

ENTREGA 2

PROYECTO ROBOCUP

GRUP03

INDICE

<u>INTRODUCCIÓN</u>	3
<u>DIAGRAMA DE CLASES</u>	5
<u>CODIGO</u>	6
CLASE JUGADOR	6
CLASE DATOS	7
Clase SeeVar	8
Clase SenseVar	8
Multithreading	8
Parseo	8
<u>ESTRATEGIA DE JUEGO</u>	9
<u>PLANIFICACIÓN</u>	9
<u>FLUJOGRAMAS</u>	10

INTRODUCCIÓN

Para la programación de nuestro equipo utilizamos una clase principal Jugador. En un principio hicimos otras cuatro clases que heredaran de esta, pero, al ver la inutilidad de esto, decidimos quitarlas y dejar solo una, Portero, ya que este tiene la función catch que es exclusiva y así evitamos posibles errores. Para iniciar un jugador se han creado cuatro programas ejecutables diferentes (portero, defensa, medio y delantero). De momento utilizaremos una estrategia 4-4-2 por lo que lanzaremos 1 portero, 4 defensas, 4 medios y 2 delanteros. Más adelante explicamos como se colocan y como se mueven cada uno de ellos.

De lo programado hasta ahora lo más importante son las funciones que se encargan de guardar los datos que recibimos del servidor. Para esto hemos creado una librería que se encarga de parsear las cadenas recibidas. Una vez encontradas las instrucciones y los atributos los guardamos en una clase Datos que hemos creado a tal efecto.

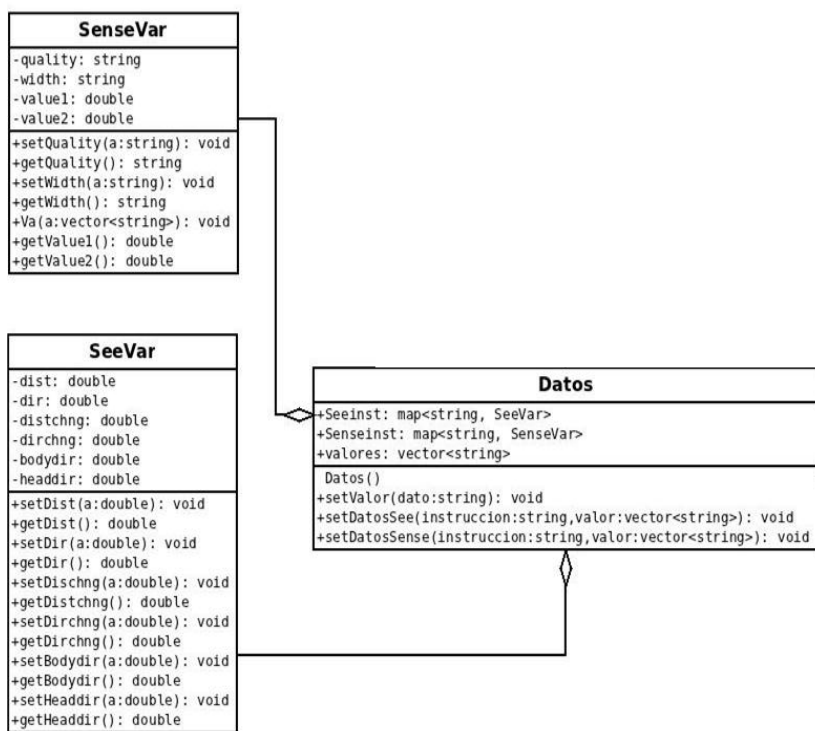


Figura 1: Diagrama de la clase Datos

Como podemos ver en la figura 1, en la clase Datos hay dos atributos de tipo `map<string, SeeVar>` `Seeinst` y `<string, SenseVar>` `Senseinst`. En estas variables guardamos los datos para cada instrucción en los distintos campos que nos ofrecen **SeeVar** y **SenseVar**. Posteriormente explicaremos en detalle de que están compuestas estas dos clases y las funciones que restan.

Para las comunicaciones con el servidor hemos utilizado una clase sockets programada a tal efecto que nos simplifica la utilización de estos. En la página 11 se puede ver el flujograma de conexión (figura 5)

También hacemos uso de threads. Con ellos somos capaces de tener 2 procesos al mismo tiempo. El que más nos interesa es el proceso mediante el cual estamos continuamente recibiendo datos y guardándolos. Por otro lado continuamos la ejecución del programa y enviamos información.

Posteriormente se explicará como hemos realizado la alternancia entre los threads enviar y recibir para evitar problemas mediante mutex y una variable booleana.

Hemos tratado de separar los datos de la lógica del programa para que estén lo más desacoplados posibles y su uso sea más sencillo y menos propenso a errores. Por eso hemos usado las distintas librerías y creado la clase Datos separada de Jugador. En la figura 2 se muestra el diagrama de clases completo.

DIAGRAMA DE CLASES:

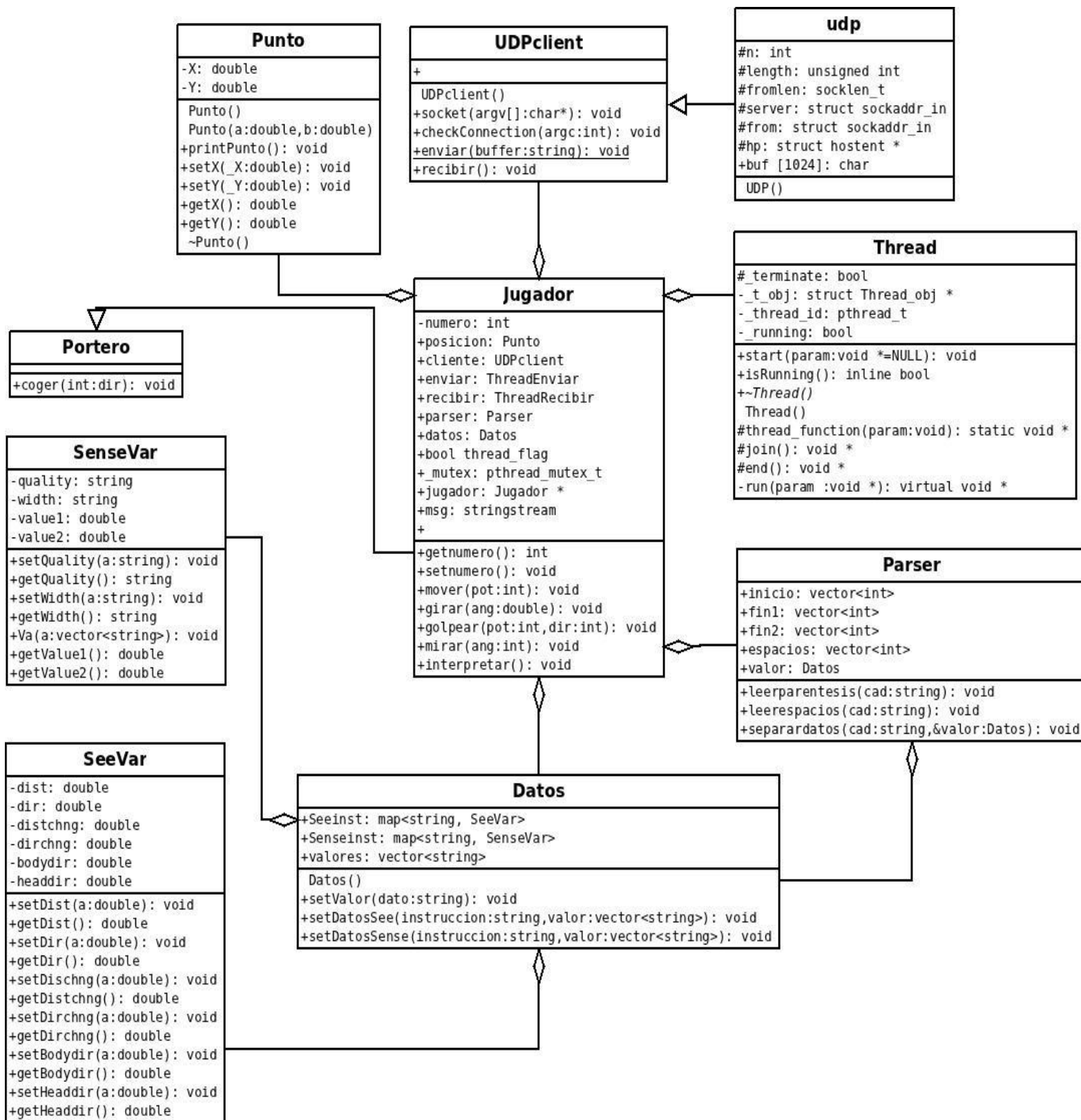


Figura 2: Diagrama de Clases

CODIGO:

CLASE JUGADOR

- **Atributos:**

- numero: único atributo privado. Es de tipo entero y en él se almacenan el numero del jugador.
- posición: objeto de tipo Punto. Se utiliza para colocar al jugador en una posición XY del campo y se utilizara para referir puntos en el espacio.
- cliente: objeto de tipo UDPclient. Se utiliza para las conexiones con el servidor.
- enviar: objeto de tipo ThreadEnviar. Lo usamos para lanzar el thread de envio.
- recibir: objeto de tipo ThreadRecibir. Lo usamos para lanzar el thread de recepción de datos.
- parser: objeto de tipo Parser. Necesario para parsear la cadena.
- datos: objeto de tipo Datos. En esta clase almacenamos los distintos datos recibidos.
- thread_flag: atributo de tipo booleno. Necesaria para controlar los threads.
- _mutex: atributo de tipo pthread_mutex_t. Necesaria para controlar la exclusión mutua de los threads y evitar la apropiación indebida de recursos.
- jugador: objeto de tipo Jugador*. Necesario para poder utilizar en los threads las funciones de esta clase.
- msg: atributo de tipo stringtam. Necesario para simplificar la introducción de datos y enviarlos desde el thread.
- inicio, fin1 y fin2: atributos de tipo vector. Tras realizar el parseo de la cadena almacenamos en ellos las posiciones en las que se encuentran los paréntesis para así poder fraccionar la cadena recibida.

- **Funciones:**

- getnumero(): devuelve el número del jugador.
- setnumero(int Nnumero): asigna a numero el valor Nnumero.
- mover(int pot): asigna a msg el string (**dash pot**) donde pot es un valor entero que indica la potencia del movimiento.
- girar(double ang): asigna a msg el string (**turn mom**) donde mom es el momento de giro. Este depende del ángulo (ang) que se quiera girar según una ecuación proporcionada en en manual de Robocup. Esta es: $mom = ang * (1 + 5 * velocidad)$
- golpear(int pot, double dir): asigna a msg el string (**kick pot dir**) donde pot es la potencia de lanzamiento y dir la dirección del mismo.

- mirar(double ang): asigna a msg el string (*turn_neck ang*) donde ang es el ángulo de giro del cuello que se quiere variar.
- interpretar(string h): esta función se encarga de parsear la cadena y colocar los datos.
- constructor: en el colocamos al jugador en la posición inicial indicada e iniciamos la conexión con el servidor.

Clase Datos:

- **Atributos:**

- Seeinst: atributo de tipo map<string,SeeVar>. El string será la instrucción que queramos almacenar. Como veremos a continuación, en seeVar se encuentran las distintas variables que pueden existir para los distintos tipos de instrucciones de tipo See.
- Senseinst: atributo análogo al anterior para instrucciones de tipo Sense.
- valores: atributo de tipo vector. En el guardamos los valores que nos llegan para una misma instrucción para, posteriormente, descargarlos en sus lugares de destino.

- **Funciones**

- setValor(string dato): añade al final del vector valores el dato deseado haciendo uso de la función pus_back();
- setDatosSee(string instruccion,vector<string>valor): Esta función guarda en *Seeinst* los valores para cada instrucción, haciendo uso de las funciones *get* de la clase SeeVar. Se ha utilizado un switch ya que el servidor no siempre manda el mismo número de datos para cada instrucción y estos pueden variar de 1 a 6. El orden de envío siempre sigue un patrón fijo: Distancia, Dirección, Cambio Distancia, Cambio Dirección, Dirección Cuerpo, Dirección Cabeza. Por lo que un case era la mejor opción, ya que, según el tamaño del vector de valores recibidos llenaremos los datos que nos hagan falta.
- setDatosSense(string instruccion,vector<string>valor): Esta función guarda en *Senseinst* los valores para cada instrucción. Diferenciamos entre tres posibles casos. El primero si la instrucción es de tipo “*view_mode*” hay que rellenar los datos *Quality* y *Width*. El segundo caso si la instrucción es de tipo “*hear*” por el momento no hacemos nada con ella. Por último para el resto de caso hacemos uso de la función setValue de la clase SenseVar (se explica a continuación).

Clase SenseVar:

Esta clase almacena los valores que pueden tener las instrucciones de tipo *sense*. Tenemos cuatro atributos privados. Dos de ellos de tipo string (*quality* y *width*) que son para las instrucciones de tipo de "*view_mode*". Los otros dos atributos son de tipo double (para el resto de instrucciones).

En cuanto a las **funciones** tenemos las funciones *get* y *set* para cada atributo y una función *setValue*.

La función *setValue(vector<string> a)* recibe un vector *a* de tipo string. El tamaño de este vector puede ser de uno o dos, según sea el numero de valores recibidos. Haciendo uso de la variable *ss* (de tipo *stringstream*) facilitamos el paso de string a double utilizando el operador *>>*. Previamente en la función hemos cargado en *ss* los valores que deseamos almacenar.

Clase SeeVar:

Esta clase almacena los valores que pueden tener las instrucciones de tipo *see*. Tenemos seis atributos privados de tipo double y como funciones sus *get* y *set* coreespondientes.

Multithreading:

Para la programación de los threads hemos hecho uso de una librería y creado dos tipos de threads diferentes. Uno para enviar y otro para recibir. Como parámetro le pasamos el objeto jugador, de la clase Jugador.

Para evitar conflictos entre threads hemos hecho uso de un mutex y una variable de tipo booleana común alojada en la clase Jugador. De tal forma que cada vez que un thread realiza un ciclo cambia el valor de la booleana impidiéndose a si mismo volver a actuar nada más terminar el ciclo. Para asegurarnos, tras comparar que es su turno, utilizamos el mutex para bloquear el thread y dejar al siguiente en espera.

La ejecución de los threads es muy simple. El que se encarga de recibir hace uso del objeto cliente para recibir información y llama a la función *interpretar* para almacenar los datos recibidos. En cuanto al que envía, hace uso del mismo objeto y envía la información que haya almacenada en el atributo *msg*.

Parseo:

Para más información sobre el parseo de cadenas ver: http://ii2-grupo3.googlecode.com/files/ENTREGA_PARSER.pdf

ESTRATEGIA DE JUEGO:

Hemos decidido jugar con una estrategia 4-4-2. Cada jugador tendrá un área delimitada de juego por lo que haremos una programación por eventos. Esto quiere decir que el jugador actuara en función de la posición en la que se encuentren tanto él como el balón. La lista de eventos está por determinar. El flujograma para los jugadores se puede ver en las figuras 3 y 4 al final del documento.

PLANIFICACIÓN:

Como ya están creadas la mayoría de funciones principales podremos centrarnos principalmente en la programación de los eventos que harán que nuestro jugador realice unas acciones u otras. En principio el reparto de tareas sigue siendo el mismo que hasta ahora. La programación del portero se realizará entre todos y el resto uno cada uno.

Puesto que vamos a basarnos en eventos vamos a estudiar la posibilidad de utilizar un patrón de eventos y ver si nos simplifica el manejo de estos.

Aunque todavía no se ha implementado hemos decidido utilizar un patrón creacional para instanciar nuestros jugadores de tal forma que solo nos haga falta una función *main()* principal y, a través de este patrón y en función del número que nos devuelva el servidor se cree automáticamente el jugador en la posición que le corresponda y actúe según se haya programado para tal jugador.

FLUJOGRAMAS:

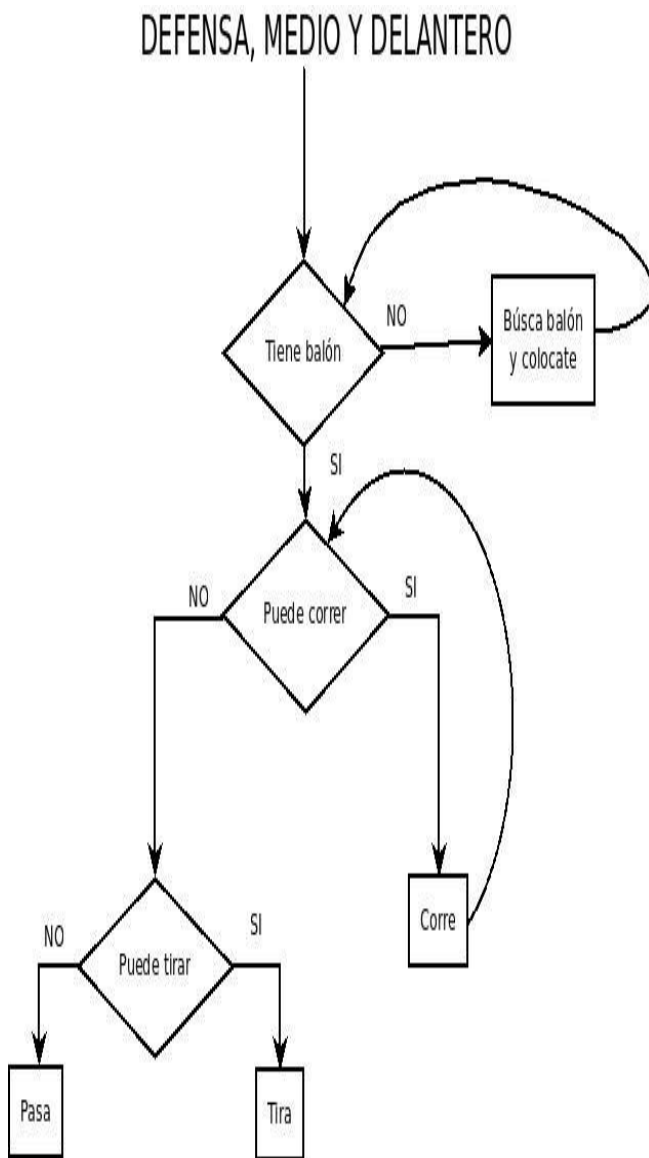


Figura 3: Flujograma para defensas, medios y delanteros

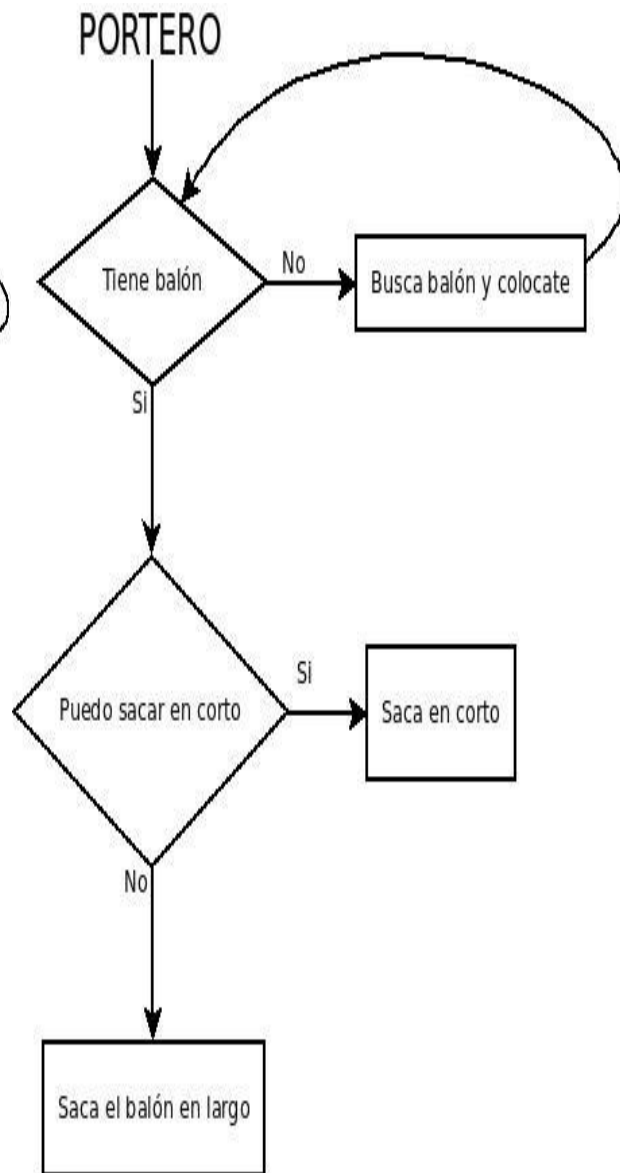


Figura 4: Flujograma para portero

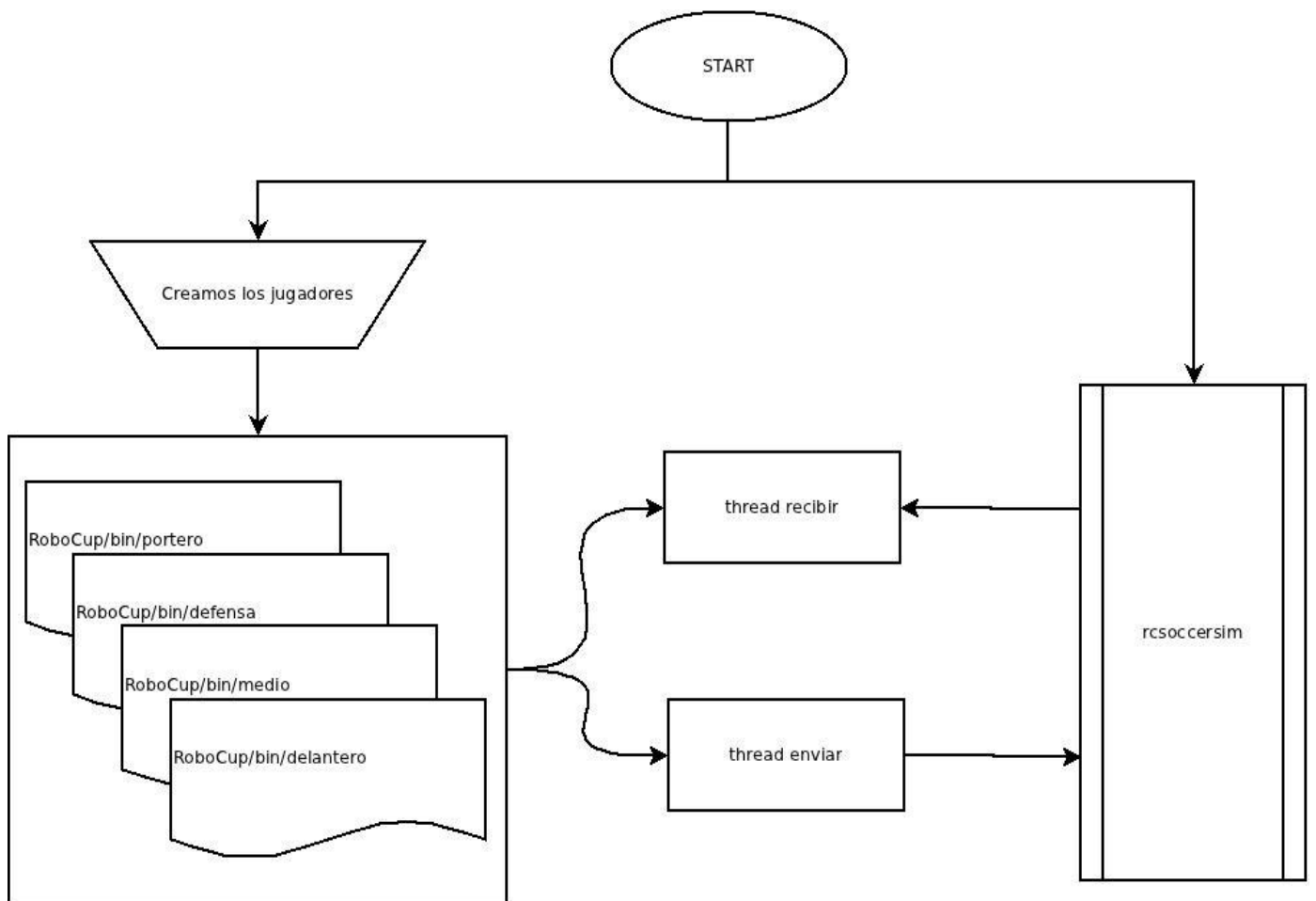


Figura 5: Flujograma de la conexión entre el jugador y el servidor