GRADO EN INGENIERÍA INFORMÁTICA

DESARROLLO DE SISTEMAS DISTRIBUIDOS

UNIVERSIDAD
DE GRANADA

P2 : Calculadora

David Serrano Domínguez

17 de marzo de 2024

Índice

1. Calculadora Básica

Esta primera calculadora contiene las cuatro operaciones básicas: Suma, Resta, Multiplicación y División. Pudiendo hacerlo hasta con un máximo de cien números a la vez

1.1. Código RPC para calculadora básica

Con esta calculadora vamos a poder realizar operaciones básicas,es decir, sumas,restas,divisiones y multiplicaciones pero no mezclar cálculos con las distintas operaciones. Lo que si se puede hacer es poder realizar con cada tipo de operación hasta un máximo de cien calulos a la vez, es decir, podemos hacer 1+45+32+67+90+... hasta cien números. Lo mismo podremos hacerlo para la resta, división y multiplicación.
A continuación se muestra el código RPC usado para generar los archivos básicos para esta calculadora

```
#define MAX_SIZE 100
#define MIN 0
```

Listing 1: Variables Globales.

En esta primera versión tenemos las operaciones mínimas que se nos pedia para la práctica.

1.2. Implementación Servidor de calculadora básica

A continuación vamos a mostrar como hemos implementado el servidor de la calculadora básica y explicar su funcionamiento.

Primero hemos definido dos variables globales que son MAX-SIZE y MIN que nos van a ayudar en las funciones de las operaciones básicas a controlar que el tamaño del vector que contiene todos los números con los que vamos a operar no sea menor que cero ni mayor que cien.

```
calc_res *
add_1_svc(double_vector arg1,  struct svc_req *rqstp)
{
  static calc_res  result;

  xdr_free(xdr_calc_res, &result);

  result.calc_res_u.result=0;

  if(arg1.size>MAX_SIZE || arg1.size<MIN){
    result.errnum=-1;
    return &result;
  }

  result.calc_res_u.result=arg1.values[0];

  for(int i=1;i<arg1.size;i++){
    result.calc_res_u.result+=arg1.values[i];
  }
  return &result;
}
```

Listing 2: Función suma en el servidor.

Va a recibir dos argumentos el primero va a ser la estructura double-vector que definimos en el archivo de calc.x y que contiene un entero con el tamaño del vector y un array de tipo double de tamaño máximo cien. Por otro lado recibimos otro argumento que es un struct que contiene información del estilo dirección del cliente que

realizo solicitud, identificador del programa,versión,etc... En nuestro contexto no nos va a ser de utilidad alguna, por tanto lo ignoraremos.
Si nos centramos en el código podemos ver que primero vamos a tener un dato de tipo calc-res que definimos nosotros en calc.x y que contiene un dato de tipo double que llamamos result y tambien el errnum que nos servire para controlar los errores y avisar al cliente cuando se produzca alguno.Posteriormente tenemos la función xdr-free() que nos ayuda a liberar la memoria asignada y por tanto poder usar result sabiendo que esta liberada la memoria que tenía asignada anteriormente.
Una vez explicado lo anterior que nos vamos a encontrare n todas las funciones del servidor al principio vamos a explicar como hemos implementado la suma.Primero comprobamos que el tamaño del vector que contiene arg1 no sea mayor que cien y ni menor que cero, en dicho caso establecemos el valor de errnum a menos uno y se lo devolvemos al cliente. Una vez comprobado el tamaño lo siguiente que vamos a hacer es inicializar el resultado al valor de la primera posición evitando así que este pueda contener valores basura que afecten al resultado final.Cuando hemos inicializaod el resultado procedemos a recorrer el vector e ir sumando todos sus valores para de forma final obtener la suma de todos y devolverselo al cliente.

A continuación la implementación de la resta y multiplicación va a ser exactamente igual que la de la suma.

```
calc_res *
substract_1_svc(double_vector arg1,  struct svc_req *rqstp)
{
  static calc_res  result;

  xdr_free(xdr_calc_res, &result);

  result.calc_res_u.result=0;

  if(arg1.size>MAX_SIZE || arg1.size<MIN){
    result.errnum=-1;
    return &result;
  }

  result.calc_res_u.result=arg1.values[0];

  for(int i=1;i<arg1.size;i++){
    result.calc_res_u.result-=arg1.values[i];
  }
  return &result;
}
```

Listing 3: Función resta en el servidor .

```
calc_res *
multiply_1_svc(double_vector arg1,  struct svc_req *rqstp)
{
  static calc_res  result;

  xdr_free(xdr_calc_res, &result);

  result.calc_res_u.result=0;

  if(arg1.size>MAX_SIZE || arg1.size<MIN){
    result.errnum=-1;
    return &result;
  }

  result.calc_res_u.result=arg1.values[0];

  for(int i=1;i<arg1.size;i++){
    result.calc_res_u.result*=arg1.values[i];
  }

  return &result;
}
```

Listing 4: Función multiplicación en el servidor .

Una vez vista las tres primeras operaciones básicas procedemos a ver como se ha implementado la división:

```
calc_res *
divide_1_svc(double_vector arg1,  struct svc_req *rqstp)
{
  static calc_res  result;

  xdr_free(xdr_calc_res, &result);

  result.calc_res_u.result=arg1.values[0];

  for(int i=1;i<arg1.size;i++){
    if(arg1.values[i]==0.0){
      result.errnum=-1;
      return &result;
    }
    result.calc_res_u.result/=arg1.values[i];
  }
  return &result;
}
```

Listing 5: Función división en el servidor .

Como podemos observar la implementación es prácticamente la misma que con las operaciones anteriores pero con un par de variaciones, la primera como podemos observar es que reseteamos a 0 el valor de errnum para que en el caso de que se haga una división a cero y se establezca a menos uno su valor, no influya en futuras operaciones que hagamos el no volver a establecer a cero su valor.
Por otro lado como acabo de decir anteriormente comprobamos en caso de dividir varios números seguidos que ningún divisor valga 0, ya que en dicho caso no se puede realizar la división y por tanto establecemos errnum a menos uno y se lo devolvemos al cliente, en caso de que esto no suceda se realizara la división con total normalidad.

1.3. Implementación Cliente de calculadora básica

Vamos a ver como hemos implementado el programa cliente para nuestra calculadora básica.

```
int menu(){
  int opcion;
  do{
    printf("\n\nBienvenido a la calculadora introduzca la operacion que desea realizar:"
      "\n ------------OPERACIONES BASICAS(MAXIMO 100 NUMEROS)------------"
        "\n 1.Introduzca 1 para realizar una suma."
      "\n 2.Introduzca 2 para realizar una resta."
      "\n 3.Introduzca 3 para realizar una multiplicacion."
      "\n 4.Introduzca 4 para realizar una division."
      "\n 5.Salir\n");
    scanf("%d",&opcion);
  }while(opcion>5 || opcion<1);
  return opcion;
}
```

Listing 6: Menú.

Lo primero que vamos a ver es la función menú que será lo que se le va a mostrar al cliente por pantalla cuando vaya a usar la calculadora, como vemos hay un total de cinco opciones las cuatro primeras representan las operaciones básicas y la opción número cinco nos va a permitir cerrar la calculadora.
Todo este menú se va a encontrar dentro de un do while para controlar que si nos dan un número que no es de ninguna opción volver a pedirselo al cliente.
Por último una vez seleccionada la opción la devolveremos para posteriormente usarla en nuestro main que veremos más adelante

Ahora vamos a proceder a ver como funciona la función que va a gestionar las distintas operaciones y como le va a mandar los argumentos al servidor.

```
void
calprog_1(char *host, double vector[], int tamanio,int opcion)
{
  CLIENT *clnt;
  calc_res  *result;
  double_vector arg1;

  arg1.size=tamanio;
  for(int i=0;i<tamanio;i++){
    arg1.values[i]=vector[i];
  }


#ifndef DEBUG
  clnt = clnt_create (host, CALPROG, CAL_VER, "udp");
  if (clnt == NULL) {
    clnt_pcreateerror (host);
    exit (1);
  }
#endif  /* DEBUG */
```

Listing 7: Primera parte de función calprog del cliente.

Vemos que recibe cuatro argumentos que son la dirección del host, un array de tipo double, el tamaño del array y opción que será la que selecciono en el menú el cliente.A continuación nos encontramos con una variable de clnt de tipo CLIENT, con otra de la estructura calc-res y con una última de tipo double-vector. Todas estas las necesitaremos a continuación.

```
switch (opcion){
  case 1:
    result = add_1(arg1, clnt);
    if (result == (calc_res *) NULL) {
      clnt_perror (clnt, "call failed");
    }
    printf("\nEl resultado es %lf \n",result->calc_res_u.result);
    break;
  case 2:
    result = substract_1(arg1, clnt);
    if (result == (calc_res *) NULL) {
      clnt_perror (clnt, "call failed");
    }
    printf("\nEl resultado es %lf \n",result->calc_res_u.result);
    break;
  case 3:
    result = multiply_1(arg1, clnt);
    if (result == (calc_res *) NULL) {
      clnt_perror (clnt, "call failed");
    }
    printf("\nEl resultado es %lf \n",result->calc_res_u.result);
    break;
  case 4:
    result = divide_1(arg1, clnt);
    if (result == (calc_res *) NULL) {
      clnt_perror (clnt, "call failed");
    }else if(result->errnum==-1){
      printf("\nNo se puede divdir por 0\n");
    }else{
      printf("\nEl resultado es %lf \n",result->calc_res_u.result);
    }
    break;
```

Listing 8: Segunda parte de función calprog del cliente.

Como vemos nos vamos a encontrar con un switch con cinco opciones, la última no sale en la captura ya que simplemente hace un exit(0). El caso uno del switch llamara a la operación suma del servidor y guardare el resultado en el struct result, para posteriormente comprobar que si no nos lo devuelve nulo o si el valor de errnum no vale menos uno.En caso dw que no haya ningún problema mostramos por la terminal el resultado.

En la resta y multiplicación se hara exactamente igual que para la suma y solo tendremos una variación en el caso de la división que añadimos una comprobación más que sera avisar al cliente si ha intentado dividir por cero.

Ahora vamos a ver el funcionamiento del main en el programa del cliente par ver como se obtienen los valores de la operación y que pasaría si mete una cantidad mayor a cien el cliente,etc...

```
do{
    opcion = menu();
    if(opcion!=5){
      do{
        printf("Cuantos numero quiere aniadir a la operacion: ");
        scanf("%d",&cant);
      }while(cant>100 || cant<0);
      for(int i=0;i<cant;i++){
        printf("Introduzca el siguiente numero: ");
        scanf("%lf",&valor);
        vector[i]=valor;
      }
      calprog_1 (host,vector,cant,opcion);
      printf("\nDesea realizar otra operacion: "
           "\n1. Introduzca cualquier numero en caso de que si: "
           "\n2. Introduzca 0 en caso de que no: \n");
      scanf("%d",&continuar);
      if(continuar==0) opcion=5;
    }
}while(opcion!=5);
exit (0);
```

Listing 9: Código main

Vemos que lo primero que hacemos es llamar a la opción menú dentro de un do while para que mientras el usuario no se salga pueda seguir haciendo operaciones. A continuación una vez que tenemos la opción escogida procedemos a preguntar con cuantos número desea hacer la operación solicitada y mediante otro do while controlamos que no nos de una cantidad mayor a cien ni menor a cero, una vez sabida la cantidad de número mediante un for vamos pidiendo uno a uno los valores con los que desea operar y los vamos guardando en un vector, para posteriormente pasarle al cal-prog el vector con dichos números, la cantidad de números que son y la opción escogida por el usuario.
Una vez hecho lo anterior preguntamos al usuario si desea realizar una nueva operación o desea salir de la calculadora.

2. Calculadora Versión Final

Vamos a pasar a hablar de nuestra versión final de la calculadora la cual de base va a tener las operaciones básicas descritas e el apartado anterior pero vamos a añadirle operaciones con vectores y matrices.
En el caso de operaciones con vectores lo que vamos a poder hacer es sumar,restar,dividir y multiplicar las componentes de dos vectores de igual tamaño de manera que si tenemos dos vectores de tamaño dos realizaremos la operación entre la componente uno del vector uno con la componente uno también del vector dos, y asi con todas las componenetes de ambos vectores, obteniendo de manera final un vector de tamaño similar a los vectores con los que operamos que contendra el resultado de las operaciones de cada componente.
Por otro lado las operaciones con matrices que vamos a poder realizar va a ser sumar,restar y multiplicar dos matrices, en el caso de las dos primeras operaciones con matrices cuadradas y las multiplicaciones no tienen que ser cuadradas solo que se cumpla el requisito de que el número de columnas de la primera matriz es igual al número de filas de la segunda matriz. Las matrices podran ser de un tamaño máximo de cinco por cinco ya que es raro que se hagan operaciones con matrices de mayor tamaño
Por último la operación que vamos a poder hacer con matrices es calcular su determinante pero en este caso las marices no van a poder ser de un tamaño mayor de tres por tres para simplificar el calcuo que hara el servidor.

2.1. Código RPC para la versión final de la calculadora

```
const MAX = 100;
const MAX_MATRIX = 25;
typedef double EXTENDED_DOUBLE;
typedef struct double_vector double_vector<MAX>;

struct double_vector{
    int size;
    double values[MAX];
};

struct double_matrix{
    int filas;
    int columnas;
    double values[MAX_MATRIX];
};

union calc_res switch (int errnum) {
    case 0:
        double result;
    default:
        void;
};

union calc_vec switch (int errnum) {
    case 0:
        double_vector result;
    default:
        void;
};

union calc_mat switch (int errnum){
    case 0:
        double_matrix result;
    default:
        void;
};
```

Listing 10: Estructuras de versión final

En primer lugar vamos a ver las estructuras que hemos creado en el código RPC para usar finalmente en nuestra calculadora. Como podemos observar se ha añadido un struct nuevo para las matrices que contiene el

número de filas y columnas de una matriz asi como un vector que la contiene, ya que las matrices las vamos a representar en vectores unidimensionales y el tamaño de dicho vector no va a ser mayor a veinticinco ya que como solo podemos tener cinco filas y columnas como máximo, el número de elementos máximo será veinticinco.
Para las operaciones con vectores hemos creado una unión para controlar cuando se produzca un error en el servidor avisar al cliente con errnum y en caso de que no lo haya devolveremos una variable de tipo double-vector que contendra el resultado. Por último en el caso de las operaciones con matrices la idea es similar solo que cuando no haya error lo que devolveremos será un variable de tipo double-matrix que contendra el resultado de la operación.

```
program CALPROG {
    version CAL_VER {
        /*Operaciones Basicas*/
        calc_res ADD(double_vector) = 1;
        calc_res SUBSTRACT(double_vector) = 2;
        calc_res MULTIPLY(double_vector) = 3;
        calc_res DIVIDE(double_vector) = 4;
        /*Operaciones con Vectores*/
        calc_vec ADD_VECTOR(double_vector,double_vector) = 5;
        calc_vec SUBSTRACT_VECTOR(double_vector,double_vector) = 6;
        calc_vec MULTIPLY_VECTOR(double_vector,double_vector) = 7;
        calc_vec DIVIDE_VECTOR(double_vector,double_vector) = 8;
        /*Operaciones con Matrices*/
        calc_mat ADD_MATRIX(double_matrix,double_matrix) = 9;
        calc_mat SUBSTRACT_MATRIX(double_matrix,double_matrix) = 10;
        calc_mat MULTIPLY_MATRIX(double_matrix,double_matrix) = 11;
        calc_res DETERMINANT_MATRIX(double_matrix) = 12;
    } = 1;
} = 0x20000157;
```

Listing 11: Procedimientos de versión final

2.2. Implentación Servidor de calculadora final

Vamos a proceder a explicar los distintos procedimientos que hemos agregado al programa para que nuestra calculadora pueda realizar operaciones con matrices y vectores.

```
#define MAX_SIZE 100
#define MIN 0
#define MAX_SIZE_MATRIX 25
#define MAX_ROWS 5
#define MAX_COLUMNS 5
#define MAX_ROWS_D 3
#define MAX_COLUMNS_D 3
```

Listing 12: Variables globales del servidor

Vamos a empezar con la operación suma de vectores, la cual es parecida a la básica solo que esta vez al recorrer el for, sumamos la posición i del primer vector más la posición i del segundo vector y lo guardamos en un tercer vector que sera el de result. Las comprobaciones que vamos a hacer en este caso para devolver o no un errnum igual a menos uno, van a ser que el tamaño de los vectores no sea mayor que cien ni menor que cero.Por otro lado vamos a comprobar que el tamaño del vector uno no sea diferente al tamaño del vector dos.

```
calc_vec *
add_vector_1_svc(double_vector arg1, double_vector arg2,  struct svc_req *rqstp)
{
  static calc_vec  result;

  xdr_free(xdr_calc_vec, &result);

  for(int i=0;i<arg1.size;i++){
    result.calc_vec_u.result.values[i]=0;
  }
```

```
11
12  if(arg1.size>MAX_SIZE || arg2.size>MAX_SIZE || arg1.size<MIN || arg2.size<MIN || arg1.size!=
      arg2.size){
13    result.errnum=-1;
14    return &result;
15  }
16
17  result.calc_vec_u.result.size=arg1.size;
18
19  for(int i=0;i<arg1.size;i++){
20    result.calc_vec_u.result.values[i]=arg1.values[i]+arg2.values[i];
21  }
22
23  return &result;
24 }
```

Listing 13: Función suma de vectores en el servidor

Y las otras tres operaciones van a ser muy similares solo que restaremos, multiplicaremos y dividiremos, en el caso de esta última al igual que con la calculadora básica comprobaremos que no sea divida por cero.

```
1  calc_vec *
2  divide_vector_1_svc(double_vector arg1, double_vector arg2,  struct svc_req *rqstp)
3  {
4    static calc_vec  result;
5
6    xdr_free(xdr_calc_vec, &result);
7
8    for(int i=0;i<arg1.size;i++){
9      result.calc_vec_u.result.values[i]=0;
10   }
11
12   if(arg1.size>MAX_SIZE || arg2.size>MAX_SIZE || arg1.size<MIN || arg2.size<MIN || arg1.size!=
       arg2.size){
13     result.errnum=-1;
14     return &result;
15   }
16
17   result.calc_vec_u.result.size=arg1.size;
18   result.errnum = 0;
19
20   for(int i=0;i<arg1.size;i++){
21     if(arg2.values[i]==0.0){
22       result.errnum=-1;
23       return &result;
24     }
25     result.calc_vec_u.result.values[i]=arg1.values[i]/arg2.values[i];
26   }
27   return &result;
28 }
```

Listing 14: Función división de vectores en el servidor

Una vez explicado el funcionamiento de las operaciones con vectores vamos a pasar a las operaciones con matrices, teniamos diferentes posibilidades pero finalmente las operaciones que hemos incorporado son las suma, resta y multiplicación de matrices de hasta un tamaño máximo de 5x5 y además el calculo de determinante de una matrices de hasta 3x3.
Vamos a pasar a explicar como las hemos implementado.

```
1  calc_mat *
2  add_matrix_1_svc(double_matrix arg1, double_matrix arg2,  struct svc_req *rqstp)
3  {
4    static calc_mat  result;
5
6    xdr_free(xdr_calc_mat, &result);
7
```

```
  //Ponemos a 0 Matriz Resultado
  for(int i=0;i<arg1.filas;i++){
    for(int j=0;j<arg1.columnas;j++){
      result.calc_mat_u.result.values[i*arg1.columnas+j]=0;
    }
  }

  if(arg1.filas>MAX_ROWS || arg1.columnas>MAX_COLUMNS || arg1.filas<MIN || arg1.columnas<MIN
    || arg1.filas!=arg2.filas && arg1.columnas!=arg2.columnas){
    result.errnum=-1;
    return &result;
  }

  result.calc_mat_u.result.filas=arg1.filas;
  result.calc_mat_u.result.columnas=arg1.columnas;

  for(int i=0;i<arg1.filas;i++){
    for(int j=0;j<arg1.columnas;j++){
      result.calc_mat_u.result.values[i*arg1.columnas+j]=arg1.values[i*arg1.columnas+j]+arg2.
    values[i*arg1.columnas+j];
    }
  }


  return &result;
}
```

Listing 15: Función suma de matrices en el servidor

Lo primero que hacemos es inicializar la matriz resultado a cero todos sus valores, para posteriormente pasar a hacer diferentes comprobaciones para sino poner errnum a menos uno y devolverselo al cliente.Las comprobaciones que haremos serán que el número de filas y columnas no sea mayor que cinco ni menor que cero, además comprobamos que el número de filas y columnas de ambas matrices sean iguales ya que para sumar dos matrices deben ser cuadradas.
Una vez realizadas dichas comprobaciones procedemos a rellenar el vector unidimensional que representa la matriz resultado e ir agregando la suma de las componentes de las matrices que pasamos como arguentos al servidor. En el caso de la función resta del servidor va a ser similar a de la suma, solo que restaremos las componentes.

Vamos a pasar a ver las dos últimas operaciones que nos quedan que es la de multiplicar matrices y calcular el determinante.

```
calc_mat *
multiply_matrix_1_svc(double_matrix arg1, double_matrix arg2,  struct svc_req *rqstp)
{
  static calc_mat  result;

  xdr_free(xdr_calc_mat, &result);

  //Ponemos a 0 Matriz Resultado
  for(int i=0;i<arg1.filas;i++){
    for(int j=0;j<arg1.columnas;j++){
      result.calc_mat_u.result.values[i*arg1.columnas+j]=0;
    }
  }


  if(arg1.filas>MAX_ROWS || arg1.columnas>MAX_COLUMNS || arg1.filas<MIN || arg1.columnas<MIN){
    result.errnum=-1;
    return &result;
  }else if(arg1.columnas!=arg2.filas){
    result.errnum=-2;
    return &result;
  }

  result.calc_mat_u.result.filas=arg1.filas;
  result.calc_mat_u.result.columnas=arg2.columnas;
```

```
26
27      for(int i=0;i<arg1.filas;i++) {
28          for(int j=0;j<arg2.columnas;j++) {
29              for(int k=0;k<arg1.columnas;k++) {
30                  result.calc_mat_u.result.values[i*arg2.columnas+j]+=
31                      arg1.values[i*arg1.columnas+k]*arg2.values[k*arg2.columnas+j];
32              }
33          }
34      }
35
36    return &result;
37  }
```

Listing 16: Función multiplicación de matrices en el servidor

Como vemos mediante esta función no solo vamos a poder multiplicar matrices cuadradas también vamos a poder hacelos con matrices en las cuales el número de columnas de la primera sea igual al número de filas de la segunda matriz ypor tanto obtener una de tamaño que tenga el mismo número de filas de la primera matriz y columnas de la segunda matriz.
Para calcular la matriz resultanto vamos a usar tres bucles anidados para multiplicar cada fila de la primera matriz por todas las columnas de la segunda matriz y asi obtener la matriz resultante y devolversela al cliente.

Vamos a pasar a ver por último como es la función que calcula el determinante.

```
1  calc_res *
2  determinant_matrix_1_svc(double_matrix arg1,  struct svc_req *rqstp)
3  {
4    static calc_res  result;
5
6    xdr_free(xdr_calc_res, &result);
7
8    if(arg1.filas>MAX_ROWS_D || arg1.columnas>MAX_COLUMNS_D || arg1.filas<MIN || arg1.columnas<
       MIN || arg1.filas!=arg1.columnas){
9      result.errnum=-1;
10     return &result;
11   }
12
13     //Caso base: si la matriz es 1x1, su determinante es el propio elemento
14     if (arg1.filas==1) {
15     result.calc_res_u.result=arg1.values[0];
16         return &result;
17     } else if (arg1.filas==2) { // Caso base: si la matriz es 2x2, aplicamos la formula
       directamente
18         result.calc_res_u.result=(arg1.values[0]*arg1.values[3])-(arg1.values[1]*arg1.values
       [2]);
19     return &result;
20     }
21
22   result.calc_res_u.result=(arg1.values[0]*arg1.values[4]*arg1.values[8])+(arg1.values[1]*arg1
       .values[5]*arg1.values[6])+(arg1.values[2]*arg1.values[3]*arg1.values[7])-
23       (arg1.values[2]*arg1.values[4]*arg1.values[6])-(arg1.values[0]*arg1.values[5]*arg1.
       values[7])-(arg1.values[1]*arg1.values[3]*arg1.values[8]);
24
25   return &result;
26 }
```

Listing 17: Función multiplicación de matrices en el servidor

Como vemos tenemos un caso base que será cuando tengamos una matriz de 1x1 y en dicho caso el valor del determinante será el propio número que forma la matriz. El siguiente caso será cuando tengamos una matriz de 2x2 y la forma de calcular la matriz de dicho tamaño es multiplicando en cruz sus componentes,es decir, componente uno por cuatro y dos por tres, en nuestro caso es lo mismo solo que empezamos en cero es vez de en uno. Y por último nos queda el caso de una matriz de 3x3 y que en dicho caso lo mismo aplicaremos lo que se llama regla de Sarrus para determinante de matrices de 3x3 y de esta forma podremos obtener el determinante de la matriz que le pasamos al servidor.

2.3. Implementación Cliente de calculadora final

Vamos a pasar a ver el cliente, aunque realmente este es muy parecido a la versión básica, principalmente lo que cambia es el menú, el main y dos funciones extra para representar los vectores y matrices resultado.

```
int menu(){
  int opcion;
  do{
    printf("\n\nBienvenido a la calculadora introduzca la operacion que desea realizar:"
        "\n ------------OPERACIONES BASICAS(MAXIMO 100 NUMEROS)------------"
          "\n 1.Introduzca 1 para realizar una suma."
        "\n 2.Introduzca 2 para realizar una resta."
        "\n 3.Introduzca 3 para realizar una multiplicacion."
        "\n 4.Introduzca 4 para realizar una division."
        "\n\n ------------OPERACIONES CON VECTORES(TAMANIO MAXIMO 100)------------"
        "\n 5.Introduzca 5 para sumar componentes de 2 vectores"
        "\n 6.Introduzca 6 para restar componentes de 2 vectores"
        "\n 7.Introduzca 7 para multiplicar componentes de 2 vectores"
        "\n 8.Introduzca 8 para dividir componentes de 2 vectores"
        "\n\n ------------OPERACIONES CON MATRICES CUADRADAS(MAXIMO 5X5)------------"
        "\n 9.Introduzca 9 para sumar 2 matrices cuadradas"
        "\n 10.Introduzca 10 para restar 2 matrices cuadrada"
        "\n 11.Introduzca 11 para multiplicar 2 matrices(NO TIENE PORQUE SER CUADRADAS)"
        "\n 12.Introduzca 12 para calcular determinante de una matriz(MAXIMO MATRICES DE 3x3)"
        "\n 13.Introduzca 13 para salir\n");
    scanf("%d",&opcion);
  }while(opcion>13 || opcion<1);

  return opcion;
}
```

Listing 18: Menú final

Como vemos es igual que el otro solo que añadiendo las opciones para los vectores y matrices.

Luego las funciones extra que tenemos es imprimirVectorResultado() y imprimirMatrizResultado() que como su propio nombre indica solo muesra por pantalla un vector o una matric con el resultado de la operación hecha.
Por otro lado el nuevo main añade un nuevo do while para controlar que tipo de operación se va a hacer,es decir, si es menor que cinco operación básica, menor que nueve será operación con vectores, luego si es menor que 12 operación con dos matrices y si es doce determinante de una matriz, de esta manera controlamos que solo tengamos que introducir por pantalla los datos necesarios para la operación que queremos hacer.
No se muestra el código en el pdf porque es muy largo y tampoco añade ninguna funcionalidad rara o dificil de entender.

```
void
calprog_1(char *host, double_vector vector1, double_vector vector2, double_matrix matrix1,
    double_matrix matrix2, int opcion)
{
  CLIENT *clnt;
  calc_res  *result;
  calc_res  *determinant;
  calc_vec  *result_v;
  calc_mat  *result_m;
  double_vector arg1;
  double_vector vector_arg1;
  double_vector vector_arg2;
  double_matrix matrix_1_arg1;
  double_matrix matrix_1_arg2;

  matrix_1_arg1=matrix_1_arg2;

  if(opcion<5){
    arg1 = vector1;
  }else if(opcion<9){
    vector_arg1 = vector1;
    vector_arg2 = vector2;
```

```
  }else{
    matrix_1_arg1 = matrix1;
    matrix_1_arg2 = matrix2;
  }
```

Listing 19: Primera parte de calprog final

Como observamos en el caso de la función calprog una de las principales diferencias con la calculadroa básica es que en vez de pasarlo un vector estático y su tamanio pasamos directamente las estructuras que definimos en RPC ñara luego asignarle mediante un if a la variable local dicha variable que pasamos como parametro en función de la opción elegida.

```
  case 5:
    result_v = add_vector_1(vector_arg1, vector_arg2, clnt);
    if (result_v == (calc_vec *) NULL) {
      clnt_perror (clnt, "call failed");
    }else if(result_v->errnum==-1){
      printf("\nError: Se produjo un error al realizar la operacion\n");
    }else{
      imprimirVectorResultado(result_v->calc_vec_u.result.values,result_v->calc_vec_u.result.
    size);
    }
    break;
  case 6:
    result_v = substract_vector_1(vector_arg1, vector_arg2, clnt);
    if (result_v == (calc_vec *) NULL) {
      clnt_perror (clnt, "call failed");
    }else if(result_v->errnum==-1){
      printf("\nError: Se produjo un error al realizar la operacion\n");
    }else{
      imprimirVectorResultado(result_v->calc_vec_u.result.values,result_v->calc_vec_u.result.
    size);
    }
    break;
  case 7:
    result_v = multiply_vector_1(vector_arg1, vector_arg2, clnt);
    if (result_v == (calc_vec *) NULL) {
      clnt_perror (clnt, "call failed");
    }else if(result_v->errnum==-1){
      printf("\nError: Se produjo un error al realizar la operacion\n");
    }else{
      imprimirVectorResultado(result_v->calc_vec_u.result.values,result_v->calc_vec_u.result.
    size);
    }
    break;
  case 8:
    result_v = divide_vector_1(vector_arg1, vector_arg2, clnt);
    if (result_v == (calc_vec *) NULL) {
      clnt_perror (clnt, "call failed");
    }else if(result_v->errnum==-1){
      printf("\nError: Intento dividir por 0 o se produjo un error al realizar la operacion\n"
    );
    }else{
      imprimirVectorResultado(result_v->calc_vec_u.result.values,result_v->calc_vec_u.result.
    size);
    }
    break;
  case 9:
    result_m = add_matrix_1(matrix_1_arg1, matrix_1_arg2, clnt);
    if (result_m == (calc_mat *) NULL) {
      clnt_perror (clnt, "call failed");
    }else if(result_m->errnum==-1){
      printf("\nError: Se produjo un error al realizar la operacion\n");
    }else{
      imprimirMatrizResultado(result_m->calc_mat_u.result.values,result_m->calc_mat_u.result.
    filas,result_m->calc_mat_u.result.columnas);
    }
    break;
```

```
51  case 10:
52    result_m = substract_matrix_1(matrix_1_arg1, matrix_1_arg2, clnt);
53    if (result_m == (calc_mat *) NULL) {
54      clnt_perror (clnt, "call failed");
55    }else if(result_m->errnum==-1){
56      printf("\nError: Se produjo un error al realizar la operacion\n");
57    }else{
58      imprimirMatrizResultado(result_m->calc_mat_u.result.values,result_m->calc_mat_u.result.
    filas,result_m->calc_mat_u.result.columnas);
59    }
60    break;
61  case 11:
62    result_m = multiply_matrix_1(matrix_1_arg1, matrix_1_arg2, clnt);
63    if (result_m == (calc_mat *) NULL) {
64      clnt_perror (clnt, "call failed");
65    }else if(result_m->errnum==-1){
66      printf("\nError: Se produjo un error al realizar la operacion\n");
67    }else if(result_m->errnum==-2){
68      printf("\nError: El numero de columnas de la primera matriz no es igual al numero de
    filas de la segundas\n");
69    }else{
70      imprimirMatrizResultado(result_m->calc_mat_u.result.values,result_m->calc_mat_u.result.
    filas,result_m->calc_mat_u.result.columnas);
71    }
72    break;
73  case 12:
74    determinant = determinant_matrix_1(matrix_1_arg1, clnt);
75    if (determinant == (calc_res *) NULL) {
76      clnt_perror (clnt, "call failed");
77    }
78    if(determinant->errnum==-1){
79      printf("\nError introdujo una matriz mayor a 3X3 o no era una matriz cuadrada");
80    }else{
81      printf("\nEl determinante de la Matriz es %lf \n",determinant->calc_res_u.result);
82    }
83    break;
```

Listing 20: Segunda parte de calprog final

Como podemos ver es muy parecido a como haciamos las operaciones básicas solo que esta vez es con vectores y matrices, y que para mostrar el resultado en vez de hacerlo directamente en calprog(), llamamos a las funciones de imprimir que explicamos anteriormente. Una vez dicho todo esto quedaría explicado todo lo respecto a la versión de la calculadora final.

3. Guia de uso de la calculadora

La calculadora tiene un uso bastante sencillo, ya que nada más ejecutarla nos va apreguntar que operación deseamos hacer aqui un ejemplo:

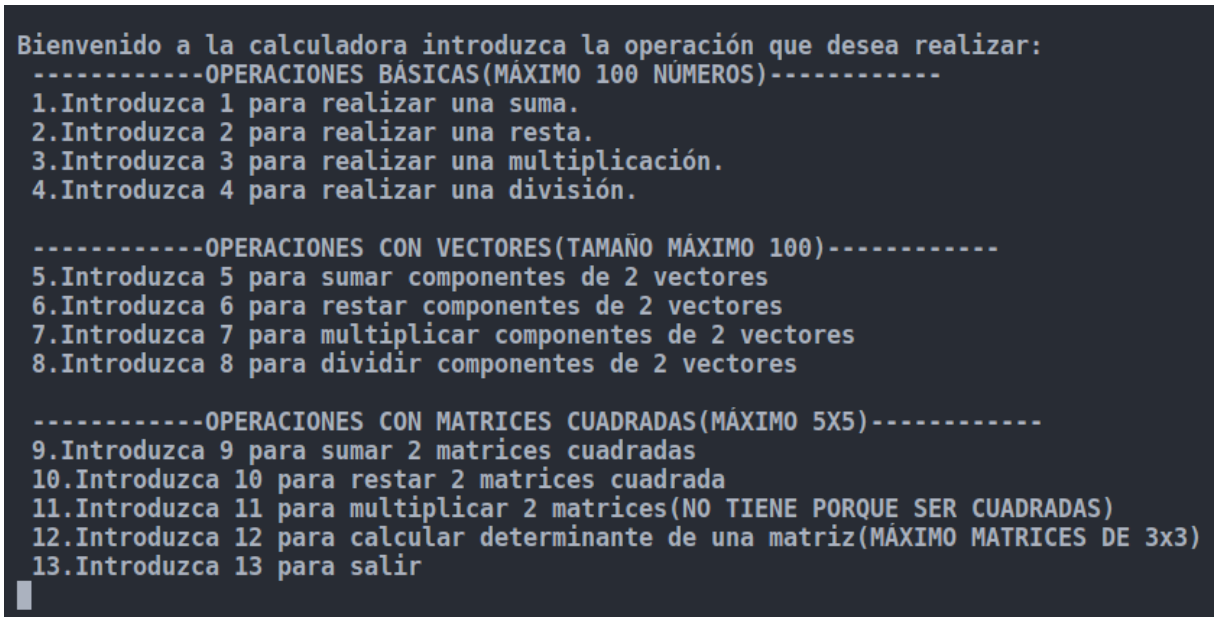

Figura 1: Menu.

Seleccionamos por ejemplo la opción 11 que es multiplicar dos matrices

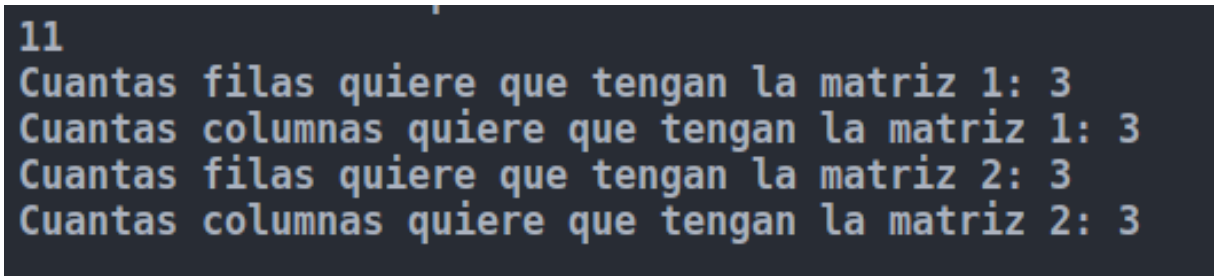

Figura 2: Introducir filas y columnas.

Nos pide las filas y columnas de cada matriz y se las introducimos(recordar que al multiplicar columnas de primera matriz debe ser igual a filas de la segunda)

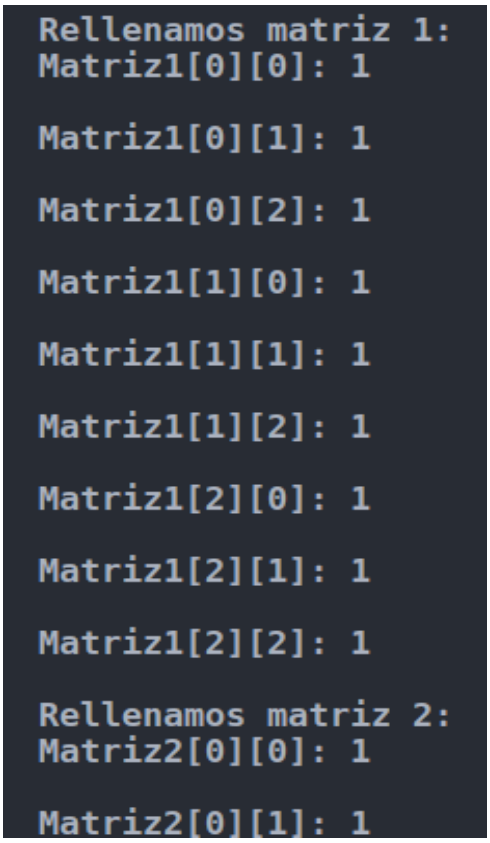

Figura 3: Rellenar filas y columnas.

Nos pide primero que rellenemos primera matriz y luego hacemos lo mismo con la segunda matriz

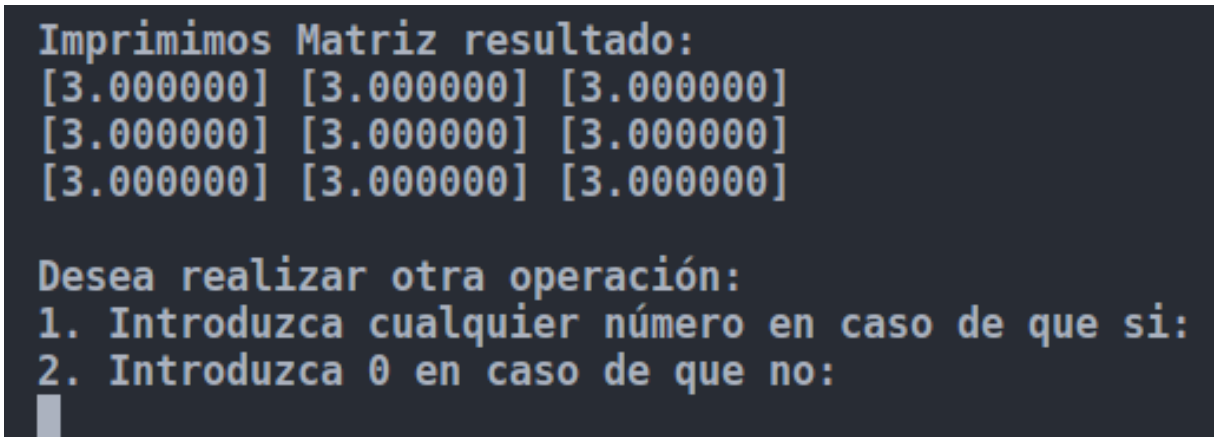

Figura 4: Solución.

Una vez rellenadas ambas matrices y demos a enter nos mostrara la solución de la operación realizada y no mostrara un menú que nos dira si queremos o no realizar una nueva operación