

RPC Programming

1.0 Prerequisites

- Some knowledge of C & Unix
- A network with at least two connected machines.

2.0 Objectives:

1. To understand the basic principles of RPC
2. To develop a program which returns the current date and time on a remote machine.
3. To further develop this program so it may call the server if available, on any remote machine.

3.0 Introduction

This worksheet is an introduction to programming using *Remote Procedure Calls*, (RPC). RPC particularly when using the protocol compiler *rpcgen*, is an easy way to begin network programming. Whilst very sophisticated services may be created using RPC (for example the NFS (network file system) service, to port a small program to use RPC to run part of itself on a remote machine is usually straightforward.

Lets consider a simple program than obtains, formats and prints the date & time:

```
#include <stdio.h>           //include headers
#include <time.h>

char *getdatestr_1(void)    //returns string
{                           // i.e. pointer to char

    static char buf[80];    // to hold string
    static char *cp;        // to hold pointer
    time_t secs;

    secs = time(NULL);      // get current time
    strcpy(buf,ctime(&secs)); // convert to string
    cp = buf;
    return(cp);            // return pointer to static buf
}

main()
{
    char *p;
```

```

    p = getdatestr_1();

    printf("%s", p);

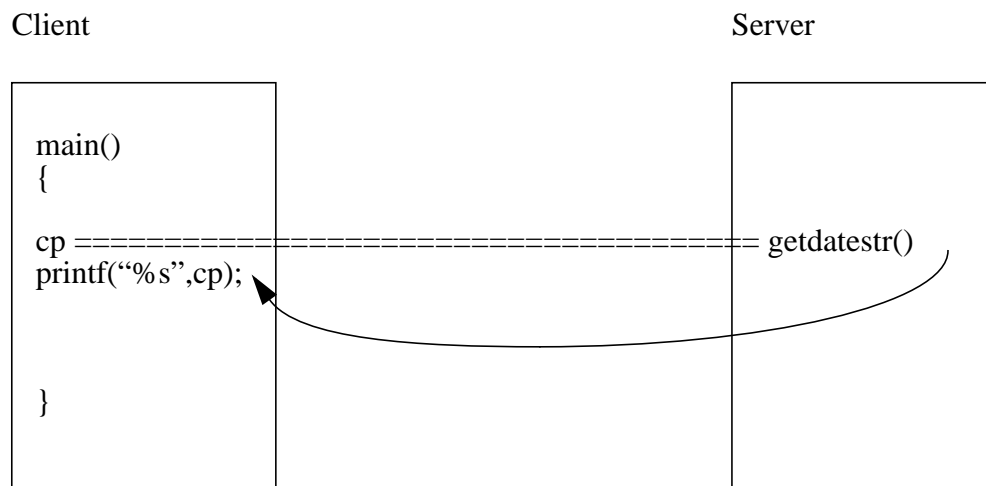
}

```

If you are familiar with C programming, you may notice that the data is stored in a static variable in `getdatestr()`. When we come to using RPC, variables we wish to communicate between computers will be REQUIRED to be static.

This simple program works on the local machine. If we want to execute part on one machine and obtain the result back, we can use RPC

:



4.0 Step 1. Define protocol.

In order to use RPC we first need to define the protocol that we are going to use to communicate. RPC protocol definitions have their own syntax, based on C. If you wish to go any significant way with RPC programming, you will need to learn this language. For sources of useful information, see the references at the end of this worksheet.

Our protocol definition:

```

/* remtime.x
   rpc protocol spec for remote time worksheet */

const MAXSTRLEN = 80;           /* max length of string */
typedef string datestr<MAXSTRLEN>; /* typedef for our ret. val.*/

program REMTIMEPROG
{

```

```

version REMTIMEEVERS
{
    datestr GETDATESTR() = 1;
} = 1;
} = 0x20650609;

```

Firstly, we have defined a type and a constant. After we have compiled the protocol, these will be available for use to use in our own code.

Next, we defined our program (server), version, and procedure numbers. These are all long ints and are used to register the service with the portmapper. Whilst version and procedure numbers are a free choice, program numbers must be chosen with care. Predefined services are listed in /etc/rpc, with protocol definitions for standard services in the directory /usr/include/rpcsvc. Have a look at the definition for nfs, how many procedures are available?

TABLE 1. Program Number Ranges

From (hex)	To (hex)	Use
0000 0000	1fff ffff	Defined by Sun for well known services. These should be common to all UNIX systems
2000 0000	3fff ffff	Defined by user. These should be used for unique local services, and development.
4000 0000	5fff ffff	Transient. This area is intended for programs which generate program numbers dynamically.
6000 0000	ffff ffff	Reserved for future use.

So for our service, we need to choose a number from 0x20000000. The example uses 0x20650609. Once the protocol has been compiled we can refer to this as REMTIME-PROG. Version numbers are to allow incompatible services to co-exist. If you are developing a service which evolves over time, different versions should have different version numbers. Procedure numbers are how the client & server keep track of what procedure should be called. Each function requires a unique number. Naming conventions recommend the procedure name to be in uppercase (in this case GETDATESTR), and the protocol compiler will generate stubs with the procedure number added; for the client to call getdatestr_1(), and for the server to provide getdate_1_svc().

RPC Protocol definitions are normally written in a file PROGRAMNAME.x

5.0 Step 2: Compile protocol

Compiling the protocol is easy:

```
rpcgen remtime.x
```

rpcgen is quite sophisticated however, so if you wish to restrict a service to TCP, alter default time-outs etc., you can do so with command line options. See the manual page for more details.

After compilation rpcgen will have generated 4 files. For our example these are:

1. remtime_xdr.c

eXternal Data Representation code. This should be linked into both client & server if needed. It provides the code to convert data structures into a neutral format for transmission, and then back to the native format for the machine. This is to reduce the problems you would get were say, one of your machines big-endian, and the other little-endian.

2. remtime_clnt.c

Client stub code. This handles interfacing with the XDR routines and provides the getdatestr() procedure ready for our client.

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

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

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

datestr *
getdatestr_1(void *argp, CLIENT *clnt)
{
    static datestr clnt_res;

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

3. remtime_svc.c

The server stubs. This is essentially the complete server without the code to perform the services we wish to offer. As a result, it is more complicated than the client stub. I've included it here for information. It's useful to understand what it does, but you can just treat it as a blackbox when programming at this level.

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

```

*/

#include "remtime.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>

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

static void
remtimeprog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
    union {
        int fill;
    } 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 GETDATESTR:
        _xdr_argument = (xdrproc_t) xdr_void;
        _xdr_result = (xdrproc_t) xdr_datestr;
        local = (char *(*)(char *, struct svc_req *))
        getdatestr_1_svc;
        break;

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

```

```

        exit (1);
    }
    return;
}

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

    pmap_unset (REMTIMEPROG, REMTIMEEVERS);

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
        fprintf (stderr, "cannot create udp service.");
        exit(1);
    }
    if (!svc_register(transp, REMTIMEPROG, REMTIMEEVERS, remtimeprog_1,
        IPPROTO_UDP)) {
        fprintf (stderr, "unable to register (REMTIMEPROG, REMTI-
        MEVERS, udp).");
        exit(1);
    }

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
        fprintf (stderr, "cannot create tcp service.");
        exit(1);
    }
    if (!svc_register(transp, REMTIMEPROG, REMTIMEEVERS, remtimeprog_1,
        IPPROTO_TCP)) {
        fprintf (stderr, "unable to register (REMTIMEPROG, REMTI-
        MEVERS, tcp).");
        exit(1);
    }

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

```

4. remtime.h

This is the header file to be included in the client & server. It includes prototypes for the services we have defined, and any constants, typedefs etc. It contains code for C++, ANSI C and K&R C. Only the ANSI C definitions are included here.

```

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

#ifdef _REMTIME_H_RPCGEN

```

```

#define _REMTIME_H_RPCGEN

#include <rpc/rpc.h>

#define MAXSTRLEN 80

typedef char *datestr;

#define REMTIMEPROG 0x20650609
#define REMTIMEEVERS 1

#if defined(__STDC__) || defined(__cplusplus)
#define GETDATESTR 1
extern datestr * getdatestr_1(void *, CLIENT *);
extern datestr * getdatestr_1_svc(void *, struct svc_req *);
extern int remtimeprog_1_freeresult (SVCXPRT *, xdrproc_t,
caddr_t);
#endif

/* the xdr functions */

#if defined(__STDC__) || defined(__cplusplus)
extern bool_t xdr_datestr (XDR *, datestr*);
#endif

#endif /* !_REMTIME_H_RPCGEN */

```

6.0 Step 3: Build the client

Our starting program was given on the first page. Firstly, we should remove the function we are going to run remotely (and save it for later incorporation into the server code. Secondly, we need to include `<rpc/rpc.h>` and our “remtime.h”. Thirdly, we need to add code to connect to the server, and modify our type definitions to match those we have defined in the protocol. The tricky part is remembering that all results are returned via pointers.

```

#include <stdio.h>
#include <rpc/rpc.h>
#include "remtime.h"

#define SERVER "localhost"

int main(void)
{
    CLIENT *client;          /* client handle - required */
    datestr *resstring;     /* pointer to a datestr to hold
                             result */

    client = clnt_create(SERVER, REMTIMEPROG, REMTIMEEVERS, "tcp");
    /* create a client handle to the specified server, program, version
       using the specified protocol */

```

```

if (client == NULL)
{
    printf("Cannot connect to server\n");
    clnt_pcreateerror(SERVER);
    /* find out why - prints to stdout */
    exit(1);
}
resstring = getdatestr_1(NULL, client);
/* as our function receives no arguments our first
argument is a pointer to void, the second a pointer
to the client handle we wish to use We should really
error check the return! */
printf("%s", *resstring);
return(0);
}

```

We can now compile our client. We require to compile and link the file above (remclient.c) which we've written, with the remtime_clnt.c and remtime_xdr.c produced by rpcgen.

```
cc -o client remtime_xdr.c remtime_clnt.c remclient.c
```

will produce an executable called client.

7.0 Step 4: Build the server

This step is easier than the client. What we need to do is define our service, (starting with our original function).

```

/* #include <stdio.h> */

#include <rpc/rpc.h>
#include <time.h>
#include "remtime.h"

datestr *getdatestr_1_svc(void *x, struct svc_req *y)
{
    static char buf[MAXSTRLEN];
    static char *cp;
    static datestr *dp;
    time_t secs;

    secs = time(NULL);
    strcpy(buf, ctime(&secs));
    cp = buf;
    dp = &cp;
    return(dp);
}

```

For our server, the above code when linked with remtime_svc.c and remtime_xdr.c provides a complete server. All we need to do is to define the functionality we wish to named

function-name_procedure-number_svc. This function will return a pointer to its result (so if our original example returned an int, the RPC version returns a pointer to int. Secondly the function will receive as an argument the struct svc_req. Unless we are engaged in advanced programming, we can basically ignore this structure (it provides info relating to the program/procedure etc. relating to the caller). Lastly, as we will return a pointer to our data, the data needs to be declared static.

8.0 To do.

1. Type in the protocol specification, compile it, build the client & server and test on local machine.
2. Modify the client source so if the remote procedure returns a null pointer (the error condition), clnt_perror is called to explain the error.
3. Modify the client source so that instead of running against localhost, client connects to another machine.
4. Modify the client so that the server can be specified on the command line.

9.0 Reference Sources

9.1 Books

Bloomer, John “*Power Programming with RPC*”, O’Reilly & Associates, 1992.

Sun Microsystems, “*Solaris 2.x: ONC+ Developers guide*”, chapters B, C & D.

Stevens, W. Richard, “*Unix Network Programming*”, Prentice Hall, 1990, chapter 18.

Peterson, L.L. & Davie, Bruce S., “*Computer Networks - A Systems Approach*”, Morgan Kaufmann, 2000.

9.2 URL’s

http://uw7doc.sco.com/SDK_netapi/CTOC-rpcpN.intro.html, Programming with RPC from SCO Unix.

http://advn2.gm.fh-koeln.de/doc_link/en_US/a_doc_lib/aixprggd/progcomc/toc.htm. AIX programming manual - see chapter 8.

<http://docs.linux.cz/programming/c/marshall/node34.html#ch:rpcgen>, Protocol compiling under Linux. A good reference source.