Subsections

Protocol Compiling and Lower Level RPC Programming

This chapter introduces the rpcgen tool and provides a tutorial with code examples and usage of the available compile-time flags. We also introduce some further RPC programming routines.

What is rpcgen

The rpcgen tool generates remote program interface modules. It compiles source code written in the RPC Language. RPC Language is similar in syntax and structure to C. rpcgen produces one or more C language source modules, which are then compiled by a C compiler.

The default output of rpcgen is:

- A header file of definitions common to the server and the client
- A set of XDR routines that translate each data type defined in the header file
- A stub program for the server
- A stub program for the client

rpcgen can optionally generate (although we ***do not*** consider these issues here -- see man pages or receommended reading):

- Various transports
- A time-out for servers
- Server stubs that are MT safe
- Server stubs that are not main programs
- C-style arguments passing ANSI C-compliant code
- An RPC dispatch table that checks authorizations and invokes service routines

rpcgen significantly reduces the development time that would otherwise be spent developing low-level routines. Handwritten routines link easily with the rpcgen output.

An rpcgen Tutorial

rpcgen provides programmers a simple and direct way to write distributed applications. Server procedures may be written in any language that observes procedure-calling conventions. They are linked with the server stub produced by rpcgen to form an executable server program. Client procedures are written and linked in the same way. This section presents some basic rpcgen programming examples. Refer also to the man rpcgen online manual page.

Converting Local Procedures to Remote Procedures

Assume that an application runs on a single computer and you want to convert it to run in a "distributed" manner on a network. This example shows the stepwise conversion of this program that writes a message to the system console.

Single Process Version of printmesg.c:

```
/* printmsg.c: print a message on the console */
#include <stdio.h>
main(int argc, char *argv[])

{
  char *message;
  if (argc != 2) {
    fprintf(stderr, "usage: %s <message>\n",argv[0]);
    exit(1);
   }
  message = argv[1];
  if (!printmessage(message)) {
   fprintf(stderr,"%s: couldn�t print your message\n",argv[0]);
   exit(1);
   }
   printf("Message Delivered!\n");
   exit(0);
}

/* Print a message to the console.
* Return a boolean indicating whether
* the message was actually printed. */

printmessage(char *msg)

{
  FILE *f;
  f = fopen("/dev/console", "w");
  if (f == (FILE *)NULL) {
   return (0);
   }
  fprintf(f, "%s\n", msg);
  fclose(f);
  return(1);
}
```

For local use on a single machine, this program could be compiled and executed as follows:

```
$ cc printmsg.c -o printmsg
$ printmsg "Hello, there."
Message delivered!
$
```

If the printmessage() function is turned into a ***remote procedure***, it can be called from anywhere in the network. rpcgen makes it easy to do this:

First, determine the data types of all procedure-calling arguments and the result argument. The calling argument of printmessage() is a string, and the result is an integer. We can write a protocol specification in RPC language that describes the remote version of printmessage. The RPC language source code for such a specification is:

```
/* msg.x: Remote msg printing protocol */
program MESSAGEPROG {
  version PRINTMESSAGEVERS {
    int PRINTMESSAGE(string) = 1;
  } = 1;
} = 0x20000001;
```

Remote procedures are always declared as part of remote programs. The code above declares an entire remote program that contains the single procedure PRINTMESSAGE.

In this example,

- PRINTMESSAGE procedure is declared to be:
  - the procedure 1,
  - in version 1 of the remote program
- MESSAGEPROG, with the program number 0x20000001.

Version numbers are incremented when functionality is changed in the remote program. Existing procedures can be changed or new ones can be added. More than one version of a remote program can be defined and a version can have more than one procedure defined.

Note: that the program and procedure names are declared with all capital letters. This is not required, but is a good convention to follow. Note also that the argument type is string and not char * as it would be in C. This is because a char * in C is ambiguous. char usually means an array of characters, but it could also represent a pointer to a single character. In RPC language, a null-terminated array of char is called a string.

There are just two more programs to write:

- The remote procedure itself

  Th RPC Version of printmsg.c:

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

#include <stdio.h>
#include "msg.h" /* msg.h generated by rpcgen */

int * printmessage_1(char **msg, struct svc_req *req)

{
  static int result; /* must be static! */
  FILE *f;

  f = fopen("/dev/console", "w");
  if (f == (FILE *)NULL) {
   result = 0;
   return (&result);
   }
  fprintf(f, "%s\n", *msg);
  fclose(f);
  result = 1;
  return (&result);
}
```

Note that the declaration of the remote procedure printmessage_1 differs from that of the local procedure printmessage in four ways:

- It takes a pointer to the character array instead of the pointer itself. This is true of all remote procedures when the '-' N option is not used: They always take pointers to their arguments rather than the arguments themselves. Without the '-' N option, remote procedures are always called with a single argument. If more than one argument is required the arguments must be passed in a struct.
- It is called with two arguments. The second argument contains information on the context of an invocation: the program, version, and procedure numbers, raw and canonical credentials, and an SVCXPRT structure pointer (the SVCXPRT structure contains transport information). This information is made available in case the invoked procedure requires it to perform the request.
- It returns a pointer to an integer instead of the integer itself. This is also true of remote procedures when the '-' N option is not used: They return pointers to the result. The result should be declared static unless the '-' M (multithread) or '-' A (Auto mode) options are used. Ordinarily, if the result is declared local to the remote procedure, references to it by the server stub are invalid after the remote procedure returns. In the case of '-' M and '-' A options, a pointer to the result is passed as a third argument to the procedure, so the result is not declared in the procedure.
- An _1 is appended to its name. In general, all remote procedures calls generated by rpcgen are named as follows: the procedure name in the program definition (here PRINTMESSAGE) is converted to all lowercase letters, an underbar (_) is appended to it, and the version number (here 1) is appended. This naming scheme allows multiple versions of the same

procedure.

- The main client program that calls it:

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

#include <stdio.h>
#include "msg.h" /* msg.h generated by rpcgen */

main(int argc, char **argv)

{
 CLIENT *clnt;
 int *result;
 char *server;
 char *message;

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

 server = argv[1];
 message = argv[2];

 /*
  * Create client "handle" used for
  * calling MESSAGEPROG on the server
  * designated on the command line.
  */

 clnt = clnt_create(server, MESSAGEPROG, PRINTMESSAGEVERS, "visible");

 if (clnt == (CLIENT *)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, clnt);
  if (result == (int *)NULL) {
    /*
     * An error occurred while calling
     * the server.
     * Print error message and die.
     */

    clnt_perror(clnt, 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: could not print your message\n",argv[0]);
   exit(1);
   }

  /* The message got printed on the
   * server's console
   */

   printf("Message delivered to %s\n", server);
   clnt_destroy( clnt );
   exit(0);
}
```

Note the following about Client Program to Call printmsg.c:

- First, a client handle is created by the RPC library routine clnt_create(). This client handle is passed to the stub routine that calls the remote procedure. If no more calls are to be made using the client handle, destroy it with a call to clnt_destroy() to conserve system resources.
- The last parameter to clnt_create() is visible, which specifies that any transport noted as visible in /etc/netconfig can be used.
- The remote procedure printmessage_1 is called exactly the same way as it is declared in msg_proc.c, except for the inserted client handle as the second argument. It also returns a pointer to the result instead of the result.
- The remote procedure call can fail in two ways. The RPC mechanism can fail or there can be an error in the execution of the remote procedure. In the former case, the remote procedure printmessage_1 returns a NULL. In the latter case, the error reporting is application dependent. Here, the error is returned through *result.

To compile the remote rprintmsg example:

- compile the protocol defined in msg.x: rpcgen msg.x.

  This generates the header files (msg.h), client stub (msg_clnt.c), and server stub (msg_svc.c).

- compile the client executable:

```
cc rprintmsg.c msg_clnt.c -o rprintmsg -lnsl
```

- compile the server executable:

```
cc msg_proc.c msg_svc.c -o msg_server -lnsl
```

The C object files must be linked with the library libnsl, which contains all of the networking functions, including those for RPC and XDR.

In this example, no XDR routines were generated because the application uses only the basic types that are included in libnsl . Let us consider further what rpcgen did with the input file msg.x:

- It created a header file called msg.h that contained #define statements for MESSAGEPROG, MESSAGEVERS, and PRINTMESSAGE for use in the other modules. This file**must** be included by both the client and server modules.
- It created the client stub routines in the msg_clnt.c file. Here there is only one, the printmessage_1 routine, that was called from the rprintmsg client program. If the name of an rpcgen input file is prog.x, the client stub's output file is called prog_clnt.c.
- It created the server program in msg_svc.c that calls printmessage_1 from msg_proc.c. The rule for naming the server output file is similar to that of the client: for an input file called prog.x, the output server file is named prog_svc.c.

Once created, the server program is installed on a remote machine and run. (If the machines are homogeneous, the server binary can just be copied. If they are not, the server source files must be copied to and compiled on the remote machine.)

Passing Complex Data Structures

rpcgen can also be used to generate XDR routines -- the routines that convert local data structures into XDR format and vice versa.

let us consider dir.x a remote directory listing service, built using rpcgen both to generate stub routines and to generate the XDR routines.

The RPC Protocol Description File: dir.x is as follows:

```
/*
* dir.x: Remote directory listing protocol
*
* This example demonstrates the functions of rpcgen.
*/

const MAXNAMELEN = 255; /* max length of directory entry */

typedef string nametype<MAXNAMELEN>; /* director entry */
typedef struct namenode *namelist; /* link in the listing */

/* A node in the directory listing */

struct namenode {
  nametype name; /* name of directory entry */
  namelist next; /* next entry */
 };

/*
* The result of a READDIR operation
*
* a truly portable application would use
* an agreed upon list of error codes
* rather than (as this sample program
* does) rely upon passing UNIX errno's
* back.
*
* In this example: The union is used
* here to discriminate between successful
* and unsuccessful remote calls.
*/

union readdir_res switch (int errno) {
  case 0:
    namelist list; /* no error: return directory listing */
  default:
    void; /* error occurred: nothing else to return */
  };

/* The directory program definition */

program DIRPROG {
  version DIRVERS {
   readdir_res
   READDIR(nametype) = 1;
  } = 1;
} = 0x20000076;
```

You can redefine types (like readdir_res in the example above) using the struct, union, and enum RPC language keywords. These keywords are not used in later declarations of variables of those types. For example, if you define a union, my_un, you declare using only my_un, and not union my_un. rpcgen compiles RPC unions into C structures. Do not declare C unions using the union keyword.

Running rpcgen on dir.x generates four output files:

- the header file, dir.h,
- the client stub, dir_clnt.c,
- the server skeleton, dir_svc.c ,and
- the XDR routines in the file dir_xdr.c.

This last file contains the XDR routines to convert declared data types from the host platform representation into XDR format, and vice versa. For each RPCL data type used in the .x file, rpcgen assumes that libnsl contains a routine whose name is the name of the data type, prepended by the XDR routine header xdr_ (for example, xdr_int). If a data type is defined in the .x file, rpcgen generates the required xdr_ routine. If there is no data type definition in the .x source file (for example, msg.x, above), then no _xdr.c file is generated. You can write a .x source file that uses a data type not supported by libnsl, and deliberately omit defining the type (in the .x file). In doing so, you must provide the xdr_ routine. This is a way to provide your own customized xdr_ routines.

The server-side of the READDIR procedure, dir_proc.c is shown below:

```
/*
* dir_proc.c: remote readdir
* implementation
*/

#include <dirent.h>
#include "dir.h" /* Created by rpcgen */

extern int errno;

extern char *malloc();
extern char *strdup();

readdir_res *
readdir_1(nametype *dirname, struct svc_req *req)

{
 DIR *dirp;
 struct dirent *d;
 namelist nl;
 namelist *nlp;

 static readdir_res res; /* must be static! */

 /* Open directory */
 dirp = opendir(*dirname);

 if (dirp == (DIR *)NULL) {
   res.errno = errno;
  return (&res);
  }

 /* Free previous result */
 xdr_free(xdr_readdir_res, &res);

 /*
  * Collect directory entries.
  * Memory allocated here is free by
  * xdr_free the next time readdir_1
  * is called
  */

  nlp = &res.readdir_res_u.list;
  while (d = readdir(dirp)) {
   nl = *nlp = (namenode *)
   malloc(sizeof(namenode));
   if (nl == (namenode *) NULL) {
      res.errno = EAGAIN;
      closedir(dirp);
      return(&res);
     }
   nl->name = strdup(d->d_name);
   nlp = &nl->next;
  }

  *nlp = (namelist)NULL;

  /* Return the result */
  res.errno = 0;
  closedir(dirp);
  return (&res);
}
```

The Client-side Implementation of implementation of the READDIR procedure, rls.c is given below:

```
/*
* rls.c: Remote directory listing client
*/

#include <stdio.h>
#include "dir.h" /* generated by rpcgen */

extern int errno;

main(int argc, char *argv[])


{
  CLIENT *clnt;
  char *server;
  char *dir;
  readdir_res *result;
  namelist nl;

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

  server = argv[1];
  dir = argv[2];

  /*
   * Create client "handle" used for
   * calling MESSAGEPROG on the server
   * designated on the command line.
   */

  cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");

  if (clnt == (CLIENT *)NULL) {
    clnt_pcreateerror(server);
    exit(1);
   }

  result = readdir_1(&dir, clnt);

  if (result == (readdir_res *)NULL) {
    clnt_perror(clnt, server);
    exit(1);
  }

  /* Okay, we successfully called
   * the remote procedure.
   */

  if (result->errno != 0) {
   /* Remote system error. Print
    * error message and die.
    */

   errno = result->errno;
   perror(dir);
   exit(1);
  }

  /* Successfully got a directory listing.
   * Print it.
   */

  for (nl = result->readdir_res_u.list;
      nl != NULL;
      nl = nl->next) {
        printf("%s\n", nl->name);
     }

  xdr_free(xdr_readdir_res, result);
  clnt_destroy(cl);
  exit(0);
}
```

As in other examples, execution is on systems named local and remote. The files are compiled and run as follows:

```
remote$ rpcgen dir.x
remote$ cc -c dir_xdr.c
remote$ cc rls.c dir_clnt.c dir_xdr.o -o rls -lnsl
remote$ cc dir_svc.c dir_proc.c dir_xdr.o -o dir_svc -lnsl
remote$ dir_svc
```

When you install rls on system local, you can list the contents of /usr/share/lib on system remote as follows:

```
local$ rls remote /usr/share/lib
ascii
eqnchar
greek
kbd
```

```
marg8
tabclr
tabs
tabs4
local$
```

rpcgen generated client code does not release the memory allocated for the results of the RPC call. Call xdr_free() to release the memory when you are finished with it. It is similar to calling the free() routine, except that you pass the XDR routine for the result. In this example, after printing the list, xdr_free(xdr_readdir_res, result); was called.

Note - Use xdr_free() to release memory allocated by malloc(). Failure to use xdr_free to() release memory results in memory leaks.

Preprocessing Directives

rpcgen supports C and other preprocessing features. C preprocessing is performed on rpcgen input files before they are compiled. All standard C preprocessing directives are allowed in the .x source files. Depending on the type of output file being generated, five symbols are defined by rpcgen. rpcgen provides an additional preprocessing feature: any line that begins with a percent sign (%) is passed directly to the output file, with no action on the line's content. Caution is required because rpcgen does not always place the lines where you intend. Check the output source file and, if needed, edit it.

The following symbols may be used to process file specific output:

RPC_HDR
-- Header file output
RPC_XDR
-- XDR routine output
RPC_SVC
-- Server stub output
RPC_CLNT
-- Client stub output
RPC_TB
-- Index table output

The following example illustrates tthe use of rpcgen�s pre-processing features.

```
/*
* time.x: Remote time protocol
*/
program TIMEPROG {
  version TIMEVERS {
    unsigned int TIMEGET() = 1;
   } = 1;
} = 0x20000044;

#ifdef RPC_SVC
%int *
%timeget_1()
%{
```

```
% static int thetime;
%
% thetime = time(0);
% return (&thetime);
%}
#endif
```

cpp Directives

rpcgen supports C preprocessing features. rpcgen defaults to use /usr/ccs/lib/cpp as the C preprocessor. If that fails, rpcgen tries to use /lib/cpp. You may specify a library containing a different cpp to rpcgen with the '-' Y flag.

For example, if /usr/local/bin/cpp exists, you can specify it to rpcgen as follows:

```
rpcgen -Y /usr/local/bin test.x
```

Compile-Time Flags

This section describes the rpcgen options available at compile time. The following table summarizes the options which are discussed in this section.

Option	Flag	Comments
C-style	'-' N	Also called Newstyle mode
ANSI C	'-' C	Often used with the -N option
MT-Safe code	'-' M	For use in multithreaded environments
MT Auto mode	'-' A	-A also turns on -M option
TS-RPC library '	-' b	TI-RPC library is default
xdr_inline count	'-' i	Uses 5 packed elements as default,
		but other number may be specified

Client and Server Templates

rpcgen generates sample code for the client and server sides. Use these options to generate the desired templates.

Flag	Function
'-' a	Generate all template files
'-' Sc	Generate client-side template
'-' Ss	Generate server-side template
'-' Sm	Generate makefile template

The files can be used as guides or by filling in the missing parts. These files are in addition to the stubs generated.

Example rpcgen compile options/templates

A C-style mode server template is generated from the add.x source by the command:

```
rpcgen -N -Ss -o add_server_template.c add.x
```

The result is stored in the file add_server_template.c.

A C-style mode, client template for the same add.x source is generated with the command line:

```
rpcgen -N -Sc -o add_client_template.c add.x
```

The result is stored in the file add_client_template.c.

A make file template for the same add.x source is generated with the command line:

```
rpcgen -N -Sm -o mkfile_template add.x
```

The result is stored in the file mkfile_template. It can be used to compile the client and the server. If the '-' a flag is used as follows:

```
rpcgen -N -a add.x
```

rpcgen generates all three template files. The client template goes into add_client.c, the server template to add_server.c, and the makefile template to makefile.a. If any of these files already exists, rpcgen displays an error message and exits.

Note - When you generate template files, give them new names to avoid the files being overwritten the next time rpcgen is executed.

Recommended Reading

The book ***Power Programming with RPC*** by John Bloomer, O'Reilly and Associates, 1992, is the most comprehensive on the topic and is essential reading for further RPC programming.

Exercises

Exercise 12834

Use rpcgen the generate and compile the rprintmsg listing example given in this chapter.

Exercise 12835

Use rpcgen the generate and compile the dir listing example given in this chapter.

Exercise 12836

Develop a Remote Procedure Call suite of programs that enables a user to search for specific files or filtererd files in a remote directory. That is to say you can search for a named file ***e.g. file.c*** or all files named `*.c` or even `*.x`.

Exercise 12837

Develop a Remote Procedure Call suite of programs that enables a user to `grep` files remotely. You may use code developed previously or unix system calls to implement `grep`.

Exercise 12838

Develop a Remote Procedure Call suite of programs that enables a user to ***list*** the contents of a named remote files.

Dave Marshall
1/5/1999