## Implementación del enlace entre el algoritmo y el motor de juego: REST API

Tal y como planteé en el planteamiento del diseño, en el apartado 3.1, el diseño iba a estar dividido en tres partes: el motor de juego (en c++), el algoritmo (en R) y el enlace entre ambos.

En este apartado voy a explicar cómo he implementado el enlace entre ambas partes del código. Para ello, he decidido establecer un REST API como método de enlace, dado que es un elemento sencillo de programar y permite el enlace entre lenguajes de programación distintos.

Para ello, he decidido que el algoritmo sea el servidor y el motor de juego el cliente. He decidido esto en base a que es el motor de juego el que solicita la información al algoritmo durante el desarrollo de la partida y es el algoritmo el que devuelve la respuesta.

Este REST API, como enlace entre ambos, tiene su parte de código en el código del algoritmo y parte de su código en el motor de juego.

Vamos a empezar con el servidor: el algoritmo en R.

### Servidor en R: rplumber

Para establecer el servidor API REST en R es necesario utilizar el paquete Plumber dentro de R. Este paquete incluye todas las funcionalidades necesarias para crear el servidor de un API REST en la dirección <http://localhost:puerto>, siendo el puerto definido en la llamada a la función de creación del servidor.

Para poder establecer este servidor, es necesario primero crear un nuevo script que va a incluir los Endpoints de la API REST. Este archivo es el plumber.R.[[1]](#footnote-1)

Para definir los valores de las funciones, la estructura es la siguiente (cabe destacar que es necesario usar #\* para distinguirlo de los comentarios normales y que plumber los identifice adecuadamente):

#\* Definición de la función

#\* tipo de salida (si no se especifica, será JSON)

#\* parámetros pi

#\* endpoint

Function(pi){

Métodos de la función

}

Siendo funciones de ejemplo las siguientes:

*#\* Echo back the input*

*#\* @param msg The message to echo*

*#\* @get /echo*

**function**(msg**=**""){

**list**(msg **=** paste0("The message is: '", msg, "'"))

}

*#\* Plot a histogram*

*#\* @png*

*#\* @get /plot*

**function**(){

rand **<-** rnorm(100)

hist(rand)

}

*#\* Return the sum of two numbers*

*#\* @param a The first number to add*

*#\* @param b The second number to add*

*#\* @post /sum*

**function**(a, b){

**as.numeric**(a) **+** **as.numeric**(b)

}

### Contenido de plumber.R

Para este proyecto, utilizaré un total de 13 funciones dentro de plumber, todas de endpoint GET ya que interactúan con dataframes y, por tanto, no necesitan cookies para trabajar.

Las funciones son las siguientes:

* **preflopini(mn1,mp1,mn2,mp2)**

Esta función se ejecuta para determinar la primera acción del algoritmo durante el preflop siendo el jugador inicial. Los parámetros de esta función son los números y palos de las cartas de mano del algoritmo, respectivamente.

Primero, se usará fijarMano con los parámetros de entrada creando así mano, se cargarán los dataframes de mano, pesos y triple con sus respectivas funciones, se descartarán las cartas de mano usando descarteCartas. Con esto, se descartan también los pesos y las triples probabilidades de las cartas que impliquen esas cartas con descartePesosMano y descarteTripleMano.

Posteriormente, se almacenan los datos de mazo, pesos, triple y mano en sus respectivos dataframes usando write.csv. Y, por último, se llama a calculaProbPreflopIni para calcular la probabilidad de acción del algoritmo, que se devolverá como salida usando return.

* **Preflop(mn1,mp1,mn2,mp2,a)**

Esta función se ejecuta para determinar la primera acción del algoritmo durante el preflop cuando no es el jugador inicial, siendo la entrada tanto las cartas, al igual que en preflopini, como la acción a. Esta función funcionará de manera similar a preflopini, pero añadiendo unos métodos.

Tras crear mano, cargar los dataframes, descartar las cartas, los pesos y los triples de la misma manera que preflopini, se actualizan los pesos con la acción a, y se recalculará las triple probabilidades con calculoProbabilidadAccion.

Después se guardan los respectivos dataframes y se calcula la salida de igual manera que preflopini.

* **Preflopact(a)**

Esta función se ejecuta para determinar posteriores acciones del algoritmo durante el preflop, independientemente de si es jugador inicial o no.

Primero, cargará los dataframes de pesos, triple y mazo. Tras eso, se actualiza la tabla de pesos con actualizaPesos y se recalcula la tabla de probabilidades triples con calculoProbabilidadAccion. Dado que solo se han modificado en esta función estos dos últimos dataframes, serán los únicos que se guarden con write.csv.

Por último, se llama a calculaProbPreflopIni para calcular la probabilidad de acción del algoritmo, que se devolverá como salida usando return.

* **flopini(mn1,mp1,mn2,mp2,mn3,mp3,aa, ao)**

Esta función se ejecuta para determinar la primera acción del algoritmo durante el flop siendo el jugador inicial. Los parámetros de esta función son los números y palos de las cartas de la mesa en esta ronda, y siendo aa y ao las apuestas del algoritmo y del oponente, respectivamente.

Primero, cargará los dataframes de pesos, triple, mano y mazo. Se fijan los valores de las cartas usando fijarMesaFlop con las tres cartas, generando la matriz Mesa. Esta matriz Mesa se utiliza con descarteCartas para filtrar las cartas restantes del mazo.

Una vez obtenido el mazo filtrado, se descartan los pesos y triple probabilidad con descartePesosFlop y descarteTripleFlop respectivamente. Se actualizan los pesos con actualizaPesos, siendo la acción a introducir la acción de ver la apuesta, ya que es el jugador inicial de la ronda.

Posteriormente, se calculan el valor de salida usando calculaProbFlop y se dan los nuevos valores de triple probabilidad con calculaProbabilidadAccion. Por último, se sobreescriben los dataframes modificados (deck, pesos, triple y mesa) y se devuelve la salida con un return.

* **Flop(mn1,mp1,mn2,mp2,mn3,mp3,a,aa, ao)**

De manera análoga a lo que ocurría con preflopini y preflop, esta es la función equivalente a flopini cuando el algoritmo no es el jugador inicial. El parámetro adicional a flopini es la acción del oponente.

El funcionamiento de esta función es idéntico al de la función flopini, únicamente cambiando el valor de actualizaPesos y calculaProbFlop predefinido por el valor a.

* **Flopact(a,aa,ao)**

Esta función es la función utilizada para determinar las posteriores acciones del algoritmo durante la ronda de flop. Dado que la actualización de acciones es idéntica en flop y turn, esta función también se utilizará para determinar acciones posteriores a la primera durante la ronda de Turn.

Primero cargará los 5 dataframes necesarios (mano, deck, mesa, pesos y triple), después actualizará los pesos con actualizaPesos y calculará la salida con calculaProbFlop. Por último, al igual que pasaba con preflopact, el único dataframe modificado es pesos, por lo que será el único que necesitará ser sobrescrito.

* **Turnini(mn,mp,aa,ao)**

Tal y como se explicó en el apartado 3.4.2.2, el procedimiento de la toma de decisiones en Flop y Turn era prácticamente idéntico, por lo que turnini tendrá un funcionamiento muy similar al de flopini.

Después de cargar los 5 dataframes (mano, deck, mesa, pesos y triple), se fija la nueva mesa con fijarMesaPostflop, se descartan las cartas de mazo, se descarta los valores de las tablas de la carta asociada con descartePesosCarta y descarteTripleCarta y se actualizan los pesos con actualizaPesos.

Por último, se calcula la salida con calculaProbFlop, se actualiza la tabla de triple probabilidad con calculoProbabilidadAccion y se guardan los 4 dataframes modificados (deck, data, triple, mesa).

* **Turn(mn,mp,aa,ao,a)**

Esta función es la equivalente a turnini para obtener la primera acción del algoritmo durante la ronda Turn cuando no es el jugador inicial.

Al igual que pasa con flop y flopini, la única diferencia entre turn y turnini es el valor que se introduce en actualizaPesos y calculaProbFlop, que en esta función es a.

* **riverini(mn,mp)**

Esta es la función para obtener la primera acción del algoritmo durante la fase de river cuando es el jugador inicial.

Se cargan los dataframes necesarios, que son mano, deck, mesa, pesos y triple; se fija la nueva mesa con fijarMesaPostflop, se descartan las cartas de mazo, se descarta los valores de las tablas de la carta asociada con descartePesosCarta y descarteTripleCarta y se actualizan los pesos con actualizaPesos.

Después de eso, se calcula la salida con calculaProbRiver y se actualizan los valores de la tabla de triple probabilidad con calculoProbabilidadAccion.

Por último, se sobrescriben los 4 dataframes modificados y se devuelve la salida.

* **river(mn,mp,a)**

De igual manera que ocurre en las otras rondas, esta es la función para obtener la primera acción del algoritmo durante la fase de river cuando no es el jugador inicial.

Su funcionamiento es idéntico al de riverini, cambiando el valor fijo de actualizaPesos y calculaProbRiver por el valor de entrada a.

* **riveract(a)**

Esta función sirve para obtener las posteriores acciones del algoritmo durante la ronda de River.

Primero se cargan los 5 dataframes, se ejecuta la actualización de pesos con actualizaPesos y se calcula la salida con calculaProbRiver.

Antes de devolver la salida con un return, se sobrescribe la tabla de pesos con write.csv.

* **reset()**

Esta función se utiliza para reiniciar los valores de los dataframes deck, pesos y triple probabilidad.

Para restaurar los valores de deck, se llama a la función crearMazo.

Para restaurar los valores de pesos y triple, se cargan los valores de los dataframes base, data\_ini.csv y triple\_csv respectivamente. Con estas tres matrices, se sobrescriben los dataframes de uso con write.csv.

* **error()**

Esta función tiene como única función el devolver un error en caso de que ocurra un error en la comunicación.

### Puesta en marcha del Servidor de API REST

La puesta en marcha del servidor, tal y como se ha programado, es bastante sencilla.

Estos comandos se tienen que utilizar desde la consola de R para poder mantener adecuadamente el servidor.

El primer paso, es inicializar todas las funciones y scripts del código. Ya que he creado la función inicializa() que inicializa todas las funciones, solo es necesario usar los siguientes comandos:

source(“inicializa.R”)

inicializa()

Tras esto, se ejecutan los comandos de plumber. Lo primero es inicializar la librería, para lo cual se utiliza el comando library(plumber), después es necesario cargar el archivo plumber.r configurado como tubería. Para eso, se utiliza el comando r<-plumb(“plumber.R”).

Con esto, r será la variable que tiene toda la información del servidor API REST, así como las funciones para cargarlo.

Por último, se ejecuta el comando r$run(port:8000) para ejecutar el servidor en la url <http://localhost:8000/>. Se puede utilizar r$run() sin asignar un puerto, entonces el programa te lo asigna a uno aleatorio, pero para poder llamarlo sin problemas desde el cliente, es necesario fijarlo a un puerto, por lo que utilizaré el puerto 8000.

Con esto, el servidor estará en marcha y devolverá al cliente la respuesta acorde a la URL llamada de manera automática.

Para salir o interrumpir el funcionamiento del servidor, se pulsa la tecla ESC en el teclado desde la consola de R, haciendo que el funcionamiento de r$run se detenga.

### Cliente en C++: curl y rapidjson

Hay varias formas de comunicar un código de C++ y una API REST. Una de las maneras más sencillas es el uso del paquete Curl[[2]](#footnote-2), que permite utilizar los comandos que se utilizan en el paquete de MSDOS del mismo nombre para comunicarse con una API.

Para poder traducir los vectores de números en formato json que devuelve la API REST programada, es necesario instalar otro paquete, que es el paquete Rapidjson[[3]](#footnote-3).

He elegido estas opciones ya que no necesita una aplicación intermedia para traducir ni conectar con el API. Además, ya que ambos se pueden codificar en c++, facilita mucho el trabajo y es mucho más cómodo trabajar con ellos.

Ambos paquetes se pueden instalar usando vcpkg, utilizando el comando vcpkg install curl y vcpkg install rapidjson respectivamente.

Ahora procedo a explicar como he implementado estos paquetes en la clase Algoritmo.

Lo primero es llamar a las librerías instaladas con #include <curl/curl.h> y #include <rapidjson/document.h> dentro del código de Algoritmo.cpp.

Tras esto, se crea una estructura llamada MemoryStruct, donde se almacenará el json después de ser analizado.

struct MemoryStruct {

char\* memory;

size\_t size;

};

El tipo size\_t se define como un valor static. Tras esto, se crea la función que va a ser utilizada para almacenar en la memoria el json: WriteMemoryCallback(void\* contents, size\_t size, size\_t nmemb, void\* userp)

Esta función crea un elemento de MemoryStruct usando el contenido de userp, y una variable size\_T que es el valor de nmemb. Reasigna el contenido de la memoria a un char\* usando realloc y, si hay suficiente espacio en memoria, copia el valor de contents a la estructura de memoria, actualizando el size\_t creado.

Una vez creados estos dos elementos auxiliares, vamos a la implementación directa. Esta implementación se realiza en las funciones obtenerTriple de la clase (obtenerTriple, obtenerTripleAccion y ObtenerTripleAct) se realizará la implementación de la misma manera, cambiando las urls a las que llamar según corresponda.

Primero se crea un objeto estructura MemoryStruct y se definen los valores de memory (usando (char\*)malloc(1)) y size(0).

Tras esto, se hacen las inicializaciones de curl. Primero se ejecuta curl\_global\_init(CURL\_GLOBAL\_ALL) para iniciar la conexión general, tras eso se crea un elemento del tipo CURL\* llamado curl que es el resultado de curl\_easy\_init() y un objeto del tipo CURLcode llamado res.

El siguiente paso es inicializar la parte de rapidjson, que consiste en crear un objeto del tipo rapidjson::Document, llamado document.

Después de las inicializaciones, se crea la url a llamar, en función de la ronda, la mano del algoritmo, las cartas de la mesa, las apuestas y la acción según convenga. Para poder concatenar los valores en un valor char\* se utilizan tanto strcpy como strcat para generar la url correspondiente.

Una vez obtenida la url, podemos realizar la petición al servidor. Para esto, se llamará a la función curl\_easy\_setopt 3 veces, cada una con una inicialización distinta con el objetivo de almacenar en la variable curl los comandos a ejecutar:

* curl\_easy\_setopt(curl, CURLOPT\_URL, url): sirve para establecer la conexión en el valor de url.
* curl\_easy\_setopt(curl, CURLOPT\_WRITEFUNCTION, WriteMemoryCallback) se utiliza para determinar que se va a utilizar una función de escritura de memoria, llamando a la función WriteMemoryCallback.
* curl\_easy\_setopt(curl, CURLOPT\_WRITEDATA, (void\*)&chunk): sirve para definir dónde se va a almacenar los datos del comando WRITEFUNCTION, es decir, donde se almacenará el resultado de WriteMemoryCallback.

Una vez definidos los comandos, se ejecuta curl usando la función curl\_easy\_perform, cuya salida se almacenará en la variable res.

Ahora que ya hemos establecido la conexión y recibido el json, es necesario analizarlo y traducirlo.

Para esto, se utiliza document.Parse con entrada el valor memory de la estructura creada. Tras esto, se comprueba que document incluye un array de números usando assert(document.IsArray()), que devolverá un error si, tras la traducción, es un array.

Ahora que tenemos el array traducido, se asocian los resultados a la variable triple de la clase algoritmo.

Por último, se implementan las limpiezas de memoria y los cierres de conexiones.

Para la limpieza de memoria, es necesario llamar a la función free(memoria), que vacia la variable memoria de la estructura creada.

Para limpiar las conexiones, es necesario llamar un cleanup por cada conexión hecha. Es decir, es necesario llamar a curl\_easy\_cleanup(curl) para cerrar curl\_easy\_init() y curl\_global\_cleanup() como cierre de curl\_global\_init(CURL\_GLOBAL\_ALL).

1. <https://www.rplumber.io/> [↑](#footnote-ref-1)
2. <https://curl.haxx.se/> [↑](#footnote-ref-2)
3. <https://rapidjson.org/> [↑](#footnote-ref-3)