Taylor Pedretti, Jeremy Noble

Lab 5 Report

1. External Data Representation, abbreviated XDR, is a standard data serialization format used in computer network protocols so that computer systems of different types can communicate with each other. Convert EPC Source command generates C code from a given input file and writes code in the Remote Procedure Call (RPC) language that then can be compiled into a C program.

2. Finished. Copied the code and compiled with the appropriate flags.

3. -C is the switch to generate code in ANSI C and also code that can compiled using the C/C++ compiler. The -a switch generates all files including sample code for the client and server sides.

4. **make -f Makefile.rand** runs the make command on the .rand file that then complies our code from the temples that were created from the rand.x file and rpcgen command. The output from being compiled from the make file:

```
[005488635@csusb.edu@jb358-1 Step1-8]$ make -f Makefile.rand
cc -g -c -o rand_clnt.o rand_clnt.c
cc -g -c -o rand_client.o rand_client.c
cc -g       -o rand_client  rand_clnt.o rand_client.o -lnsl -ltirpc
cc -g -c -o rand_svc.o rand_svc.c
cc -g -c -o rand_server.o rand_server.c
cc -g       -o rand_server  rand_svc.o rand_server.o -lnsl -ltirpc
```

When ./rand_server and ./rand_client are ran nothing happens because only a shell was created no real functions were made from the template.

5. On this step we are adding code to our functions which have nothing within them to add some meat to them to make them do things when called. In double *get_next_random_1_svc(void *argp, struct svc_req *rqstp) function we're making it so it will always out up a number below 1.0, but these numbers will not be truly random. In the main function we're adding a for loop which calls the server 20 times asking for 20 random numbers.

6. **rpcinfo** shows us programs that are running with ports open waiting for things to connect to them. These could be from web servers or ssh servers waiting for connections to happen. Shows us which ports the program is running on and who the owner of the program is.

7. This step wasn't all that hard because the videos the professor had uploaded helped out a lot as long as you followed them you could get this step done. To get the extra credit all you had to change was the N from 3 to 4 and run one extra server on a different computer.

rand.h

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

#ifndef _RAND_H_RPCGEN
#define _RAND_H_RPCGEN

#include <rpc/rpc.h>


#ifdef __cplusplus
extern "C" {
#endif


struct params {
    int xleft;
    int xright;
};
typedef struct params params;

#define RAND_PROG 0x30000000
#define RAND_VERS 1

#if defined(__STDC__) || defined(__cplusplus)
#define GET_NEXT_RANDOM 1
extern  int * get_next_random_1(params *, CLIENT *);
extern  int * get_next_random_1_svc(params *, struct svc_req *);
extern int rand_prog_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define GET_NEXT_RANDOM 1
extern  int * get_next_random_1();
extern  int * get_next_random_1_svc();
extern int rand_prog_1_freeresult ();
#endif /* K&R C */

/* the xdr functions */

#if defined(__STDC__) || defined(__cplusplus)
extern  bool_t xdr_params (XDR *, params*);

#else /* K&R C */
extern bool_t xdr_params ();
#endif /* K&R C */

#ifdef __cplusplus
}
#endif

#endif /* !_RAND_H_RPCGEN */
```

rand_client.c

```
#include <SDL/SDL.h>
#include <SDL/SDL_thread.h>
#include "rand.h"

#define N 4
char *hosts[N];
SDL_mutex *mutex;
SDL_cond *barrierQueue;
int count = 0;
int era = 0;
int x[N];                      //The random number of server i
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;

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

    result_1 = get_next_random_1(&get_next_random_1_arg, clnt);
    if (result_1 == (int *) NULL) {
       clnt_perror (clnt, "call failed");
    }

    clnt_destroy (clnt);

    return *result_1;
```

```
}

void barrier()
{
    int myEra;                  //local variable
    SDL_LockMutex(mutex);

    count++;
    if(count < N)
    {
       myEra = era;

       while(myEra == era)
             SDL_CondWait(barrierQueue, mutex);
    }
    else
    {
       count = 0;               //reset count
       era++;
       SDL_CondBroadcast(barrierQueue);    //signal all threads in queue
    }

    SDL_UnlockMutex(mutex);

}

int threads(void *data)
{
    int k, i_minus_l, i_plus_l, id, xleft, xright;
    id = *((int *) data);
    printf("Thread %d", id);

    for(k = 0; k < 10; k++)
    {
       i_minus_l = id - 1;

       if(i_minus_l < 0)
             i_minus_l += N;

       xleft = x[i_minus_l];
       i_plus_l = (id+1)%N;
       xright = x[i_plus_l];
       x[id] = rand_prog_1(hosts[id], xleft, xright);
       printf("(%d: %d ) ", id, x[id]);
       rns[id][k] = x[id];
       barrier();
    }
```

```c
    return 0;
}

int main (int argc, char *argv[])
{
    int i, j;
    SDL_Thread *ids[N];

    if (argc < 5) {
       printf ("usage: %s server_host1 host2 hoist3 host4 ... \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 buffers
    printf("\nRandom Numbers:");
    for(i = 0; i < N; i++)
    {
       printf("\nFrom Server %d:\n", i);

       for(j = 0; j < 10; ++j)
             printf("%d, ", rns[i][j]);
    }

    printf("\n");
    exit (0);
}
```

rand_clnt.c

```c
#include <memory.h> /* for memset */
#include "rand.h"
```

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

int *
get_next_random_1(params *argp, CLIENT *clnt)
{
    static int clnt_res;

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

rand_server.c

```
#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;
    printf("%d\n", result);

    return &result;
}
```

rand_svc.c

```
#include "rand.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
```

```c
#include <netinet/in.h>

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

static void
rand_prog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
    union {
       params get_next_random_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 GET_NEXT_RANDOM:
       _xdr_argument = (xdrproc_t) xdr_params;
       _xdr_result = (xdrproc_t) xdr_int;
       local = (char *(*)(char *, struct svc_req *)) get_next_random_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;
}
```

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

    pmap_unset (RAND_PROG, RAND_VERS);

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
       fprintf (stderr, "%s", "cannot create udp service.");
       exit(1);
    }
    if (!svc_register(transp, RAND_PROG, RAND_VERS, rand_prog_1, IPPROTO_UDP)) {
       fprintf (stderr, "%s", "unable to register (RAND_PROG, RAND_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, RAND_PROG, RAND_VERS, rand_prog_1, IPPROTO_TCP)) {
       fprintf (stderr, "%s", "unable to register (RAND_PROG, RAND_VERS, tcp).");
       exit(1);
    }

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

rand_xdr.c

```c
#include "rand.h"

bool_t
xdr_params (XDR *xdrs, params *objp)
{
    register int32_t *buf;

     if (!xdr_int (xdrs, &objp->xleft))
        return FALSE;
     if (!xdr_int (xdrs, &objp->xright))
        return FALSE;
    return TRUE;
}
```