Práctica 1, Parte 1. Haciendo una calculadora con Sun RPC

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.

Para las pruebas, se ha reorganizado el Makefile para compilarlo todo en distintas carpetas y se ha creado un *script* de *Shell* para compilar y ejecutar todo.

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:

```
    double suma(double,double) = 1;
    double resta(double,double) = 2;
    double producto(double,double) = 3;
    double cociente(double,double) = 4;
```

¿Y por qué uso double en vez de int? Más que nada por el caso de la división, donde importan los decimales.

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 :

```
double *
suma_1_svc(double arg1, double arg2,  struct svc_req *rqstp)
{
     static double  result;

     result = arg1 + arg2 ;
```

```
    return &result;
}
```

En el cliente se usa la función rpc_calc_1, 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.

Al hacer pruebas, obtenemos estos resultados:

```
./tests.sh: 5-13

./bin/server &
./bin/cliente localhost 2 + 2
./bin/cliente localhost 2 - 2
./bin/cliente localhost 2 x 2
./bin/cliente localhost 2 : 2
./bin/cliente localhost 34 + 67
./bin/cliente localhost 34 - 67
./bin/cliente localhost 34 x 67
./bin/cliente localhost 34 : 67
```

```
Operación a localhost -> 2.000000 + 2.000000

Sumamos
localhost -> 2.000000 + 2.000000 = 4.000000
Operación a localhost -> 2.000000 - 2.000000

Restamos
localhost -> 2.000000 - 2.000000 = 0.000000
Operación a localhost -> 2.000000 x 2.000000

Multiplicamos
localhost -> 2.000000 x 2.000000 = 4.000000
Operación a localhost -> 2.000000 : 2.000000

Dividimos
localhost -> 2.000000 : 2.000000 = 1.000000
Operación a localhost -> 34.000000 +
67.000000
Sumamos
localhost -> 34.000000 + 67.000000 =
101.000000
Operación a localhost -> 34.000000 -
67.000000
Restamos
localhost -> 34.000000 - 67.000000 = -
33.000000
Operación a localhost -> 34.000000 x
67.000000
Multiplicamos
localhost -> 34.000000 x 67.000000 =
2278.000000
Operación a localhost -> 34.000000 :
67.000000
Dividimos
localhost -> 34.000000 : 67.000000 =
0.507463
```

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.x** y una función que devuelva uno de ellos:

```
struct funcion_cuadratica{
    int a;
    int b;
    int c;
    double valor_1;
    double valor_2;
};

/* ... */

funcion_cuadratica ecuacionGradoDos(int,int,int) = 5;
```

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, $x=\frac{-c}{b}$.

```
funcion_cuadratica *
ecuaciongradodos_1_svc(int arg1, int arg2, int arg3,  struct svc_req *rqstp)
{
        static funcion_cuadratica  result;
        double x1,x2;
        result.a = arg1;
        result.b = arg2;
        result.c = arg3;

        if(result.a != 0){
            double discriminante = (result.b * result.b) - (4 * result.a *
result.c) ;

            if (discriminante < 0){
                result.valor_1 = result.valor_2 = INT_MIN;
            }
            else{
                double raiz = sqrt(discriminante) ;
                printf("Raiz = %f", raiz);
                x1 = (-result.b + raiz) / (2*result.a) ;
                x2 = (-result.b - raiz)/ (2*result.a) ;
                result.valor_1 = x1;
                result.valor_2 = x2;
            }
        }
```

```
        else{
            result.valor_1 = result.valor_2 = (-
result.c*1.0)/(result.b*1.0);
        }


        return &result;
}
```

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

Al hacer los tests:

```
./tests.sh : 15-19
./bin/cliente localhost -s 3 -5 1
./bin/cliente localhost -s 0 2 -1
./bin/cliente localhost -s 3 0 -1
./bin/cliente localhost -s 3 2 0
./bin/cliente localhost -s 1 1 1
```

```
Ecuación de segundo grado:

localhost : Para la ecuación 3x^2 + -5x
+ 1 = 0 -> {X1 = 1.434259 , X2 =
0.232408}
Ecuación de segundo grado:

localhost : Para la ecuación 0x^2 + 2x +
-1 = 0 -> {X1 = 0.500000 , X2 =
0.500000}
Ecuación de segundo grado:

localhost : Para la ecuación 3x^2 + 0x +
-1 = 0 -> {X1 = 0.577350 , X2 = -
0.577350}
Ecuación de segundo grado:

localhost : Para la ecuación 3x^2 + 2x +
0 = 0 -> {X1 = 0.000000 , X2 = -
0.666667}
Ecuación de segundo grado:

localhost : Para la ecuación 1x^2 + 1x +
1 = 0 -> No hay soluciones reales
```

3. Vectores : Operaciones acumuladas

En el caso de los vectores, habría que definirlo en calculadora.x usando **typedef [tipo] [nombre]<>**.
Así pues, definimos un array de double y algunas operaciones para las que usar un solo vector.

```
typedef double secuencia<>;

double sumaAcumuladaVector(secuencia) = 6;
double restaAcumuladaVector(secuencia) = 7;
double productoAcumuladaVector(secuencia) = 8;
```

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**.

```
double *
sumaacumuladavector_1_svc(secuencia arg1,  struct svc_req *rqstp)
{
     static double  result;
     result = arg1.secuencia_val[0] * 1.0;

     for (int i = 1 ; i < arg1.secuencia_len ; i++){
          result += (arg1.secuencia_val[i] * 1.0);
     }


     return &result;
}
```

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*, y para liberar la memoria dinámica.

El formato sería **<Programa> <Host_server> -v1 <Archivo_vector> <Operacion>**

```
secuencia read_ints (const char* file_name)
{
  FILE* file = fopen (file_name, "r");
  secuencia vector_resultado;
  int i = 0;
  int iterador = 0;

  fscanf (file, "%d", &i);
  vector_resultado.secuencia_len = i;
  printf ("Tamaño -> %d :", vector_resultado.secuencia_len);
  vector_resultado.secuencia_val = malloc(vector_resultado.secuencia_len *
sizeof(double));
  while (!feof (file))
```

```
    {
      fscanf (file, "%d", &i);
      vector_resultado.secuencia_val[iterador] = i ;
      iterador++;
    }
  fclose (file);

  return vector_resultado;
}
```

```
void liberarVector (secuencia aBorrar){
    aBorrar.secuencia_len = 0;
    free(aBorrar.secuencia_val);
}
```

Al hacer los tests:

./tests.sh, 21-23 ./bin/cliente localhost -v1 ./input_files/vector1.txt + ./bin/cliente localhost -v1 ./input_files/vector1.txt - ./bin/cliente localhost -v1 ./input_files/vector1.txt x	Tamaño -> 30 :Sumamos localhost -> Operación acumulada [+] = 15497.000000 Tamaño -> 30 :Restamos localhost -> Operación acumulada [-] = - 14621.000000 Tamaño -> 30 :Multiplicamos localhost -> Operación acumulada [x] = 14684500605517800286770268064890366568429904849789967052015908443694430955765 76.000000
./tests.sh, 25-27 ./bin/cliente localhost -v1 ./input_files/vector2.txt + ./bin/cliente localhost -v1 ./input_files/vector2.txt - ./bin/cliente localhost -v1 ./input_files/vector2.txt x	Tamaño -> 30 :Sumamos localhost -> Operación acumulada [+] = 1610.000000 Tamaño -> 30 :Restamos localhost -> Operación acumulada [-] = - 1602.000000 Tamaño -> 30 :Multiplicamos localhost -> Operación acumulada [x] = 13017350021816772134248268981043777467463646576 64.000000

4. Vectores : Operaciones entre vectores

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

En el calculadora.x:

```
     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.

```
secuencia *
sumavectores_1_svc(secuencia arg1, secuencia arg2,  struct svc_req *rqstp)
{
          static secuencia  result;

          result.secuencia_len = arg1.secuencia_len;
          result.secuencia_val = malloc(result.secuencia_len *
sizeof(double));

          for (int i = 0 ; i < result.secuencia_len ; i++){
               result.secuencia_val[i] = arg1.secuencia_val[i] +
arg2.secuencia_val[i] ;
          }

          return &result;
}
```

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.

Para usar la función , la estructura sería

```
/* ... */
else if (argc == 6 && strcmp(argv[2],"-v2") == 0){
          host = argv[1];
          char *filename1 = argv[3] ;
          char *filename2 = argv[5] ;
          operacion = argv[4];
          vector1 = read_ints(filename1);
          vector2 = read_ints(filename2);
          vector_solucion =
rpc_calc_vectorial(host,vector1,operacion,vector2);
          printf ("\n%s -> Operación vectorial [%s] =\n", host, operacion);
          printf("[");
          for (int i = 0 ; i < vector_solucion.secuencia_len ; i++){
               printf(" %f ", vector_solucion.secuencia_val[i]);
          }
          printf("]\n\n");
```

```
        liberarVector(vector1);
        liberarVector(vector2);
        liberarVector(vector_solucion);
    }
/* ... */
```

Al hacer los tests:

```
./tests.sh, 29-32
./bin/cliente localhost -v2
./input_files/vector1.txt +
./input_files/vector2.txt
./bin/cliente localhost -v2
./input_files/vector1.txt -
./input_files/vector2.txt
./bin/cliente localhost -v2
./input_files/vector1.txt x
./input_files/vector2.txt
./bin/cliente localhost -v2
./input_files/vector1.txt :
./input_files/vector2.txt
```

```
localhost -> Operación vectorial [+] =
[ 442.000000  983.000000  356.000000
301.000000  629.000000  776.000000
1014.000000  947.000000  521.000000
354.000000  1084.000000  595.000000
122.000000  827.000000  455.000000
644.000000  948.000000  136.000000
712.000000  236.000000  429.000000
164.000000  776.000000  784.000000
756.000000  106.000000  507.000000
677.000000  421.000000  405.000000 ]

Tamaño -> 30 :Tamaño -> 30 :Restamos
localhost -> Operación vectorial [-] =
[ 434.000000  871.000000  296.000000
133.000000  437.000000  660.000000
850.000000  913.000000  425.000000
318.000000  906.000000  485.000000  -
24.000000  771.000000  311.000000
552.000000  934.000000  92.000000
524.000000  60.000000  289.000000
112.000000  772.000000  604.000000
608.000000  -48.000000  419.000000
487.000000  363.000000  333.000000 ]

Tamaño -> 30 :Tamaño ->
30 :Multiplicamos
localhost -> Operación vectorial [x] =
[ 1752.000000  51912.000000  9780.000000
18228.000000  51168.000000  41644.000000
76424.000000  15810.000000  22704.000000
6048.000000  88555.000000  29700.000000
3577.000000  22372.000000  27576.000000
27508.000000  6587.000000  2508.000000
58092.000000  13024.000000  25130.000000
3588.000000  1548.000000  62460.000000
50468.000000  2233.000000  20372.000000
55290.000000  11368.000000  13284.000000
]

Tamaño -> 30 :Tamaño -> 30 :Dividimos
localhost -> Operación vectorial [:] =
[ 109.500000  16.553571  10.866667
2.583333  5.552083  12.379310  11.365854
54.705882  9.854167  18.666667
11.179775  9.818182  0.671233  28.535714
```

```
5.319444  13.000000  134.428571
5.181818  6.574468  1.681818  5.128571
5.307692  387.000000  7.711111  9.216216
0.376623  10.522727  6.126316  13.517241
10.250000 ]
```

5. Matrices : Operaciones entre matrices

En el caso de las matrices, habría que definirlo en calculadora.x usando **`typedef [NOMBRE_ARRAY] [nombre_matriz]<>`**, habiendo definido previamente el array con el **`typedef`** correspondiente.

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

```
typedef double secuencia<>;
typedef secuencia matriz<>;

matriz sumaMatrices(matriz,matriz) = 13;
 matriz restaMatrices(matriz,matriz) = 14;
matriz productoMatrices(matriz,matriz) = 15;
```

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

```
matriz *
sumamatrices_1_svc(matriz arg1, matriz arg2,  struct svc_req *rqstp)
{
          static matriz  result;
          int filas, columnas ;

          filas = arg1.matriz_len;
          columnas = arg1.matriz_val[0].secuencia_len ;

          result.matriz_len = filas;
          result.matriz_val = malloc(filas * sizeof(secuencia)) ;

          for (size_t i = 0; i < filas; i++)
          {

               secuencia fila;
               fila.secuencia_len = columnas ;
               fila.secuencia_val = malloc(columnas * sizeof(double));
               result.matriz_val[i] = fila ;
```

```
            for (size_t j = 0; j < columnas; j++)
            {
                fila.secuencia_val[j] = arg1.matriz_val[i].secuencia_val[j]
+ arg2.matriz_val[i].secuencia_val[j] ;
            }


        }


        return &result;
}
```

En el cliente, hay que leer la matriz desde archivo, desde donde también se reserva la memoria dinámica, y luego se libera:

```
matriz read_matrix (const char* file_name)
{
  FILE* file = fopen (file_name, "r");
  matriz matriz_resultado;
  int filas, columnas ;
  int i = 0;
  int iterador = 0;

  fscanf (file, "%d", &i);
  filas = i ;
  fscanf (file, "%d", &i);
  columnas = i;
  printf ("Tamaño de matriz-> %d x %d :\n", filas, columnas);

  matriz_resultado.matriz_len = filas;
  matriz_resultado.matriz_val = malloc(columnas * sizeof(secuencia)) ;

  for (int sec = 0 ; sec < matriz_resultado.matriz_len ; sec++){
       secuencia fila;
       fila.secuencia_len = columnas ;
       fila.secuencia_val = malloc(columnas * sizeof(double));
       matriz_resultado.matriz_val[sec] = fila ;
  }

  while (!feof (file))
    {
       fscanf (file, "%d", &i);
       matriz_resultado.matriz_val[(int)(iterador/filas)].secuencia_val[(int)
```

```
(iterador%filas)] = i;
        iterador++;
    }
  fclose (file);

  return matriz_resultado;
}
```

```
void liberarMatriz (matriz aBorrar){
    for (int i = 0 ; i < aBorrar.matriz_len ; i++){
        liberarVector(aBorrar.matriz_val[i]);
    }
    aBorrar.matriz_len = 0;
    free(aBorrar.matriz_val);
}
```

Al hacer los tests:

```
./bin/cliente localhost -m
./input_files/matriz1.txt +
./input_files/matriz2.txt
./bin/cliente localhost -m
./input_files/matriz1.txt -
./input_files/matriz2.txt
./bin/cliente localhost -m
./input_files/matriz1.txt x
./input_files/matriz2.txt
```

```
Tamaño de matriz-> 4 x 4 :
Tamaño de matriz-> 4 x 4 :
Sumamos
localhost -> Operación vectorial [+] =
[ 10.000000  82.000000  39.000000  79.000000
]
[ 54.000000  19.000000  91.000000  73.000000
]
[ 26.000000  51.000000  20.000000  62.000000
]
[ 64.000000  100.000000  49.000000
18.000000 ]


[ 21.000000  19.000000  91.000000  99.000000
]
[ 41.000000  57.000000  83.000000  76.000000
]
[ 15.000000  31.000000  33.000000  16.000000
]
[ 70.000000  59.000000  82.000000  84.000000
]


Resultado:
[ 31.000000  101.000000  130.000000
178.000000 ]
[ 95.000000  76.000000  174.000000
149.000000 ]
[ 41.000000  82.000000  53.000000  78.000000
]
[ 134.000000  159.000000  131.000000
102.000000 ]


Tamaño de matriz-> 4 x 4 :
```

```
Tamaño de matriz-> 4 x 4 :
Restamos
localhost -> Operación vectorial [-] =
[ 10.000000  82.000000  39.000000  79.000000
]
[ 54.000000  19.000000  91.000000  73.000000
]
[ 26.000000  51.000000  20.000000  62.000000
]
[ 64.000000  100.000000  49.000000
18.000000 ]


[ 21.000000  19.000000  91.000000  99.000000
]
[ 41.000000  57.000000  83.000000  76.000000
]
[ 15.000000  31.000000  33.000000  16.000000
]
[ 70.000000  59.000000  82.000000  84.000000
]


Resultado:
[ -11.000000  63.000000  -52.000000  -
20.000000 ]
[ 13.000000  -38.000000  8.000000  -3.000000
]
[ 11.000000  20.000000  -13.000000
46.000000 ]
[ -6.000000  41.000000  -33.000000  -
66.000000 ]


Tamaño de matriz-> 4 x 4 :
Tamaño de matriz-> 4 x 4 :
Multiplicamos
localhost -> Operación vectorial [x] =
[ 10.000000  82.000000  39.000000  79.000000
]
[ 54.000000  19.000000  91.000000  73.000000
]
[ 26.000000  51.000000  20.000000  62.000000
]
[ 64.000000  100.000000  49.000000
18.000000 ]


[ 21.000000  19.000000  91.000000  99.000000
]
[ 41.000000  57.000000  83.000000  76.000000
]
[ 15.000000  31.000000  33.000000  16.000000
]
[ 70.000000  59.000000  82.000000  84.000000
]


Resultado:
[ 10598.000000  12267.000000  12503.000000
10616.000000 ]
[ 8604.000000  16730.000000  9232.000000
12609.000000 ]
```

	[7502.000000 10321.000000 9093.000000 8133.000000] [8466.000000 11447.000000 13458.000000 15718.000000]

6. Matrices : Determinante de una matriz

Siguiendo con las matrices, probemos a hacer el determinante de una matriz cuadrada.

En calculadora.x , declararemos la función

```
double determinanteMatrices(matriz) = 16;
```

Y en el servidor, requeriremos de dos funciones extra, aparte de la cual se llama. Como usamos el método de eliminación de Gauss, tendremos que ir creando matrices dinámicamente:

```
double *
determinantematrices_1_svc(matriz arg1, struct svc_req *rqstp)
{
	static double  result;

	result = determinante(arg1) ;

	return &result;
}

int determinante(matriz arg1){

    double result = 0.0 ;
    int orden = arg1.matriz_len;

	if(orden == 1){
	    result = arg1.matriz_val[0].secuencia_val[0];
	}
	else{
	     for (int j = 0; j < orden; j++) {
	        result = result + arg1.matriz_val[0].secuencia_val[j] *
cofactor(arg1, 0, j);
	     }
	}

    return result ;
}


int cofactor(matriz input, int fila, int columna)
```

```
{
    matriz submatriz;
    int orden = input.matriz_len;
    int n = orden - 1;
    int i, j;


          submatriz.matriz_len = n;
          submatriz.matriz_val = malloc(n * sizeof(secuencia)) ;

          for (size_t i = 0; i < n; i++)
          {
               secuencia fila;
               fila.secuencia_len = n ;
               fila.secuencia_val = malloc(n * sizeof(double));
               submatriz.matriz_val[i] = fila ;
          }

    int x = 0;
    int y = 0;
    for (i = 0; i < orden; i++) {
        for (j = 0; j < orden; j++) {
            if (i ≠ fila && j ≠ columna) {
                submatriz.matriz_val[x].secuencia_val[y] =
input.matriz_val[i].secuencia_val[j];
                y++;
                if (y ≥ n) {
                    x++;
                    y = 0;
                }
            }
        }
    }

    return pow(-1.0, fila + columna) * determinante(submatriz);
}
```

En el cliente, sólo tenemos que usar la orden **<Programa> <Host_server> -dm <Archivo_matriz>**

Al hacer los tests:

`./bin/cliente localhost -dm ./input_files/matriz1.txt`
`./bin/cliente localhost -dm ./input_files/matriz2.txt`
	`Tamaño de matriz-> 4 x 4 :`
	
	`localhost -> Determinante = -17345874.000000`
	`Tamaño de matriz-> 4 x 4 :`
	
	`localhost -> Determinante = -923660.000000`

7. Conclusiones

La llamada a procedimiento remoto podría permitir realizar tareas complejas que podrían requerir procesadores potentes usando un PC cualquiera como cliente. En el caso de esta calculadora, si quisiéramos hacer operaciones con matrices gigantes desde el propio PC, tal vez podría tardar mucho. Pero... ¿y si el servidor es un nodo de cómputo o un superordenador? ¿Y si distribuimos el cálculo entre varios servidores? Gracias a los RPC podríamos tener los resultados de forma sencila y posiblemente transparente al usuario.