PRÁCTICA 2 - RPC

Desarrollo de Sistemas Distribuidos

Sergio Hervas Cobo - A1

Abril 2022

1 Introduction

En este documento se recogerá la memoria realizada para la segunda práctica de la asignatura de Desarrollo de Sistemas Distribuidos, la cual está basada en la implementación de sistemas distribuidos con llamadas a procedimientos remotos (RPC). En la primera parte de la práctica se utilizará SunRP para construir un sistema distribuido implementandolo en lenguaje X, mientras que en la segunda se hará uso de un IDL que nos permitirá programar el cliente y el servidor en distintos lenguajes.

2 Primera parte - SunRPC

Para esta parte se debía implementar un sistema distribuido que cumpliese la función de una calculadora. Para ello, se deben especificar los procedimientos que se pretenden implementar y las distintas estructuras de datos que se utilizarán en un fichero *.x*, el cual generará los respectivos ficheros de C para el cliente y para el servidor donde se desarrollarán las distintas funciones de la calculadora.

En mi caso, en *dir.x* he especificado 21 procedimientos (Figura 1), que recogen operaciones simples, con fracciones, con vectores o con matrices. Además, se han declarado las estructuras de datos que se han necesitado para cada procedimiento (tanto para sus argumentos como para el dato que devuelve).

Una vez especificado todo lo que se pretende implementar en el programa, hay que generar los distintos ficheros en lenguaje C. Esto se ha realizado mediante el comando *rpcgen -NCa dir.x*. Como resultado, se crearon en su directorio los siguientes documentos:

- Cabecera donde se especifican las definiciones que comparten el cliente y el servidor (estructuras de datos y procedimientos)
- Ficheros xdr que se encargarán de traducir las estructuras de datos definidas
- Un stub para el cliente y otro para el servidor, que hacen de intermediario entre los dos programas para la llamada a procedimiento remoto.

```
program DIRPROG {
        version DIRVER {
                tipo_simple SUMA(operacion) = 1;
                tipo_simple RESTA(operacion) = 2;
                tipo_simple MULTIPLICACION(operacion) = 3;
                tipo_simple DIVISION(operacion) = 4;
                tipo_simple POTENCIA(operacion_2) = 5;
                tipo_simple RAIZ(operacion_2) = 6;
                tipo_simple LOGARITMO(operacion_2) = 7;
                tipo_vector SUMA_VECTORES(vectores) = 8;
                tipo_vector RESTA_VECTORES(vectores) = 9;
                tipo_simple PRODUCTO_ESCALAR(vectores) = 10;
                tipo_vector MULTI_VECTOR_ESCALAR(vectoryescalar) = 11;
                tipo_vector DIVI_VECTOR_ESCALAR(vectoryescalar) = 12;
                tipo_matriz SUMA_MATRICES(matrices) = 13;
                tipo_matriz RESTA_MATRICES(matrices) = 14;
                tipo_matriz PRODUCTO_MATRICES(matrices) = 15;
                tipo_matriz MULTI_MATRIZ_ESCALAR(matrizyescalar) = 16;
                tipo_matriz DIVI_MATRIZ_ESCALAR(matrizyescalar) = 17;
                tipo_fraccion SUMA_FRACCIONES(fracciones) = 18;
                tipo_fraccion RESTA_FRACCIONES(fracciones) = 19;
                tipo_fraccion MULTI_FRACCIONES(fracciones) = 20;
                tipo_fraccion DIVI_FRACCIONES(fracciones) = 21;
                tipo_fraccion SIMPLIFICAR_FRACCION(fraccion) = 22;S
        } =1;
} = 0x20000155;
```

Figure 1: Declaración de los procedimientos en el fichero dir.x

Cuando se obtuvieron todos estos ficheros, se implementaron las operaciones necesarias en cada procedimiento del servidor, asi como el menú de interacción del cliente para la elección del procedimiento a ejecutar. El programa es interactivo y, por tanto, el usuario decidirá qué operaciones hacer y qué datos utilizar, con la entrada por teclado y la salida de los resultados por la pantalla de la terminal. Esto lo detallaré a continuación:

2.1 Cliente

Mediante bucles anidados y órdenes switch, he implementado un menú interactivo para que el usuario de la calculadora elija las operaciones por teclado en la terminal. Las opciones son:

- **Operaciones simples:** Pedirá al usuario una operación simple y cuando devuelva el resultado, preguntará al usuario si quiere volver a introducir otra operación. En caso afirmativo, el usuario tendrá que volver a introducir otra operación. Ejemplo: *"3 + 4"* (suma) ó *"5 l 6"* (logaritmo). Incorpora las siguientes operaciones para números flotantes:
  - Suma
  - Resta
  - Producto
  - División

  - Potencia
  - Raíz n-sima
  - Logaritmo en base n

- **Operaciones simples (interactivo):** Igual que el anterior pero esta vez simulará una calculadora realista. Es decir, una vez devuelva el resultado y el usuario acepte volver a introducir otra operación, se deberá escribir otra operación **sobre** el resultado anterior. Ejemplo: *"3 + 4 = 12"* → *"12 * 3 = 36"*

- **Operaciones con fracciones:** Se ha creado una estructura de datos específica para estos procedimientos remotos, donde para representar a una fracción se utilizan dos enteros en un struct. El usuario deberá introducir la operación que desea realizar sobre las fracciones y el valor del numerador y denominador de estas. Se realizan las siguientes operaciones con fracciones:

  - Suma de fracciones
  - Resta de fracciones
  - Multiplicación de fracciones
  - División de fracciones
  - Simplificación de fracciones (para ello se ha implementado una función que realiza el máximo común divisor)

- **Operaciones con vectores:** Para las operaciones con vectores se utiliza una estructura que almacena el tamaño del vector y el array. El usuario deberá introducir el tamaño del vector y cada uno de los valores de este. Se implementan las siguientes operaciones:

  - Suma de vectores
  - Resta de vectores
  - Producto escalar
  - Multiplicación de vector por escalar
  - División de vector por escalar

- **Operaciones con matrices:** Se ha utilizado una estructura que contiene un vector, el número de columnas y el número de filas de la matriz. El vector será indexado como una matriz mediante la fórmula *fila * tam_columnas + columna*. El usuario introducirá el número de filas y de columnas y los valores de cada componente de la matriz (fila por fila), además de la operación deseada. Las operaciones posibles son:

  - Suma de matrices
  - Resta de matrices
  - Producto de matrices
  - Multiplicación de matriz por escalar
  - División de matriz por escalar

2.2 Servidor (1)

Para implementar el primer servidor solamente tuve que tener en cuenta los parámetros que se le pasaba y el objeto que devolvía. La implementación de la mayoría de los métodos es sencilla, exceptuando unos cuantos que requieren de algoritmos algo más complejos como la multiplicación de dos maltrices, la simplificación de fracciones o el producto escalar de vectores.

Algunos métodos se han implementado utilizando la librería *math* de C por simplicidad y eficiencia como la potencia de números o la raíz n-sima.

En cuanto a las matrices, al principio fueron implementadas en este servidor. Posteriormente, decidí crear otro servidor para que "soportase" las operaciones de matrices (suponiendo que tienen un coste significativamente más alto y el nuevo servidor tiene unas características más óptimas para almacenar estos datos y operar con ellos). A continuación hablaremos de este segundo servidor.

2.3 Servidor (2)

Como hemos dicho antes, si partimos de la suposición de que este nuevo servidor tiene mejores especificaciones, sería lógico delegar las operaciones "más complejas" a este para distribuir correctamente la carga de trabajo entre las máquinas. Por esto, haremos que el servidor 1 sea cliente de este nuevo servidor, que implementará las operaciones de las matrices. Para ello, se ha creado un nuevo fichero .x llamado *dir2.x* que especifica las operaciones de las matrices junto con las estructuras de datos que utilizarán. Estas, son las mismas que las especificadas en el archivo *dir.x* pero en esta ocasión el nombre será distinto para evitar errores.

Se generaron los distintos archivos al utilizar rpcgen y se modificaron los métodos del fichero C del segundo servidor con la implementación de las operaciones de las matrices. Además, se modificaron estos métodos en el primer servidor para llamar desde estos a los servicios del nuevo server, convirtiéndolo en cliente.

2.4 Pruebas de ejecución

Para lanzar el cliente se utiliza "./dir_server" y para ejecutar el cliente "./dir_client localhost"

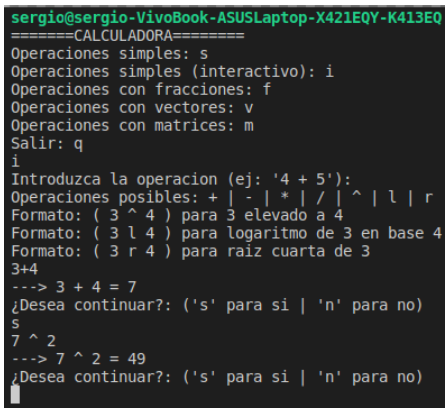

Figure 2: Salidas de pantalla del modo interactivo y del modo fracciones

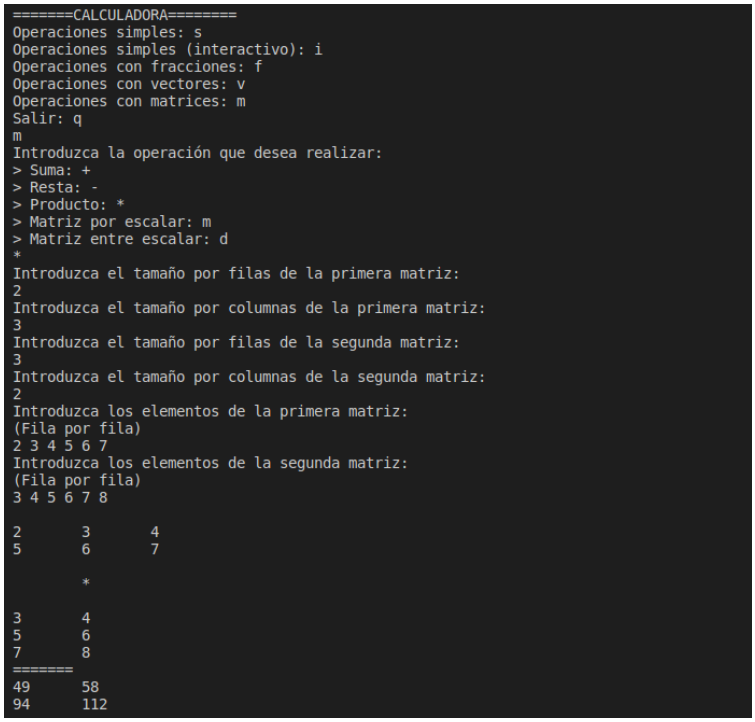

Figure 3: Salida de pantalla del modo matrices

3 Segunda parte - IDL

La filosofía de esta parte es muy semejante a lo anterior y las funciones implementadas son exactamente las mismas. Por ello, me abstendré de comentar esos detalles que ya se han explicado antes.

Para implementar la calculadora con thrift se ha creado el archivo *calculadora.thrift* donde se especificarán, al igual que anteriormente en el archivo dir.x, los métodos de las clases y las estructuras de datos necesarias. En este caso solo se ha necesitado una ya que no hay ningún tipo de límite en el número de parámetros y, por tanto, solo se ha creado una para las fracciones por comodidad. A través de este archivo trift, se generaron tres carpetas:

- **gen-java:** Archivos que contienen la calculadora en Java
- **gen-rb:** Archivos que contienen la calculadora en Ruby
- **gen-py:** Archivos que contienen la calculadora en Python

```
struct Fraccion {
   1: required i32 num,
   2: required i32 den
}

service Calculadora{
   void ping(),
   double suma(1:double num1, 2:double num2),
   double resta(1:double num1, 2:double num2),
   double multiplicacion(1:double num1, 2:double num2);
   double division(1:double num1, 2:double num2);
   double raices(1:double num1, 2:i32 num2);
   double potencia(1:double num1, 2:i32 num2);
   double logaritmo(1:double num1, 2:i32 num2);
   list<double> suma_vectores(1:list<double> vector1, 2:list<double> vector2);
   list<double> resta_vectores(1:list<double> vector1, 2:list<double> vector2);
   double producto_escalar(1:list<double> vector1, 2:list<double> vector2);
   list<double> multi_vector_escalar(1:list<double> vector1, 2:double escalar);
   list<double> division_vector_escalar(1:list<double> vector1, 2:double escalar);
   list<list<double>> suma_matrices(1:list<list<double>> matriz1, 2:list<list<double>> matriz2);
   list<list<double>> resta_matrices(1:list<list<double>> matriz1, 2:list<list<double>> matriz2);
   list<list<double>> producto_matrices(1:list<list<double>> matriz1, 2:list<list<double>> matriz2);
   list<list<double>> multi_matriz_escalar(1:list<list<double>> matriz1, 2:double escalar);
   list<list<double>> division_matriz_escalar(1:list<list<double>> matriz1, 2:double escalar);
   Fraccion suma_fracciones(1:Fraccion f1, 2:Fraccion f2);
   Fraccion resta_fracciones(1:Fraccion f1, 2:Fraccion f2);
   Fraccion multiplicacion_fracciones(1:Fraccion f1, 2:Fraccion f2);
   Fraccion division_fracciones(1:Fraccion f1, 2:Fraccion f2);
   Fraccion simplificacion_fracciones(1:Fraccion f1);
}
```

Figure 4: Declaración de los procedimientos y la estructura de datos Fraccion en el fichero calculadora.thrift

Además, al igual que en la primera parte, se ha realizado un reenvío de llamada al servicio de un servidor a otro. Es decir, tendremos un servidor intermedio que actuará de cliente de otro servidor para algunas operaciones (de nuevo, supondremos que la máquina del nuevo servidor tiene mejores características y nos podría venir bien para operaciones más complejas). Cada máquina ha sido implementada en un lenguaje distinto, que se detallarán a continuación.

3.1 Cliente - Python

Una vez generados los procedimientos, se creó el cliente de Python que implementará el programa para el menú interactivo del usuario. Además, debido a la mayor facilidad del lenguaje, se ha modularizado el programa con funciones para una mayor comprensión del código.

El cliente abre una conexión sobre el puerto 9090 para lanzar la petición al servidor que esté escuchando en dicho puerto y llamará al método que el usuario elija. (Las distintas operaciones ya se comentaron en la primera parte).

3.2 Servidor - Java

Este servidor fue el primero en crearse, implementándose todos los métodos en él originalmente (posteriormente se tuvieron que borrar los de las matrices para utilizarlos en ruby). La forma de implementarlo no difiere mucho de C, solamente se reduce la complejidad para trabajar con matrices y vectores. Para recibir las peticiones del cliente, el servidor escuchará en el puerto 9090.

Para conectarse con el nuevo servidor, abrirá otra conexión sobre un puerto distinto (9093). Así, se evita que el cliente pueda llamar al servidor de ruby y se garantiza que la petición debe pasar si o si por este servidor.

3.3 Servidor (2) - Ruby

Por último, como función extra he decidido crear otro servidor para realizar las operaciones con las matrices debido a la suposición mencionada anteriormente.

El servidor escuchará en el puerto 9093.

3.4 Pruebas de ejecución

Para lanzar este sistema primero se deben lanzar los servidores primero y por último el cliente de Python (por cuestiones de conexión entre ellos).

Se deberán introducir los datos uno a uno. (Por ejemplo: 3 Enter; + Enter; 5 Enter;)

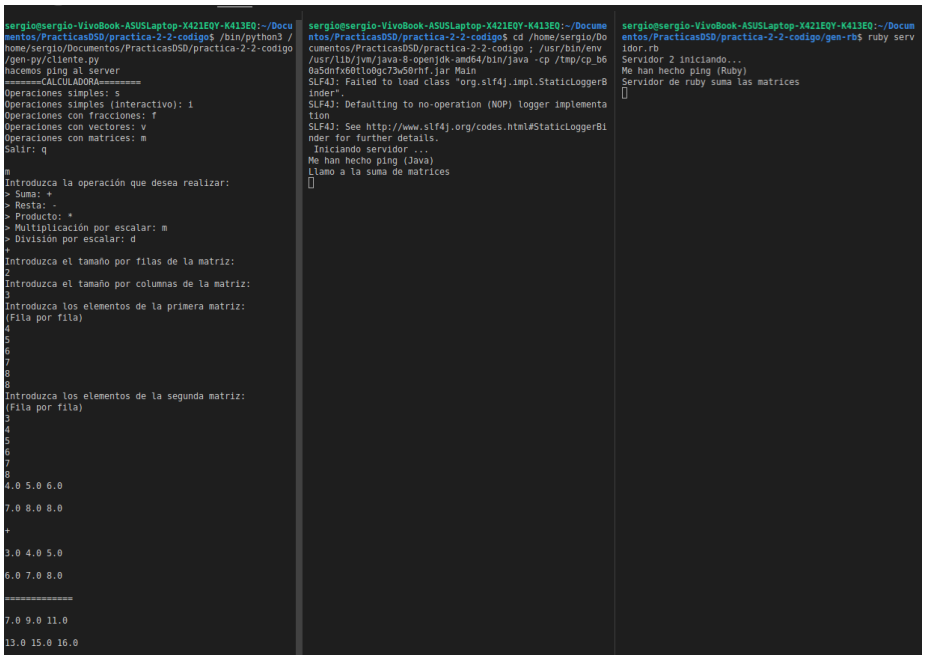

Figure 5: Ejecución de suma de matrices. Cliente a la izquierda, al lado el servidor de Java y por último el servidor de Ruby.

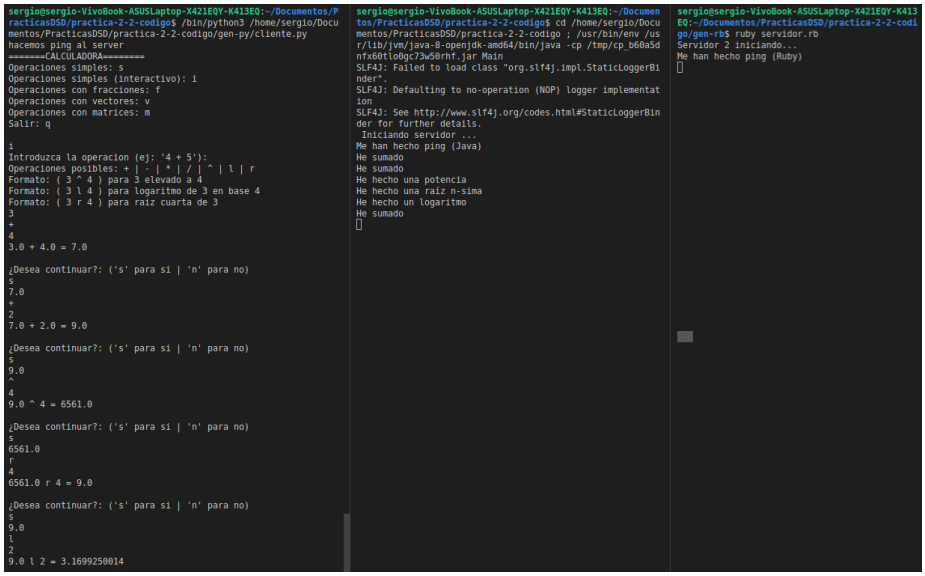

Figure 6: Ejecución del modo interactivo en operaciones simples. Cliente a la izquierda, al lado el servidor de Java y por último el servidor de Ruby.

4 Conclusiones

La práctica me ha resultado bastante interesante y considero que me ha ayudado a aprender sobre el funcionamiento de los sistemas distribuidos y la conexión entre sistemas con distintos lenguajes de programación. Además, me ha provocado mucha curiosidad sobre Python como nuevo lenguaje y su uso para este tipo de programas.

5 Comentarios

- Se ha tenido que modificar el primer Makefile de RPC para poder realizar la conexión entre los dos servidores.
- Se han incluido las librerías de Thrift para java para el correcto funcionamiento
- En python se deben introducir los datos línea por línea y sin ningúna introducción errónea.