Tecnologías de la Información. Redes, Comunicaciones y Arquitecturas Distribuidas.

AÑO 2010

Trabajo Práctico N° 2

Programación de Clientes y Servidores utilizando RPC en LINUX

Profesor:

- MSc. Pablo Pessolani

Integrantes:

- Lorena Diorio
lorenadiorio@gmail.com

- Ariel Rossanigo
arielrossanigo@gmail.com

- Román Zenobi
rozenobi@hotmail.com

Desarrollo del Práctico

Configuración de rpc (pingRPC.x)

```
program DATE_PROG {
    version DATE_VERS {
                string ping(string) = 1;       /* procedure number = 1 */
                } = 1;                         /* version number = 1 */
} = 0x31234567;                                /* program number = 0x31234567 */
```

Servidor RPC (servidorRPC.c)

```c
#include "pingRPC.h"
#include "utiles_ping.c"


char **
ping_1_svc(char **argp, struct svc_req *rqstp)
{
        static char * result;

        printf("Se recibio peticion en el puerto %d, desde la IP: %s\n",
                (svc_getcaller(rqstp->rq_xprt))->sin_port,
                inet_ntoa((svc_getcaller(rqstp->rq_xprt))->sin_addr));
        printf("TS: %s,  Datos:%s \n", str_hora_actual(), *argp);

        result= *argp;
        return &result;
}
```

Cliente RPC (clienteRPC.c)

```
#include "pingRPC.h"
#include "utiles_ping.c"

void date_prog_1(char *host, int tamanio, int repeticiones)
{
  CLIENT *clnt;
  char * *result_1;
  char *msg = generar_paquete(tamanio);

  clnt = clnt_create (host, DATE_PROG, DATE_VERS, "udp");
  if (clnt == NULL) {
    clnt_pcreateerror (host);
    exit (1);
  }

  int i;
  for (i=0; i< repeticiones;i++)
  {
    iniciar_temporizador();
    result_1 = ping_1(&msg, clnt);
    if (result_1 == (char **) NULL) {clnt_perror (clnt, "call failed");}

    printf("RTT paquete %d: %d uSeg\n", i+1,finalizar_temporizador());

  }
  imprimir_resultados_ping(host);
  clnt_destroy (clnt);
}


int main (int argc, char *argv[])
{
  char *ip_server;
  int puerto;
  int repeticiones = 5;
  int tamanio=100;

  if (parsear_parametros_cliente(argc, argv, &ip_server, &puerto, &repeticiones,
      &tamanio) < 0)
  {
    return -1;
  }

  date_prog_1 (ip_server, tamanio, repeticiones);
  exit (0);
}
```

PING RPC (PingRpc_svc.c)

```c
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "pingRPC.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void
date_prog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
  union {
    char *ping_1_arg;
  } argument;
  char *result;
  xdrproc_t _xdr_argument, _xdr_result;
  char *(*local)(char *, struct svc_req *);

  switch (rqstp->rq_proc) {
  case NULLPROC:
    (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
    return;

  case ping:
    _xdr_argument = (xdrproc_t) xdr_wrapstring;
    _xdr_result = (xdrproc_t) xdr_wrapstring;
    local = (char *(*)(char *, struct svc_req *)) ping_1_svc;
    break;

  default:
    svcerr_noproc (transp);
    return;
  }
  memset ((char *)&argument, 0, sizeof (argument));
  if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
    svcerr_decode (transp);
    return;
  }
  result = (*local)((char *)&argument, rqstp);
  if (result != NULL && !svc_sendreply(transp, (xdrproc_t) _xdr_result, result)) {
    svcerr_systemerr (transp);
  }
  if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
    fprintf (stderr, "%s", "unable to free arguments");
    exit (1);
  }
  return;
}

int
main (int argc, char **argv)
{
  register SVCXPRT *transp;

  pmap_unset (DATE_PROG, DATE_VERS);

  transp = svcudp_create(RPC_ANYSOCK);
  if (transp == NULL) {
```

```
    fprintf (stderr, "%s", "cannot create udp service.");
    exit(1);
  }
  if (!svc_register(transp, DATE_PROG, DATE_VERS, date_prog_1, IPPROTO_UDP)) {
    fprintf (stderr, "%s", "unable to register (DATE_PROG, DATE_VERS, udp).");
    exit(1);
  }

  transp = svctcp_create(RPC_ANYSOCK, 0, 0);
  if (transp == NULL) {
    fprintf (stderr, "%s", "cannot create tcp service.");
    exit(1);
  }
  if (!svc_register(transp, DATE_PROG, DATE_VERS, date_prog_1, IPPROTO_TCP)) {
    fprintf (stderr, "%s", "unable to register (DATE_PROG, DATE_VERS, tcp).");
    exit(1);
  }

  svc_run ();
  fprintf (stderr, "%s", "svc_run returned");
  exit (1);
  /* NOTREACHED */
}
```

PING RPC (PingRpc_clnt.c)

```c
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <memory.h> /* for memset */
#include "pingRPC.h"

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

char **
ping_1(char **argp, CLIENT *clnt)
{
  static char *clnt_res;

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

Utilies PING (Utiles_Ping.c)

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

//completa puerto de acuerdo a los parametros de entrada
int parsear_parametros_servidor(int argc, char* argv[], int *puerto)
{
  int c;
  while ((c= getopt(argc, argv, "p::")) != -1)
  {
    switch(c)
    {
      case 'p':
      {
        *puerto = atoi(optarg);
        if (*puerto <1024 || *puerto > 65535)
        {
          printf ("El puerto debe estar comprendido entre 1024 y 65535\n");
          return -1;
        }
        break;
      }
    }
  }
    return 0;
}

//completa IP, puerto, repeticiones y tamanio de acuerdo a los parametros de entrada
int parsear_parametros_cliente(int argc, char* argv[],  char **ip, int *puerto, int
*repeticiones, int *tamanio)
{
  if (argc < 2)
  {
    printf ("Error. Uso: %s <IPservidor> [-p <puerto>] [-r <repeticiones>] [-s
<tamaño>]\n", argv[0]);
    return -1;
  }
  *ip= argv[1];

  int c;
  while ((c= getopt(argc, argv, "r::p::s::")) != -1)
  {
    switch(c)
    {
      case 'r':
      {
        *repeticiones = atoi(optarg);
        if (*repeticiones < 1 || *repeticiones > 101)
        {
          printf ("El nro de repeticiones debe estar comprendido entre 1 y 101\n");
          return -1;
        }
        break;
      }
      case 's':
      {
        *tamanio = atoi(optarg);
        if (*tamanio <1 || *tamanio > 10001)
        {
          printf ("El tamano debe estar comprendido entre 1 y 10001\n");
          return -1;
        }
        break;
      }
      case 'p':
      {
        *puerto = atoi(optarg);
```

```
        if (*puerto <1024 || *puerto > 65535)
        {
          printf ("El puerto debe estar comprendido entre 1024 y 65535\n");
          return -1;
        }
        break;
      }
    }
  }
  return 0;
}

//Genera un string con los digitos del 0 al 9, de tamanio caracteres
char* generar_paquete(int tamanio)
{
  char* res = malloc((tamanio+1)*sizeof(char));
  int i;
  for (i = 0; i<tamanio; i++)
  {
    res[i] = (char)(i%10 + 48);
  }
  res[tamanio]='\0';
  return res;
}

//Variables utilizadas en los temporizadores
struct timeval start;
struct timeval stop;
struct timezone tz;

//Calcula la diferencia entre dos valores de tiempo y la almacena en out
void tvsub( out, in )
struct timeval *out, *in;
{
  if( (out->tv_usec -= in->tv_usec) < 0 )
  {
    out->tv_sec--;
    out->tv_usec += 1000000;
  }
  out->tv_sec -= in->tv_sec;
}

struct ResultadosTemporizador
{
  int Minimo;
  int Maximo;
  int Total;
  int Cantidad;
  float Promedio;
};

//variable para llevar los resultados acumulados del todas las mediciones
struct ResultadosTemporizador res;

//inicializamos res
void iniciar_resultados_temporizador()
{
  res.Minimo = 0;
  res.Maximo = -1;
  res.Cantidad= 0;
  res.Total = 0;
}

//Iniciamos una medicion, colocamos en start la hora del dia
void iniciar_temporizador()
{
  gettimeofday( &start, &tz );
}
```

```
//finalizamos una medicion, calculamos la diferencia entre la hora del dia y start
//actualizamos los valores acumulados
int finalizar_temporizador()
{
  int tiempo;
  gettimeofday( &stop, &tz );
  tvsub( &stop, &start );
  tiempo =  stop.tv_sec * 1000000 + stop.tv_usec;
  if (res.Minimo > tiempo || res.Cantidad==0) res.Minimo = tiempo;
  if (res.Maximo < tiempo || res.Cantidad==0) res.Maximo = tiempo;
  res.Total += tiempo;
  res.Cantidad++;
  return tiempo;
}

//Obtenemos los resultados acumulados de todas las mediciones
struct ResultadosTemporizador obtener_resultados_temporizador()
{
    if (res.Cantidad!= 0)
        res.Promedio = (float)res.Total/res.Cantidad;
    else
        res.Promedio = 0;
    return res;
}

//Imprimimos los resultados acumulados
void imprimir_resultados_ping(char *IP)
{
    struct ResultadosTemporizador r= obtener_resultados_temporizador();
    printf("=========================================================\n");
    printf("Ping a: %s\n", IP);
    printf("RTT min: %d uSeg, max: %d uSeg, prom: %.2f uSeg\n", r.Minimo, r.Maximo,
r.Promedio);
    printf("=========================================================\n");
}

char * str_hora_actual()
{
  struct tm *ptr;
  time_t lt;
  //obtenemos tiempo actual
  lt = time(NULL);
  //obtenemos tiempo local
  ptr = localtime(&lt);
  //obtenemos un string formateado con el tiempo local
  char * res= malloc(9*sizeof(char));
  strftime(res, 9, "%H:%M:%S", ptr);
  return res;
}
```

Makefile

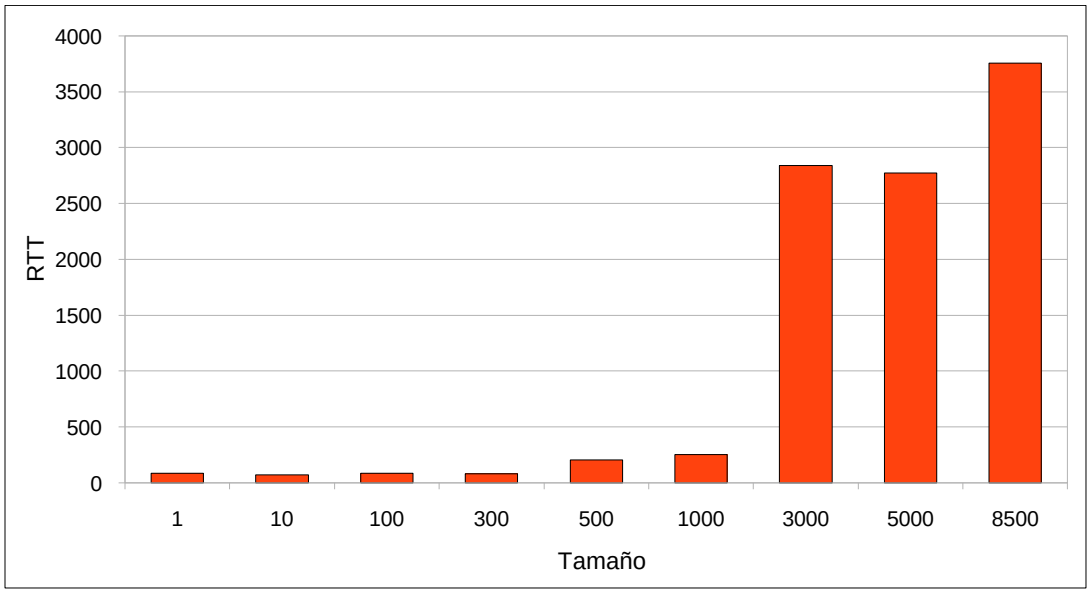
```
TP2:
	gcc -o ../obj/rpcping ../src/clienteRPC.c ../src/pingRPC_clnt.c
	gcc -o ../obj/rpcpingd ../src/servidorRPC.c ../src/pingRPC_svc.c

clean:
	rm ../obj/*

copy:
	if test -d /home/sod/2010/tp2/Zenobi-DIorio-Rossanigo;
		then echo "el path ya existe";
		else mkdir /home/sod/2010/tp2/Zenobi-DIorio-Rossanigo;
	fi
	cp -R -f -u ../* /home/sod/2010/tp2/Zenobi-DIorio-Rossanigo
```

Mediciones

Tamaño	RTT Promedio (uSeg)
1	85
10	72
100	86
300	83
500	204
1000	255
3000	2838
5000	2773
8500	3757



Descargos.

En las mediciones no pudimos utilizar paquetes de 10.000 caracteres porque RPC no podía codificar los argumentos. Este valor lo reemplazamos por 8.500 caracteres.

El error devuelto por RPC era: ***call failed: RPC: Can't encode arguments.***