rpcgen Programming Guide 3

This chapter introduces the rpcgen tool and provides a tutorial with code examples and usage of the available compile-time flags. See “Glossary” on page 331 for the definition of the terms used in this chapter.

SunOS 5.x Features	*page 22*
An rpcgen Tutorial	*page 23*
Compile-Time Flags	*page 37*
rpcgen Programming Techniques	*page 50*

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:

- 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. (For a discussion of RPC programming without `rpcgen`, see Chapter 4, "The Programmer's Interface to RPC.")

SunOS 5.x Features

This section lists the features found in the SunOS 5.x `rpcgen` code generator that are not found in the SunOS 4.x version.

Template Generation

`rpcgen` generates client-side, server-side, and makefile templates. See "Client and Server Templates" on page 38 for the list of options.

C-style Mode

`rpcgen` has two compilation modes, C-style and default. C-style mode lets arguments be passed by value, instead of as pointers to a structure. It also supports passing multiple arguments. The default mode is the same as in previous releases. See "C-style Mode" on page 39 for the example code for both modes.

Multithread-Safe Code

`rpcgen` can now generate MT-safe code for use in a threaded environment. By default, the code generated by `rpcgen` is not MT-safe. See "MT-Safe Code" on page 42 for the description and example code.

Multithread Auto Mode

`rpcgen` can generate MT-safe server stubs that operate in the MT Auto mode. See "MT Auto Mode" on page 48 for the definition and example code.

Library Selection

`rpcgen` can use library calls for either TS-RPC or TI-RPC. See "TI-RPC or TS-RPC Library Selection" on page 49.

ANSI C -compliant Code

The output generated by `rpcgen` conforms to ANSI C standards. The code can also be used in the SPARCompiler™ C++ 3.0 environment. See "ANSI C-compliant Code" on page 49.

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 `rpcgen(1)` manpage.

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. Code Example 3-1 shows the original program.

Code Example 3-1 Single Process Version of `printmesg.c`

```
/* printmsg.c: print a message on the console */

#include <stdio.h>

main(argc, argv)
    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(msg)
    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 message 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, the `PRINTMESSAGE` procedure is declared to be procedure `1`, in version `1` of the remote program `MESSAGEPROG`, with the program number `0x20000001`. (See Appendix B, "RPC Protocol and Language Specification" for guidance on choosing program numbers.) 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
- The main client program that calls it

Code Example 3-2 is a remote procedure that implements the `PRINTMESSAGE` procedure in Code Example 3-1.

Code Example 3-2 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(msg, req)
```

```
    char **msg;
    struct svc_req *req;          /* details of call */
{
    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:

1. 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`.

2. 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.

3. 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.

4. 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.

Code Example 3-3 shows the main client program that calls the remote procedure.

Code Example 3-3 Client Program to Call printmsg.c

```
/*
 * rprintmsg.c: remote version of "printmsg.c"
 */
#include <stdio.h>
#include "msg.h"               /* msg.h generated by rpcgen */

main(argc, argv)
    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 Code Example 3-3:

1. 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. (The client handle can be created in other ways as well. See Chapter 4, "The Programmer's Interface to RPC" for details.) If no more calls are to be made using the client handle, destroy it with a call to `clnt_destroy()` to conserve system resources.

2. The last parameter to `clnt_create()` is `visible`, which specifies that any transport noted as visible in `/etc/netconfig` can be used. For further information on this, see the `/etc/netconfig` file and its description in *Transport Interfaces Programming Guide*.

3. 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.

4. 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`.

Here are the compile commands for the `printmsg` example:

```
$   rpcgen msg.x
$   cc rprintmsg.c msg_clnt.c -o rprintmsg -lnsl
$   cc msg_proc.c msg_svc.c -o msg_server -lnsl
```

First, `rpcgen` was used to generate the header files (`msg.h`), client stub (`msg_clnt.c`), and server stub (`msg_svc.c`). Then, two programs are compiled: the client program `rprintmsg` and the server program `msg_server`. 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`.

Here is what `rpcgen` did with the input file `msg.x`:

1. 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.

2. 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 `FOO.x`, the client stub's output file is called `FOO_clnt.c`.

3. 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 `FOO.x`, the output server file is named `FOO_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.) For this example, the remote machine is called `remote` and the local machine is called `local`. The server is started from the shell on the remote system:

```
remote$ msg_server
```

Server processes generated with `rpcgen` always run in the background. It is not necessary to follow the server's invocation with an ampersand (`&`). Servers generated by `rpcgen` can also be invoked by port monitors like `listen`() and `inetd`(), instead of from the command line.

Thereafter, a user on `local` can print a message on the console of machine `remote` as follows:

```
local$ rprintmsg remote "Hello, there."
```

Using `rprintmsg`, a user can print a message on any system console (including the `local` console) when the server `msg_server` is running on the target system.

Passing Complex Data Structures

"Converting Local Procedures to Remote Procedures" on page 23 shows how to generate client and server RPC code. `rpcgen` can also be used to generate XDR routines (the routines that convert local data structures into XDR format and vice versa).

Code Example 3-4 presents a complete RPC service: a remote directory listing service, built using `rpcgen` both to generate stub routines and to generate the XDR routines.

Code Example 3-4 RPC Protocol Description File: `dir.x`

```
/*
 *  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>;          /* directory 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, `foo`, you declare using only `foo`, and not `union foo`.

`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: (1) the header file, (2) the client stub, (3) the server skeleton, and (4) 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.

3

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`), 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. See Chapter 4, "The Programmer's Interface to RPC," for more details on passing arbitrary data types. The server-side of the `READDIR` procedure is shown in Code Example 3-5.

Code Example 3-5 Server `dir_proc.c` Example

```
/*
 * 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(dirname, req)
    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 freed 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);
}
```

Code Example 3-6 shows the client-side implementation of the `READDIR` procedure.

Code Example 3-6 Client-side Implementation of `rls.c`

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

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

extern int errno;

main(argc, argv)
    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.

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.

Table 3-1 rpcgen Preprocessing Directives

Symbol	Use
RPC_HDR	Header file output
RPC_XDR	XDR routine output
RPC_SVC	Server stub output
RPC_CLNT	Client stub output
RPC_TBL	Index table output

Code Example 3-7 is a simple rpcgen example. Note the use of rpcgen's preprocessing features.

Code Example 3-7 Time Protocol rpcgen Source

```
/*
 * 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 *Directive*

`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.

Table 3-2 `rpcgen` Compile-time Flags

Option	Flag	Comments
Templates	-a, -Sc, -Ss, -Sm	See Table 3-3 on page 38
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

3

Client and Server Templates

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

Table 3-3 `rpcgen` Template Selection Flags

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.

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.

C-style Mode

Also called Newstyle mode, The -N flag causes `rpcgen` to produce code in which arguments are passed by value and multiple arguments are passed without a `struct`. These changes allow RPC code that is more like C and other high-level languages. For compatibility with existing programs and make files, the previous (standard) mode of argument passing is the default. The following examples demonstrate the new feature. The source modules for both modes, C-style and default, are given in Code Example 3-8 and in Code Example 3-9 respectively.

Code Example 3-8 C-style Mode Version of `add.x`

```
/*
 * This program contains a procedure to add 2 numbers. It
 * demonstrates the C-style mode argument passing. Note that add()
 * has 2 arguments.
 */
program ADDPROG {                                   /* program number */
    version ADDVER {                                 /* version number */
        int add(int, int) = 1;                       /* procedure */
    } = 1;
} = 0x20000199;
```

Code Example 3-9 Default Mode Version of `add.x`

```
/*
 * This program contains a procedure to add 2 numbers
 * It demonstrates the "default" mode argument passing.
 * In this mode rpcgen can process only one argument.
 */
struct add_arg {
    int first;
    int second;
};
program ADDPROG {                                   /* program number */
    version ADDVER {                                 /* version number */
        int add (add_arg) = 1;                       /* procedure */
    } = 1;
} = 0x20000199;
```

The next four figures show the resulting client-side templates.

Code Example 3-10 C-style Mode Client Stub for `add.x`

```
/*
 * The C-style client side main routine calls the  add() function on
 * the remote rpc server
 */
#include <stdio.h>
#include "add.h"

main(argc, argv)
int argc;
char *argv[];
{
    CLIENT *clnt;
    int *result,x,y;

    if(argc != 4) {
        printf("usage: %s host num1 num2\n" argv[0]);
        exit(1);
    }
    /* create client handle - bind to server */
    clnt = clnt_create(argv[1], ADDPROG, ADDVER, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror(argv[1]);
        exit(1);
    }
    x = atoi(argv[2]);
    y = atoi(argv[3]);
    /*
     * invoke remote procedure: Note that multiple arguments can be
     * passed to add_1() instead of a pointer
     */
    result = add_1(x, y, clnt);
    if (result == (int *) NULL) {
        clnt_perror(clnt, "call failed:");
        exit(1);
    } else {
        printf("Success: %d + %d = %d\n", x, y, *result);
    }
    exit(0);
}
```

Code Example 3-11 shows how the default mode code differs from C-style mode code.

Code Example 3-11 Default Mode Client

```
    arg.first = atoi(argv[2]);
    arg.second = atoi(argv[3]);
    /*
     * invoke remote procedure -- note that a pointer to the argument
     * has to be passed to the client stub
     */
    result = add_1(&arg, clnt);
```

The server-side procedure in C-style mode is shown in Code Example 3-12.

Code Example 3-12 C-style Mode Server

```
#include "add.h"

int *
add_1(arg1, arg2, rqstp)
    int arg1;
    int arg2;
    struct svc_req *rqstp;
{
    static int result;

    result = arg1 + arg2;
    return(&result);
}
```

The server side procedure in default mode is shown in Code Example 3-13.

Code Example 3-13 Default Mode Server Stub

```
#include "add.h"
int *
add_1(argp, rqstp)
    add_arg *argp;
    struct svc_req *rqstp;
{
    static int result;

    result = argp->first + argp->second;
    return(&result);
}
```

3

MT-Safe Code

By default, the code generated by `rpcgen` is not MT safe. It uses unprotected global variables and returns results in the form of static variables. The `-M` flag generates MT-safe code which can be used in a multithreaded environment. This can be used with the C-style flag, the ANSI C flag, or both.

An example of an MT-safe program with this interface follows. The `rpcgen` protocol file is `msg.x`, shown in Code Example 3-14.

Code Example 3-14 MT-Safe Program: `msg.x`

```
program MESSAGEPROG {
version PRINTMESSAGE {
        int PRINTMESSAGE(string) = 1;
        } = 1;
} = 0x4001;
```

A string is passed to the remote procedure, which prints it and returns the length of the string to the client. The MT-Safe stubs are generated with:

```
% rpcgen -M msg.x
```

A possible client-side code that could be used with this is shown in Code Example 3-15.

Code Example 3-15 MT-Safe Client Stub

```
#include "msg.h"

void
messageprog_1(host)
    char *host;
{
    CLIENT *clnt;
    enum clnt_stat retval_1;
    int result_1;
    char * printmessage_1_arg;

    clnt = clnt_create(host, MESSAGEPROG, PRINTMESSAGE, "netpath");
    if (clnt == (CLIENT *) NULL) {
        clnt_pcreateerror(host);
        exit(1);
    }
    printmessage_1_arg = (char *) malloc(256);
```

```
    strcpy(printmessage_1_arg, "Hello World");

    retval_1 = printmessage_1(&printmessage_1_arg, &result_1,clnt);
    if (retval_1 != RPC_SUCCESS) {
        clnt_perror(clnt, "call failed");
    }
    printf("result = %d\n", result_1);

    clnt_destroy(clnt);
}

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

    if (argc < 2) {
        printf("usage:  %s server_host\n", argv[0]);
        exit(1);
    }
    host = argv[1];
    messageprog_1(host);
}
```

Note that a pointer to both the arguments and the results needs to be passed in to the `rpcgen`-generated code. This is to preserve reentrancy. The value returned by the stub function indicates whether this call is a success or a failure. The stub returns `RPC_SUCCESS` if the call is successful. Compare the MT-safe client stub (generated with the -M option) and the not MT-safe client stub shown in Code Example 3-16. The client stub that is not MT safe uses a static to store returned results and can use only one thread at a time.

Code Example 3-16 Client Stub (Not MT Safe)

```
int *
printmessage_1(argp, clnt)
    char **argp;
    CLIENT *clnt;
{
    static int clnt_res;

    memset((char *)&clnt_res, 0, sizeof (clnt_res));
    if (clnt_call(clnt, PRINTMESSAGE,
        (xdrproc_t) xdr_wrapstring, (caddr_t) argp,
```

```
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}
```

The server side code is shown in Code Example 3-17.

Note – When compiling a server that uses MT-safe mode, you must link in the threads library. To do this, specify the `-lthread` option in the compile command.

Code Example 3-17 MT-Safe Server Stub

```
#include "msg.h"
#include <syslog.h>

bool_t
printmessage_1_svc(argp, result, rqstp)
    char **argp;
    int *result;
    struct svc_req *rqstp;
{
    int retval;

    if (*argp == NULL) {
        syslog(LOG_INFO, "argp is NULL\n");
        *result = 0;
    }
    else {
        syslog("argp is %s\n", *argp);
        *result = strlen (*argp);
    }
    retval = 1;
    return (retval);
}

int
messageprog_1_freeresult(transp, xdr_result, result)
    SVCXPRT *transp;
    xdrproc_t xdr_result;
    caddr_t result;
{
    /*
     * Insert additional freeing code here, if needed
```

```
     */
    (void) xdr_free(xdr_result, result);
}
```

The server side code should not use statics to store returned results. A pointer to the result is passed in and this should be used to pass the result back to the calling routine. A return value of 1 indicates success to the calling routine, while 0 indicates a failure.

In addition, the code generated by `rpcgen` also generates a call to a routine to free any memory that may have been allocated when the procedure was called. To prevent memory leaks, any memory allocated in the service routine needs to be freed in this routine. `messageprog_1_freeresult` frees the memory.

Normally, `xdr_free` frees any allocated memory for you (in this case, no memory was allocated, so no freeing needs to take place).

As an example of the use of the -M flag with the C-style and ANSI C flag, consider the following file, `add.x`, shown in Code Example 3-18.

Code Example 3-18 MT-Safe Program: `add.x`

```
program ADDPROG {
    version ADDVER {
    int add(int, int) = 1;
    } = 1;
}= 199;
```

This program adds two numbers and returns its result to the client. `rpcgen` is invoked on it, with the following command:

```
% rpcgen -N -M -C add.x
```

The multithreaded client code to call this is shown in Code Example 3-19.

Code Example 3-19 MT-Safe Client: `add.x`

```
/*
 * This client-side main routine starts up a number of threads, each
 * of which calls the server concurrently.
 */

#include "add.h"

CLIENT *clnt;
#define NUMCLIENTS 5
struct argrec {
    int arg1;
    int arg2;
```

```
};

/* Keeps count of number of threads running */
int numrunning;
mutex_t numrun_lock;
cond_t condnum;

void
addprog(struct argrec *args)
{
    enum clnt_stat retval;
    int result;
    /* call server code */
    retval = add_1(args->arg1, args->arg2, &result, clnt);
    if (retval != RPC_SUCCESS) {
        clnt_perror(clnt, "call failed");
    } else
        printf("thread #%x call succeeded, result = %d\n",
        thr_getself(),
               result);
    /* decrement the number of running threads */
    mutex_lock(&numrun_lock);
    numrunning--;
    cond_signal(&condnum);
    mutex_unlock(&numrun_lock);
    thr_exit(NULL);
}

main(int argc, char *argv[])
{
    char *host;
    struct argrec args[NUMCLIENTS];
    int i;
    thread_t mt;
    int ret;

    if (argc < 2) {
        printf("usage:  %s server_host\n", argv[0]);
        exit(1);
    }
    host = argv[1];
    clnt = clnt_create(host, ADDPROG, ADDVER, "netpath");
    if (clnt == (CLIENT *) NULL) {
        clnt_pcreateerror(host);
        exit(1);
```

```
    };
    mutex_init(&numrun_lock, USYNC_THREAD, NULL);
    cond_init(&condnum, USYNC_THREAD, NULL);
    numrunning = 0;

    /* Start up separate threads */
    for (i = 0; i < NUMCLIENTS; i++) {
        args[i].arg1 = i;
        args[i].arg2 = i + 1;
        ret = thr_create(NULL, NULL, addprog, (char *) &args[i],
                 THR_NEW_LWP, &mt);
        if (ret == 0)
            numrunning++;
    }

    mutex_lock(&numrun_lock);
    /* are any threads still running ? */
    while (numrunning != 0)
        cond_wait(&condnum, &numrun_lock);
    mutex_unlock(&numrun_lock);
    clnt_destroy(clnt);
}
```

The server-side procedure is shown in Code Example 3-20.

Note – When compiling a server that uses MT-safe mode, you must link in the threads library. To do this, specify the `-lthread` option in the compile command.

Code Example 3-20 MT-Safe Server: `add.x`

```
add_1_svc(int arg1, int arg2, int *result, struct svc_req *rqstp)
{
    bool_t retval;
    /* Compute result */
    *result = arg1 + arg2;
    retval = 1;
    return (retval);
}

/* Routine for freeing memory that may be allocated in the server
   procedure */
int
addprog_1_freeresult(SVCXPRT *transp, xdrproc_t xdr_result, caddr_t
result)
```

```
{
    (void) xdr_free(xdr_result, result);
}
```

MT Auto Mode

MT Auto mode enables RPC servers to automatically use Solaris threads to process client requests concurrently. Use the `-A` option to generate RPC code in MT Auto mode. The `-A` option also has the effect of turning on the `-M` option, so `-M` does not need to be explicitly specified. The `-M` option is necessary because any code generated has to be multithread safe.

Further discussion on multithreaded RPC begins on page 111; see also "MT Auto Mode" on page 119.

Here is an example of an Auto mode program generated by `rpcgen`. The `rpcgen` protocol file`time.x` is shown in Code Example 3-21.

Code Example 3-21 MT Auto Mode: `time.x`

```
program TIMEPROG {
    version TIMEVERS {
        unsigned int TIMEGET(void) = 1;
        void TIMESET(unsigned) = 2;
    } = 1;
} = 0x20000044;
```

A string is passed to the remote procedure, which prints it and returns the length of the string to the client. The MT-safe stubs are generated with:

```
% rpcgen -A time.x
```

When the `-A` option is used, the generated server code will contain instructions for enabling MT Auto mode for the server.

Note – When compiling a server that uses MT Auto mode, you must link in the threads library. To do this, specify the `-lthread` option in the compile command.

TI-RPC or TS-RPC Library Selection

In older SunOS releases, `rpcgen` created stubs that used the socket functions. With the current SunOS release, you can use either the transport-independent RPC (TI-RPC) or the transport-specific socket (TS-RPC) routines. This provides backward compatibility with previous releases. The default uses the TI-RPC interfaces. The `-b` flag tells `rpcgen` to create TS-RPC variant source code as its output.

ANSI C-compliant Code

`rpcgen` can also produce output that is compatible with ANSI C or SPARCompiler C++ 3.0. This feature is selected with the `-C` compile flag and is most often used with the `-N` flag, described in "C-style Mode" on page 39.

The `add.x` example of the server template is generated by the command:

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

It is important to note that on the C++ 3.0 server, remote procedure names require an `_svc` suffix. In the following example, the `add.x` template and the `-C` compile flag produce the client side `add_1` and the server stub `add_1_svc`.

Code Example 3-22 `rpcgen` ANSI C Server Template

```
/*
 * This is a template. Use it to develop your own functions.
 */
#include <c_varieties.h>
#include "add.h"

int *
add_1_svc(int arg1, int arg2, struct svc_req *rqstp)
{
    static int result;
    /*
     * insert server code here
     */
    return(&result);
}
```

This output conforms to the syntax requirements and structure of ANSI C. The header files that are generated when this option is invoked can be used with ANSI C or with C++.

`xdr_inline` Count

`rpcgen` tries to generate more efficient code by using `xdr_inline` when possible (see `xdr_admin`(3) manpage). When a structure contains elements that `xdr_inline` can be used on (for example `integer`, `long`, `bool`), the relevant portion of the structure is packed with `xdr_inline`(). A default of five or more packed elements in sequence causes in-line code to be generated. This default can be changed with the `-i` flag. For example:

```
rpcgen -i 3 test.x
```

causes `rpcgen` to start generating in-line code after three qualifying elements are found in sequence. The example:

```
rpcgen -i 0 test.x
```

prevents any in-line code from being generated.

In most situations, there is no reason to use the `-i` flag. The `_xdr.c` stub is the only file affected by this feature.

`rpcgen` Programming Techniques

This section suggests some common RPC and `rpcgen` programming techniques. Each topic is covered in its own subsection.

- Network Type
  `rpcgen` can produce server code for specific transport types.
- Define Statements
  C-preprocessing symbols can be defined on `rpcgen` command lines.
- Broadcast Calls
  Servers need not send error replies to broadcast calls.
- Debugging Applications
  Debug as normal function calls, then change to a distributed application.
- Port Monitor Support
  Port monitors can “listen” on behalf of RPC servers.
- Dispatch Tables
  Programs can access server dispatch tables.

- Time-out Changes
  Client default time-out periods can be changed.
- Authentication
  Clients may authenticate themselves to servers; interested servers can examine client authentication information.

Network Types/Transport Selection

`rpcgen` takes optional arguments that allow a programmer to specify desired network types or specific network identifiers. (For details of network selection, see *Transport Interfaces Programming Guide*).

The `-s` flag creates a server that responds to requests on the specified type of transport. For example, the invocation

```
rpcgen -s datagram_n prot.x
```

writes a server to standard output that responds to any of the connectionless transports specified in the `NETPATH` environment variable (or in `/etc/netconfig`, if `NETPATH` is not defined). A command line can contain multiple `-s` flags and their network types.

Similarly, the `-n` flag creates a server that responds only to requests from the transport specified by a single network identifier.

Caution – Be careful using servers created by `rpcgen` with the `-n` flag. Network identifiers are host specific, so the resulting server may not run as expected on other hosts.

Command Line Define Statements

You can define C-preprocessing symbols and assign values to them from the command line. Command line define statements can, for example, be used to generate conditional debugging code when the `DEBUG` symbol is defined. For example:

```
$ rpcgen -DDEBUG proto.x
```

Server Response to Broadcast Calls

When a procedure has been called through broadcast RPC and cannot provide a useful response, the server should send no reply to the client. This reduces network traffic. To prevent the server from replying, a remote procedure can return `NULL` as its result. The server code generated by `rpcgen` detects this and sends no reply.

Code Example 3-23 is a procedure that replies only if it is an NFS server.

Code Example 3-23 NFS Server Response to Broadcast Calls

```
void *
reply_if_nfsserver()
{
    char notnull; /*only here so we can use its address*/

    if( access( "/etc/dfs/sharetab", F_OK ) < 0 ) {
        /* prevent RPC from replying */
        return( (void *) NULL );
    }
    /* assign notnull a non-null value so RPC will send a reply */
    return( (void *) &notnull );
}
```

A procedure *must* return a non-`NULL` pointer when it wants RPC library routines to send a reply.

In Code Example 3-23, if the procedure `reply_if_nfsserver` is defined to return non void values, the return value (`&notnull`) should point to a static variable.

Port Monitor Support

Port monitors such as `inetd` and `listen` can monitor network addresses for specified RPC services. When a request arrives for a particular service, the port monitor spawns a server process. After the call has been serviced, the server can exit. This technique conserves system resources. The `main` server function generated by `rpcgen` allows invocation by `inetd`. See “Using inetd” on page 105“for details.

It may be useful for services to wait for a specified interval after satisfying a service request, in case another request follows. If there is no call in the specified time, the server exits, and some port monitors, like `inetd`, continue

to monitor for the server. If a later request for the service occurs, the port monitor gives the request to a waiting server process (if any), rather than spawning a new process.

Note – When monitoring for a server, some port monitors, like `listen`, always spawn a new process in response to a service request. If a server is used with such a monitor, it should exit immediately on completion.

By default, services created using `rpcgen` wait for 120 seconds after servicing a request before exiting. The programmer can change the interval with the `-K` flag. In this example,

```
$ rpcgen -K 20 proto.x
```

the server waits for 20 seconds before exiting. To create a server that exits immediately, zero value can be used for the interval period:

```
$ rpcgen -K 0 proto.x.
```

To create a server that never exits, the value is `-K -1.`

Time-out Changes

After sending a request to the server, a client program waits for a default period (25 seconds) to receive a reply. This time-out may be changed using the `clnt_control()` routine. See "Standard Interfaces" on page 68," for additional uses of the `clnt_control` routine. See also the `rpc`(3N) manpage. When considering time-out periods, be sure to allow the minimum amount of time required for "round-trip" communications over the network.
Code Example 3-24 illustrates the use of `clnt_control()`.

Code Example 3-24 `clnt_control` Routine

```
struct timeval tv;

CLIENT *clnt;
clnt = clnt_create( "somehost", SOMEPROG, SOMEVERS, "visible" );

if (clnt == (CLIENT *)NULL)
    exit(1);
tv.tv_sec = 60;/* change time-out to 60 seconds */
tv.tv_usec = 0;
clnt_control(clnt, CLSET_TIMEOUT, &tv);
```

3

Client Authentication

The client create routines do not have any facilities for client authentication. Some clients may have to authenticate themselves to the server.

The following example illustrates one of the least secure authentication methods in common use. See "Authentication" on page 95," for information on the more secure DES authentication technique.

Code Example 3-25 `AUTH_SYS` Authentication Program

```
CLIENT *clnt;
clnt = clnt_create( "somehost", SOMEPROG, SOMEVERS, "visible" );
if (clnt != (CLIENT *)NULL) {
    /* To set AUTH_SYS style authentication */
        clnt->cl_auth = authsys_createdefault();
}
```

Authentication information is important to servers that have to achieve some level of security. This extra information is supplied to the server as a second argument.

Code Example 3-26 is a server that checks client authentication data. It is modified from `printmessage_1()` in "An rpcgen Tutorial" on page 23 and only allows superusers to print a message to the console.

Code Example 3-26 `printmsg_1` for Superuser

```
int *
printmessage_1(msg, req)
    char **msg;
    struct svc_req  *req;
{
    static int result;/* Must be static */
    FILE *f;
    struct authsys_parms *aup;

    aup = (struct authsys_parms *)req->rq_clntcred;
    if (aup->aup_uid != 0) {
        result = 0;
        return (&result)
    }

/* Same code as before. */
}
```

Dispatch Tables

It is sometimes useful for programs to have access to dispatch tables used by the RPC package. For example, the server dispatch routine may check authorization and then invoke the service routine; or a client library may deal with the details of storage management and XDR data conversion.

When invoked with the `-T` option, `rpcgen` generates RPC dispatch tables for each program defined in the protocol description file, `proto.x`, in the file `proto_tbl.i`. The suffix `.i` stands for "index." `rpcgen` may be invoked with the `-t` option to build only the header file. `rpcgen` cannot be invoked in C-style mode (`-N`) with either the `-T` or `-t` flag.

Each entry in the dispatch table is a `struct rpcgen_table`, defined in the header file `proto.h` as follows:

```
struct rpcgen_table {
    char *(*proc)();
    xdrproc_t xdr_arg;
    unsigned len_arg;
    xdrproc_t xdr_res;
    xdrproc_t len_res
};
```

where:

`proc`	is a pointer to the service routine
`xdr_arg`	is a pointer to the input (argument) `xdr` routine
`len_arg`	is the length in bytes of the input argument
`xdr_res`	is a pointer to the output (result) `xdr` routine
`len_res`	is the length in bytes of the output result

The table, named `dirprog_1_table` for the `dir.x` example, is indexed by procedure number. The variable `dirprog_1_nproc` contains the number of entries in the table.

An example of how to locate a procedure in the dispatch tables is shown by the routine `find_proc()`:

```
struct rpcgen_table *
find_proc(proc)
    u_long proc;
{
    if (proc >= dirprog_1_nproc)
        /* error */
```

```
    else
        return (&dirprog_1_table[proc]);
}
```

Each entry in the dispatch table contains a pointer to the corresponding service routine. However, that service routine is usually not defined in the client code. To avoid generating unresolved external references, and to require only one source file for the dispatch table, the `rpcgen` service routine initializer is `RPCGEN_ACTION(proc_ver)`.

This way, the same dispatch table can be included in both the client and the server. Use the following define statement when compiling the client:

```
#define RPCGEN_ACTION(routine) 0
```

And use the following define when writing the server:

```
#define RPCGEN_ACTION(routine)routine
```

Debugging Applications

You can simplify the testing and debugging process. First test the client program and the server procedure in a single process by linking them with each other rather than with the client and server skeletons. Comment out calls to the client create RPC library routines (see the `rpc_clnt_create`(3N) manpage) and the authentication routines. Do not link with `libnsl`.

Link the procedures from previous example by:

```
cc rls.c dir_clnt.c dir_proc.c -o rls
```

With the RPC and XDR functions commented out, the procedure calls execute as ordinary local function calls, and the program is debugged with a local debugger such as `dbxtool`. When the program works, the client program is linked to the client skeleton produced by `rpcgen` and the server procedures are linked to the server skeleton produced by `rpcgen`.

You can also use the Raw RPC mode to test the XDR routines. See “Testing Programs Using Low-level Raw RPC” on page 85 for details.

There are two kinds of errors that can happen in an RPC call. The first kind of error is caused by a problem with the mechanism of the remote procedure calls. Examples of these are (1) the procedure is not available, (2) the remote server is not responding, and (3) the remote server is unable to decode the

arguments. In Code Example 3-26, an RPC error happens if `result` is `NULL`. The reason for the failure can be displayed by using `clnt_perror()`, or an error string can be returned through `clnt_sperror()`.

The second type of error is caused by the server itself. In Code Example 3-26, an error can be returned by `opendir`(). The handling of these errors is application specific and is the responsibility of the programmer.

Note that the mechanism illustrated by the paragraphs above does not function with the `-C` option because of the `_svc` suffix added to the server-side routines.

3