Yousef Jarrar, Jose Perez

Professor Tong Yu

CSE 461

March 12, 2019

Points: 20/20

Lab 5: RPC

Part 1: Twenty Random Numbers using rpcgen

Running the command man rpcgen, we are given the documentation:

$man rpcgen

NAME

rpcgen - an RPC protocol compiler

SYNOPSIS

rpcgen infile

rpcgen [-Dname[=value]] [-T] [-K secs] infile

rpcgen -c|-h|-l|-m|-M|-t [-o outfile ] infile

rpcgen [-I] -s nettype [-o outfile] infile

rpcgen -n netid [-o outfile] infile

DESCRIPTION

rpcgen is a tool that generates C code to implement an RPC protocol. The input to rpcgen is a language similar to C known as RPC Language (Remote Procedure Call Language).

rpcgen is normally used as in the first synopsis where it takes an input file and generates up to four output files. If the infile is named proto.x, then rpcgen will generate a header file in proto.h, XDR routines in

proto\_xdr.c, server-side stubs in proto\_svc.c, and client-side stubs in proto\_clnt.c. With the -T option, it will also generate the RPC dispatch table in proto\_tbl.i. With the -Sc option, it will also generate sample code

which would illustrate how to use the remote procedures on the client side. This code would be created in proto\_client.c. With the -Ss option, it will also generate a sample server code which would illustrate how to write

the remote procedures. This code would be created in proto\_server.c.

The server created can be started both by the port monitors (for example, inetd or listen) or by itself. When it is started by a port monitor, it creates servers only for the transport for which the file descriptor 0 was

passed. The name of the transport must be specified by setting up the environmental variable PM\_TRANSPORT. When the server generated by rpcgen is executed, it creates server handles for all the transports specified in NET‐

PATH environment variable, or if it is unset, it creates server handles for all the visible transports from /etc/netconfig file. Note: the transports are chosen at run time and not at compile time.

When built for a port monitor (rpcgen -I), and that the server is self-started, it backgrounds itself by default. A special define symbol RPC\_SVC\_FG can be used to run the server process in foreground.

The second synopsis provides special features which allow for the creation of more sophisticated RPC servers. These features include support for user provided #defines and RPC dispatch tables. The entries in the RPC dis‐

patch table contain:

· pointers to the service routine corresponding to that procedure,

· a pointer to the input and output arguments

· the size of these routines

A server can use the dispatch table to check authorization and then to execute the service routine; a client library may use it to deal with the details of storage management and XDR data conversion.

The other three synopses shown above are used when one does not want to generate all the output files, but only a particular one. Some examples of their usage is described in the EXAMPLE section below. When rpcgen is exe‐

cuted with the -s option, it creates servers for that particular class of transports. When executed with the -n option, it creates a server for the transport specified by netid. If infile is not specified, rpcgen accepts

the standard input.

The C preprocessor, cc -E [see cc(1)], is run on the input file before it is actually interpreted by rpcgen. For each type of output file, rpcgen defines a special preprocessor symbol for use by the rpcgen programmer:

RPC\_HDR defined when compiling into header files

RPC\_XDR defined when compiling into XDR routines

RPC\_SVC defined when compiling into server-side stubs

RPC\_CLNT defined when compiling into client-side stubs

RPC\_TBL defined when compiling into RPC dispatch tables

Any line beginning with `%' is passed directly into the output file, uninterpreted by rpcgen.

For every data type referred to in infile, rpcgen assumes that there exists a routine with the string xdr\_ prepended to the name of the data type. If this routine does not exist in the RPC/XDR library, it must be provided.

Providing an undefined data type allows customization of XDR routines.

* With this newly installed package, we create the rand.x that was requested

rand.x

/\* rand.x \*/

program RAND\_PROG {

    version RAND\_VERS {

        void INITIALIZE\_RANDOM ( long ) = 1;

        double GET\_NEXT\_RANDOM ( void ) = 2;

    } = 1;

} = 0x30000000;

* Use $rpcgen -C -a rand.x to create the rest of the files.
* Use $make -f Makefile.rand to compile all the files to run the ./rand\_server and ./rand\_client with the host name added. It should not output anything as the rand\_server.c and rand\_client.c files do not have added in the code needed.

rand\_server.c (with added code)

#include "rand.h"

void \*

initialize\_random\_1\_svc(long \*argp, struct svc\_req \*rqstp)

{

    static char \* result;

    return (void \*) &result;

}

double \*

get\_next\_random\_1\_svc(void \*argp, struct svc\_req \*rqstp)

{

    static double result;

    result +=0.31;

    if(result >= 1.0)

        result -= 0.713;

    return &result;

}

rand\_client.c (with added code)

/\*

\* 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 "rand.h"

double

rand\_prog\_1(char \*host)

{

    CLIENT \*clnt;

    void \*result\_1;

    long initialize\_random\_1\_arg;

    double \*result\_2;

    char \*get\_next\_random\_1\_arg;

#ifndef DEBUG

    clnt = clnt\_create (host, RAND\_PROG, RAND\_VERS, "udp");

    if (clnt == NULL) {

        clnt\_pcreateerror (host);

        exit (1);

    }

#endif  /\* DEBUG \*/

/\*

\*  result\_1 = initialize\_random\_1(&initialize\_random\_1\_arg, clnt);

    if (result\_1 == (void \*) NULL) {

        clnt\_perror (clnt, "call failed");

    }

\*/

    result\_2 = get\_next\_random\_1((void\*)&get\_next\_random\_1\_arg, clnt);

    if (result\_2 == (double \*) NULL) {

        clnt\_perror (clnt, "call failed");

    }

#ifndef DEBUG

    clnt\_destroy (clnt);

#endif   /\* DEBUG \*/

    return \*result\_2;

}

int

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

{

    char \*host;

    if (argc < 2) {

        printf ("usage: %s server\_host\n", argv[0]);

        exit (1);

    }

    host = argv[1];

    //rand\_prog\_1 (host);

    double x;

    int i;

    printf("\n Twenty Random Numbers");

    for ( i = 0; i<20; ++i){

        x = rand\_prog\_1(host);

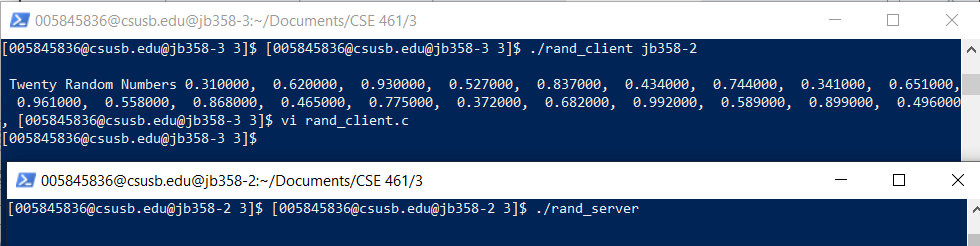
        printf(" %f\n, ", x);

    }

exit (0);

}

* $make -f Makefile.rand again and run both the server and client to get the 20 random numbers.



Part 2: Parallel Random Number Generator

rand.x (for part 2)

/\*rand.x\*/

struct params{

    int xleft;

    int xright;

};

program RAND\_PROG{

    version RAND\_VERS {

     int GET\_NEXT\_RANDOM ( params ) = 1; /\*Service #1\*/

    } = 1;

} = 0x30000000; /\* program # \*/

rand\_server.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 "rand.h"

int \*

get\_next\_random\_1\_svc(params \*argp, struct svc\_req \*rqstp)

{

    static int result;

    int xl, xr;

    xl = argp->xleft;

    xr = argp->xright;

    result = (11 \* xl + 13 \* result + 5 \* xr ) % 31;

    return &result;

}

rand\_client.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 <SDL/SDL.h>

#include <SDL/SDL\_thread.h>

#include "rand.h"

#define N 3 //Number of hosts/threads

char \*hosts[N]; //servers

SDL\_mutex \*mutex;

SDL\_cond \*barrierQueue; //Condition Variable

int count = 0;

int era = 0;

int x[N];

int rns[N][10];

int

rand\_prog\_1(char \*host, int xl, int xr)

{

    CLIENT \*clnt;

    int \*result\_1;

    params get\_next\_random\_1\_arg;

    get\_next\_random\_1\_arg.xleft = xl;

    get\_next\_random\_1\_arg.xright = xr;

//#ifndef   DEBUG

    clnt = clnt\_create (host, RAND\_PROG, RAND\_VERS, "udp");

    if (clnt == NULL) {

        clnt\_pcreateerror (host);

        exit (1);

    }

//#endif DEBUG

    result\_1 = get\_next\_random\_1(&get\_next\_random\_1\_arg, clnt);

    if (result\_1 == (int \*) NULL) {

        clnt\_perror (clnt, "call failed");

    }

//#ifndef   DEBUG

    clnt\_destroy (clnt);

//#endif DEBUG

    return \*result\_1;

}

void barrier(){

    int myEra; //a local variable

    SDL\_LockMutex ( mutex );

    count ++;

    if ( count < N ){

        myEra = era;

        while ( myEra == era )

            SDL\_CondWait ( barrierQueue, mutex );

    } else {

        count = 0; //reset the count

        era ++;

        SDL\_CondBroadcast ( barrierQueue ); // Signal all threads in queue

     }

    SDL\_UnlockMutex ( mutex );

}

int threads ( void \*data ){

    int k, i\_minus\_1, i\_plus\_1, id, xleft, xright;

    id = \*( (int \*) data );

    printf ("Thread %d", id);

    for ( k = 0; k < 10; k++ ){

     i\_minus\_1 = id - 1;

     if (i\_minus\_1 < 0 )

         i\_minus\_1 += N;

     xleft = x[i\_minus\_1];

     i\_plus\_1 = ( id + 1 ) %N;

     xright = x[i\_plus\_1];

     x[id] = rand\_prog\_1 (hosts[id], xleft, xright );

     printf ("(%d: %d )", id, x[id] );

     rns[id][k] = x[id];

     barrier();

    }

    return 0;

}

int

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

{

    int i, j;

    SDL\_Thread \*ids[N];

    //char \*host;

    if (argc < 4) {

        printf ("usage: %s server\_host1 host2 host3 ... \n", argv[0]);

        exit (1);

    }

    mutex = SDL\_CreateMutex();

    barrierQueue = SDL\_CreateCond();

    for ( i = 0; i < N; i++ )

        x[i] = rand() % 31; //Initial Values

    for ( i = 0; i < N; i++ ){

        hosts[i] = argv[i+1];

        ids[i] = SDL\_CreateThread ( threads, &i );

    }

    for ( i = 0; i < N; i++)

        SDL\_WaitThread ( ids[i], NULL );

    //print out results in buffer

    printf("\n Random Numbers: ");

    for ( i = 0; i < N; i++){

        printf("\n From Server %d:\n", i);

        for (j = 0; j < 10; ++j )

            printf("%d, ", rns[i][j] );

    }

    printf("\n");

    //host = argv[1];

    //rand\_prog\_1 (host);

exit (0);

}

Makefile.rand

# This is a template Makefile generated by rpcgen

# Parameters

CLIENT = rand\_client

SERVER = rand\_server

SOURCES\_CLNT.c =

SOURCES\_CLNT.h =

SOURCES\_SVC.c =

SOURCES\_SVC.h =

SOURCES.x = rand.x

TARGETS\_SVC.c = rand\_svc.c rand\_server.c rand\_xdr.c

TARGETS\_CLNT.c = rand\_clnt.c rand\_client.c rand\_xdr.c

TARGETS = rand.h rand\_xdr.c rand\_clnt.c rand\_svc.c rand\_client.c rand\_server.c

OBJECTS\_CLNT = $(SOURCES\_CLNT.c:%.c=%.o) $(TARGETS\_CLNT.c:%.c=%.o)

OBJECTS\_SVC = $(SOURCES\_SVC.c:%.c=%.o) $(TARGETS\_SVC.c:%.c=%.o)

# Compiler flags

CFLAGS += -g

LDLIBS += -lnsl -lSDL -lpthread -ltirpc

RPCGENFLAGS =

# Targets

all : $(CLIENT) $(SERVER)

$(TARGETS) : $(SOURCES.x)

rpcgen $(RPCGENFLAGS) $(SOURCES.x)

$(OBJECTS\_CLNT) : $(SOURCES\_CLNT.c) $(SOURCES\_CLNT.h) $(TARGETS\_CLNT.c)

$(OBJECTS\_SVC) : $(SOURCES\_SVC.c) $(SOURCES\_SVC.h) $(TARGETS\_SVC.c)

$(CLIENT) : $(OBJECTS\_CLNT)

$(LINK.c) -o $(CLIENT) $(OBJECTS\_CLNT) $(LDLIBS)

$(SERVER) : $(OBJECTS\_SVC)

$(LINK.c) -o $(SERVER) $(OBJECTS\_SVC) $(LDLIBS)

clean:

$(RM) core $(TARGETS) $(OBJECTS\_CLNT) $(OBJECTS\_SVC) $(CLIENT) $(SERVER)

Output:

