxito.

Cuando el juego(**Game**) se complete este pasará a ser **RunGame,** todo RunGame tiene su **GameLogic** propia y es en esta la que se implementa la lógica del manejo del juego multijugador. Apenas RunGame se inicie crea los **GamingClient**, que estos a su vez inician sus hilos de protocolos y como primer mensaje envío n_cars(cantidad de jugadores en RunGame) y el id de cada Cliente.

Nota Aclaratoria: AcceptThread,RunGame y las clases protocolos de todos los clientes, son **Threads**.

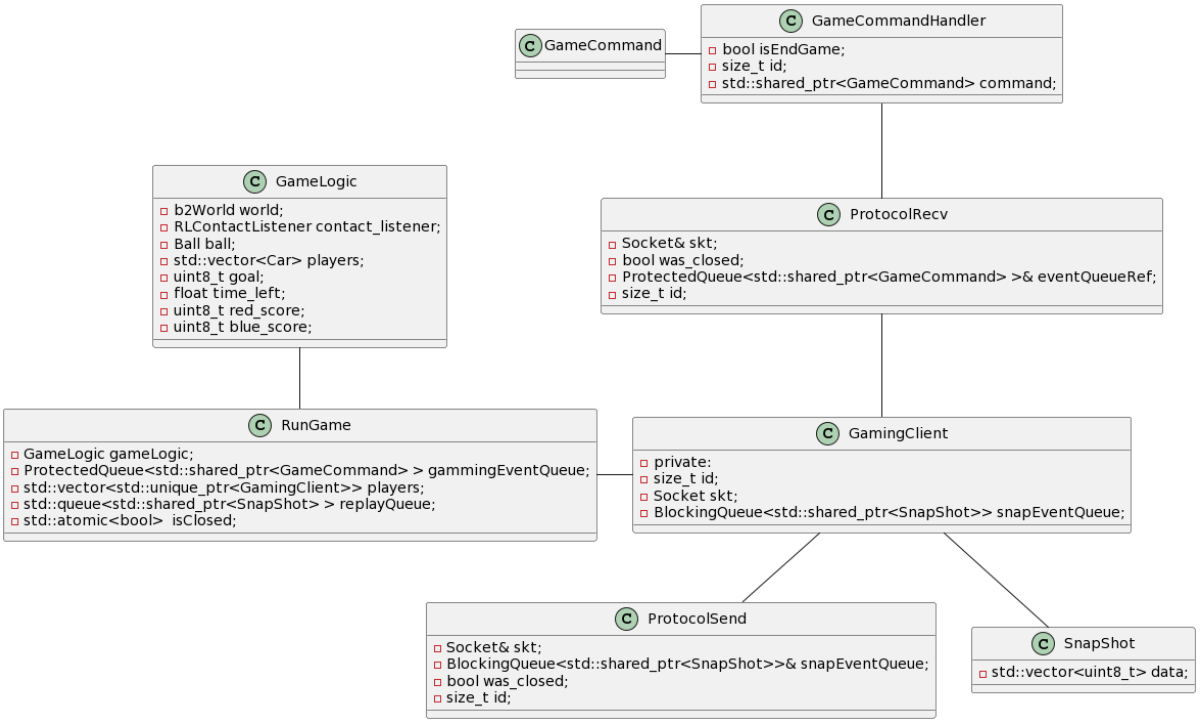


Fig.2

Parte dos del Server, partimos desde RunGame, en este hilo se accion eventos de juego (con **GameCommand** de forma parecida a MenuCommand, pero esta vez estos se administran por un **ProtectedQueue**) y se pide a **GameLogic** que simule un step para luego pedir una foto del estado del juego (**Snapshot**); está snap tiene que ser distribuida a cada Cliente(actualmente **GamingClient**).

4.Cliente

Una vez entendido el Server el Cliente es más fácil, al menos en su idea base, ya que el mismo debe seguir las reglas del protocolo y agregarle la dificultad de combinar dos interfaces visuales para brindarle al usuario una experiencia agradable a la vista.

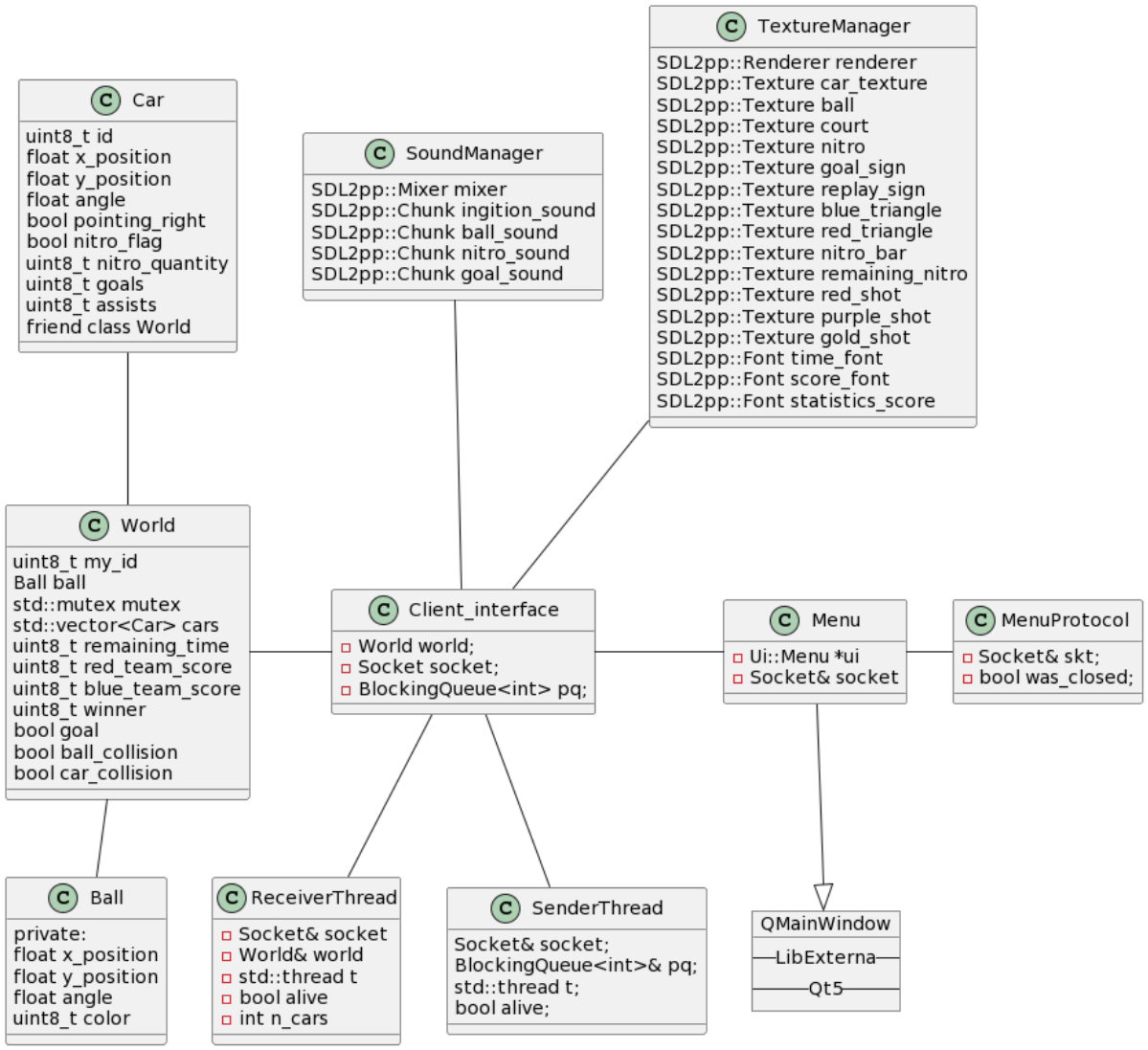


Fig. 3

Coordinando con lo visto en Server, el Cliente utiliza **Client_interface**(que a su vez delega toda la primera parte a **Menu** junto su **MenuProtocol**) mientras que del lado del server cuando el Client es un MenuClient(finaliza al unirse o crear a una partida).

Cuando del lado del Server pasamos a StanByClient aca finalizamos la parte de Menu, y pasamos a renderizar **SoundManager** y **TextureManager**(estos renderizados se realizan solo una vez en el ciclo de juego, y diría en el scope total del programa), al finalizar el renderizado esperamos el resto de la respuesta del server para obtener los n_cars y mi id

Recibimos los n_cars y tu id personal y … ¡Enhorabuena! ya inicia el verdadero juego, consta en recibir por **ReciverThread** datos que modifican el **World**(incluye modificaciones en **Ball** y **Car**) para luego dibujarlo con los managers mientras que envió eventos de teclado desde Client_inteface a **SenderThread**. Cabe aclarar que World en realidad no tiene un Car sino un Car por cada cliente de la partida, y la cantidad de Car a crear depende del n_car recibido.

Finalizando el tiempo que recibimos, mostramos unas estadísticas que proveerá el Server. Para finalizar completamente el Cliente deberá presionar la “q” desde la interfaz.

5.Protocolo

Por lo visto anteriormente se sabe que hay tres partes en los protocolos mínimamente(parte de menu, parte de juego parte final).

MenuProtocol

Resumiendo a groso modo esta parte del protocolo es enviar el largo del mensaje y luego el mensaje en sí, sin embargo los mensajes tienen que seguir el siguiente orden del lado del cliente:
<Command> <parameters>
“UNIR 4 tomako”

UNIR es el Command mientras que el resto serán parámetros del mismo.
Por parte del Server la mayoría de las respuestas enviadas al los cliente es en función de los que la clase **Command** almacene en su response.

Recordar envías largo y luego el mensaje.

GameProtocol

Acá la cosa se pone peliaguda en el Server el Snapshot es cargado por GameLogic al finalizar la emulación de su step y es mandado a cada GamingClient, como el mensaje de SnapShot solo varía en función de n_cars, se puede considerar el largo de los mensajes constantes. Del lado del cliente solo envia 1 byte para iniciar el ciclo de GameCommand.

```
primero     y una sola vez     [ncars]  [id]

byte partido   [ tiemporestante:uint8 ] [ scoreteam1:uint8 ] [ scoreteam2:uint8 ]
               [ flagGol:bool ] [ pelotaSound:bool ] [ flagutoChoquePiso:bool ]


ball           [ posx : uint32_t ] [ posy : uint32_t] [ ang :uint32_t ]
               [ colorPelota:uint8 ] 0 1 gold  2 red 3 purple

id 0 primer id
car            [ id :uint8 ] [ posx : uint32_t ] [ posy: uint32_t ] [ ang: uint32_t ]
               [ flagOrientacion:bool ][ flagnitro:bool ] [ nitro:uint8 ]
```

FinishProtocol

En este caso utilizamos la clase SnapShot, pero le cargamos:

```
[TeamGanador:uint8] (0 Rojo 1 Blue 3empate)

lo siguente va n_veces
[golesdeljugador:uint8]
[cantidad asistencias del jugado:uint8]
```

6. Detalles y consejos

1.Detalle

Siendo sinceros Snapshot solo es un vector de uint8 solo almacena data, el Cliente literalmente castio cada float, bool y uint8 a manopla en la etapa de juego y finalización.

Dicho esto, más allá del casteo y de tener cuidado como cargas el SnapShot .. claramente del lado del Client quedará a refactor el siguiente Sprint modificar esto.

```
//UPDATE FLAGS
this->remaining_time = data[0];
this->blue_team_score = data[1];
this->red_team_score = data[2];
this->goal = data[3];
this->ball_collision = data[4];
this->car_collision = data[5];

//UPDATE BALL
this->ball.x_position = FC(buf+6);
this->ball.y_position = FC(buf+10);
this->ball.angle = (-180/PI)*FC(buf+14);
this->ball.color = data[18];

if (this->ball.color != 0) std::cout << (int)this->ball.color << std::e

//UPDATE ALL CARS
for(size_t i=0; i < this->cars.size(); i++){
    this->cars[i].id = (uint8_t) data[((i) * 16 + 19)];
    this->cars[i].x_position = FC(buf + (i*16 + 20));
    this->cars[i].y_position = FC(buf + (i*16 + 24));
    this->cars[i].angle = (-180/PI)*FC(buf + (i*16 + 28));
    this->cars[i].pointing_right = (uint8_t) data[i*16 + 32];
    this->cars[i].nitro_flag = (uint8_t) data[i*16 + 33];
    this->cars[i].nitro_quantity = data[i*16 + 34];
}
```

2.Detalle

El ciclo de GameCommand es el Cliente toca una tecla se envía, el Server recibe los parsea por id a GameCommand con el GameHandler lo pushea a una cola protegida, se popea en RunGame y actua sobre Gamelogic, y se libera.

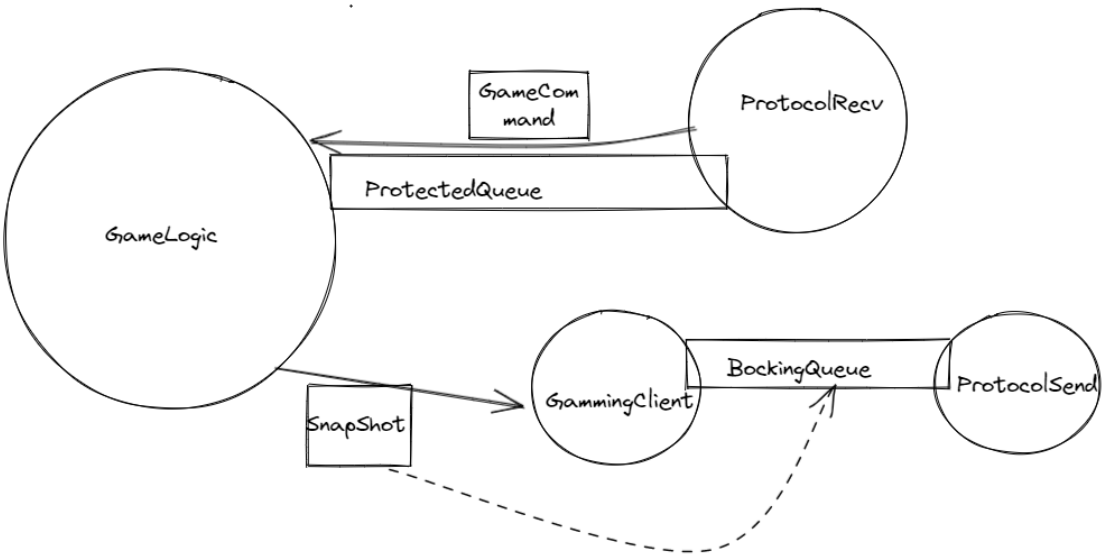
En el caso de los Command(comandos para menu) es se parsea y se intenta correr en GameHandler(monitor de Games), en el caso de error se catchea en el mismo Command para enviar un mensaje de fail Request.

3.Detalle

Falta limpiar el RunGame del Servidor al finalizar el juego, para que no sea una carga innecesaria en el mismo, por favor realizar en el siguiente Sprint.

4.Detalle

La comunicación en el Server al momento de la lógica se puede resumir a:



1.Consejo

Al agregar CommandNuevos tanto del lado de Menu como del RunGame, revisar las entidades Command y GameCommand.

Conclusión

Es un Tp interezante, bastante abarcativo en el que teníamos que coordinar distintas tareas concurrentes y aprender a utilizar distintas librerías en un mismo proyecto, y todo esto intentando ser los más RAII posible.