

Llamada a Procedimientos Remotos RPC/Sun

Ejemplos

Sistemas Distribuidos
Grado en Ingeniería Informática



Ejemplo 1. Calculadora Remota mediante RPC

En este ejemplo vamos a ver como implementar un servicio de calculadora sencilla mediante llamadas a procedimientos remotos (RPC).

Antes de nada hay que tener instalado en los equipos el paquete **rpcbind** y el paquete **portmap** para poder utilizar los puertos UDP y TCP mediante RPC. Para ello tendremos que ejecutar el comando.

```
sudo apt-get install rpcbind portmap
```

Para poder generar un servidor de calculadora hay que especificar mediante el lenguaje de definición de interfaces IDL (*Interface Definition Language*) el conjunto de procedimientos que el servidor va a proporcionar a los clientes. Vemos a continuación la interfaz IDL para la calculadora.

Fichero calculadora.x

```
struct entrada {  
    int arg1;  
    int arg2;  
};
```

Estructura con dos campos, cada uno es un operando de un procedimiento suma, resta, multiplicación o división.

```
struct entrada_c {  
    int arg1;  
    char operador;  
    int arg2;  
};
```

Estructura con tres campos, **arg1** y **arg2** son los operandos y la operación se indica en el campo **operado**.

```
program CALCULADORA {  
    version CALCULADORA_VER {  
        int sumar(entrada) = 1;  
        int restar(entrada) = 2;  
        int multiplicar(entrada) = 3;  
        int dividir(entrada) = 4;  
        int operacion(entrada_c) = 5;  
    } = 1;  
} = 0x30000001;
```

Definición de los procedimientos que podrán ser ejecutados remotamente. En la definición se indica su valor devuelto, su parámetro y un número que se utiliza como identificador.

Este fichero (**calculadora.x**) lo guardamos en un directorio que llamaremos *idl*. Además crearemos dos directorios adicionales que denominaremos *cliente* y *servidor* respectivamente.

Podemos observar en el fichero IDL que:

1. Cada procedimiento hay que asignarle un valor numérico empezando por 1. Los procedimientos se agrupan en programas y dentro de estos en versiones.
2. Cada versión debe ser enumerada igualmente como los procedimientos.
3. Los programas también deben ser enumerados pero el nº de posibles identificadores numéricos a utilizar están restringidos al rango **0x20000000 – 0x3FFFFFFF**. Dos programas en RPC no pueden tener el mismo identificador y versión en un mismo servidor. Si se utiliza el servidor *RPC_Server* aconsejo poner los últimos dígitos de vuestro DNI para que no tengáis problemas. Por ejemplo 0x20462728.

El siguiente paso es generar el código *stub* de cliente (**calculadora_clnt.c**) y el *skeleton* de la parte servidora (**calculadora_svc.c**). El *stub* y el *skeleton* son *representantes* de la parte servidora y cliente respectivamente en los procesos cliente y servidor respectivamente (ver figura en la página 13), es decir, el cliente se comunica localmente con el *stub* (representante del servidor) y el servidor se

comunica con el skeleton (representante del cliente). Internamente el stub y el skeleton se comunican haciendo transparente toda la gestión de red, la gestión de llamada remota, el pase de parámetros a los procedimientos remotos, así como el retorno de valores de obtenidos de la ejecución de dichos procedimientos.

Además se necesita el código (**calculadora_xdr.c**) que realiza el *aplanamiento/desaplanamiento* (*marshalling/unmarshalling*) de los tipos de datos definidos en el fichero de definición de interfaces **calculadora.x**. Este fichero (XDR) se necesita en el caso que se definan tipos de datos estructurados y se utilicen en la llamada a procedimientos remotos, en caso de utilizar parámetros con datos simples puede que no se genere dicho fichero.

Para generar automáticamente estos ficheros a partir del fichero de definición de interfaz **calculadora.x** se utiliza el comando **rpcgen** de la forma: **rpcgen calculadora.x**

Automáticamente se generarán los siguientes ficheros:

1. **calculadora.h**: Fichero con la definición de datos y procedimientos. Necesario tanto en el servidor como en el cliente.
2. **calculadora_clnt.c**: Código del stub del cliente. Se necesitará compilar junto con el código del programa cliente que se generará más tarde.
3. **calculadora_svc.c**: Código del skeleton del servidor. Se necesitara compilar junto con el código del programa servidor que se generará más tarde.
4. **calculadora_xdr.c**: Código que realiza el *marshalling* (serialización de datos) que será necesario en la compilación tanto del código del programa cliente como del código del programa servidor.

Para compartir los ficheros generados tanto en la parte servidor como la parte cliente vamos a realizar unos enlaces *simbólicos* en vez de copiarlos directamente.

Nos situamos en el directorio cliente y ejecutamos los siguientes comandos para realizar enlaces simbólicos a los ficheros comentados previamente:

```
ln -s ../idl/calculadora.h calculadora.h
ln -s ../idl/calculadora_clnt.c calculadora_clnt.c
ln -s ../idl/calculadora_xdr.c calculadora_xdr.c
```

Y en el directorio servidor un proceso similar creando los siguientes enlaces simbólicos a los ficheros comentados previamente:

```
ln -s ../idl/calculadora.h calculadora.h
ln -s ../idl/calculadora_svc.c calculadora_svc.c
ln -s ../idl/calculadora_xdr.c calculadora_xdr.c
```

El siguiente paso consiste en generar los ficheros que realizarán el servicio de servidor (**calculadorad.c**) y el cliente de la calculadora (**calculadora.c**).

Para ello en el directorio idl utilizamos el comando **rpcgen** de la siguiente manera:

```
rpcgen -Sc calculadora.x >../cliente/calculadora.c
rpcgen -Ss calculadora.x >../servidor/calculadorad.c
```

Ya tenemos generados un fichero con la estructura necesaria para servir de servidor y otro como cliente. Dichos ficheros han de rellenarse con el código que dará el servicio y el que necesita dicho servicio.

Antes de nada es recomendable generar unos ficheros Makefile tanto en el cliente como en el servidor. Los proporcionamos a continuación para que sirvan de modelo para posteriores códigos:

Fichero Makefile del cliente

```
CFLAGS=-Wall
CC=gcc

all: calculadora

calculadora: calculadora.o calculadora_clnt.o calculadora_xdr.o
    $(CC) -o $@ calculadora.o calculadora_clnt.o calculadora_xdr.o

calculadora.o: calculadora.h

clean:
    rm -f *.o calculadora
```

Fichero Makefile del servidor

```
CFLAGS=-Wall
CC=gcc

all: calculadorad

calculadorad: calculadorad.o calculadora_svc.o calculadora_xdr.o
    $(CC) -o $@ calculadorad.o calculadora_svc.o calculadora_xdr.o

calculadorad.o: calculadora.h

clean:
    rm -f *.o calculadorad
```

Empezaremos a rellenar el código del servidor. Señalamos con una flecha en color Rojo indicamos las líneas que hemos añadido, el resto son las generadas automáticamente por el programa **rpcgen**.

Fichero calculadorad.c del servidor

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "calculadora.h"

int *
sumar_1_svc(entrada *argp, struct svc_req *rqstp)
{
    static int result;
    result=argp->arg1+argp->arg2;
    return &result;
}

int *
restar_1_svc(entrada *argp, struct svc_req *rqstp)
{
    static int result;
    result=argp->arg1-argp->arg2;
    return &result;
}

int *
multiplicar_1_svc(entrada *argp, struct svc_req *rqstp)
{
    static int result;
    result=argp->arg1*argp->arg2;
    return &result;
}
```

```
int *
dividir_1_svc(entrada *argp, struct svc_req *rqstp)
{
    static int result;
    result=argp->arg1/argp->arg2;
    return &result;
}

int *
operacion_1_svc(entrada_c *argp, struct svc_req *rqstp)
{
    static int result;
    switch(argp->operador)
    {
        case '+': result=argp->arg1+argp->arg2;
                  break;
        case '-': result=argp->arg1-argp->arg2;
                  break;
        case '*': result=argp->arg1*argp->arg2;
                  break;
        case '/': result=argp->arg1/argp->arg2;
                  break;
        default:  result=0;
    };
    printf("SERVIDOR: Realizado %d %c %d = %d\n",
           argp->arg1, argp->operador, argp->arg2,
           result);
    return &result;
}
```

Antes de presentar el código de cliente, presentamos a continuación el código generado automáticamente, posteriormente presentaremos el código del cliente totalmente implementado. Se han insertado comentarios del código en verde.

Fichero inicial calculadora.c del cliente

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "calculadora.h"

void
calculadora_1(char *host)
{
    CLIENT *clnt;
    int *result_1;
    entrada sumar_1_arg;
    int *result_2;
    entrada restar_1_arg;
    int *result_3;
    entrada multiplicar_1_arg;
    int *result_4;
    entrada dividir_1_arg;
    int *result_5;
    entrada_c operacion_1_arg;
#ifdef DEBUG
    clnt = clnt_create (host, CALCULADORA, CALCULADORA_VER, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */
}
```

- El puntero clnt es el stub.
- Define una variable result por cada procedimiento remoto
- Define una variable entrada por cada procedimiento remoto

- Por defecto utiliza el protocolo UDP, es mejor utilizar el protocolo TCP


```
result_1 = sumar_1(&sumar_1_arg, clnt);
if (result_1 == (int *) NULL) {
    clnt_perror (clnt, "call failed");
}
result_2 = restar_1(&restar_1_arg, clnt);
if (result_2 == (int *) NULL) {
    clnt_perror (clnt, "call failed");
}
result_3 = multiplicar_1(&multiplicar_1_arg, clnt);
if (result_3 == (int *) NULL) {
    clnt_perror (clnt, "call failed");
}
result_4 = dividir_1(&dividir_1_arg, clnt);
if (result_4 == (int *) NULL) {
    clnt_perror (clnt, "call failed");
}
result_5 = operacion_1(&operacion_1_arg, clnt);
if (result_5 == (int *) NULL) {
    clnt_perror (clnt, "call failed");
}
#ifdef DEBUG
    clnt_destroy (clnt);
#endif /* DEBUG */
}

int
main (int argc, char *argv[])
{
    char *host;

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    calculadora_1 (host);
    exit (0);
}
```

- Define una llamada a un procedimiento remoto con las variables creadas anteriormente para cada procedimiento remoto.
- Añade a cada llamada una comprobación y un mensaje de error por si ha ocurrido algún fallo.
- Este código nos es útil como base para programar el código del cliente final.
- **Hay que tener en cuenta que este código, tal como está, fallará con toda seguridad si se intenta ejecutar, ya que en ningún momento se genera código para inicializar las variables que se utilizan en las llamadas a los procedimientos remotos.**

- El programa principal comprueba que como mínimo se introduzca por línea de comando el nombre del servidor, después realiza la llamada al código que comprueba todos los procedimientos.

Continuamos con el código del cliente, más largo por tener que realizar la interacción con el usuario. Igualmente se ha señalado con una flecha en azul el código que se ha añadido, el resto es generado por el programa **rpcgen**.

Fichero calculadora.c del cliente

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "calculadora.h"
#include <stdlib.h>
#define rnd (random()/(float) RAND_MAX)

int Menu()
{
    int opcion;
    do
    {
        printf("1.- Sumar\n");
        printf("2.- Restar\n");
        printf("3.- Multiplicar");
        printf("4.- Dividir");
        printf("5.- Automáticas");
        printf("6.- Salir\n\n");
        printf("Elige opcion:");
        scanf("%d",&opcion);
    } while (opcion<1 || opcion>6);
    return opcion;
}

int
main (int argc, char *argv[])
{
    char *host;

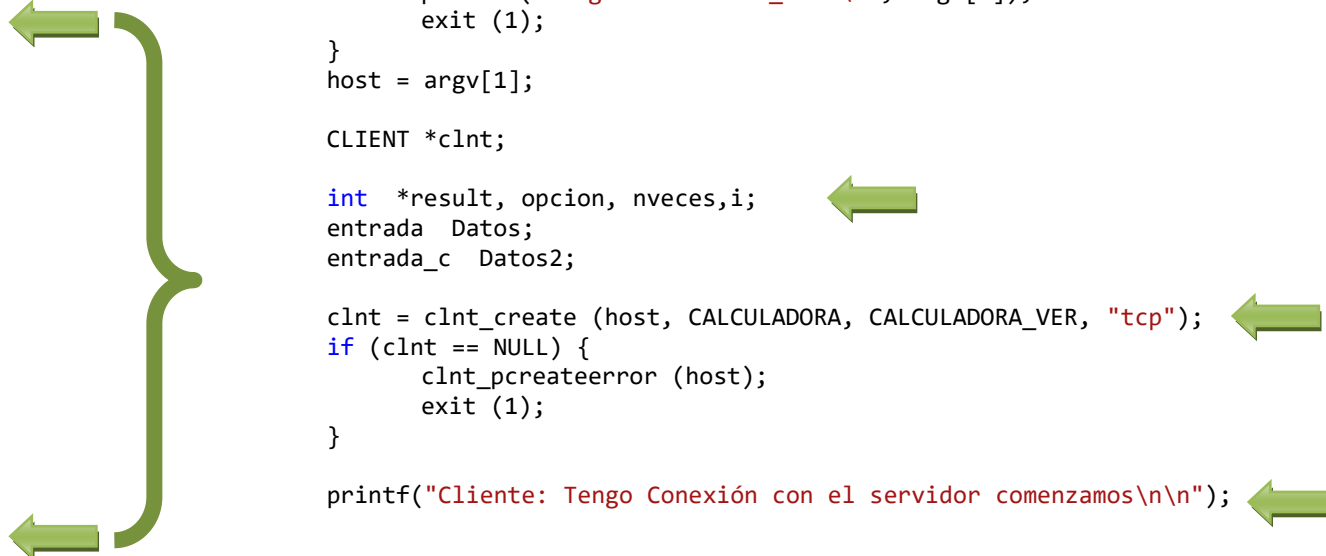
    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];

    CLIENT *clnt;

    int *result, opcion, nveces,i;
    entrada Datos;
    entrada_c Datos2;

    clnt = clnt_create (host, CALCULADORA, CALCULADORA_VER, "tcp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }


    printf("Cliente: Tengo Conexión con el servidor comenzamos\n\n");
```



```
do
{
    opcion=Menu();

    if (opcion<5)
    {
        printf("Cliente: Introduce el valor del primer argumento: ");
        scanf("%d",&Datos.arg1);
        printf("Cliente: Introduce el valor del segundo argumento: ");
        scanf("%d",&Datos.arg2);
    };

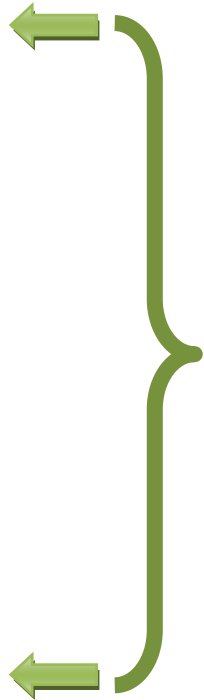
    switch (opcion)
    {
        case 1: result = sumar_1(&Datos, clnt);
            if (result == (int *) NULL)
                clnt_perror (clnt, "Cliente: Error en la llamada al procedimiento Sumar");
            printf("Cliente: Resultado de la operación: %d\n",*result);
            break;
        case 2: result = restar_1(&Datos, clnt);
            if (result == (int *) NULL)
                clnt_perror (clnt, "Cliente: Error en la llamada al procedimiento Restar");
            printf("Cliente: Resultado de la operación: %d\n",*result);
            break;
        case 3: result = multiplicar_1(&Datos, clnt);
            if (result == (int *) NULL)
                clnt_perror (clnt, "Cliente: Error en la llamada al procedimiento Multiplicar");
            printf("Cliente: Resultado de la operación: %d\n",*result);
            break;
        case 4: result = dividir_1(&Datos, clnt);
            if (result == (int *) NULL)
                clnt_perror (clnt, "Cliente: Error en la llamada al procedimiento Dividir");
            printf("Cliente: Resultado de la operación: %d\n",*result);
            break;
    }
}
```



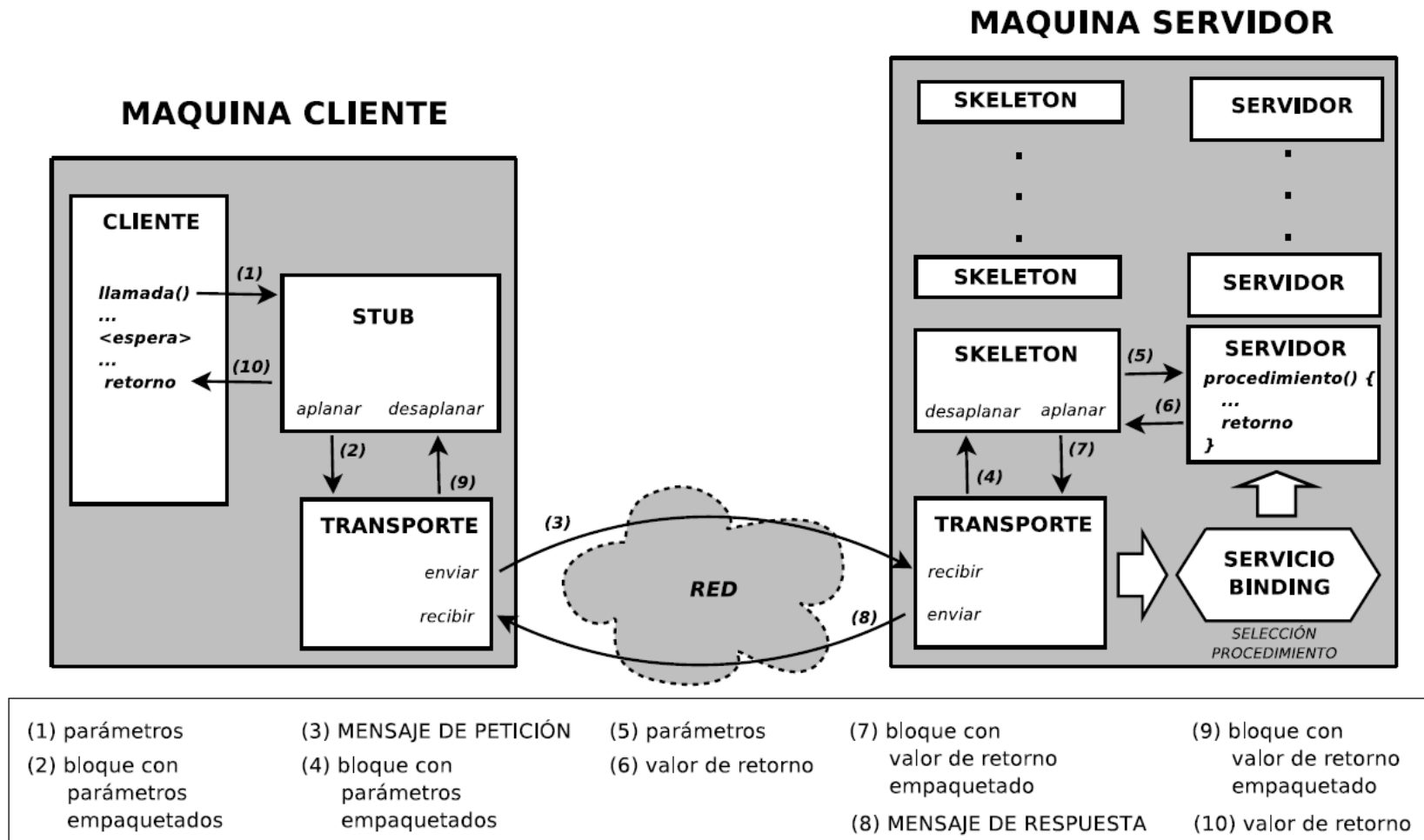
```
case 5: printf("Cliente: Introduce el numero de veces a ejecutar el comando operacion: ");
scanf("%d",&nveces);
for (i=0; i<nveces; i++)
{
    Datos2.arg1=1+rnd*100;
    Datos2.arg2=1+rnd*100;
    switch((int)(rnd*4))
    {
        case 0: Datos2.operador='+'; break;
        case 1: Datos2.operador='-'; break;
        case 2: Datos2.operador='*'; break;
        case 3: Datos2.operador='/'; break;
    }
    result = operacion_1(&Datos2, cInt);
    if (result == (int *) NULL)
        cInt_perror (cInt, "Cliente: Error en la llamada al procedimiento Operacion");
    else
        printf("Cliente: Resultado de la operación %d es: %d\n",i,*result);
    }
}

} while (opcion!=6);
printf("Cliente: Hemos terminado. Hasta otra\n\n");

cInt_destroy (cInt);
exit (0);
}
```

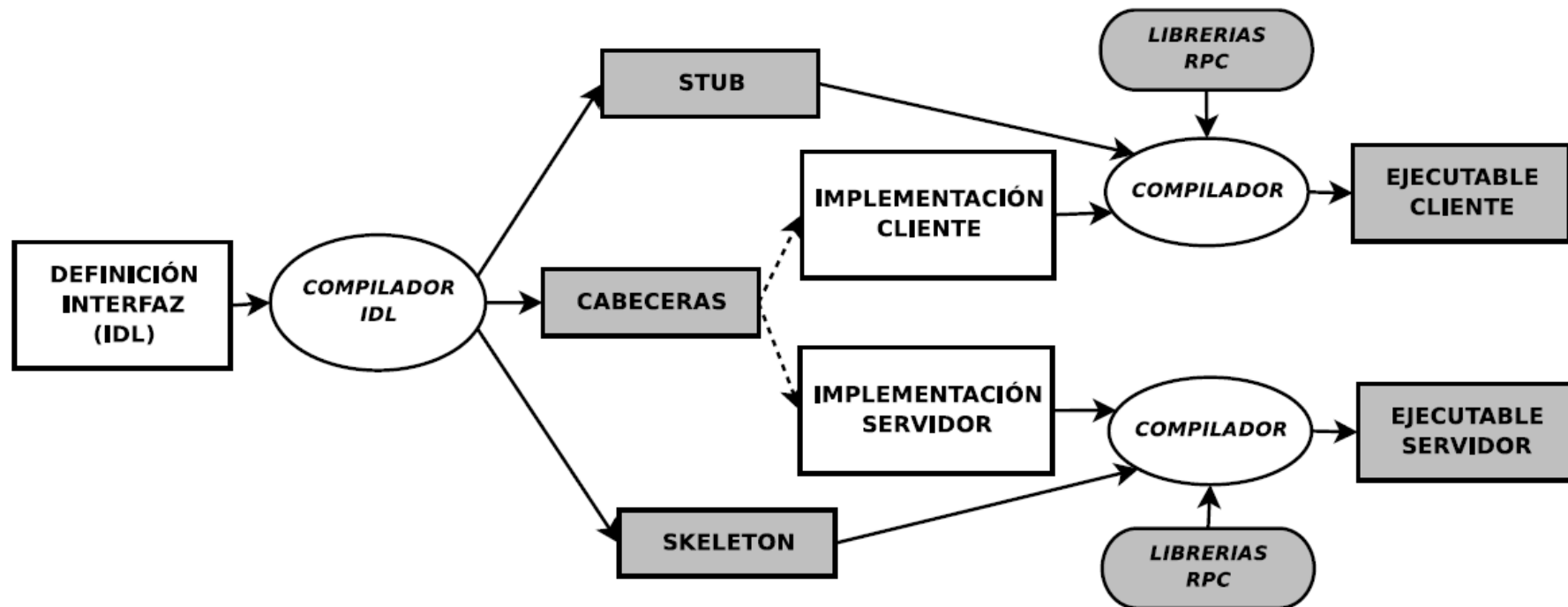


En resumen, podemos ver este diagrama los componentes típicos del proceso de llamada a procedimientos remotos.

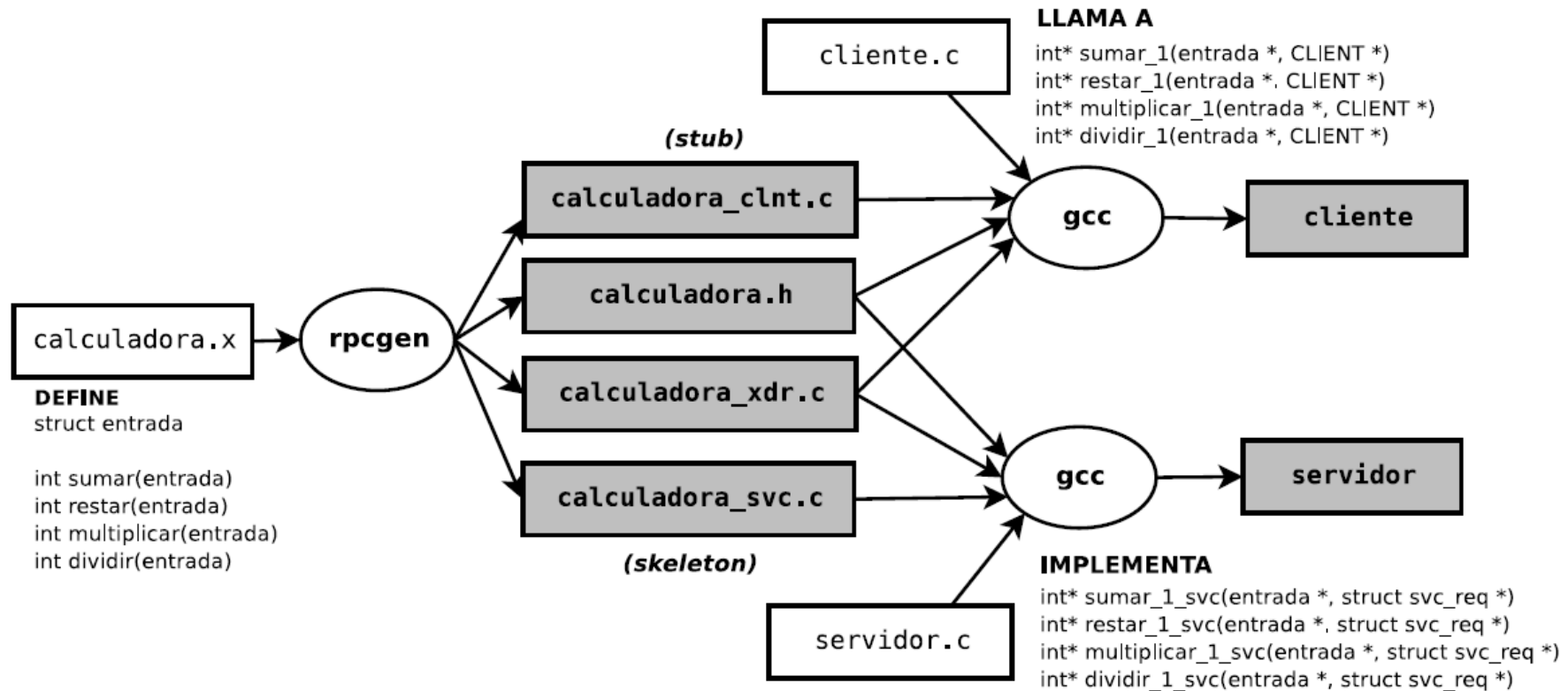


En el siguiente diagrama vemos el proceso general para utilizar RPC donde sólo hemos de implementar la definición de la interfaz IDL, el cliente y el servidor, el resto de código se encarga de generarlo el compilador de interfaces **rpcgen**.

Funcionamiento del compilador de interfaces



Resumen del proceso realizado en el ejemplo de la calculadora RPC



Ejemplo 2. Producto escalar y suma de vectores dinámicos mediante RPC

En este ejemplo se presenta sólo el fichero de definición de interfaces y los ficheros finales que implementan el servidor y el cliente. Para poder ejecutarlos, se deben realizar los mismos pasos que en el ejemplo 1.

La principal característica que presenta este código es cómo trabajar con estructuras dinámicas con un tamaño que en principio no se conocen su número de elementos aunque si es necesario conocer su tamaño máximo

Fichero vector.x

```
const MAX_VECTOR=100;

typedef float t_vector<>;

struct entrada1 {
    t_vector v;
    float    c;
};

struct entrada2 {
    t_vector v1;
    t_vector v2;
};

program OPER_VECTOR {
    version OPER_VECTORVER {
        t_vector escalado_vector(entrada1) = 1;
        t_vector suma_vectorial(entrada2)  = 2;
    } = 1;
} = 0x30000002;
```


Fichero servidor_vector.c

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "vector.h"
#include <malloc.h>

/*
 * Implementación del escalado de vectores
 * multiplica cada elemento del vector por la constante indicada
 */
t_vector * escalado_vector_1_svc(entrada1 *argp, struct svc_req *rqstp) {
    static t_vector result; /* Valor de retorno */
    static float vector_aux[MAX_VECTOR]; /* Espacio estático para almacenar el vector resultado*/

    /* Recuperar argumentos: vector + constante */
    t_vector vector = argp->v;
    float constante = argp->c;

    /* Inicializar vector de salida (indicar tamaño + pedir espacio) */
    result.t_vector_len = vector.t_vector_len;
    result.t_vector_val = vector_aux;

    /* Multiplicar cada componente por la constante */
    int i;
    for (i=0; i< result.t_vector_len; i++) {
        result.t_vector_val[i] = constante * vector.t_vector_val[i];
    }

    /* Devolver valor de retorno */
    return &result;
}
```

```
/*
 * Implementacion de la suma vectorial de 2 vectores
 *
 */
t_vector * suma_vectorial_1_svc(entrada2 *argp, struct svc_req *rqstp) {
    static t_vector  result;          /* Valor de retorno */
    static float     vector_aux[MAX_VECTOR]; /* Espacio estático para almacenar el vector resultado*/

    /* Recuperar argumentos: 2 vectores */
    t_vector vector1 = argp->v1;
    t_vector vector2 = argp->v2;

    /* Inicializar vector de salida (indicar tamaño + pedir espacio) */
    result.t_vector_len = vector1.t_vector_len;
    result.t_vector_val = vector_aux;

    /* Sumar componente a componente */
    int i;
    for (i=0; i< result.t_vector_len; i++) {
        result.t_vector_val[i] = vector1.t_vector_val[i] + vector2.t_vector_val[i];
    }

    /* Devolver valor de retorno */
    return &result;
}
```

Fichero cliente_vector.c

```
#include "vector.h"

#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

/* prototipos */
float * crear_vector_aleatorio(int);
void volcar_vector(t_vector);
```

```
int main(int argc, char ** argv) {
    char *host;

    if (argc != 2) {
        fprintf(stderr, "ERROR: formato incorrecto \ncliente nombre_servidor\n");
        exit(1);
    }

    host = argv[1];

    CLIENT *clnt;
    t_vector *resultado1;
    entrada1 args1;
    entrada2 args2;

    clnt = clnt_create (host, OPER_VECTOR, OPER_VECTORVER, "tcp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }

    /* Crear vectores aleatorios */
    t_vector vector1;
    vector1.t_vector_len = 5;
    vector1.t_vector_val = crear_vector_aleatorio(5);
    printf("VECTOR 1:");
    volcar_vector(vector1);
    printf("\n");

    t_vector vector2;
    vector2.t_vector_len = 5;
    vector2.t_vector_val = crear_vector_aleatorio(5);
    printf("VECTOR 2:");
    volcar_vector(vector2);
    printf("\n");
}
```

```
/* llamada a escalado_vector */
args1.v = vector1;
args1.c = 100.0;
resultado1 = escalado_vector_1(&args1, clnt);
if (resultado1 == (t_vector *) NULL) {
    clnt_perror (clnt, "call failed");
}
printf("ESCALADO DE VECTOR (*100)\n vector resultado:");
volcar_vector(*resultado1);
printf("\n");

/* llamada a suma vectorial */
args2.v1 = vector1;
args2.v2 = vector2;
resultado1 = suma_vectorial_1(&args2, clnt);
if (resultado1 == (t_vector *) NULL) {
    clnt_perror (clnt, "call failed");
}
printf("SUMA VECTORIAL DE 2 VECTORES\n vector resultado:");
volcar_vector(*resultado1);
printf("\n");

clnt_destroy (clnt);
}

float * crear_vector_aleatorio(int tamano) {
    float * aux = (float *) malloc(tamano * sizeof(float));
    int i;

    for (i=0; i < tamano; i++){
        aux[i] = 10.0 * (random()/(float) RAND_MAX);
    }
    //    aux[i] = i;
    return(aux);
}
```

```
void volcar_vector(t_vector aux){
    int i;

    printf("( ");
    for (i = 0; i < aux.t_vector_len; i++) {
        printf("%6.2f ", aux.t_vector_val[i]);
    }
    printf(")");
}
```