Práctica 1, Parte 2. Haciendo una calculadora con Apache Thrift

0. Introducción

Para esta práctica se trata de hacer una calculadora donde a través de un cliente que envía una petición a un servidor para realizar varios cálculos y devolver el resultado al cliente, quien lo imprime en pantalla.
En principio se trataba de hacer las operaciones básicas (sumas, restas, productos y cocientes), al cual poco a poco se ha ido añadiendo más complejidad haciendo operaciones acumulativas, con structs, con vectores y con matrices.

Además, como variación con respecto a la parte 1 de esta práctica, se ha tratado de implementar el cliente en un lenguaje (Python en este caso) y el servidor en otro (se intentó con Dart primero, pero por cuestiones que detallaré más tarde decidí hacerlo en Ruby).

1. Una calculadora básica.

Lo primero que se hizo fue tratar de hacer que la calculadora pueda sumar, restar, multiplicar y dividir. Para ello se necesitan dos números y la operación a realizar, junto con el host del servidor, ya que sin el servidor el cliente no funcionaría adecuadamente (cogería las operaciones pero no haría nada más).

Por ello, en el archivo de definiciones se establecieron 4 funciones:

```
    void ping(),
   i32 suma(1:i32 num1, 2:i32 num2),
   i32 resta(1:i32 num1, 2:i32 num2),
   i32 producto(1:i32 num1, 2:i32 num2),
   double cociente(1:i32 num1, 2:i32 num2),
```

Al generar las plantillas, en el servidor se crean las funciones arriba mencionadas, con un atributo de salida return que recomiendan rellenar.

Para las operaciones básicas sólo había que relacionar ambos operandos con la operación de la función. Así, en el caso de la suma tendríamos :

Ruby

```ruby
def suma(n1,n2)
 puts "Sumando"
 return n1+n2
end
```

Python

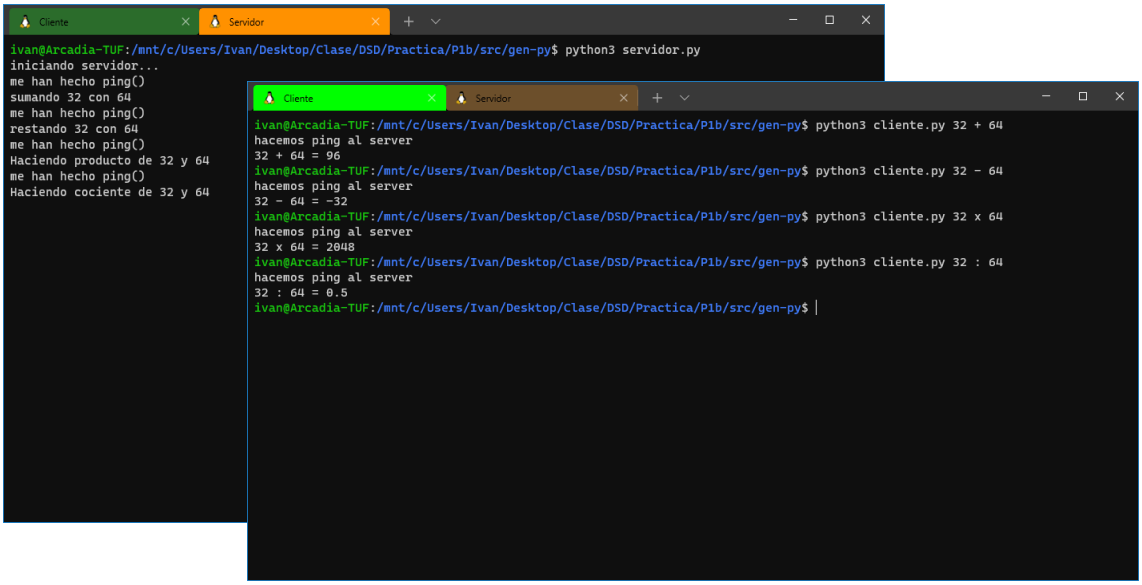
```python
def suma(self, n1, n2):
  print("sumando ")
  return n1 + n2
```

Dart

```dart
@override
Future<int> suma(int num1, int
num2) async {
 print("Suma");
 return num1 + num2;
}
```

En el cliente se usa la función switch_basico el cual observa la operación a realizar de entre las cuatro posibles (+,-,x,:) y según ese operando llama a una u otra función pasando como parámetros los operandos y el operador.

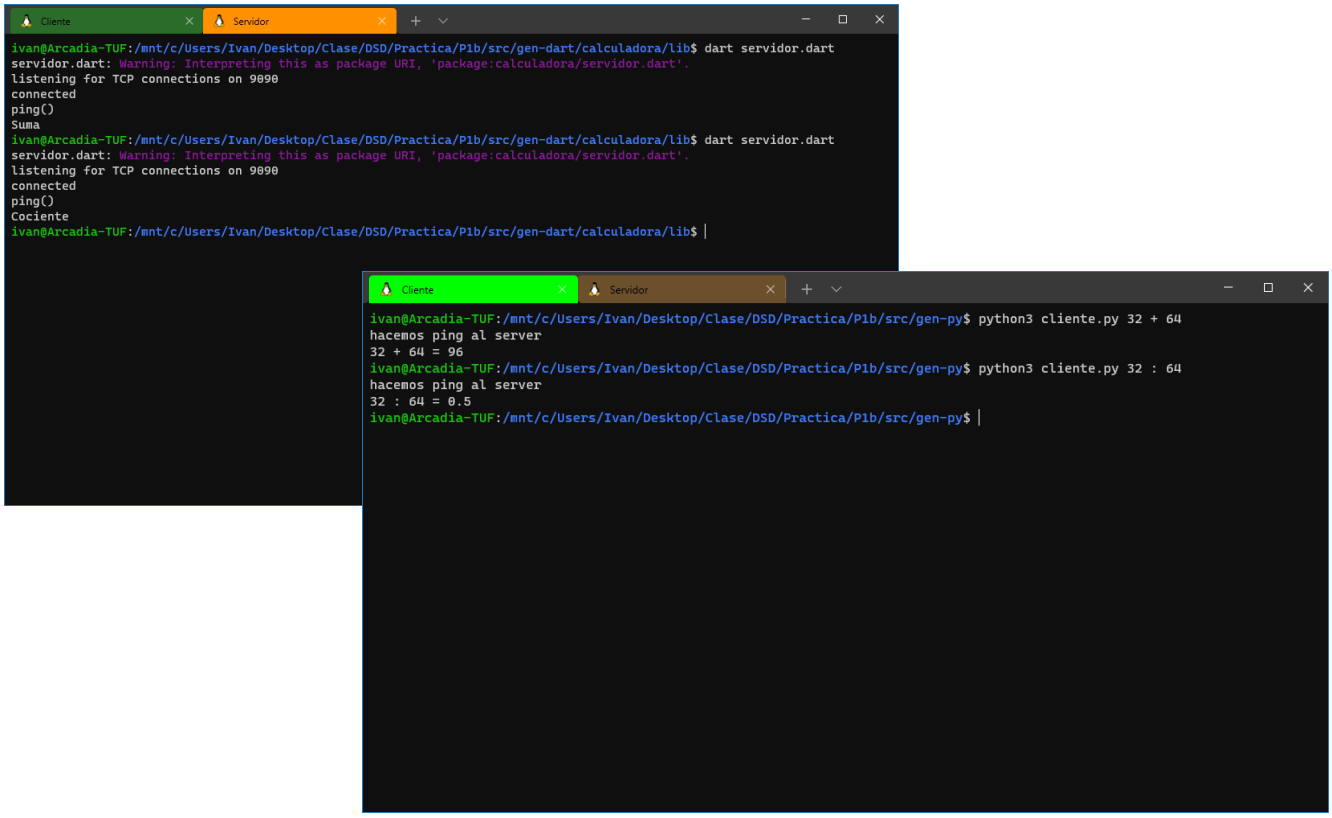
```
def switch_basico(operacion, oper1, oper2):
  if(operacion == '+'):
    resultado = client.suma(oper1,oper2)
  elif(operacion == '-'):
    resultado = client.resta(oper1,oper2)
  elif(operacion == 'x'):
    resultado = client.producto(oper1,oper2)
  elif(operacion == ':'):
    resultado = client.cociente(oper1,oper2)
  else:
    resultado = None
  return resultado
```



1.1 Intentando hacer el servidor en otro lenguaje

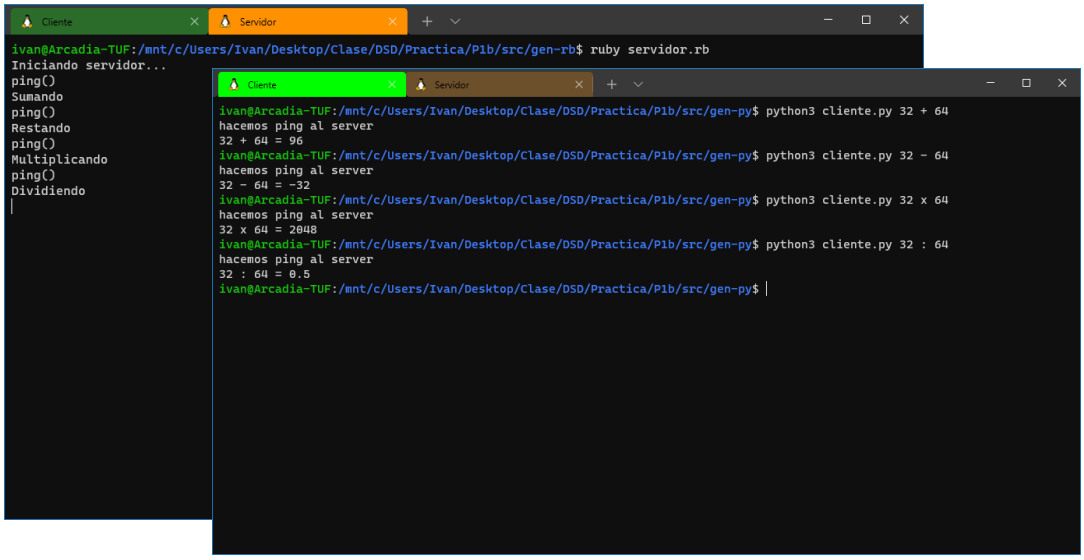
Llegados a este punto, se barajó la posibilidad de hacer el programa del servidor en otro lenguaje. Por tanto, nada más ver que funcionaban las operaciones básicas en un servidor escrito con Python, me aventuré a intentar hacerlo con Dart.

Para ello, me basé en el ejemplo que ofrecía Apache Thrift sobre ese lenguaje, pero al adaptarlo para que se ajustara a la práctica, observé que sólo podía hacer una operación antes de salirse al Terminal o Bash. En un principio pensé en que simplemente se hiciera un bucle infinito ejecutando el programa, de forma de que si acabara no hubiera que ejecutarlo manualmente, pero pensé : *¿Y si mientras se rearma el servidor llega una petición? No lo podría atender y daría error en el cliente.*



También fue interesante fijarse en que, si bien lenguajes como Python, Ruby o Java tenían una clase denominada **`TSimpleServer`**, Dart no lo tenía. Así pues, descarté avanzar con el desarrollo del servidor en Dart (aunque se quedará adjunto en la entrega) y hacerlo en Ruby.

Dado que en Ruby si permitía seguir la ejecución del servidor tras servir una petición del cliente, seguí con él.



2. Introduciendo una función con *structs* : Resolver ecuaciones cuadráticas

En una ecuación cuadrática hay dos soluciones, cosa que no se podría hacer con una función ya que no puede devolver dos enteros de por sí. Pero... ¿y si los metemos en un *struct* junto con los valores que se pasan, para comprobar que lo que se recibe es el resultado de lo que se envía?

Así, lo primero que hay que hacer es declarar el struct en **`calculadora.thrift`** y una función que devuelva uno de ellos:

```
struct funcion_cuadratica{
   1: required i32 a
   2: required i32 b
   3: required i32 c
   4: required double x1
   5: required double x2
}
/* ... */

funcion_cuadratica ecuacionGradoDos(1: i32 a, 2: i32 b, 3: i32 c),
```

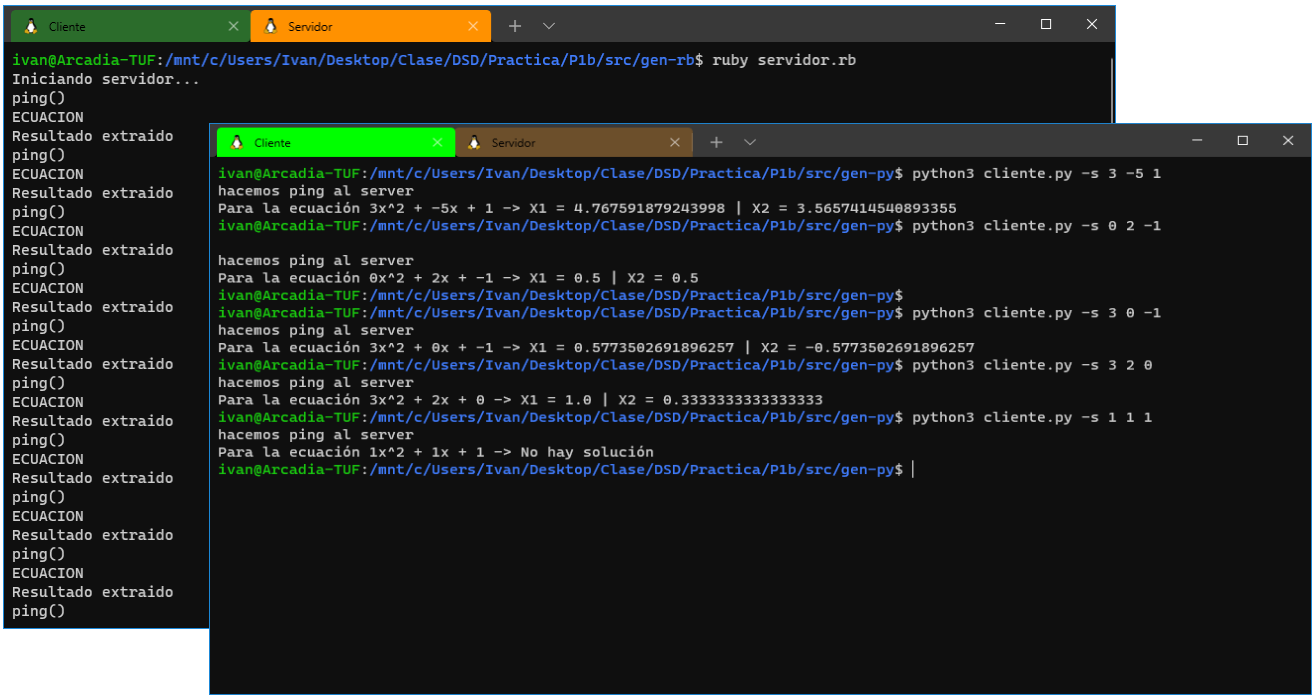
Luego, en el servidor se resuelve usando la fórmula $x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$, que en el caso de que a = 0, $x=\frac{-c}{b}$.

```
def ecuacionGradoDos(a,b,c)
  puts "ECUACION"
  result = Funcion_cuadratica.new
  result.a = a
  result.b = b
  result.c = c

  if (result.a == 0)
   result.x1 = (-result.c * 1.0)/(result.b * 1.0)
   result.x2 = (-result.c * 1.0)/(result.b * 1.0)
  else
   discriminante = (result.b*result.b) - 4*result.a*result.c
   if (discriminante < 0)
    result.x1 = -12345678
    result.x2 = -12345678
   else
    raiz = Math.sqrt(discriminante)
```

```
    result.x1 = (((result.b * result.b) + raiz)*1.0)/(2.0*result.a)
    result.x2 = (((result.b * result.b) - raiz)*1.0)/(2.0*result.a)
   end
  end
  puts "Resultado extraido"
  return result
end
```

Y en el cliente se usa el modificador "-s", cambiando así el input de los argumentos al formato **`python3 cliente.py -s <Ax^2> <Bx> <C>.`**



3. Vectores : Operaciones acumuladas

En el caso de los vectores, habría que definirlo en calculadora.thrift usando list**<TIPO>**. Así pues, definimos un array de enteros y algunas operaciones para las que usar un solo vector.

```
double sumaAcumulada(1: list<i32> acumulado),
double restaAcumulada(1: list<i32> acumulado),
double productoAcumulado(1: list<i32> acumulado),
```

Así pues, **`secuencia`** se encuentra como un *struct* con un tamaño y un puntero al *array*.

En el servidor, se coge el *array*, sacamos la primera variable como referencia y se va sumando los demás, devolviendo la operación acumulada en **`result`**.

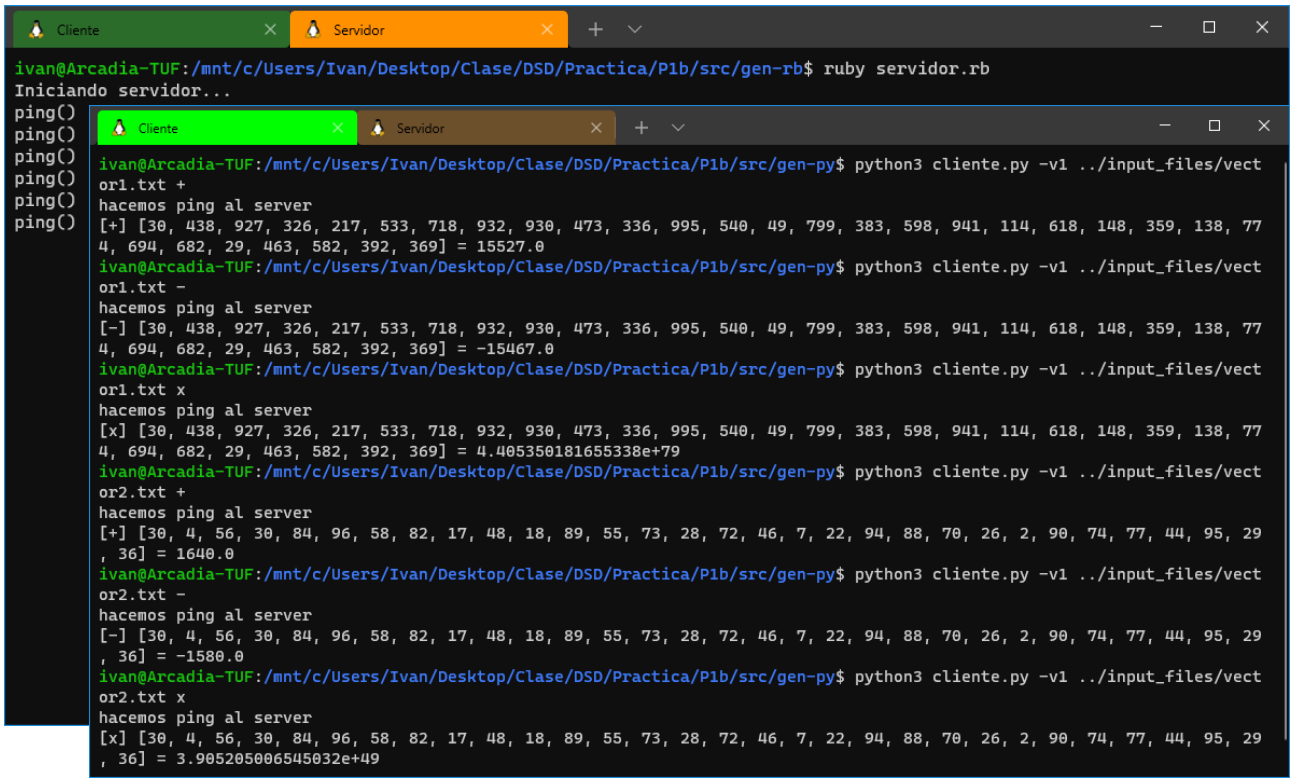
```
  def sumaAcumulada(arrayAcumulativo)
  acumulado = 0
  for item in arrayAcumulativo do
   acumulado = acumulado + item
  end
  return acumulado
 end
```

En el cliente, además de poner otro modificador para aceptar leer los archivos, se han tenido que hacer funciones para leer desde archivo los valores del *array.*

El formato sería **`python3 <Programa> -v1 <Archivo_vector> <Operacion>`**

```
with open(argv[2]) as f:
      vector = [int(x) for x in f.read().split()]
```

```
def switch_acumulativo(operacion, acumulativo):
  if(operacion == '+'):
    resultado = client.sumaAcumulada(acumulativo)
  elif(operacion == '-'):
    resultado = client.restaAcumulada(acumulativo)
  elif(operacion == 'x'):
    resultado = client.productoAcumulado(acumulativo)
  else:
    resultado = None
  return resultado
```



4. Vectores : Operaciones entre vectores

Siguiendo con los vectores, ahora trataremos de hacer operaciones con dos vectores y devolverla en otro.

En el calculadora.thrift:

```
    secuencia sumaVectores(secuencia,secuencia) = 9;
    secuencia restaVectores(secuencia,secuencia) = 10;
    secuencia productoVectores(secuencia,secuencia) = 11;
    secuencia cocienteVectores(secuencia,secuencia) = 12;
```

En el servidor, cuando recibimos ambos arrays, reservamos memoria para otro array en base a la longitud de los dos argumentos.

```
  def sumaVectores(v1, v2)
  total = Array.new(v1.length)
  for i in 0..v1.length-1 do
   total[i] = v1[i] + v2[i]
  end
  return total
end
```

En el cliente hacemos las comprobaciones de longitud, para ver que una no sea mayor o menor que la otra. En caso contrario, avisará y terminará la ejecución. Su ejecución sería **`python3 cliente.py -v2 <archivo1> <operación> <archivo2>`**

Para usar la función , la estructura sería

```
/* ... */
elif (argc == 5 and argv[1] == '-v2'):
    operacion = argv[3]
    ficheroAcumulado1 = argv[2]
    ficheroAcumulado2 = argv[4]

    with open(argv[2]) as f:
      vector1 = [int(x) for x in f.read().split()]

    with open(argv[4]) as f2:
      vector2 = [int(x) for x in f2.read().split()]

    resultado = switch_vectorial(operacion,vector1,vector2)

    if (resultado != None):
      print(f"{vector1} {operacion} {vector2} = {resultado}")
    else:
      uso()

/* ... */
```

5. Matrices : Operaciones entre matrices

En el caso de las matrices, habría que definirlo en calculadora.thrift usando **`list<list<TIPO>>`**.

Así pues, definimos una matriz de double (que sería un *array* de *arrays* de **`double`**) y algunas operaciones para las que usar dos matrices.

```
   list<double> restaVectores(1: list<i32> vector1, 2:list<i32>
vector2),
   list<double> productoVectores(1: list<i32> vector1, 2:list<i32>
vector2),
   list<double> cocienteVectores(1: list<i32> vector1, 2:list<i32>
vector2),
```

En el servidor, como hay que generar una nueva matriz, se debe reservar la memoria. Por ejemplo, al sumar matrices:

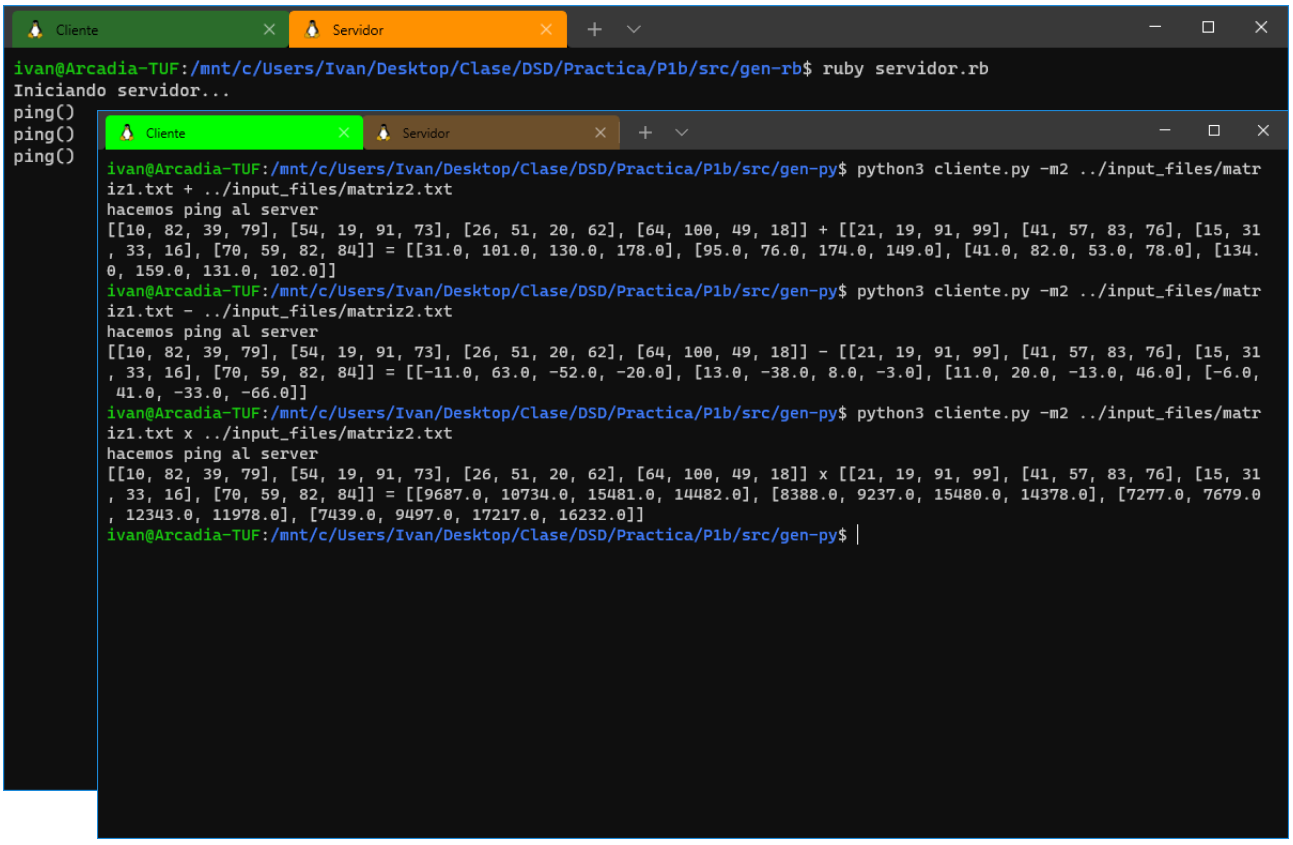
```
  def sumaMatrices(m1,m2)
 total = Array.new(m1.length){Array.new(m2.length)}
 for i in 0..m1.length-1 do
  for j in 0..m2.length-1 do
   total[i][j] = m1[i][j] + m2[i][j]
```

```
    end
  end

  return total
end
```

En el cliente, hay que leer la matriz desde archivo. En Python se ejecuta con la orden **`python3 cliente.py -m2 <archivo1> <operación> <archivo2>`**:

```
ficheroAcumulado1 = argv[2]
#...
matriz1 = []
#...
with open(argv[2]) as f:
      for linea in f:
        linea_anadir = [int(x) for x in linea.split()]
        matriz1.append(linea_anadir)
```



7. Conclusiones

La llamada a procedimiento remoto podría permitir realizar tareas complejas que podrían requerir procesadores potentes usando un PC cualquiera como cliente. Además, a diferencia de Sun RPC, permite usar sus funciones y stubs en varios lenguajes, permitiendo así una comodidad e integración con otras tecnologías, como (Node)JS o Ruby (on Rails). Si junto con ello permitimos una distribución de la potencia de cómputo, acabaríamos con una buena versatilidad