Práctica 2
Programación Distribuida y Tiempo Real

Lucas Di Cunzolo
Santiago Tettamanti

Abstract—RPC - Introducción.

Index Terms—Programación Distribuida y Tiempo Real, c, LaTeX, rpc

1 Punto 1

1.1 Inciso A

Si los procedimientos fuesen locales estarían todos dentro de un mismo archivo como cualquier otro programa "común" de C. Como se muestra en el apendice A No habría latencia en la comunicación ni perdida de comunicación ya que no habría comunicación alguna con ningún host server/client (sin tener en cuenta el SO local); todas las llamadas se harían dentro del mismo espacio de direcciones.

1.2 Inciso B

Capturas de las salidas de tanto los clientes como los servidores de cada uno de los casos:

Caso 1: Figura 2 - Figura 3

Caso 2: Figura 4

No sacamos captura del server ui, no habia output en este caso

Caso 3: Figura 5 - Figura 6

Caso 4: Figura 7 - Figura 8

1.3 Inciso C

Con UDP(Figura 9): Vemos que al poner en el server en el medio de la comunicación un mensaje de exit (Figura 10), el servidor termina bruscamente en el medio de la operación (Figura 11); entonces el cliente, al terminar el timeout definido en su proceso(Figura 12), termina la comunicacion, tal como se ve en esta captura de la ejecución del cliente(Figura 13). Lo mismo pasa si en vez de terminar su proceso el servidor tarda mucho en procesar lo pedido, lo que simularemos con un mensaje de sleep dentro del proceso servidor(Figura 14) de tal manera que el sleep tenga una duracion mayor a la definida en el timeout del cliente; entonces como el servidor no da su respuesta antes que termine el timeout, el cliente cerrará la conexión(Figura 13). En este caso el servidor, a diferencia de con el mensaje exit, serguirá su ejecución esperando el próximo request(Figura 15).

Con TCP(Figura 16) el resultado es exactamente el mismo tanto cuando el servidor termina su ejecución en medio de la comunicación (agregando el mensaje exit) como cuando tarda más de lo debido (agregando mensaje sleep)

2 Punto 2

2.1 2.A

El flag -N intenta de generar código en un estándar más nuevo que el ANSI (flag -C). Al intentar compilar el código con ese flag, utilizando el código original, falla. Esto se debe a que rcpgen, genera un .h las funciones con el tipo SIN puntero, a diferencia del flag -C, que los genera como punteros a operand.

2.2 2.B

El flag -M sirve para generar código seguro para la concurrencia, utilizando un paŕametro extra al servicio, de tipo int*.
El flag -A, es la configuracion por default, que dependiendo el sistema en el que se compila, va a ser (o no), seguro para la concurrencia multihilo.

Figura 17

3 PUNTO 3

rpcgen utiliza la estructura definida en el archivo .x para generar estructuras C en ambos puntos (cliente y servidor), con las cuales va a trabajar casteando.

El servicio recibe punteros a estas estructuras, las cuales va a trabajar y retornar nuevamente casteando a (caddr_t), el cual es equivalente a un void* Esto nos permite trabajar con cualquier tipo de C, siempre volviendo a castear a la estructura definida a partir del .x

4 PUNTO 4

Para este punto, se implementaron las funciones básicas, `write|add`, `read|get` y `list|ls`

Como primera instancia, se definió nuestro archivo `.x`, disponible para ver en el apéndice B.1

Se busco manejar cualquier tipo de archivos, por eso el uso del tipo opaque.

Luego se definieron el cliente junto a los comandos disponibles (seccion B.2) y los servicios (seccion B.3).

Cabe aclarar que para mejorar la legibilidad del codigo, se opto por utilizar un arreglo de estructuras con punteros a funciones para diferenciar rapidamente los comandos.

El parseo de los argumentos al cliente cuenta con 2 etapas, la primera busca algun comando valido, y se realiza a recorriendo manualmente `argv`.
La segunda busca las flags posibles para los comandos, esto se hizo usando la libreria `getopt` para mayor facilidad.

La conexion con el servidor tuvo que ser obligatoriamente del tipo **TCP** para soportar los grandes volumenes de datos.

5 PUNTO 5

5.1 A

5.2 A.1

Para el experimento del timeout usaremos el caso 1-simple. Vemos que si ponemos en el proceso servidor un delay de 25 mediante un sleep(25) en el medio de la comunicación:

Figura 18

Entonces la comunicación se realiza exitosamente y la llamada al proceso remoto termina de forma correcta:

Figura 19

Figura 20

En cambio si ponemos en el proceso servidor un delay de 26 mediante un sleep(26) en el medio de la comunicación: Ver Figura 21 Entonces la comunicación se corta por parte del cliente y la llamada no se completa:

Ver Figura 22

Ver Figura 23

Mediante estos resultados podemos afirmar que el timeout definido en el proceso cliente es el 25 segundos, por los que si el proceso servidor o la comunicación tardan más de ese tiempo entonces el cliente finalizará la comunicación abruptamente dando por sentado una falla en la misma. Esto lo podemos corroborar en el codigo de llamada al proceso por parte del cliente:

Ver Figura 12

5.3 A.2

También utilizaremos para calcular el promedio de una llamada rpc el caso 1-simple. Cambiaremos el codigo del cliente de tal manera que solo llame a un proceso remoto y agregaremos una función para calcular el tiempo que tarda la llamada al proceso remoto: Funcion para calcular el tiempo:

```
double dwalltime (){
  double sec;
  struct timeval tv;
  gettimeofday(&tv,NULL);
  sec=tv.tv_sec + tv.tv_usec /1000000.0;
  return sec;
}
```

Llamada a proceso remoto:

```
x = atoi(argv[2]);
y = atoi(argv[3]);

double time = dwalltime();
printf("%d + %d = %d\n",x,y,
        add(clnt,x,y));

printf("%g\n", dwalltime() - time);

return(0);
```

Se ejecutó 50 veces el proceso cliente, el resultado se puede ver en el apendice C Este es el tiempo que tardó la llamada en 50 ejecuciones distintas. Debajo, la suma y el promedio de ellas, todo expresado en segundos.

5.4 B

Teniendo en cuenta los datos del inciso anterior, se redujo el timeout del proceso cliente del caso 1-simple un 10% menos que el promedio encontrado (0.00025433064 - 0.00025433064*10/100 = 0.00022889757):

```
/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 0.00022889757, 0 };
```

Fig. 1. Experiment reduced timeout

Y se ejecutó 10 veces el cliente, haciendo por lo tanto 10 llamadas remotas, pero al parecer el struct TIMEOUT no permite floats, por lo que todo lo menor a 1 será tomado como 0 y ninguna llamada será menor al tiempo de timeout.

5.5 C

Se realizó un cliente/servidor RCP en donde siempre se corta la comunicación por exceso de tiempo. Para ello se definió solo una función en el servidor, llamada sleep_time, que recibe como parámetro la cantidad de segundo que esta función se quedará "dormida" antes de seguir con la ejecución, esto se implementó con la función sleep(). El cliente por su parte lo que hace es setear su tiempo de timeout en un valor determinado (5 segundos en nuestro ejemplo), y luego llamar a la función del servidor pasando como parámetro un valor mayor al seteado en su timeout (6 segundos en nuestro ejemplo). De esta forma nos aseguramos que el servidor tardará en ejecutar el proceso más tiempo que el definido en el timeout del cliente, por lo que la llamada siempre fallará. Parte codigo cliente:

```
int main( int argc, char *argv[]) {
  CLIENT *clnt;
  if (argc!=2) {
    fprintf(stderr,"Usage: %s hostname\n",
    exit(0);
  }

  /* Create a CLIENT data structure that r
     procedure SIMP_PROG, version SIMP_VER
     host specified by the 1st command lin

  clnt = clnt_create(argv[1], SLEEPER_PROG

  /* Make sure the create worked */
  if (clnt == (CLIENT *) NULL) {
    clnt_pcreateerror(argv[1]);
    exit(1);
  }
  struct timeval tv;
  tv.tv_sec = 5;
/* change time-out to 5 seconds */
  tv.tv_usec = 0;
  clnt_control(clnt, CLSET_TIMEOUT, &tv);
  sleep_time(clnt,6);

  return(0);
}
```

Codigo servidor:

```
#include <stdio.h>
#include "sleeper.h"

int *
sleep_time_1_svc(sleeping_time *argp, stru
  static int  result;

  sleep(argp->time);
  result = argp->time;

  return (&result);
}
```

APPENDIX

Código C

.1 Punto 1

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

int add( int x, int y ) {
  return(x+y);      /* Note the use of the '+' operator to achieve addition! */
}

int subtract( int x, int y ) {
  return(x-y);      /* This is a little harder, we have to use '-' */
}


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

  int x,y;

  if (argc!=3) {
    fprintf(stderr,"Usage: simp num1 num\n");
    fprintf(stderr,"        num1 and num2 must be integer values (for now)\n");
    fprintf(stderr,"        Floating point arithmetic is coming soon - preregis
    fprintf(stderr,"        receive our new integrated multiplication program a
    exit(0);
  }

  x = atoi(argv[1]);
  y = atoi(argv[2]);

  printf("%d + %d = %d\n",x,y,add(x,y));
  printf("%d - %d = %d\n",x,y,subtract(x,y));
  return(0);
}
```

.2 Punto 4

.2.1 Definicion

```
#define VERSION_NUMBER 1

%#define DATA_SIZE UINT_MAX

struct ftp_file {
  string name<PATH_MAX>;
        opaque data<>;
        uint64_t checksum;
};
```

```
program FTP_PROG {
    version FTP_VERSION {
      ftp_file READ(string) = 1;
      int WRITE(ftp_file) = 2;
      string LIST(string) = 3;
    } = VERSION_NUMBER;
} = 555555555;

#define FTP_PROG 555555555
```

.2.2 Cliente

```
/* RPC client for simple addition example */

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <getopt.h>
#include "commands.h"

typedef int (*command_function)(CLIENT*, char*, char*);

typedef struct command {
  char name[15];
  char description[100];
  command_function callback;
} command_t;


int main( int argc, char *argv[]) {
  CLIENT *clnt;
  int i, c;

  // Declare commands with functions
  command_t commands[] = {
    {
      .name="write",
      .description="Add a file from --src to --dest",
      .callback=&ftp_write
    },
    {
      .name="add",
      .description="Add a file from --src to --dest",
      .callback=&ftp_write
    },
    {
      .name="read",
      .description="Store a file from --src to --dest",
```

```
    .callback=&ftp_read
  },
  {
    .name="get",
    .description="Store a file from --src to --dest",
    .callback=&ftp_read
  },
  {
    .name="list",
    .description="List files from --src",
    .callback=&ftp_list
  },
  {
    .name="ls",
    .description="List files from --src",
    .callback=&ftp_list
  },
};

// Check parameters
if (argc < 2) {
  fprintf(stderr,"Usage: %s\n",argv[0]);
  for (i = 0; i < sizeof(commands)/sizeof(command_t); i++) {
    fprintf(stderr, "\t- %s: %s\n", commands[i].name, commands[i].description
  }
  exit(0);
}

int command = -1;
for (i = 0; i < sizeof(commands)/sizeof(command_t); i++) {
  if (!strcmp(commands[i].name, argv[1])) {
    command = i;
    break;
  }
}

if (command == -1) {
  fprintf(stderr, "Invalid command");
  exit(1);
}

// Config getopt
int option_index = 0;
static struct option long_options[] = {
  {"src", required_argument, 0, 's'},
  {"dest", required_argument, 0, 'd'},
  {"host", required_argument, 0, 'h'},
  {0,      0,                 0,  0 }
};
```

```
// Variables for ftp command
char src[PATH_MAX];
strcpy(src, "");
char dest[PATH_MAX];
strcpy(dest, "");
char hostname[PATH_MAX];
strcpy(hostname, "");

// Parse arguments
while ((c = getopt_long(argc, argv, "s:d:h:", long_options, &option_index))
  switch (c) {
    case 's':
      strcpy(src, optarg);
      break;
    case 'd':
      strcpy(dest, optarg);
      break;
    case 'h':
      strcpy(hostname, optarg);
      break;
    default:
      abort();
  }
}

if (!strlen(src)) {
  fprintf(stderr, "Specify a --src path\n");
  exit(1);
}

if (!strlen(dest)) {
  fprintf(stderr, "--dest setted to tmp1\n");
  strcpy(dest, "tmp1");
}

// Config RPC
/* Create a CLIENT data structure that reference the RPC
   procedure FTP_PROG, version FTP_VERSION running on the
   host specified by the 1st command line arg. */
if(!strlen(hostname)) {
  strcpy(hostname, "localhost");
}
clnt = clnt_create(hostname, FTP_PROG, FTP_VERSION, "tcp");

/* Make sure the create worked */
if (clnt == (CLIENT *) NULL) {
  clnt_pcreateerror(argv[1]);
  exit(1);
```

```
  }

  printf("Connecting to server with host %s\n", hostname);
  // Call command
  commands[command].callback(clnt, src, dest);
  return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>
#include "utils.h"
#include "ftp.h"  /* Created for us by rpcgen - has everything we need ! */

/* Wrapper function takes care of calling the RPC procedure */
int ftp_write(CLIENT *clnt, char *src, char *dest) {
  double time = dwalltime();
  #ifdef DEBUG
  printf("write - Args: \n\t- src: %s\n\t- dest: %s\n\n", src, dest);
  #endif
  FILE* file;

  file = fopen(src, "r");
  if (file == NULL) {
    fprintf(stderr, "Error opening file %s\n", src);
    exit(1);
  }

  ftp_file ftp_file_data;
  int *result;

  /* Gather everything into a single data structure to send to the server */
  ftp_file_data.data.data_val = malloc(DATA_SIZE);
  ftp_file_data.data.data_len = fread(ftp_file_data.data.data_val, sizeof(char
  ftp_file_data.name = malloc(PATH_MAX);
  ftp_file_data.name = strcpy(ftp_file_data.name, dest);
  ftp_file_data.checksum = hash(ftp_file_data.data.data_val);

  fclose(file);

  #ifdef DEBUG
  printf("dest: %s\ndata: %s\nsize: %d\nchecksum: %" PRIu64 "\n",
         ftp_file_data.name,
         ftp_file_data.data.data_val,
         ftp_file_data.data.data_len,
         ftp_file_data.checksum);
  #endif
```

```
  /* Call the client stub created by rpcgen */
  result = write_1(ftp_file_data,clnt);
  if (result == NULL) {
    fprintf(stderr,"Trouble calling remote procedure\n");
    exit(0);
  } else if (*result == -1) {
    fprintf(stderr, "Error creating file 'store/%s' in server\n", dest);
  }
  printf("File stored at 'store/%s'\n", dest);
  fprintf(stderr, "Took %g ms\n\n", dwalltime()-time);
  return(*result);
}

/* Wrapper function takes care of calling the RPC procedure */
int ftp_read(CLIENT *clnt, char *src, char *dest) {
  double time = dwalltime();
  #ifdef DEBUG
  //printf("write - Args: \n\t- data: %s\n\t- dest: %s\n\n", data, dest);
  #endif
  FILE* file;
  ftp_file *ftp_file_data;

  ftp_file_data = read_1(src,clnt);
  if (ftp_file_data == NULL) {
    fprintf(stderr,"Trouble calling remote procedure\n");
    exit(0);
  }

  printf("Storing file %s...\n", dest);
  file = fopen(dest, "w");
  if (file == NULL) {
    fprintf(stderr, "Error opening file %s\n", src);
    exit(1);
  }
  fwrite(ftp_file_data->data.data_val, sizeof(char), ftp_file_data->data.data_l
  fclose(file);

  fprintf(stderr, "Took %g ms\n\n", dwalltime()-time);
  return 1;
}

/* Wrapper function takes care of calling the RPC procedure */
int ftp_list(CLIENT *clnt, char *src, char *dest) {
  double time = dwalltime();
  #ifdef DEBUG
  //printf("write - Args: \n\t- data: %s\n\t- dest: %s\n\n", data, dest);
  #endif
  char **paths;
  paths = list_1(src, clnt);
```

```
  printf("%s\n", paths[0]);
  fprintf(stderr, "Took %g ms\n\n", dwalltime()-time);
  return 0;
}
```

.2.3 Servidor

```
/* Definition of the remote add and subtract procedure used by
   simple RPC example
   rpcgen will create a template for you that contains much of the code
   needed in this file is you give it the "-Ss" command line arg.
*/
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>
#include "utils.h"
#include "ftp.h"

/* Here is the actual remote procedure */
/* The return value of this procedure must be a pointer to int! */
/* we declare the variable result as static so we can return a
   pointer to it */

int *
write_1_svc(ftp_file argp, struct svc_req *rqstp)
{
    double time = dwalltime();
    #ifdef DEBUG
    printf("write_1_svc - Args:\n\targp:\n\t\t- name: %s\n\t\t- data: %s\n\t\t-
            argp.name, argp.data.data_val, argp.data.data_len, argp.checksum);
    #endif

    // Declare variables
        FILE *file;
    DIR *dir;
    static int result;
    char path[PATH_MAX];

    // Set path
    snprintf(path, PATH_MAX, "%s/%s", "store", argp.name);

    #ifdef DEBUG
    printf("Path: %s\n\n", path);
    #endif

    dir = opendir("store");
```

```
    if (dir) {
        closedir(dir);
    } else if (ENOENT == errno) {
        mkdir("store", 0777);
    } else {
        fprintf(stderr, "Error creating file '%s'\n", path);
        result = -1;
        return &result;
    }

    // Open file and check errors
    file = fopen(path, "w");
    if (file == NULL) {
        fprintf(stderr, "Error creating file '%s'\n", path);
        result = -1;
        return &result;
    }

    // Check checksum
    uint64_t checksum = hash(argp.data.data_val);
    if (checksum != argp.checksum) {
        fprintf(stderr, "Error in checksum!!\nOriginal: %"PRIu64"\nOwn: %"PRIu
    }

    // Write file
    result = fwrite(argp.data.data_val, sizeof(char), argp.data.data_len, file

    fclose(file);
        printf("Storing %s...\n", path);

    fprintf(stderr, "Took %g ms\n\n", dwalltime()-time);
        return((int*)&result);
}

ftp_file *
read_1_svc(char *path, struct svc_req *rqstp)
{

    double time = dwalltime();
        printf("Reading %s...\n", path);
    FILE *file;
    ftp_file *file_struct;
    file_struct = malloc(sizeof(char*) + sizeof(u_int) + sizeof(char*) + sizeo
    file_struct->data.data_val = malloc(DATA_SIZE);

    file = fopen(path, "r");
    if (file == NULL) {
        fprintf(stderr, "Error opening file %s\n", path);
        file_struct->data.data_len = -1;
```

```
        return file_struct;
    }
    file_struct->data.data_len = fread(file_struct->data.data_val, sizeof(char
    file_struct->name = malloc(PATH_MAX);
    file_struct->name = strcpy(file_struct->name, path);

    fprintf(stderr, "Took %g ms\n\n", dwalltime()-time);
    return file_struct;
}

char **
list_1_svc(char *path, struct svc_req *rqstp)
{
    double time = dwalltime();
        printf("Listing files '%s'\n", path);
    DIR *dir;
    char **paths;
    paths = (char**)malloc(sizeof(char*));
    *paths = (char*)malloc(PATH_MAX);
    *paths = strcpy(*paths, "");
    struct dirent *dir_str;

    dir = opendir(path);
    if(dir) {
        while((dir_str = readdir(dir)) != NULL) {
            if (strcmp(dir_str->d_name, ".") && strcmp(dir_str->d_name, ".."))
                strcat(paths[0], dir_str->d_name);
                strcat(*paths, "\t");

                #ifdef DEBUG
                printf("%s\n", dir_str->d_name);
                #endif
            }
        }
        closedir(dir);
    } else {
        snprintf(*paths, PATH_MAX, "list: cannot access %s: No such directory"
    }
    fprintf(stderr, "Took %g ms\n\n", dwalltime()-time);
    return paths;
}
```

.3 Punto 5

```
0.000288963
0.000247002
0.000273943
0.000135899
0.000260115
0.000271082
```

0.000247955
0.000322104
0.00015521
0.000272989
0.000265121
0.000277042
0.000202179
0.000264883
0.000272989
0.00028801
0.000247955
0.000257015
0.000270844
0.000267029
0.000267982
0.000257969
0.000261068
0.00027895
0.000280142
0.000283957
0.000282764
0.000194073
0.000265837
0.000252008
0.000263929
0.000252962
0.000270128
0.000262022
0.000276089
0.000256062
0.00026989
0.000192881
0.000282049
0.000222921
0.00019002
0.000277996
0.000178099
0.000316143
0.000213146
0.000187159
0.000271082
0.000237942
0.000293016
0.000289917

Sum: 0.012716532
Average: 0.00025433064

.4 Figuras

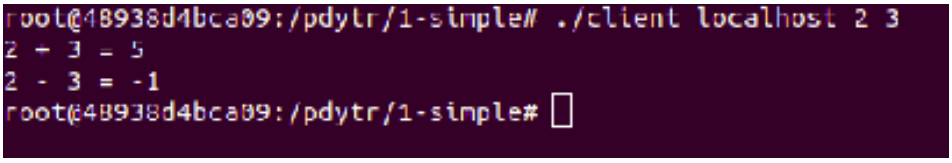

Fig. 2. Simple client

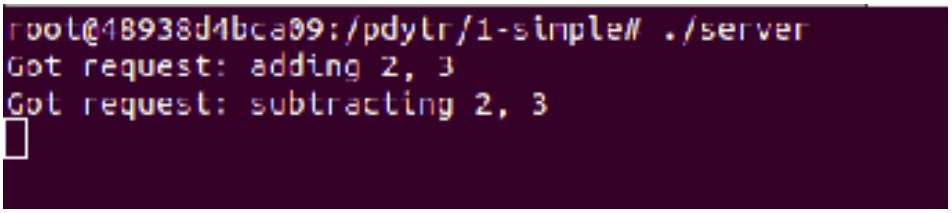

Fig. 3. Simple server

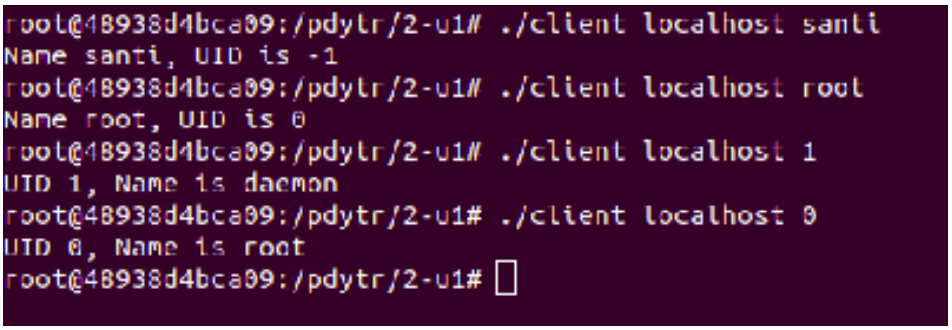

Fig. 4. UI client

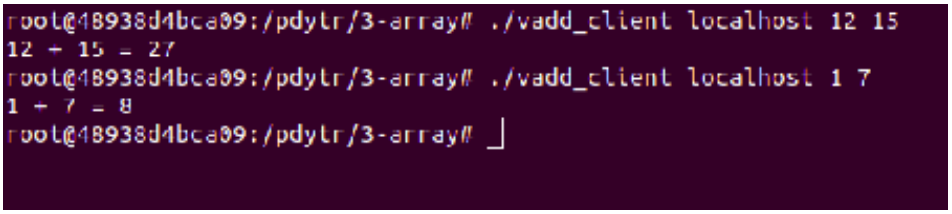

Fig. 5. Array client

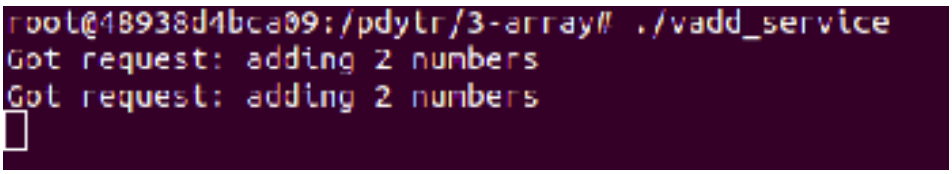

Fig. 6. Array server

```
root@48938d4bca09:/pdytr/4-list# ./client localhost 2 5
2 5
Sum is 7
root@48938d4bca09:/pdytr/4-list# ./client localhost 9878 324
9878 324
Sum is 10202
root@48938d4bca09:/pdytr/4-list#
```

Fig. 7. List client

```
root@48938d4bca09:/pdytr/4-list# ./server
```

Fig. 8. List server

```
   /* Create a CLIENT data structure that reference the RPC
      procedure SIMP_PROG, version SIMP_VERSION running on the
      host specified by the 1st command line arg. */

   clnt = clnt_create(argv[1], SIMP_PROG, SIMP_VERSION, "udp");

   /* Make sure the create worked */
   if (clnt == (CLIENT *) NULL) {
     clnt_pcreateerror(argv[1]);
     exit(1);
   }

   /* get the 2 numbers that should be added */
   x = atoi(argv[2]);
   y = atoi(argv[3]);


   printf("%d + %d = %d\n",x,y, add(clnt,x,y));
   printf("%d - %d = %d\n",x,y, sub(clnt,x,y));
   return(0);
}
```

Fig. 9. UDP client

```
int *
add_1_svc(operands *argp, struct svc_req *rqstp)
{
    static int  result;

    printf("Got request: adding %d, %d\n",
           argp->x, argp->y);

    result = argp->x + argp->y;
    exit(0);

    return (&result);
}
```

Fig. 10. Server exit

```
root@18938d4bca09:/pdylr/1-simple# ./server
Got request: adding 1, 5
root@18938d4bca09:/pdylr/1-simple#
```

Fig. 11. server trouble

```
/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

int *
add_1(operands *argp, CLIENT *clnt)
{
    static int clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, ADD,
        (xdrproc_t) xdr_operands, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}
```

Fig. 12. Client timeout

```
root@48938d4bca09:/pdylr/1-simple# ./client localhost 1 5
trouble calling remote procedure
root@48938d4bca09:/pdylr/1-simple#
```

Fig. 13. Client trouble

```
/* Here is the actual remote procedure */
/* The return value of this procedure must be a pointer to int! */
/* we declare the variable result as static so we can return a
   pointer to it */

int *
add_1_svc(operands *argp, struct svc_req *rqstp)
{
    static int  result;

    printf("Got request: adding %d, %d\n",
           argp->x, argp->y);

    result = argp->x + argp->y;
    sleep(10);
    return (&result);
}
```

Fig. 14. Server sleep

```
root@48938d4bca09:/pdylr/1-simple# ./server
Got request: adding 3, 4
```

Fig. 15. Server waiting

```
/* Create a CLIENT data structure that reference the RPC
   procedure SIMP_PROG, version SIMP_VERSION running on the
   host specified by the 1st command line arg. */

clnt = clnt_create(argv[1], SIMP_PROG, SIMP_VERSION, "tcp");

/* Make sure the create worked */
if (clnt == (CLIENT *) NULL) {
  clnt_pcreateerror(argv[1]);
  exit(1);
}

/* get the 2 numbers that should be added */
x = atoi(argv[2]);
y = atoi(argv[3]);


printf("%d + %d = %d\n",x,y, add(clnt,x,y));
printf("%d - %d = %d\n",x,y, sub(clnt,x,y));
return(0);
}
```

Fig. 16. TCP client

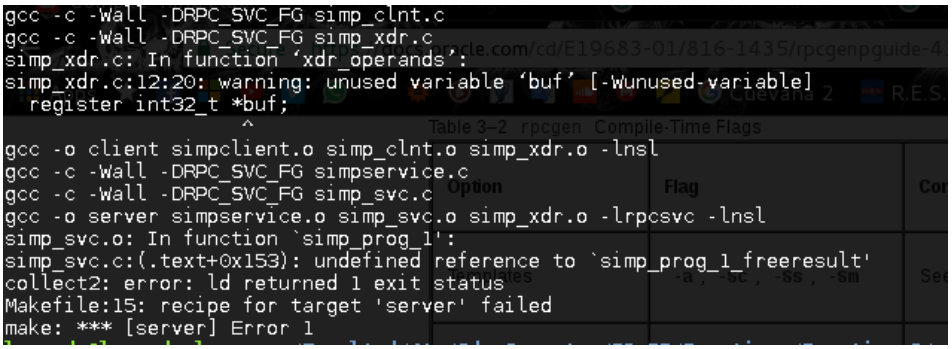

Fig. 17. Servidor

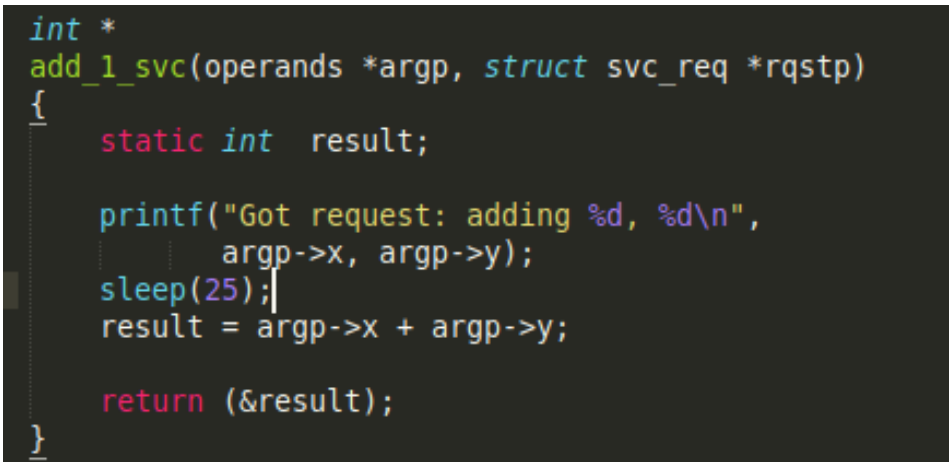

Fig. 18. Experiment timeout

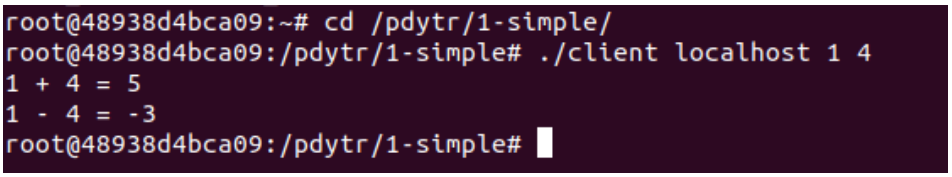

Fig. 19. Experiment timeout client1

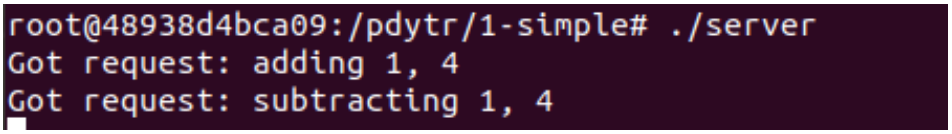

Fig. 20. Experiment timeout server1

```c
int *
add_1_svc(operands *argp, struct svc_req *rqstp)
{
    static int  result;

    printf("Got request: adding %d, %d\n",
           argp->x, argp->y);
    sleep(26);
    result = argp->x + argp->y;

    return (&result);
}
```

Fig. 21. Experiment timeout

```
root@48938d4bca09:/pdytr/1-simple# ./client localhost 1 4
Trouble calling remote procedure
root@48938d4bca09:/pdytr/1-simple#
```

Fig. 22. Experiment timeout client2

```
root@48938d4bca09:/pdytr/1-simple# ./server
Got request: adding 1, 4
```

Fig. 23. Experiment timeout serve2