Subsections

Remote Procedure Calls (RPC)

This chapter provides an overview of Remote Procedure Calls (RPC) RPC.

What Is RPC

RPC is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

RPC makes the client/server model of computing more powerful and easier to program. When combined with the ONC RPCGEN protocol compiler (Chapter 33) clients transparently make remote calls through a local procedure interface.

How RPC Works

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. Figure 32.1 shows the flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

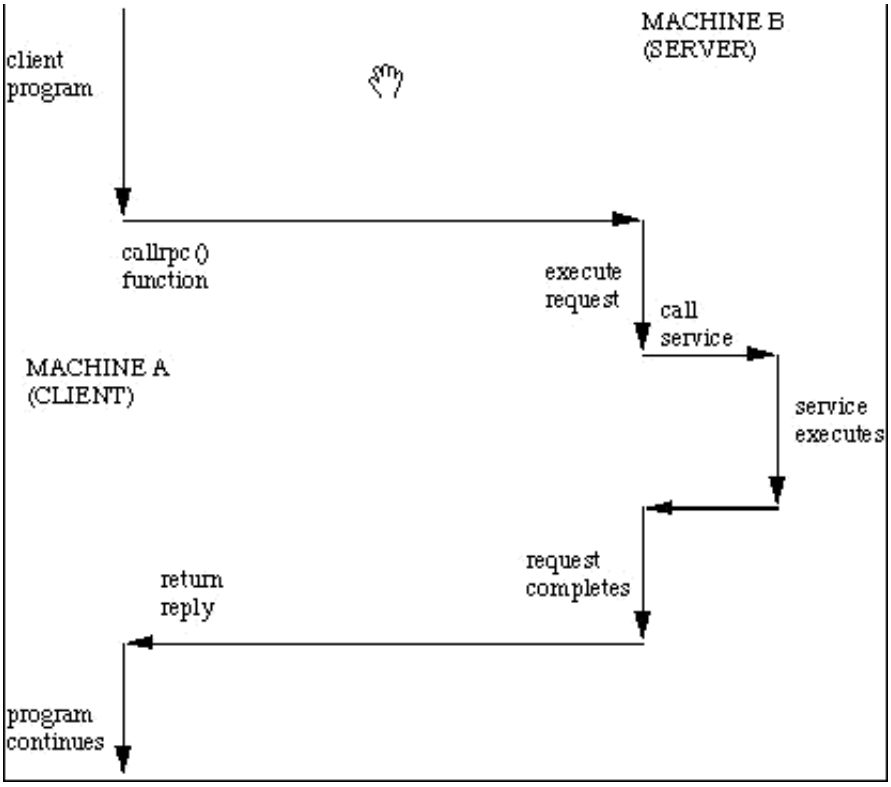

Fig. 32.1 Remote Procedure Calling Mechanism A remote procedure is uniquely identified by the triple: (program number, version number, procedure number) The program number identifies a group of related remote procedures, each of which has a unique procedure number. A program may consist of one or more versions. Each version consists of a collection of procedures which are available to be called remotely. Version numbers enable multiple versions of an RPC protocol to be available simultaneously. Each version contains a a number of procedures that can be called remotely. Each procedure has a procedure number.

RPC Application Development

Consider an example:

A client/server lookup in a personal database on a remote machine. Assuming that we cannot access the database from the local machine (via NFS).

We use UNIX to run a remote shell and execute the command this way. There are some problems with this method:

- the command may be slow to execute.
- You require an login account on the remote machine.

The RPC alternative is to

- establish an server on the remote machine that can repond to queries.
- Retrieve information by calling a query which will be quicker than previous approach.

To develop an RPC application the following steps are needed:

- Specify the protocol for client server communication
- Develop the client program
- Develop the server program

The programs will be compiled seperately. The communication protocol is achieved by generated stubs and these stubs and rpc (and other libraries) will need to be linked in.

Defining the Protocol

The easiest way to define and generate the protocol is to use a protocol complier such as rpcgen which we discuss is Chapter 33.

For the protocol you must identify the name of the service procedures, and data types of parameters and return arguments.

The protocol compiler reads a definitio and automatically generates client and server stubs.

rpcgen uses its own language (RPC language or RPCL) which looks very similar to preprocessor directives.

rpcgen exists as a standalone executable compiler that reads special files denoted by a .x prefix.

So to compile a RPCL file you simply do

rpcgen rpcprog.x

This will generate possibly four files:

- rpcprog_clnt.c -- the client stub
- rpcprog_svc.c -- the server stub
- rpcprog_xdr.c -- If necessary XDR (external data representation) filters
- rpcprog.h -- the header file needed for any XDR filters.

The external data representation (XDR) is an data abstraction needed for machine independent communication. The client and server need not be machines of the same type.

Defining Client and Server Application Code

We must now write the the client and application code. They must communicate via procedures and data types specified in the Protocol.

The service side will have to register the procedures that may be called by the client and receive and return any data required for processing.

The client application call the remote procedure pass any required data and will receive the retruned data.

There are several levels of application interfaces that may be used to develop RPC applications. We will briefly disuss these below before exapnading thhe most common of these in later chapters.

Compliling and running the application

Let us consider the full compilation model required to run a RPC application. Makefiles are useful for easing the burden of compiling RPC applications but it is necessary to understand the complete model before one can assemble a convenient makefile.

Assume the the client program is called rpcprog.c, the service program is rpcsvc.c and that the protocol has been defined in rpcprog.x and that rpcgen has been used to produce the stub and filter files: rpcprog_clnt.c, rpcprog_svc.c, rpcprog_xdr.c, rpcprog.h.

The client and server program must include (#include "rpcprog.h"

You must then:

- compile the client code:

  ```
  cc -c  rpcprog.c
  ```

- compile the client stub:

  ```
  cc -c  rpcprog_clnt.c
  ```

- compile the XDR filter:

  ```
  cc -c  rpcprog_xdr.c
  ```

- build the client executable:

  ```
  cc -o rpcprog rpcprog.o rpcprog_clnt.o rpcprog_xdr.c
  ```

- compile the service procedures:

  ```
  cc -c  rpcsvc.c
  ```

- compile the server stub:

  ```
  cc -c  rpcprog_svc.c
  ```

- build the server executable:

  ```
  cc -o rpcsvc rpcsvc.o rpcprog_svc.o rpcprog_xdr.c
  ```

Now simply run the programs rpcprog and rpcsvc on the client and server respectively. The server procedures must be registered before the client can call them.

Overview of Interface Routines

RPC has multiple levels of application interface to its services. These levels provide different degrees of control balanced with different amounts of interface code to implement. In order of increasing control and complexity. This section gives a summary of the routines available at each level. Simplified Interface Routines

The simplified interfaces are used to make remote procedure calls to routines on other machines, and specify only the type of transport to use. The routines at this level are used for most applications. Descriptions and code samples can be found in the section, Simplified Interface @ 3-2.

Simplified Level Routine Function

rpc_reg() -- Registers a procedure as an RPC program on all transports of the specified type.

rpc_call() -- Remote calls the specified procedure on the specified remote host.

rpc_broadcast() -- Broadcasts a call message across all transports of the specified type. Standard Interface Routines The standard interfaces are divided into top level, intermediate level, expert level, and bottom level. These interfaces give a developer much greater control over communication parameters such as the transport being used, how long to wait beforeresponding to errors and retransmitting requests, and so on.

Top Level Routines

At the top level, the interface is still simple, but the program has to create a client handle before making a call or create a server handle before receiving calls. If you want the application to run on all transports, use this interface. Use of these routines and code samples can be found in Top Level Interface

clnt_create() -- Generic client creation. The program tells clnt_create() where the server is located and the type of transport to use.

clnt_create_timed() Similar to clnt_create() but lets the programmer specify the maximum time allowed for each type of transport tried during the creation attempt.

svc_create() -- Creates server handles for all transports of the specified type. The program tells svc_create() which dispatch function to use.

clnt_call() -- Client calls a procedure to send a request to the server.

Intermediate Level Routines

The intermediate level interface of RPC lets you control details. Programs written at these lower levels are more complicated but run more efficiently. The intermediate level enables you to specify the transport to use.

clnt_tp_create() -- Creates a client handle for the specified transport.

clnt_tp_create_timed() -- Similar to clnt_tp_create() but lets the programmer specify the maximum time allowed. svc_tp_create() Creates a server handle for the specified transport.

clnt_call() -- Client calls a procedure to send a request to the server.

Expert Level Routines

The expert level contains a larger set of routines with which to specify transport-related parameters. Use of these routines

clnt_tli_create() -- Creates a client handle for the specified transport.

svc_tli_create() -- Creates a server handle for the specified transport.

rpcb_set() -- Calls rpcbind to set a map between an RPC service and a network address.

rpcb_unset() -- Deletes a mapping set by rpcb_set().

rpcb_getaddr() -- Calls rpcbind to get the transport addresses of specified RPC services.

svc_reg() -- Associates the specified program and version number pair with the specified dispatch routine.

svc_unreg() -- Deletes an association set by svc_reg().

clnt_call() -- Client calls a procedure to send a request to the server.

Bottom Level Routines

The bottom level contains routines used for full control of transport options.

clnt_dg_create() -- Creates an RPC client handle for the specified remote program, using a connectionless transport.

svc_dg_create() -- Creates an RPC server handle, using a connectionless transport.

clnt_vc_create() -- Creates an RPC client handle for the specified remote program, using a connection-oriented transport.

svc_vc_create() -- Creates an RPC server handle, using a connection-oriented transport.

clnt_call() -- Client calls a procedure to send a request to the server.

The Programmer's Interface to RPC

This section addresses the C interface to RPC and describes how to write network applications using RPC. For a complete specification of the routines in the RPC library, see the rpc and related man pages.

Simplified Interface

The simplified interface is the easiest level to use because it does not require the use of any other RPC routines. It also limits control of the underlying communications mechanisms. Program development at this level can be rapid, and is directly supported by the rpcgen compiler. For most applications, rpcgen and its facilities are sufficient. Some RPC services are not available as C functions, but they are available as RPC programs. The simplified interface library routines provide direct access to the RPC facilities for programs that do not require fine levels of control.

Routines such as rusers are in the RPC services library librpcsvc. rusers.c, below, is a program that displays the number of users on a remote host. It calls the RPC library routine, rusers.

The program.c program listing:

```
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <stdio.h>

/*
* a program that calls the
* rusers() service
*/

main(int argc,char **argv)

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

if ((num = rnusers(argv[1])) < 0) {
  fprintf(stderr, "error: rusers\n");
  exit(1);
 }

fprintf(stderr, "%d users on %s\n", num, argv[1] );
exit(0);
}
```

Compile the program with:

```
cc program.c -lrpcsvc -lnsl
```

The Client Side

There is just one function on the client side of the simplified interface rpc_call().

It has nine parameters:

```
int
rpc_call (char *host /* Name of server host */,
   u_long prognum /* Server program number */,
   u_long versnum /* Server version number */,
   xdrproc_t inproc /* XDR filter to encode arg */,
   char *in /* Pointer to argument */,
   xdr_proc_t outproc /* Filter to decode result */,
   char *out /* Address to store result */,
   char *nettype /* For transport selection */);
```

This function calls the procedure specified by prognum, versum, and procnum on the host. The argument to be passed to the remote procedure is pointed to by the in parameter, and inproc is the XDR filter to encode this argument. The out parameter is an address where the result from the remote procedure is to be placed. outproc is an XDR filter which will decode the result and place it at this address.

The client blocks on rpc_call() until it receives a reply from the server. If the server accepts, it returns RPC_SUCCESS with the value of zero. It will return a non-zero value if the call was unsuccessful. This value can be cast to the type clnt_stat, an enumerated type defined in the RPC include files (<rpc/rpc.h>) and interpreted by the clnt_sperrno() function. This function returns a pointer to a standard RPC error message corresponding to the error code. In the example, all "visible" transports listed in /etc/netconfig are tried. Adjusting the number of retries requires use of the lower levels of the RPC library. Multiple arguments and results are handled by collecting them in structures.

The example changed to use the simplified interface, looks like

```
#include <stdio.h>
#include <utmp.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

/* a program that calls the RUSERSPROG
* RPC program
*/

main(int argc, char **argv)

{
  unsigned long nusers;
  enum clnt_stat cs;
  if (argc != 2) {
    fprintf(stderr, "usage: rusers hostname\n");
    exit(1);
   }

  if( cs = rpc_call(argv[1], RUSERSPROG,
       RUSERSVERS, RUSERSPROC_NUM, xdr_void,
       (char *)0, xdr_u_long, (char *)&nusers,
       "visible") != RPC_SUCCESS ) {
          clnt_perrno(cs);
```

```
        exit(1);
      }

  fprintf(stderr, "%d users on %s\n", nusers, argv[1] );
  exit(0);
}
```

Since data types may be represented differently on different machines, rpc_call() needs both the type of, and a pointer to, the RPC argument (similarly for the result). For RUSERSPROC_NUM, the return value is an unsigned long, so the first return parameter of rpc_call() is xdr_u_long (which is for an unsigned long) and the second is &nusers (which points to unsigned long storage). Because RUSERSPROC_NUM has no argument, the XDR encoding function of rpc_call() is xdr_void() and its argument is NULL.

The Server Side

The server program using the simplified interface is very straightforward. It simply calls rpc_reg() to register the procedure to be called, and then it calls svc_run(), the RPC library's remote procedure dispatcher, to wait for requests to come in.

rpc_reg() has the following prototype:

```
rpc_reg(u_long prognum /* Server program number */,
    u_long versnum /* Server version number */,
    u_long procnum /* server procedure number */,
    char *procname /* Name of remote function */,
    xdrproc_t inproc /* Filter to encode arg */,
    xdrproc_t outproc /* Filter to decode result */,
    char *nettype /* For transport selection */);
```

svc_run() invokes service procedures in response to RPC call messages. The dispatcher in rpc_reg() takes care of decoding remote procedure arguments and encoding results, using the XDR filters specified when the remote procedure was registered. Some notes about the server program:

- Most RPC applications follow the naming convention of appending a _1 to the function name. The sequence _n is added to the procedure names to indicate the version number n of the service.
- The argument and result are passed as addresses. This is true for all functions that are called remotely. If you pass NULL as a result of a function, then no reply is sent to the client. It is assumed that there is no reply to send.
- The result must exist in static data space because its value is accessed after the actual procedure has exited. The RPC library function that builds the RPC reply message accesses the result and sends the value back to the client.
- Only a single argument is allowed. If there are multiple elements of data, they should be wrapped inside a structure which can then be passed as a single entity.
- The procedure is registered for each transport of the specified type. If the type parameter is (char *)NULL, the procedure is registered for all transports specified in NETPATH.

You can sometimes implement faster or more compact code than can rpcgen. rpcgen handles the generic code-generation cases. The following program is an example of a hand-coded registration routine. It registers a single procedure and enters svc_run() to

service requests.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

void *rusers();

main()
{
 if(rpc_reg(RUSERSPROG, RUSERSVERS,
      RUSERSPROC_NUM, rusers,
      xdr_void, xdr_u_long,
      "visible") == -1) {
        fprintf(stderr, "Couldn't Register\n");
         exit(1);
       }
  svc_run(); /* Never returns */
 fprintf(stderr, "Error: svc_run returned!\n");
 exit(1);
}
```

rpc_reg() can be called as many times as is needed to register different programs, versions, and procedures.

Passing Arbitrary Data Types

Data types passed to and received from remote procedures can be any of a set of predefined types, or can be programmer-defined types. RPC handles arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, by always converting them to a standard transfer format called external data representation (XDR) before sending them over the transport. The conversion from a machine representation to XDR is called serializing, and the reverse process is called deserializing. The translator arguments of rpc_call() and rpc_reg() can specify an XDR primitive procedure, like xdr_u_long(), or a programmer-supplied routine that processes a complete argument structure. Argument processing routines must take only two arguments: a pointer to the result and a pointer to the XDR handle.

The following XDR Primitive Routines are available:

```
xdr_int() xdr_netobj() xdr_u_long() xdr_enum()
xdr_long() xdr_float() xdr_u_int() xdr_bool()
xdr_short() xdr_double() xdr_u_short() xdr_wrapstring()
xdr_char() xdr_quadruple() xdr_u_char() xdr_void()
```

The nonprimitive xdr_string(), which takes more than two parameters, is called from xdr_wrapstring().

For an example of a programmer-supplied routine, the structure:

```
struct simple {
  int a;
  short b;
 } simple;
```

contains the calling arguments of a procedure. The XDR routine xdr_simple() translates the argument structure as shown below:

```
#include <rpc/rpc.h>
#include "simple.h"

bool_t xdr_simple(XDR *xdrsp, struct simple *simplep)

{
  if (!xdr_int(xdrsp, &simplep->a))
    return (FALSE);
  if (!xdr_short(xdrsp, &simplep->b))
    return (FALSE);
  return (TRUE);
}
```

An equivalent routine can be generated automatically by rpcgen (See Chapter 33).

An XDR routine returns nonzero (a C TRUE) if it completes successfully, and zero otherwise.

For more complex data structures use the XDR prefabricated routines:

```
xdr_array() xdr_bytes() xdr_reference()
xdr_vector() xdr_union() xdr_pointer()
xdr_string() xdr_opaque()
```

For example, to send a variable-sized array of integers, it is packaged in a structure containing the array and its length:

```
struct varintarr {
int *data;
int arrlnth;
} arr;
```

Translate the array with xdr_array(), as shown below:

```
bool_t xdr_varintarr(XDR *xdrsp, struct varintarr *arrp)

{
   return(xdr_array(xdrsp, (caddr_t)&arrp->data,
         (u_int *)&arrp->arrlnth, MAXLEN, sizeof(int), xdr_int));
}
```

The arguments of xdr_array() are the XDR handle, a pointer to the array, a pointer to the size of the array, the maximum array size, the size of each array element, and a pointer to the XDR routine to translate each array element. If the size of the array is known in advance, use xdr_vector() instread as is more efficient:

```
int intarr[SIZE];

bool_t xdr_intarr(XDR *xdrsp, int intarr[])
{
  return (xdr_vector(xdrsp, intarr, SIZE, sizeof(int), xdr_int));
}
```

XDR converts quantities to 4-byte multiples when serializing. For arrays of characters, each character occupies 32 bits. xdr_bytes() packs characters. It has four parameters similar to the first four parameters of xdr_array().

Null-terminated strings are translated by xdr_string(). It is like xdr_bytes() with no length parameter. On serializing it gets the string length from strlen(), and on deserializing it creates a null-terminated string.

xdr_reference() calls the built-in functions xdr_string() and xdr_reference(), which translates pointers to pass a string, and struct simple from the previous examples. An example use of xdr_reference() is as follows:

```
struct finalexample {
   char *string;
   struct simple *simplep;
  } finalexample;

bool_t xdr_finalexample(XDR *xdrsp, struct finalexample *finalp)

{  if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
       return (FALSE);
   if (!xdr_reference( xdrsp, &finalp->simplep, sizeof(struct simple), xdr_simple))
      return (FALSE);
   return (TRUE);
}
```

Note thatxdr_simple() could have been called here instead of xdr_reference() .

Developing High Level RPC Applications

Let us now introduce some further functions and see how we develop an application using high level RPC routines. We will do this by studying an example.

We will develop a remote directory reading utility.

Let us first consider how we would write a local directory reader. We have seem how to do this already in Chapter 19.

Consider the program to consist of two files:

- lls.c -- the main program which calls a routine in a local module read_dir.c

```
/*
 * ls.c: local directory listing main - before RPC
 */
#include <stdio.h>
#include <strings.h>
#include "rls.h"

main (int argc, char **argv)

{
      char    dir[DIR_SIZE];
```

```
    /* call the local procedure */
    strcpy(dir, argv[1]);    /* char dir[DIR_SIZE] is coming and going... */
    read_dir(dir);

    /* spew-out the results and bail out of here! */
    printf("%s\n", dir);

    exit(0);
}
```

- read_dir.c -- the file containing the ***local*** routine read_dir().

```
/* note - RPC compliant procedure calls take one input and
  return one output. Everything is passed by pointer.  Return
  values should point to static data, as it might have to
  survive some while. */
#include <stdio.h>
#include <sys/types.h>
#include <sys/dir.h>    /* use <xpg2include/sys/dirent.h> (SunOS4.1) or
      <sys/dirent.h> for X/Open Portability Guide, issue 2 conformance */
#include "rls.h"

read_dir(char    *dir)
  /* char dir[DIR_SIZE] */
{
    DIR * dirp;
    struct direct *d;
            printf("beginning ");

    /* open directory */
    dirp = opendir(dir);
    if (dirp == NULL)
        return(NULL);

    /* stuff filenames into dir buffer */
    dir[0] = NULL;
    while (d = readdir(dirp))
        sprintf(dir, "%s%s\n", dir, d->d_name);

    /* return the result */
         printf("returning ");
    closedir(dirp);
    return((int)dir);  /* this is the only new line from Example 4-3 */
}
```

- the header file rls.h contains only the following (for now at least)

```
#define DIR_SIZE 8192
```

  Clearly we need to share the size between the files. Later when we develop RPC versions more information will need to be added to this file.

This local program would be compiled as follows:

```
cc lls.c read_dir.c -o lls
```

Now we want to modify this program to work over a network: Allowing us to inspect directories of a remote server accross a network.

The following steps will be required:

- We will have to convert the read_dir.c, to run on the server.
  - We will have to register the server and the routine read_dir() on the server/.
- The client lls.c will have to call the routine as a remote procedure.
- We will have to define the protocol for communication between the client and the server programs.

Defining the protocol

We can can use simple NULL-terminated strings for passing and receivong the directory name and directory contents. Furthermore, we can embed the passing of these parameters directly in the client and server code.

We therefore need to specify the program, procedure and version numbers for client and servers. This can be done automatically using rpcgen or relying on prdefined macros in the simlified interface. Here we will specify them manually.

The server and client must agree ***ahead of time*** what logical adresses thney will use (The physical addresses do not matter they are hidden from the application developer)

Program numbers are defined in a standard way:

- 0*x*00000000 - 0*x*1*FFFFFFF*: Defined by Sun
- 0*x*20000000 - 0*x*3*FFFFFFF*: User Defined
- 0*x*40000000 - 0*x*5*FFFFFFF*: Transient
- 0*x*60000000 - 0*xFFFFFFFF*: Reserved

We will simply choose a ***user deifnined value*** for our program number. The version and procedure numbers are set according to standard practice.

We still have the DIR_SIZE definition required from the local version as the size of the directory buffer is rewquired by bith client and server programs.

Our new rls.h file contains:

```
#define DIR_SIZE 8192
#define DIRPROG ((u_long) 0x20000001)   /* server program (suite) number */
#define DIRVERS ((u_long) 1)    /* program version number */
#define READDIR ((u_long) 1)    /* procedure number for look-up */
```

Sharing the data

We have mentioned previously that we can pass the data a simple strings. We need to define an XDR filter routine xdr_dir() that shares the data. Recall that only one encoding and decoding argument can be handled. This is easy and defined via the standard xdr_string() routine.

The XDR file, rls_xrd.c, is as follows:

```
#include <rpc/rpc.h>

#include "rls.h"

bool_t xdr_dir(XDR *xdrs, char *objp)

{ return ( xdr_string(xdrs, &objp, DIR_SIZE) ); }
```

The Server Side

We can use the original read_dir.c file. All we need to do is register the procedure and start the server.

The procedure is registered with registerrpc() function. This is prototypes by:

```
registerrpc(u_long prognum /* Server program number */,
      u_long versnum /* Server version number */,
      u_long procnum /* server procedure number */,
      char *procname /* Name of remote function */,
      xdrproc_t inproc /* Filter to encode arg */,
      xdrproc_t outproc /* Filter to decode result */);
```

The parameters a similarly defined as in the rpc_reg simplified interface function. We have already discussed the setting of the parametere with the protocol rls.h header files and the rls_xrd.c XDR filter file.

The svc_run() routine has also been discussed previously.

The full rls_svc.c code is as follows:

```
#include <rpc/rpc.h>
#include "rls.h"

main()
{
      extern bool_t xdr_dir();
      extern char * read_dir();

      registerrpc(DIRPROG, DIRVERS, READDIR,
                  read_dir, xdr_dir, xdr_dir);

      svc_run();
}
```

The Client Side

At the client side we simply need to call the remote procedure. The function callrpc() does this. It is prototyped as follows:

```
callrpc(char *host /* Name of server host */,
   u_long prognum /* Server program number */,
   u_long versnum /* Server version number */,
```

```
  char *in /* Pointer to argument */,
  xdrproc_t inproc /* XDR filter to encode arg */,
  char *out /* Address to store result */
  xdr_proc_t outproc /* Filter to decode result */);
```

We call a local function read_dir() which uses callrpc() to call the remote procedure that has been registered READDIR at the server.

The full rls.c program is as follows:

```
/*
 * rls.c: remote directory listing client
 */
#include <stdio.h>
#include <strings.h>
#include <rpc/rpc.h>
#include "rls.h"

main (argc, argv)
int argc; char *argv[];
{
char   dir[DIR_SIZE];

      /* call the remote procedure if registered */
      strcpy(dir, argv[2]);
      read_dir(argv[1], dir); /* read_dir(host, directory) */

      /* spew-out the results and bail out of here! */
      printf("%s\n", dir);

      exit(0);
}

read_dir(host, dir)
char  *dir, *host;
{
      extern bool_t xdr_dir();
      enum clnt_stat clnt_stat;

      clnt_stat = callrpc ( host, DIRPROG, DIRVERS, READDIR,
                  xdr_dir, dir, xdr_dir, dir);
      if (clnt_stat != 0) clnt_perrno (clnt_stat);
}
```

Exercise

Exercise 12833

Compile and run the remote directory example rls.c ***etc***. Run both the client ande srever locally and if possible over a network.

Dave Marshall
1/5/1999