Sun Remote Procedure Call Mechanism

Originally developed by Sun, but now widely available on other platforms (including Digital Unix). Also known as Open Network Computing (ONC).

Sun RPC package has an RPC compiler (rpcgen) that automatically generates the client and server stubs.

RPC package uses XDR (eXternal Data Representation) to represent data sent between client and server stubs.

Has built-in representation for basic types (int, float, char).

Also provides a declarative language for specifying complex data types.

Remote Date Example

originally adapted from Stevens Unix Networking book.

Running *rpcgen date.x* generates `date.h`, `date_clnt.c` and `date_svc.c`. The header file is included with both client and server. The respective C source files are linked with client and server code.

```
/*
 * date.x  Specification of the remote date and time server
 */

/*
 * Define two procedures
 *      bin_date_1() returns the binary date and time (no arguments)
 *      str_date_1() takes a binary time and returns a string
 *
 */

program DATE_PROG {
    version DATE_VERS {
        long BIN_DATE(void) = 1;    /* procedure number = 1 */
        string STR_DATE(long) = 2;  /* procedure number = 2 */
    } = 1;                          /* version number = 1 */
} = 0x31234567;                     /* program number = 0x31234567 */
```

Notes:

- Start numbering proceedures at 1 (procedure 0 is always the "null procedure".
- Program number is defined by the user. Use range 0x20000000 to 0x3fffffff.
- Provide a prototype for each function. Sun RPC allows only a single parameter and a single result. Must use a structure for more parameters or return values (see XDRC++ example).
- use *clnt_create()* to get handle to remote procedure.
- do not have to use *rpcgen*. Can handcraft own routines.
- program number is a unique number chosen by the user.
- extra level of indirection on return values
- get a handle to remote server with `clnt_create()`. Also use this call to choose TCP or UDP.

- use of `static` is important in the server code so the storage is retained after the procedure return.

```
/*
 * rdate.c  client program for remote date program
 */

#include <stdio.h>
#include <rpc/rpc.h>    /* standard RPC include file */
#include "date.h"       /* this file is generated by rpcgen */

main(int argc, char *argv[])
{
    CLIENT *cl;         /* RPC handle */
    char *server;
    long *lresult;      /* return value from bin_date_1() */
    char **sresult;     /* return value from str_date_1() */

    if (argc != 2) {
        fprintf(stderr, "usage: %s hostname\n", argv[0]);
        exit(1);
    }

    server = argv[1];

    /*
     * Create client handle
     */

    if ((cl = clnt_create(server, DATE_PROG, DATE_VERS, "udp")) == NULL) {
        /*
         * can't establish connection with server
         */
         clnt_pcreateerror(server);
         exit(2);
    }

    /*
     * First call the remote procedure "bin_date".
     */

    if ( (lresult = bin_date_1(NULL, cl)) == NULL) {
        clnt_perror(cl, server);
        exit(3);
    }
    printf("time on host %s = %ld\n",server, *lresult);

    /*
```

```
     * Now call the remote procedure str_date
     */

    if ( (sresult = str_date_1(lresult, cl)) == NULL) {
        clnt_perror(cl, server);
        exit(4);
    }
    printf("time on host %s = %s", server, *sresult);

    clnt_destroy(cl);         /* done with the handle */
    exit(0);
}
```

```
/*
 * dateproc.c   remote procedures; called by server stub
 */

#include <time.h>
#include <rpc/rpc.h>        /* standard RPC include file */
#include "date.h"           /* this file is generated by rpcgen */

/*
 * Return the binary date and time
 */

 long *bin_date_1_svc(void *arg, struct svc_req *s)
 {
    static long timeval;    /* must be static */

    timeval = time((long *) 0);
    return(&timeval);
 }

/*
 * Convert a binary time and return a human readable string
 */

char **str_date_1_svc(long *bintime, struct svc_req *s)
{
    static char *ptr;       /* must be static */
    ptr = ctime((const time_t *)bintime);   /* convert to local time */
    return(&ptr);
}
```

Notes:

- No *main()* routine in server code. It is in the *_svc.c* code generated by *rpcgen*.
- Extra level of indirection. *bin_date()* does not return a long, but returns a pointer to a long. Have a pointer to a string in *str_date()*.
- The variable being returned needs to be declared `static` otherwise it will be deallocated on the stack when the function returns.

```
< /cs/cs4513/public/example/Date >ls

dateproc.c  date.x  rdate.c
< /cs/cs4513/public/example/Date >rpcgen date.x

< /cs/cs4513/public/example/Date >ls

date_clnt.c  date.h  dateproc.c  date_svc.c  date.x  rdate.c
< /cs/cs4513/public/example/Date >gcc -o rdate rdate.c date_clnt.c

< /cs/cs4513/public/example/Date >gcc -o dateproc dateproc.c date_svc.c

< /cs/cs4513/public/example/Date >dateproc&

[1] 15380
< /cs/cs4513/public/example/Date >rdate ccc3

rdate: [ccc3] Fri Nov 11 14:52:26 2005

< /cs/cs4513/public/example/Date >./rdate ccc3

time on host ccc3 = 1131738761
time on host ccc3 = Fri Nov 11 14:52:41 2005
< /cs/cs4513/public/example/Date >exit
```

What happens with these processes:

1. Server process creates a UDP socket and binds to a local port. It calls *svc_register()* routine to register its program number and version with the local *port mapper process*. This process should have been started as a daemon at system boot time. Server then waits for requests.
2. Client program contacts the port mapper on the designated machine using UDP. Gets port number for the server process.
3. Client program calls the client stub for the remote procedure (first *bin_date_1_svc()* and then *str_date_1_svc*. Defaults for how long to wait, how many times to retry, etc.

Remote Message Printing Example

Adaptation of the sample program in the *rpcgen* Programming Guide.

Also have a C++ version available.

```
/*
 * msg.x Remote message printing protocol
 */

program MESSAGEPROG {
    version MESSAGEVERS {
        int PRINTMESSAGE(string) = 1;
    } = 1;
} = 0x20000099;
```

```
/*
 * rprintmsg.c   remote printing version of "printmsg.c"
 */

#include <stdio.h>
#include <rpc/rpc.h>        /* always needed */
#include "msg.h"            /* msg.h wil be generated by rpcgen */

main(int argc, char *argv[])
{
    CLIENT *cl;
    int *result;
    char *server;
    char *message;

    if (argc != 3) {
        fprintf(stderr,
        "usage %s host message\n",argv[0]);
    }
    server = argv[1];
    message = argv[2];

    /* Creare client handle for calling MESSAGEPROG on the server
     * designated on the command line.  We tell the RPC package
     * to use the tcp protocol when contacting the server.
     */
    cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");
    if (cl == NULL) {
        /*
         * Couldn't establish connection with server.
         * Print error message and die.
         */
        clnt_pcreateerror(server);
        exit(1);
    }

    /*
     * Call the remote procedure "printmessage" on the server.
     */
    result = printmessage_1(&message, cl);
    if (result == NULL) {
        /*
         * An error occurred while calling the server.
         * Print error message and die
         */
```

```
        clnt_perror(cl, server);
        exit(1);
    }
    /*
     * Okay, we successfully called the remote procedure
     */
    if (*result == 0) {
        /*
         * Server was unable to print our message.
         * Print error message and die.
         */
        fprintf(stderr, "%s: %s couldn't print your message\n",
        argv[0], server);
        exit(1);
    }
    /*
     * Message got printed at server
     */
     printf("Message delivered to %s!\n", server);
     exit(0);
}
```

```
/*
 * msg_proc.c  implementation of the remote procedure call "printmessage"
 */

#include <stdio.h>
#include <rpc/rpc.h>        /* always needed */
#include "msg.h"            /* msg.h will be generated by rpcgen */

/*
 * Remote version of "printmessage"
 */

int * printmessage_1_svc(char **msg, struct svc_req *s)
{
    static int result;       /* must be static! */

    printf("%s\n", *msg);
    result = 1;
    return(&result);
}
```

```
< /cs/cs4513/public/example/Message >rpcgen msg.x

< /cs/cs4513/public/example/Message >ls

msg_clnt.c  msg.h  msg_proc.c  msg_svc.c  msg.x  rprintmsg.c
< /cs/cs4513/public/example/Message >gcc -o msg_server msg_proc.c msg_svc.c

< /cs/cs4513/public/example/Message >gcc -o rprintmsg rprintmsg.c msg_clnt.c

< /cs/cs4513/public/example/Message >./msg_server&

[1] 15474
< /cs/cs4513/public/example/Message >./rprintmsg ccc3 "Hello to ccc3"

Hello to ccc3
Message delivered to ccc3!
```

Note that the client can run on other machines of different types all sending a message to server.

XDR/C++ Example

```
/*
 * testxdr.x
 */

/*
 * Define a procedure
 *      str_test_1() takes a structure parameter and returns a string
 *
 */

struct testxdr{
    long long_arg;
    string string_arg < 128 >;
};

program TEST_PROG {
    version TEST_VERS {
    string STR_TEST(testxdr) = 1;  /* procedure number = 1 */
    } = 1;                          /* version number = 1 */
} = 0x31234567;                     /* program number = 0x31234567 */
```

So here is an example of how to pass multiple arguments. A long and a string in a single argument. Note that *rpcgen* will create another file `testxdr_xdr.c` that must be compiled and linked.

```
/* client.C  */

#include <iostream>
using namespace std;
#include <string.h>

#include <rpc/rpc.h>

extern "C" {
#include "testxdr.h"
}

main(int argc, char *argv[])

{
  CLIENT *c1;
  char *server;

  char **sresult;


  if (argc !=2){
    cerr <<"usage:" << argv[0] <<" hostname\n";
    exit(1);
  }

  server = argv[1];

  if ((c1 = clnt_create(server, TEST_PROG, TEST_VERS, "udp")) == NULL){
       clnt_pcreateerror(server);
    exit(2);
  }

  testxdr xdrmessage; //structure testxdr defined in testxdr.x

  long temp_long = 1;
  char *temp_str = "Client is testing";

  //initialize xdrmessage
  xdrmessage.long_arg = temp_long;
  xdrmessage.string_arg = temp_str;

  if ((sresult = str_test_1(&xdrmessage, c1)) == NULL){
       clnt_perror(c1, server);
       exit(4);
```

```
    }

 cout << "Client call server successfully\n ";
 cout << "Server send message back:\n " << server << " = " <<*sresult<<"\n";

 clnt_destroy(c1);
 exit(0);
}
```

```
//server.C   remote procedures; called by server stub

#include <iostream>
using namespace std;
#include <string.h>
#include <rpc/rpc.h>        //* standard RPC include file */

extern "C"{
#include "testxdr.h"
}                          //* this file is generated by rpcgen */


/*
 * Accept and print out client message and return a server string
 */

char **str_test_1_svc(testxdr *xdrm, struct svc_req *s)
{
    static char *ptr = "Server say Hi to client!"; /* must be static */

    cout << "Message from client: "<< xdrm->string_arg << "\n";

    return(&ptr);
}
```

```
< /cs/cs4513/public/example/XDRC++ >rpcgen testxdr.x

< /cs/cs4513/public/example/XDRC++ >ls

client.C  testxdr_clnt.c  testxdr_svc.c  testxdr_xdr.c
server.C  testxdr.h   testxdr.x
< /cs/cs4513/public/example/XDRC++ >gcc -c *.c

< /cs/cs4513/public/example/XDRC++ >g++ -o server server.C testxdr_xdr.o testxdr_svc.o

< /cs/cs4513/public/example/XDRC++ >g++ -o client client.C testxdr_xdr.o testxdr_clnt.o

< /cs/cs4513/public/example/XDRC++ >./server&
```

Sun RPC Summary

- Parameter passing: single argument passed and received (must use structures for more).
- Binding: done through port mapper daemon process. Must know remote server name.
- Transport protocol: Can use either UDP or TCP, Total size of arguments must be less than 8192 bytes for UDP. *rpcgen* defaults to UDP.
- Exception handling: If UDP is used and the client times out then it resends the request. If TCP is used and an error occurs then the request is not resent.
- Data representation: uses XDR.
- Security. Set `cl->cl_auth =` (3 basic types). :
  - `AUTH_NULL`. null authentification (default)
  - `AUTH_SYS`. Unix authentification causes following fields to be included with each RPC request: time stamp, name of the local host, Default used by NFS (network file system). client's effective user id, list of all client groups.
  - `AUTH_DES`. exchanges secure information using DES (data encryption standard) for standard. Also `AUTH_KERBEROS`.

More information in the way of Internet drafts:

- RPC: RFC1057
- XDR: RFC1014

Look at `www.ietf.org`.