Memoria P2

Salvador Romero Cortés

SUN RPC

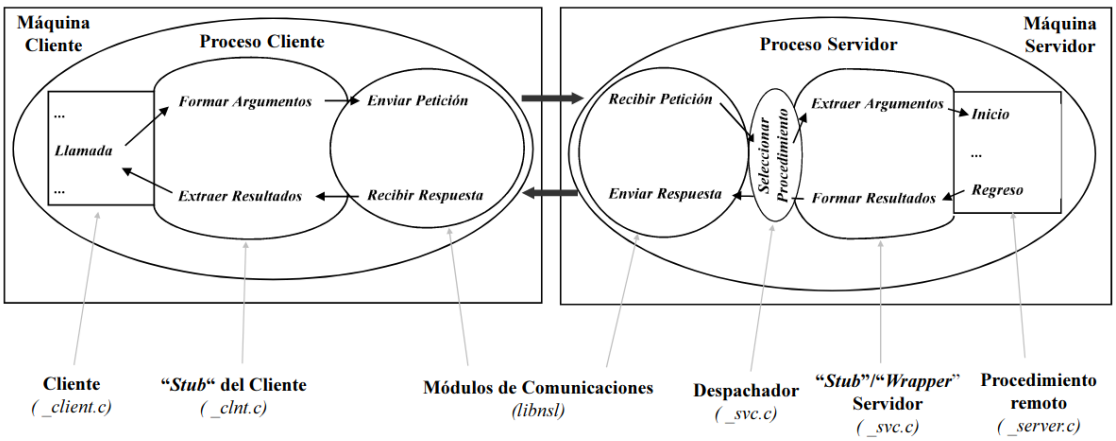
Para esta primera práctica usamos SUN RPC con el lenguaje de programación C. Primero definimos una serie de structs para almacenar los datos de los parámetros de entrada de cada procedimiento:

```
struct pareja_double{
    double num1;
    double num2;
};

struct pareja_int{
    int num1;
    int num2;
};

struct vec3{
    double x;
    double y;
    double z;
};

struct pareja_vec3{
    vec3 v1;
    vec3 v2;
};

struct vec2{
    double x;
    double y;
};

struct pareja_vec2{
    vec2 v1;
    vec2 v2;
};

struct vec2escalar{
    vec2 v;
    double escalar;
};

struct vec3escalar{
    vec3 v;
    double escalar;
};
```

De esta forma podemos ajustarnos al requisito de RPC de tener un único parámetro de entrada. Una vez creados los structs, definimos las operaciones de la calculadora:

```
program CALCULADORA {
    version CALCVER {
        double SUMAR(pareja_double) = 1;
        double RESTAR(pareja_double) = 2;
        double MULTIPLICAR(pareja_double) = 3;
        double DIVIDIR (pareja_double) = 4;
        int DIVISIONENTERA(pareja_int) = 5;
        int MODULO (pareja_int) = 6;
        vec3 SUMAVEC3(pareja_vec3) = 7;
        vec2 SUMAVEC2(pareja_vec2) = 8;
        vec3 RESTAVEC3(pareja_vec3) = 9;
        vec2 RESTAVEC2(pareja_vec2) = 10;
        vec2 PESCALAR2(vec2escalar) = 11;
        vec3 PESCALAR3(vec3escalar) = 12;
    } =1;
} = 0x20000155;
```

Ahora usamos la orden `rpcgen -NCa calculadora.x` para generar los ficheros necesarios para programar el sistema. Los ficheros que se generan incluyen los ficheros de cliente servidor así como sus stubs necesarios para funcionar. Un diagrama de como funciona es:



Una vez tenemos los ficheros generados ya consiste en implementar las funciones en el servidor de la calculadora y hacer llamadas a él desde el cliente.

Funcionamiento del programa

Para ejecutar el programa ejecutamos `rpcgen`, luego compilamos con `make -f Makefile.calculadora` y sustituimos el `calculadora_client.c` y el `calculadora_server.c` por los que se incluyen en la práctica (rpcgen los sustituye por unos en blanco, por eso es necesario copiarlos despues de hacer el rpcgen).

Una vez compilado ejecutamos primero el servidor con `./calculadora_server` y desde otra terminal lanzamos el `./calculadora_client`.

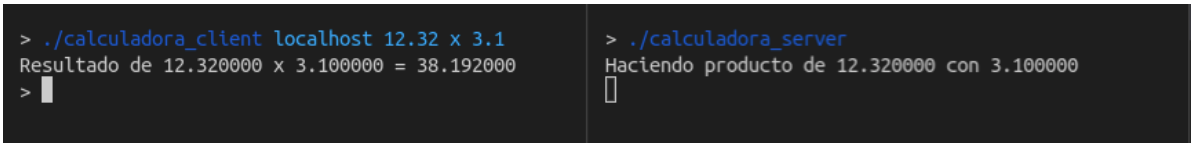
Para lanzar el cliente es necesario pasarle una serie de argumentos. Para ello es necesario saber primero si vamos a trabajar con operaciones más sencillas (suma, resta, producto...) o si vamos a trabajar con vectores. En caso de que sean operaciones sencillas pondremos

```
./calculadora_client localhost num1 operador num2
```

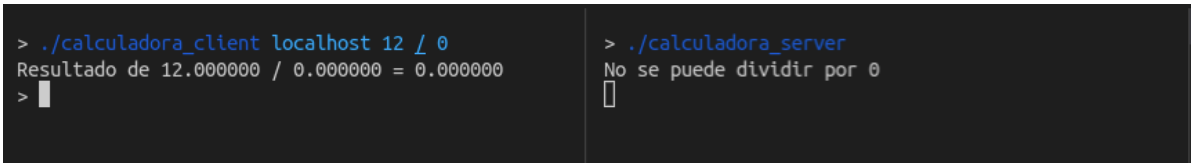
Donde localhost es el servidor donde esté el calculadora_server, num1 y num2 son los números que utilizamos en el cálculo y operador es un carácter que indica que operación se va a ejecutar. Los valores posibles del operador son:

```
+ para la suma
- para la resta
x para el producto
/ para la division
e para la division entera
% para el módulo
```

Una vez ejecutado se nos devolverá el resultado y el servidor indicará que operación está realizando. Por ejemplo:



Se incluyen también una serie de medidas para que no se pueda dividir por 0.



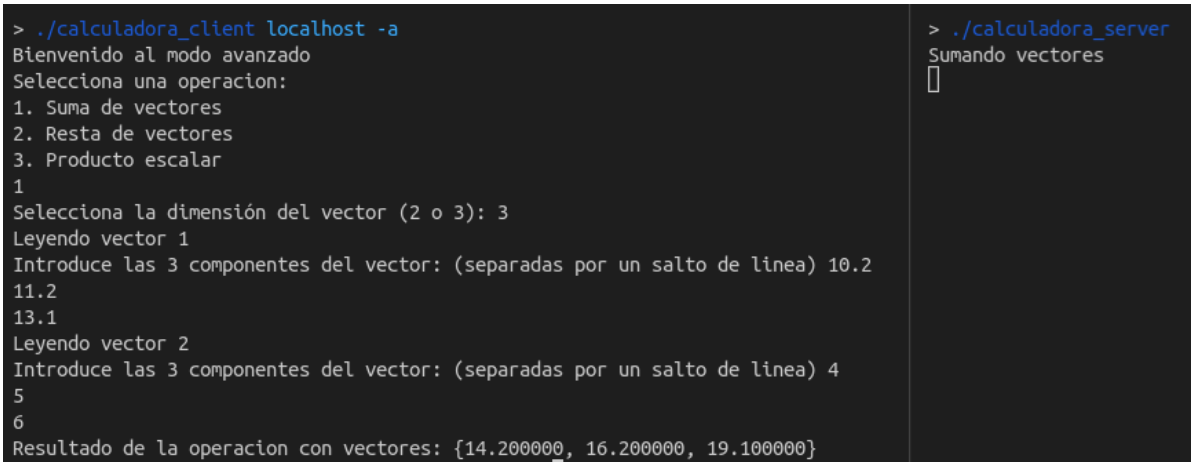
Para acceder al modo de operaciones con vectores lanzaremos el cliente con:

```
./calculadora_client localhost -a
```

De esta forma se nos presentará un menú interactivo donde escoger la operación con vectores que queramos usar. Están disponibles:

```
1. Suma de vectores
2. Resta de vectores
3. Producto escalar
```

Para cada operación con vectores se nos preguntará si queremos vectores de dimensión 2 o 3. Una vez elegido, introduciremos los valores de los vectores y nos devolverá el resultado.



Apache THRIFT

En esta segunda parte usaremos python con Apache Thrift para hacer un sistema similar. Usaremos la flexibilidad y comodidad de python para introducir más operaciones con más facilidad. Primero, definimos las operaciones

```
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),
   i32 divisionEntera(1:i32 num1, 2:i32 num2),
   double division(1:double num1, 2:double num2),
   i32 modulo(1:i32 num1, 2:i32 num2),
   list<double> sumaVectores(1:list<double> v1, 2:list<double> v2),
   list<double> restaVectores(1:list<double> v1, 2:list<double> v2),
   list<double> productoVectorial(1:list<double> v1, 2:list<double> v2),
   list<double> productoEscalar(1:list<double> v1, 2:double num2),
   double determinante(1:list<double> matriz)
}
```

Generamos los ficheros con

```
thrift -gen py calculadora.thrift
```

Incorporamos los ficheros cliente y servidor a la carpeta gen-py.

Funcionamiento

Primero lanzamos el servidor con

```
python3 gen-py/servidor.py
```

La sintaxis del cliente es la misma que la del programa RPC, solo que no se indica el servidor.

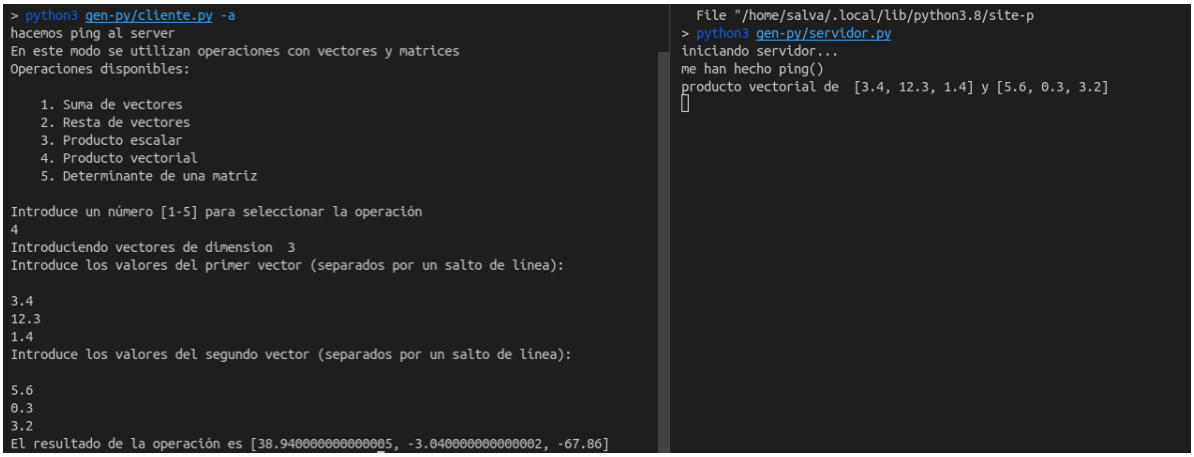
```
python3 gen-py/cliente.py num1 operador num2
```

O para el modo avanzado:

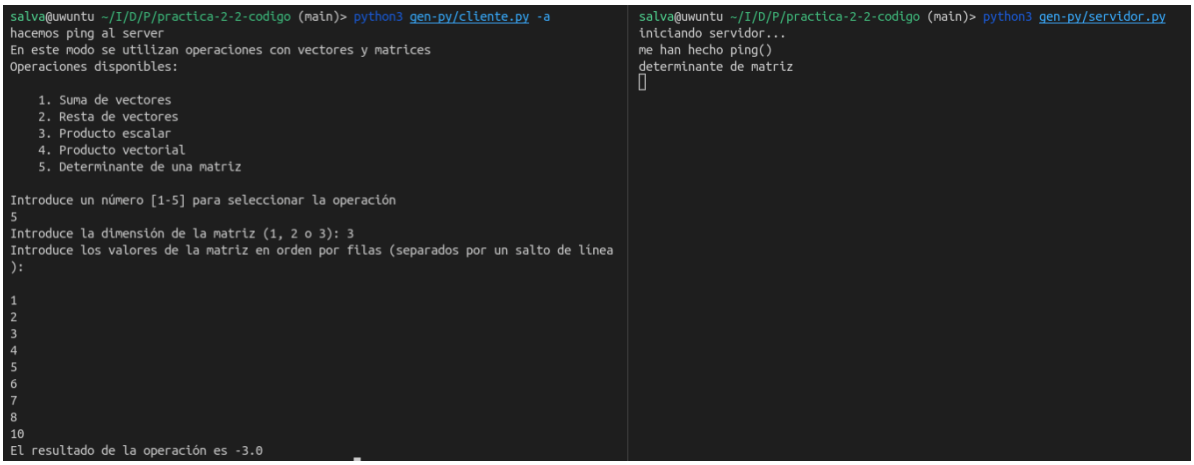
```
python3 gen-py/cliente.py -a
```

Ejemplo de modo básico:

Ejemplo del modo avanzado:

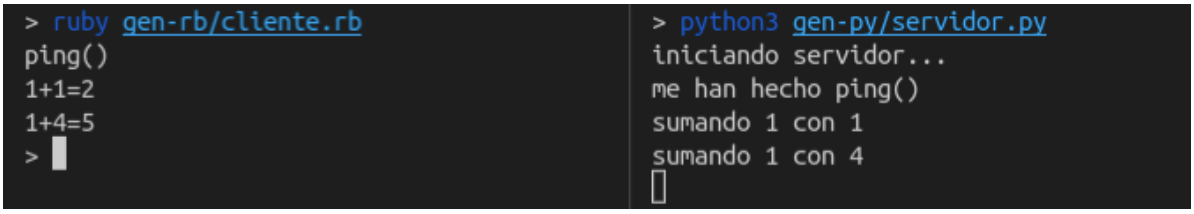


En el caso del determinante de la matriz se podrá elegir la dimensión de la matriz (hasta 3) y se calculará.



Probando con otros lenguajes

Es fácil tratar con varios lenguajes con Apache Thrift gracias a que con el mismo fichero .thrift podemos generar la estructura necesaria en varios lenguajes. Es interesante notar que incluso se pueden mezclar los lenguajes usados en el cliente y en el servidor. Por ejemplo, aquí tenemos un cliente escrito en ruby y un servidor escrito en python



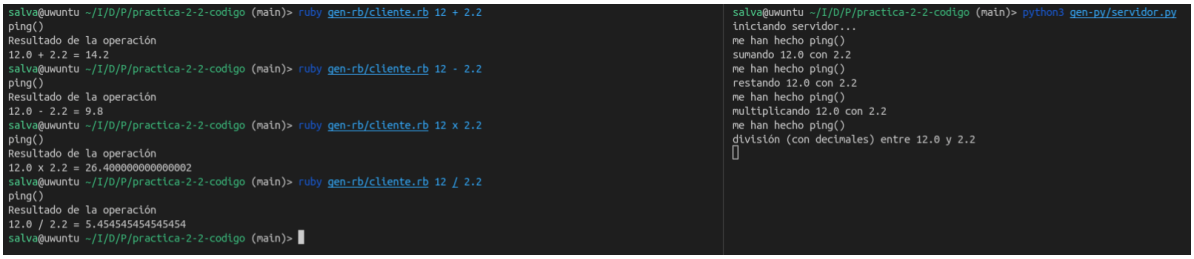
He hecho un cliente básico (suma, resta, producto, división) en ruby para usar con el servidor en python. Tiene la misma sintaxis que el cliente de python. Primero generamos con apache thrift los archivos necesarios para ruby con

```
thrift -gen rb calculadora.thrift
```

Luego, incorporamos el cliente.rb en la carpeta gen-rb.

Ejecutamos el servidor en python como antes y el cliente en ruby como:

```
ruby gen-rb/cliente.rb <num1> <operacion> <num2>
```


salva@uwuntu ~/I/D/P/practica-2-2-codigo (main)> ruby gen-rb/cliente.rb 12 + 2.2
ping()
Resultado de la operación
12.0 + 2.2 = 14.2
salva@uwuntu ~/I/D/P/practica-2-2-codigo (main)> ruby gen-rb/cliente.rb 12 - 2.2
ping()
Resultado de la operación
12.0 - 2.2 = 9.8
salva@uwuntu ~/I/D/P/practica-2-2-codigo (main)> ruby gen-rb/cliente.rb 12 x 2.2
ping()
Resultado de la operación
12.0 x 2.2 = 26.400000000000002
salva@uwuntu ~/I/D/P/practica-2-2-codigo (main)> ruby gen-rb/cliente.rb 12 / 2.2
ping()
Resultado de la operación
12.0 / 2.2 = 5.454545454545454
salva@uwuntu ~/I/D/P/practica-2-2-codigo (main)>
salva@uwuntu ~/I/D/P/practica-2-2-codigo (main)> python3 gen-py/servidor.py
iniciando servidor...
me han hecho ping()
sumando 12.0 con 2.2
me han hecho ping()
restando 12.0 con 2.2
me han hecho ping()
multiplicando 12.0 con 2.2
me han hecho ping()
división (con decimales) entre 12.0 y 2.2