The Programmer's Interface to RPC 4

This chapter 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(3N)` and related man pages.

Simplified Interface	*page 60*
Standard Interfaces	*page 68*
Testing Programs Using Low-level Raw RPC	*page 85*
Advanced RPC Programming Techniques	*page 88*
Multithreaded RPC Programming	*page 111*
MT Auto Mode	*page 119*
MT User Mode	*page 123*
Porting From TS-RPC to TI-RPC	*page 136*

RPC Is Multithread Safe

The client and server interfaces described in this chapter are multithread safe, except where noted (such as raw mode). This means that applications that contain RPC function calls can be used freely in a multithreaded application.

4

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`. Code Example 4-1 is a program that displays the number of users on a remote host. It calls the RPC library routine, `rusers()`.

Code Example 4-1 `rusers` Program

```
#include <stdio.h>

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

main(argc, argv)
    int argc;
    char **argv;
{
    int num;

    if (argc != 2) {
        fprintf(stderr, "usage: %s hostname\n", argv[0]);
        exit(1);
    }
    if ((num = rusers(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 in Code Example 4-1 with:

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

Client

There is just one function on the client side of the simplified interface: `rpc_call()`. It has nine parameters:

```
int                                     0 or error code

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 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 in Code Example 4-1, changed to use the simplified interface, looks like Code Example 4-2.

Code Example 4-2 `rusers` Program Using Simplified Interface

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

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

main(argc, argv)
    int argc;
    char **argv;
{
    unsigned long nusers;
    enum clnt_stat;

    if (argc != 2) {
        fprintf(stderr, "usage: rusers hostname\n");
        exit(1);
    }
    if( clnt_stat = rpc_call(argv[1], RUSERSPROG, RUSERSVERS,
      RUSERSPROC_NUM, xdr_void, (char *)0, xdr_u_long,
      (char *)&nusers, "visible") != RPC_SUCCESS) {
        clnt_perrno(clnt_stat);
        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`.

Server

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 arguments:

```
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`.

Hand-Coded Registration Routine

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.

Code Example 4-3 Hand-Coded Registration Server

```
#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.

Table 4-1 XDR Primitive Type Routines

XDR Primitive Routines			
`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 in Code Example 4-4.

Code Example 4-4 `xdr_simple` Routine

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

bool_t
xdr_simple(xdrsp, simplep)
    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`.

An XDR routine returns nonzero (a C `TRUE`) if it completes successfully, and zero otherwise. A complete description of XDR is provided in Appendix C, “XDR Protocol Specification."

Table 4-2 XDR Building Block Routines

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_varintarr()`, as shown in Code Example 4-5.

Code Example 4-5 `xdr_varintarr` Syntax Use

```
bool_t
xdr_varintarr(xdrsp, arrp)
    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()`, as shown in Code Example 4-6.

Code Example 4-6 `xdr_vector` Syntax Use

```
int intarr[SIZE];

bool_t
```

```
xdr_intarr(xdrsp, 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.

Code Example 4-7 calls the built-in functions `xdr_string()` and `xdr_reference()`, which translates pointers to pass a string, and `struct simple` from the previous examples.

Code Example 4-7 `xdr_reference` Syntax Use

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

bool_t
xdr_finalexample(xdrsp, finalp)
    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 that `xdr_simple()` could have been called here instead of `xdr_reference()`.

4

Standard Interfaces

Interfaces to standard levels of the RPC package provide increasing control over RPC communications. Programs that use this control are more complex. Effective programming at these lower levels requires more knowledge of computer network fundamentals. The top, intermediate, expert, and bottom levels are part of the standard interfaces.

This section shows how to control RPC details by using lower levels of the RPC library. For example, you can select the transport protocol, which can be done at the simplified interface level only through the `NETPATH` variable. You should be familiar with the TLI in order to use these routines.

The routines shown in Table 4-3 cannot be used through the simplified interface because they require a transport handle. For example, there is no way to allocate and free memory while serializing or deserializing with XDR routines at the simplified interface.

Table 4-3 XDR Routines Requiring a Transport Handle

Do Not Use With Simplified Interface		
`clnt_call()`	`clnt_destroy()`	`clnt_control()`
`clnt_perrno()`	`clnt_pcreateerror()`	`clnt_perror()`
`svc_destroy()`		

Top Level Interface

At the top level, the application can specify the *type* of transport to use but not the specific transport. This level differs from the simplified interface in that the application creates its own transport handles, in both the client and server.

Client

Assume the header file in Code Example 4-8.

Code Example 4-8 `time_prot.h` Header File

```
/* time_prot.h */
#include <rpc/rpc.h>
#include <rpc/types.h>
```

```
struct timev {
    int second;
    int minute;
    int hour;
};
typedef struct timev timev;
bool_t xdr_timev();

#define TIME_PROG ((u_long)0x40000001)
#define TIME_VERS ((u_long)1)
#define TIME_GET ((u_long)1)
```

Code Example 4-9 shows the client side of a trivial date service using top-level service routines. The transport type is specified as an invocation argument of the program.

Code Example 4-9 Client for Trivial Date Service

```
#include <stdio.h>
#include "time_prot.h"

#define TOTAL (30)
/*
 * Caller of trivial date service
 * usage: calltime hostname
 */
main(argc, argv)
    int argc;
    char *argv[];
{
    struct timeval time_out;
    CLIENT *client;
    enum clnt_stat stat;
    struct timev timev;
    char *nettype;

    if (argc != 2 && argc != 3) {
        fprintf(stderr,"usage:%s host[nettype]\n",argv[0]);
        exit(1);
    }
    if (argc == 2)
        nettype = "netpath";                    /* Default */
    else
        nettype = argv[2];
    client = clnt_create(argv[1], TIME_PROG, TIME_VERS, nettype);
    if (client == (CLIENT *) NULL) {
```

```
            clnt_pcreateerror("Couldn't create client");
            exit(1);
        }
        time_out.tv_sec = TOTAL;
        time_out.tv_usec = 0;
        stat = clnt_call( client, TIME_GET, xdr_void, (caddr_t)NULL,
            xdr_timev, (caddr_t)&timev, time_out);
        if (stat != RPC_SUCCESS) {
            clnt_perror(client, "Call failed");
            exit(1);
        }
        fprintf(stderr,"%s: %02d:%02d:%02d GMT\n", nettype, timev.hour,
            timev.minute, timev.second);
        (void) clnt_destroy(client);
        exit(0);
}
```

If `nettype` is not specified in the invocation of the program, the string `netpath` is substituted. When RPC libraries routines encounter this string, the value of the `NETPATH` environment variable governs transport selection.

If the client handle cannot be created, display the reason for the failure with `clnt_pcreateerror()`, or get the error status by reading the contents of the global variable `rpc_createerr`.

After the client handle is created, `clnt_call()` is used to make the remote call. Its arguments are the remote procedure number, an XDR filter for the input argument, the argument pointer, an XDR filter for the result, the result pointer, and the time-out period of the call. The program has no arguments, so `xdr_void()` is specified. Clean up by calling `clnt_destroy()`.

In the above example, if the programmer wished to bound the time allowed for client handle creation to thirty seconds, the call to clnt_create() should be replaced with a call to clnt_create_timed() as shown in the following code segment:

```
struct timeval timeout;
timeout.tv_sec = 30; /* 30 seconds */
timeout.tv_usec = 0;

client = clnt_create_timed(argv[1], TIME_PROG, TIME_VERS, nettype,
        &timeout);
```

Server

Code Example 4-10 shows a top-level implementation of a server for the trivial date service.

Code Example 4-10 Server for Trivial Date Service

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "time_prot.h"

static void time_prog();

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

    if (argc > 2) {

        fprintf(stderr, "usage: %s [nettype]\n", argv[0]);
        exit(1);
    }
    if (argc == 2)
        nettype = argv[1];
    else
        nettype = "netpath";                        /* Default */
    transpnum = svc_create(time_prog,TIME_PROG,TIME_VERS,nettype);
    if (transpnum == 0) {
        fprintf(stderr,"%s: cannot create %s service.\n", argv[0],
                  nettype);
        exit(1);
    }
    svc_run();
}

/*
 * The server dispatch function
 */
static void
time_prog(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
```

```
{
    struct timev rslt;
    time_t thetime;

    switch(rqstp->rq_proc) {
        case NULLPROC:
            svc_sendreply(transp, xdr_void, NULL);
            return;
        case TIME_GET:
            break;
        default:
            svcerr_noproc(transp);
            return;
    }
    thetime = time((time_t *) 0);
    rslt.second = thetime % 60;
    thetime /= 60;
    rslt.minute = thetime % 60;
    thetime /= 60;
    rslt.hour = thetime % 24;
    if (!svc_sendreply( transp, xdr_timev, &rslt)) {
        svcerr_systemerr(transp);
    }
}
```

`svc_create()` returns the number of transports on which it created server handles. `time_prog()` is the service function called by `svc_run()` when a request specifies its program and version numbers. The server returns the results to the client through `svc_sendreply()`.

When `rpcgen` is used to generate the dispatch function, `svc_sendreply()` is called after the procedure returns, so `rslt` (in this example) must be declared `static` in the actual procedure. `svc_sendreply()` is called from inside the dispatch function, so `rslt` is not declared `static`.

In this example, the remote procedure takes no arguments. When arguments must be passed, the calls:

```
svc_getargs( SVCXPRT_handle, XDR_filter, argument_pointer);
svc_freeargs( SVCXPRT_handle, XDR_filter argument_pointer );
```

fetch, deserialize (XDR decode), and free the arguments.

Intermediate Level Interface

At the intermediate level, the application directly chooses the transport to use.

Client

Code Example 4-11 shows the client side of the time service from “Top Level Interface” on page 68, written at the intermediate level of RPC. In this example, the user must name the transport over which the call is made on the command line.

Code Example 4-11 Client for Time Service, Intermediate Level

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>/* For netconfig structure */
#include "time_prot.h"

#define TOTAL (30)

main(argc,argv)
    int argc;
    char *argv[];
{
    CLIENT *client;
    struct netconfig *nconf;
    char *netid;
    /* Declarations from previous example */

    if (argc != 3) {
        fprintf(stderr, "usage: %s host netid\n”, argv[0]);
        exit(1);
    }
    netid = argv[2];
    if ((nconf = getnetconfigent( netid)) ==
         (struct netconfig *) NULL) {
        fprintf(stderr, "Bad netid type: %s\n", netid);
        exit(1);
    }
    client = clnt_tp_create(argv[1], TIME_PROG, TIME_VERS, nconf);
    if (client == (CLIENT *) NULL) {
        clnt_pcreateerror("Could not create client");
        exit(1);
    }
```

```
    freenetconfigent(nconf);
    /* Same as previous example after this point */
}
```

In this example, the `netconfig` structure is obtained by a call to `getnetconfigent(netid)`. (See the `getnetconfig(3N)` man page and *Transport Interfaces Programming Guide* for more details.) At this level, the program explicitly selects the network.

In the above example, if the programmer wished to bound the time allowed for client handle creation to thirty seconds, the call to `clnt_tp_create()` should be replaced with a call to `clnt_tp_create_timed()` as shown in the following code segment:

```
struct timeval timeout;
timeout.tv_sec = 30; /* 30 seconds */
timeout.tv_usec = 0;

client = clnt_tp_create_timed(argv[1], TIME_PROG, TIME_VERS, nconf,
        &timeout);
```

Server

Code Example 4-12 shows the corresponding server. The command line that starts the service must specify the transport over which the service is provided.

Code Example 4-12 Server for Time Service, Intermediate Level

```
/*
 * This program supplies Greenwich mean time to the client
 * that invokes it. The call format is: server netid
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>    /* For netconfig structure */
#include "time_prot.h"

static void time_prog();

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

```
{
    SVCXPRT *transp;
    struct netconfig *nconf;

    if (argc != 2) {
        fprintf(stderr, "usage: %s netid\n", argv[0]);
        exit(1);
    }
    if ((nconf = getnetconfigent( argv[1])) ==
      (struct netconfig *) NULL) {
        fprintf(stderr, "Could not find info on %s\n", argv[1]);
        exit(1);
    }
    transp = svc_tp_create(time_prog, TIME_PROG, TIME_VERS, nconf);
    if (transp == (SVCXPRT *) NULL) {
        fprintf(stderr, "%s: cannot create %s service\n",
            argv[0], argv[1]);
        exit(1)
    }
    freenetconfigent(nconf);
    svc_run();
}

    static
    void time_prog(rqstp, transp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
/* Code identical to Top Level version */
}
```

Expert Level Interface

At the expert level, network selection is done the same as at the intermediate level. The only difference is in the increased level of control that the application has over the details of the `CLIENT` and `SVCXPRT` handles. These examples illustrate this control, which is exercised using the `clnt_tli_create()` and `svc_tli_create()` routines. For more information on TLI, see *Transport Interfaces Programming Guide*.

Client

Code Example 4-13 shows a version of `clntudp_create()` (the client creation routine for UDP transport) using `clnt_tli_create()`. The example shows how to do network selection based on the family of the transport you choose. `clnt_tli_create()` is used to create a client handle and to:

- Pass an open TLI file descriptor, which may or may not be bound
- Pass the server's address to the client
- Specify the send and receive buffer size

Code Example 4-13 Client for RPC Lower Level

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>
#include <netinet/in.h>
/*
 * In earlier implementations of RPC, only TCP/IP and UDP/IP were
 * supported. This version of clntudp_create() is based on
 * TLI/Streams.
 */
CLIENT *
clntudp_create(raddr, prog, vers, wait, sockp)
    struct sockaddr_in *raddr;     /* Remote address */
    u_long prog;                   /* Program number */
    u_long vers;                   /* Version number */
    struct timeval wait;           /* Time to wait */
    int *sockp;                    /* fd pointer */
{
    CLIENT *cl;                     /* Client handle */
    int madefd = FALSE;             /* Is fd opened here */
    int fd = *sockp;                /* TLI fd */
    struct t_bind *tbind;           /* bind address */
    struct netconfig *nconf;        /* netconfig structure */
    void *handlep;

    if ((handlep = setnetconfig() ) == (void *) NULL) {
        /* Error starting network configuration */
        rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
        return((CLIENT *) NULL);
    }
    /*
     * Try all the transports until it gets one that is
     * connectionless, family is INET, and preferred name is UDP
     */
```

```
while (nconf = getnetconfig( handlep)) {
    if ((nconf->nc_semantics == NC_TPI_CLTS) &&
         (strcmp( nconf->nc_protofmly, NC_INET ) == 0) &&
         (strcmp( nconf->nc_proto, NC_UDP ) == 0))
     break;
}
if (nconf == (struct netconfig *) NULL)
    rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
    goto err;
}
if (fd == RPC_ANYFD) {
    fd = t_open(nconf->nc_device, O_RDWR, &tinfo);
    if (fd == -1) {
        rpc_createerr.cf_stat = RPC_SYSTEMERROR;
        goto err;
    }
}
if (raddr->sin_port == 0) { /* remote addr unknown */
    u_short sport;
    /*
     * rpcb_getport() is a user-provided routine that calls
     * rpcb_getaddr and translates the netbuf address to port
     * number in host byte order.
     */
    sport = rpcb_getport(raddr, prog, vers, nconf);
    if (sport == 0) {
        rpc_createerr.cf_stat = RPC_PROGUNAVAIL;
        goto err;
    }
    raddr->sin_port = htons(sport);
}
/* Transform sockaddr_in to netbuf */
tbind = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR);
if (tbind == (struct t_bind *) NULL)
    rpc_createerr.cf_stat = RPC_SYSTEMERROR;
    goto err;
}
if (t_bind->addr.maxlen < sizeof( struct sockaddr_in))
    goto err;
(void) memcpy( tbind->addr.buf, (char *)raddr,
                sizeof(struct sockaddr_in));
tbind->addr.len = sizeof(struct sockaddr_in);
/* Bind fd */
if (t_bind( fd, NULL, NULL) == -1) {
    rpc_createerr.ct_stat = RPC_TLIERROR;
    goto err;
```

```
    }
    cl = clnt_tli_create(fd, nconf, &(tbind->addr), prog, vers,
                         tinfo.tsdu, tinfo.tsdu);
    /* Close the netconfig file */
    (void) endnetconfig( handlep);
    (void) t_free((char *) tbind, T_BIND);
    if (cl) {
        *sockp = fd;
        if (madefd == TRUE) {
            /* fd should be closed while destroying the handle */
            (void)clnt_control(cl,CLSET_FD_CLOSE, (char *)NULL);
        }
        /* Set the retry time */
        (void) clnt_control( l, CLSET_RETRY_TIMEOUT,
                             (char *) &wait);
        return(cl);
    }
err:
    if (madefd == TRUE)
        (void) t_close(fd);
    (void) endnetconfig(handlep);
    return((CLIENT *) NULL);
}
```

The network is selected using `setnetconfig()`, `getnetconfig()`, and `endnetconfig()`.

Note – `endnetconfig()` is not called until after the call to `clnt_tli_create()`, near the end of the example.

`clntudp_create()` can be passed an open TLI `fd`. If passed none `(fd == RPC_ANYFD)`, it opens its own using the `netconfig` structure for UDP to find the name of the device to pass to `t_open()`.

If the remote address is not known `(raddr->sin_port == 0)`, it is obtained from the remote `rpcbind`.

After the client handle has been created, you can modify it using calls to `clnt_control()`. The RPC library closes the file descriptor when destroying the handle (as it does with a call to `clnt_destroy()` when it opens the `fd` itself) and sets the retry time-out period.

Server

Code Example 4-14 shows the server side of Code Example 4-13. It is called `svcudp_create()`. The server side uses `svc_tli_create()`.

`svc_tli_create()` is used when the application needs a fine degree of control, particularly to:

- Pass an open file descriptor to the application.
- Pass the user's bind address.
- Set the send and receive buffer sizes. The `fd` argument may be unbound when passed in. If it is, then it is bound to a given address, and the address is stored in a handle. If the bind address is set to `NULL` and the `fd` is initially unbound, it will be bound to any suitable address.

Use `rpcb_set()` to register the service with `rpcbind`.

Code Example 4-14 Server for RPC Lower Level

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>
#include <netinet/in.h>

SVCXPRT *
svcudp_create(fd)
    register int fd;
{
    struct netconfig *nconf;
    SVCXPRT *svc;
    int madefd = FALSE;
    int port;
    void *handlep;
    struct  t_info tinfo;

    /* If no transports available */
    if ((handlep = setnetconfig() ) == (void *) NULL) {
        nc_perror("server");
        return((SVCXPRT *) NULL);
    }
    /*
     * Try all the transports until it gets one which is
     * connectionless, family is INET and, name is UDP
     */
    while (nconf = getnetconfig( handlep)) {
        if ((nconf->nc_semantics == NC_TPI_CLTS) &&
```

```
            (strcmp( nconf->nc_protofmly, NC_INET) == 0 )&&
            (strcmp( nconf->nc_proto, NC_UDP) == 0 ))
           break;
    }
    if (nconf == (struct netconfig *) NULL) {
        endnetconfig(handlep);
        return((SVCXPRT *) NULL);
    }
    if (fd == RPC_ANYFD) {
        fd = t_open(nconf->nc_device, O_RDWR, &tinfo);
        if (fd == -1) {
            (void) endnetconfig();
            return((SVCXPRT *) NULL);
        }
        madefd = TRUE;
    } else
        t_getinfo(fd, &tinfo);
    svc = svc_tli_create(fd, nconf, (struct t_bind *) NULL,
                         tinfo.tsdu, tinfo.tsdu);
    (void) endnetconfig(handlep);
    if (svc == (SVCXPRT *) NULL) {
        if (madefd)
            (void) t_close(fd);
        return((SVCXPRT *)NULL);
    }
    return (svc);
}
```

The network selection here is accomplished similar to `clntudp_create()`. The file descriptor is not bound explicitly to a transport address because `svc_tli_create()` does that.

`svcudp_create()` can use an open `fd`. It will open one itself using the selected `netconfig` structure, if none is provided.

Bottom Level Interface

The bottom-level interface to RPC lets the application control all options. `clnt_tli_create()` and the other expert-level RPC interface routines are based on these routines. You rarely use these low-level routines.

Bottom-level routines create internal data structures, buffer management, RPC headers, and so on. Callers of these routines, like the expert level routine `clnt_tli_create()`, must initialize the `cl_netid` and `cl_tp` fields in the

client handle. For a created handle, `cl_netid` is the network identifier (for example `udp`) of the transport and `cl_tp` is the device name of that transport (for example `/dev/udp`). The routines `clnt_dg_create()` and `clnt_vc_create()` set the `clnt_ops` and `cl_private` fields.

Client

Code Example 4-15 shows calls to `clnt_vc_create()` and `clnt_dg_create()`.

Code Example 4-15 Client for Bottom Level

```
/*
 * variables are:
 * cl: CLIENT *
 * tinfo: struct t_info returned from either t_open or t_getinfo
 * svcaddr: struct netbuf *
 */
   switch(tinfo.servtype) {
       case T_COTS:
       case T_COTS_ORD:
           cl = clnt_vc_create(fd, svcaddr,
            prog, vers, sendsz, recvsz);
           break;
       case T_CLTS:
           cl = clnt_dg_create(fd, svcaddr,
            prog, vers, sendsz, recvsz);
           break;
       default:
           goto err;
   }
```

These routines require that the file descriptor is open and bound. `svcaddr` is the address of the server.

Server

Code Example 4-16 is an example of creating a bottom-level server.

Code Example 4-16 Server for Bottom Level

```
/*
 * variables are:
 * xprt: SVCXPRT *
```

```
 */
switch(tinfo.servtype) {
    case T_COTS_ORD:
    case T_COTS:
        xprt = svc_vc_create(fd, sendsz, recvsz);
        break;
    case T_CLTS:
        xprt = svc_dg_create(fd, sendsz, recvsz);
        break;
    default:
        goto err;
}
```

Server Caching

`svc_dg_enablecache()` initiates service caching for datagram transports. Caching should be used only in cases where a server procedure is a "once only" kind of operation, because executing a cached server procedure multiple times will yield different results.

```
svc_dg_enablecache(xprt, cache_size)
    SVCXPRT *xprt;
    unsigned long cache_size;
```

This function allocates a duplicate request cache for the service endpoint `xprt`, large enough to hold `cache_size` entries. A duplicate request cache is needed if the service contains procedures with varying results. Once enabled, there is no way to disable caching.

Low-Level Data Structures

The following are for reference only. The implementation may change.

First is the client RPC handle, defined in `<rpc/clnt.h>`. Low-level implementations must provide and initialize one handle per connection, as shown in Code Example 4-17.

Code Example 4-17 RPC Client Handle Structure

```
typedef struct {
    AUTH *cl_auth;                      /* authenticator */
    struct clnt_ops {
        enum clnt_stat(*cl_call)();    /* call remote procedure */
        void       (*cl_abort)();      /* abort a call */
```

```
        void        (*cl_geterr)();     /* get specific error code */
        bool_t      (*cl_freeres)();    /* frees results */
        void        (*cl_destroy)();    /* destroy this structure */
        bool_t      (*cl_control)();    /* the ioctl() of rpc */
    } *cl_ops;
    caddrt_t    cl_private;             /* private stuff */
    char        *cl_netid;              /* network token *l
    char        *cl_tp;                 /* device name */
} CLIENT;
```

The first field of the client-side handle is an authentication structure, defined in <rpc/auth.h>. By default, it is set to AUTH_NONE. A client program must initialize cl_auth appropriately, as shown in Code Example 4-18.

Code Example 4-18 Client Authentication Handle

```
typedef struct {
    struct      opaque_auth  ah_cred;
    struct      opaque_auth  ah_verf;
    union       des_block    ah_key;
    struct auth_ops {
        void    (*ah_nextverf)();
        int     (*ah_marshal)();        /* nextverf & serialize */
        int     (*ah_validate)();       /* validate varifier */
        int     (*ah_refresh)();        /* refresh credentials */
        void    (*ah_destroy)();        /* destroy this structure */
    } *ah_ops;
    caddr_t ah_private;
} AUTH;
```

In the AUTH structure, ah_cred contains the caller's credentials, and ah_verf contains the data to verify the credentials. See "Authentication" on page 95 for details.

Code Example 4-19 shows the server transport handle.

Code Example 4-19 Server Transport Handle

```
typedef struct {
    int         xp_fd;
#define xp_sock    xp_fd
    u_short xp_port; /* associated port number. Obsoleted */
    struct xp_ops {
        bool_t      (*xp_recv)();     /* receive incoming requests */
       enum xprt_stat (*xp_stat)();   /* get transport status */
        bool_t      (*xp_getargs)();    /* get arguments */
```

```
        bool_t      (*xp_reply)();      /* send reply */
        bool_t      (*xp_freeargs)();   /* free mem alloc for args */
        void        (*xp_destroy)();    /* destroy this struct */
    } *xp_ops;
    int     xp_addrlen;         /* length of remote addr. Obsoleted */
    char    *xp_tp;             /* transport provider device name */
    char    *xp_netid;            /* network token */
    struct netbuf  xp_ltaddr;   /* local transport address */
    struct netbuf  xp_rtaddr;   /* remote transport address */
    char           xp_raddr[16]; /* remote address. Now obsoleted */
    struct opaque_auth xp_verf; /* raw response verifier */
    caddr_t        xp_p1;        /* private: for use by svc ops */
    caddr_t        xp_p2;       /* private: for use by svc ops */
    caddr_t        xp_p3;       /* private: for use by svc lib */
} SVCXPRT;
```

Table 4-4 shows the fields for the server transport handle.

Table 4-4 RPC Server Transport Handle Fields

`xp_fd`	The file descriptor associated with the handle. Two or more server handles can share the same file descriptor.
`xp_netid`	The network identifier (e.g. udp) of the transport on which the handle is created and `xp_tp` is the device name associated with that transport.
`xp_ltaddr`	The server's own bind address.
`xp_rtaddr`	The address of the remote caller (and so may change from call to call).

`xp_netid`
`xp_tp`
`xp_ltaddr`
	Initialized by `svc_tli_create()` and other expert-level routines.

The rest of the fields are initialized by the bottom-level server routines `svc_dg_create()` and `svc_vc_create()`.

For connection-oriented endpoints, the fields in Table 4-5 are not valid until a connection has been requested and accepted for the server:

Table 4-5 RPC Connection-Oriented Endpoints

Fields Not Valid Until Connection Is Accepted		
`xp_fd`	`xp_ops`	`xp_p1`
`xp_p2`	`xp_verf`	`xp_tp`
`xp_ltaddr`	`xp_rtaddr`	`xp_netid`

Testing Programs Using Low-level Raw RPC

There are two pseudo-RPC interface routines that bypass all the network software. The routines shown in Code Example 4-20, `clnt_raw_create()` and `svc_raw_create()`, do not use any real transport.

Note – Do not use raw mode on production systems. Raw mode is intended as a debugging aid only. Raw mode is not MT safe.

Code Example 4-20 is compiled and linked using the following Makefile:

```
all: raw
CFLAGS += -g
raw: raw.o
cc -g -o raw raw.o -lnsl
```

Code Example 4-20 Simple Program Using Raw RPC

```
/*
 * A simple program to increment a number by 1
 */

#include <stdio.h>
#include <rpc/rpc.h>
#include <rpc/raw.h>

#define prognum 0x40000001
#define versnum 1
#define INCR 1
```

```
struct timeval TIMEOUT = {0, 0};
static void server();

main (argc, argv)
    int argc;
    char **argv;
{
    CLIENT *cl;
    SVCXPRT *svc;
    int num = 0, ans;
    int flag;

    if (argc == 2)
        num = atoi(argv[1]);
        svc = svc_raw_create();
    if (svc == (SVCXPRT *) NULL) {
        fprintf(stderr, "Could not create server handle\n");
        exit(1);
    }
    flag = svc_reg( svc, prognum, versnum, server,
            (struct netconfig *) NULL );
    if (flag == 0) {
        fprintf(stderr, "Error: svc_reg failed.\n");
        exit(1);
    }
    cl = clnt_raw_create( prognum, versnum );
    if (cl == (CLIENT *) NULL) {
        clnt_pcreateerror("Error: clnt_raw_create");
        exit(1);
    }
    if (clnt_call(cl, INCR, xdr_int, (caddr_t) &num, xdr_int,
          (caddr_t) &ans, TIMEOUT)
      != RPC_SUCCESS) {
        clnt_perror(cl, "Error: client_call with raw");
        exit(1);
    }
    printf("Client: number returned %d\n", ans);
    exit(0);
}

static void
server(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
```

```
{
    int num;

    fprintf(stderr, "Entering server procedure.\n");

    switch(rqstp->rq_proc) {
        case NULLPROC:
            if (svc_sendreply( transp, xdr_void,
                (caddr_t) NULL) == FALSE) {
                fprintf(stderr, "error in null proc\n");
                exit(1);
            }
            return;
        case INCR:
            break;
        default:
            svcerr_noproc(transp);
            return;
    }
    if (!svc_getargs( transp, xdr_int, &num)) {
        svcerr_decode(transp);
        return;
    }
    fprintf(stderr, "Server procedure: about to increment.\n");
    num++;
    if (svc_sendreply(transp, xdr_int, &num) == FALSE) {
        fprintf(stderr, "error in sending answer\n");
        exit (1);
    }
    fprintf(stderr, "Leaving server procedure.\n");
}
```

Note the following points in Code Example 4-20:

- The server must be created before the client.
- `svc_raw_create()` has no parameters.
- The server is not registered with `rpcbind`. The last parameter to `svc_reg()` is `(struct netconfig *) NULL`, which means that it will not be registered with `rpcbind`.
- `svc_run()` is not called.
- All the RPC calls occur within the same thread of control.
- The server-dispatch routine is the same as for normal RPC servers.

4

Advanced RPC Programming Techniques

This section addresses areas of occasional interest to developers using the lower level interfaces of the RPC package. The topics are:

- `poll()` on the server— how a server can call the dispatcher directly if calling `svc_run()` is not feasible
- Broadcast RPC — how to use the broadcast mechanisms
- Batching —how to improve performance by batching a series of calls
- Authentication — what methods are available in this release
- Port monitors — how to interface with the `inetd` and `listener` port monitors
- Multiple program versions — how to service multiple program versions

`poll()` on the Server Side

This section applies only to servers running RPC in single-threaded (default) mode.

A process that services RPC requests and performs some other activity cannot always call `svc_run()`. If the other activity periodically updates a data structure, the process can set a `SIGALRM` signal before calling `svc_run()`. This allows the signal handler to process the data structure and return control to `svc_run()` when done.

A process can bypass `svc_run()` and access the dispatcher directly with the `svc_getreqset()` call. Given the file descriptors of the transport endpoints associated with the programs being waited on, the process can have its own `poll()` that waits on both the RPC file descriptors and its own descriptors.

Code Example 4-21 shows `svc_run()`. `svc_pollset` is an array of `pollfd` structures that is derived, through a call to `__rpc_select_to_poll()`, from `svc_fdset`. The array can change every time *any* RPC library routine is called, because descriptors are constantly being opened and closed. `svc_getreq_poll()` is called when `poll()` determines that an RPC request has arrived on some RPC file descriptors.

Note – The functions `__rpc_dtbsize()` and `__rpc_select_to_poll()` are not part of the SVID, but they are available in the `libnsl` library. The descriptions of these functions are included here so that you may create versions of these functions for non-Solaris implementations.

```
int __rpc_select_to_poll(int fdmax, fd_set *fdset, struct pollfd
*pollset)
```

Given an `fd_set` pointer and the number of bits to check in it, this function initializes the supplied `pollfd` array for RPC's use. RPC polls only for input events. The number of `pollfd` slots that were initialized is returned.

```
int __rpc_dtbsize()
```

This function calls the `getrlimit()` function to determine the maximum value that the system may assign to a newly created file descriptor. The result is cached for efficiency.

For more information on the SVID routines in this section, see the `rpc_svc_calls(3N)` and the `poll(2)` man pages.

Code Example 4-21 `svc_run()` and `poll()`

```
void
svc_run()
{
    int nfds;
    int dtbsize = __rpc_dtbsize();
    int i;
    struct pollfd svc_pollset[fd_setsize];

    for (;;) {
        /*
         * Check whether there is any server fd on which we may have
         * to wait.
         */
        nfds = __rpc_select_to_poll(dtbsize, &svc_fdset,
                                    svc_pollset);
        if (nfds == 0)
            break;/* None waiting, hence quit */

        switch (i = poll(svc_pollset, nfds, -1)) {
        case -1:
            /*
             * We ignore all errors, continuing with the assumption
             * that it was set by the signal handlers (or any
             * other outside event) and not caused by poll().
             */
        case 0:
            continue;
        default:
            svc_getreq_poll(svc_pollset, i);
```

```
        }
    }
}
```

Broadcast RPC

When an RPC broadcast is issued, a message is sent to all `rpcbind` daemons on a network. An `rpcbind` daemon with which the requested service is registered forwards the request to the server. The main differences between broadcast RPC and normal RPC calls are:

- Normal RPC expects one answer; broadcast RPC expects many answers (one or more answer from each responding machine).
- Broadcast RPC works only on connectionless protocols that support broadcasting, such as UDP.
- With broadcast RPC, all unsuccessful responses are filtered out; so, if there is a version mismatch between the broadcaster and a remote service, the broadcaster never hears from the service.
- Only datagram services registered with `rpcbind` are accessible through broadcast RPC; service addresses may vary from one host to another, so `rpc_broadcast` sends messages to `rpcbind`'s network address.
- The size of broadcast requests is limited by the maximum transfer unit (MTU) of the local network; the MTU for Ethernet is 1500 bytes.

Code Example 4-22 shows how `rpc_broadcast()` is used and describes its arguments.

Code Example 4-22 RPC Broadcast

```
/*
 * bcast.c: example of RPC broadcasting use.
 */

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

main(argc, argv)
    int argc;
    char *argv[];
{
    enum clnt_stat rpc_stat;
    u_long prognum, vers;
    struct rpcent *re;
```

```
    if(argc != 3) {
        fprintf(stderr, "usage : %s RPC_PROG VERSION\n", argv[0]);
        exit(1);
    }
    if (isdigit( *argv[1]))
        prognum = atoi(argv[1]);
    else {
        re = getrpcbyname(argv[1]);
        if (! re) {
            fprintf(stderr, "Unknown RPC service %s\n", argv[1]);
            exit(1);
        }
        prognum = re->r_number;
    }
    vers = atoi(argv[2]);
    rpc_stat = rpc_broadcast(prognum, vers, NULLPROC, xdr_void,
             (char *)NULL, xdr_void, (char *)NULL, bcast_proc, NULL);
    if ((rpc_stat != RPC_SUCCESS) && (rpc_stat != RPC_TIMEDOUT)) {
        fprintf(stderr, "broadcast failed: %s\n",
                  clnt_sperrno(rpc_stat));
        exit(1);
    }
    exit(0);
}
```

The function in Code Example 4-23 collects replies to the broadcast. Normal operation is to collect either the first reply or all replies. `bcast_proc` displays the IP address of the server that has responded. Since the function returns `FALSE` it will continue to collect responses, and the RPC client code will continue to resend the broadcast until it times out.

Code Example 4-23 Collect Broadcast Replies

```
bool_t
bcast_proc(res, t_addr, nconf)
    void *res;                                  /* Nothing comes back */
    struct t_bind *t_addr;                    /* Who sent us the reply */
    struct netconfig *nconf;
{
    register struct hostent *hp;
    char *naddr;

    naddr = taddr2naddr(nconf, &taddr->addr);
    if (naddr == (char *) NULL) {
```

```
            fprintf(stderr,"Responded: unknown\n");
        } else {
            fprintf(stderr,"Responded: %s\n", naddr);
            free(naddr);
        }
        return(FALSE);
}
```

If `done` is `TRUE`, then broadcasting stops, and `rpc_broadcast()` returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no responses come back, the routine returns with `RPC_TIMEDOUT`.

Batching

RPC is designed so that clients send a call message and wait for servers to reply to the call. This implies that a client is blocked while the server processes the call. This is inefficient when the client does not need each message acknowledged.

RPC batching lets clients process asynchronously. RPC messages can be placed in a pipeline of calls to a server. Batching requires that:

- The server does not respond to any intermediate message.
- The pipeline of calls is transported on a reliable transport, such as TCP.
- The result's XDR routine in the calls must be `NULL`.
- The RPC call's time-out must be zero.

Because the server does not respond to each call, the client can send new calls in parallel with the server processing previous calls. The transport can buffer many call messages and send them to the server in one `write`() system call. This decreases interprocess communication overhead and the total time of a series of calls. The client should end with a nonbatched call to flush the pipeline.

Code Example 4-24 shows the unbatched version of the client. It scans the character array, `buf`, for delimited strings and sends each string to the server.

Code Example 4-24 Unbatched Client

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "windows.h"
```

```
main(argc, argv)
    int argc;
    char **argv;
{
    struct timeval total_timeout;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    char buf[1000], *s = buf;

    if ((client = clnt_create( argv[1], WINDOWPROG, WINDOWVERS,
      "circuit_v")) == (CLIENT *) NULL) {
        clnt_pcreateerror("clnt_create");
        exit(1);
    }

    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    while (scanf( "%s", s ) != EOF) {
        if (clnt_call(client, RENDERSTRING, xdr_wrapstring, &s,
           xdr_void, (caddr_t) NULL, total_timeout) != RPC_SUCCESS) {
            clnt_perror(client, "rpc");
            exit(1);
        }
    }

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

Code Example 4-25 shows the batched version of the client. It does not wait after each string is sent to the server. It waits only for an ending response from the server.

Code Example 4-25 Batched Client

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "windows.h"

main(argc, argv)
    int argc;
    char **argv;
{
    struct timeval total_timeout;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
```

```
    char buf[1000], *s = buf;

    if ((client = clnt_create( argv[1], WINDOWPROG, WINDOWVERS,
      "circuit_v")) == (CLIENT *) NULL) {
        clnt_pcreateerror("clnt_create");
        exit(1);
    }
    timerclear(&total_timeout);
    while (scanf("%s", s) != EOF)
        clnt_call(client, RENDERSTRING_BATCHED, xdr_wrapstring,
                  &s, xdr_void, (caddr_t) NULL, total_timeout);
    /* Now flush the pipeline */
    total_timeout.tv_sec = 20;
    clnt_stat = clnt_call(client, NULLPROC, xdr_void,
           (caddr_t) NULL, xdr_void, (caddr_t) NULL, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "rpc");
        exit(1);
    }
    clnt_destroy(client);
    exit(0);
}
```

Code Example 4-26 shows the dispatch portion of the batched server. Because the server sends no message, the clients are not notified of failures.

Code Example 4-26 Batched Server

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "windows.h"

void
windowdispatch(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    char    *s = NULL;

    switch(rqstp->rq_proc) {
        case NULLPROC:
            if (!svc_sendreply( transp, xdr_void, NULL))
                fprintf(stderr, "can't reply to RPC call\n");
            return;
        case RENDERSTRING:
            if (!svc_getargs( transp, xdr_wrapstring, &s)) {
```

```
                fprintf(stderr, "can't decode arguments\n");
                /* Tell caller an error occurred */
                svcerr_decode(transp);
                break;
            }
            /* Code here to render the string s */
            if (!svc_sendreply( transp, xdr_void, (caddr_t) NULL))
                fprintf( stderr, "can't reply to RPC call\n");
            break;
        case RENDERSTRING_BATCHED:
            if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                fprintf(stderr, "can't decode arguments\n");
                /* Be silent in the face of protocol errors */
                break;
            }
            /* Code here to render string s, but send no reply! */
            break;
        default:
            svcerr_noproc(transp);
            return;
    }
    /* Now free string allocated while decoding arguments */
    svc_freeargs(transp, xdr_wrapstring, &s);
}
```

Batching Performance

To illustrate the benefits of batching, the examples in Code Example 4-24, Code Example 4-25, and Code Example 4-26 were completed to render the lines in a 25144-line file. The rendering service simply throws the lines away. The batched version of the application was four times as fast as the unbatched version.

Authentication

In all of the preceding examples in this chapter, the caller has not identified itself to the server, and the server has not required identification of the caller. Some network services, such as a network file system, require caller identification. Refer to *System Administration Guide, Volume I*, to implement any of the authentication methods described in this section.

Just as different transports can be used when creating RPC clients and servers, different “flavors” of authentication can be associated with RPC clients. The authentication subsystem of RPC is open ended. So, many flavors of authentication can be supported. The authentication protocols are further defined in Appendix B, “RPC Protocol and Language Specification."

Sun RPC currently supports the authentication flavors shown in Table 4-6.

Table 4-6 Authentication Methods Supported By Sun RPC

`AUTH_NONE`	Default. No authentication performed
`AUTH_SYS`	An authentication flavor based on UNIX operating system, process permissions authentication
`AUTH_SHORT`	An alternate flavor of AUTH_SYS used by some servers for efficiency. Client programs using AUTH_SYS authentication can receive AUTH_SHORT response verifiers from some servers. See Appendix B, “RPC Protocol and Language Specification for details
`AUTH_DES`	An authentication flavor based on DES encryption techniques
`AUTH_KERB`	Version 4 Kerberos authentication based on DES framework

When a caller creates a new RPC client handle as in:

```
clnt = clnt_create(host, prognum, versnum, nettype);
```

the appropriate client-creation routine sets the associated authentication handle to:

```
clnt->cl_auth = authnone_create();
```

If you create a new instance of authentication, you must destroy it with `auth_destroy(clnt->cl_auth)`. This should be done to conserve memory.

On the server side, the RPC package passes the service-dispatch routine a request that has an arbitrary authentication style associated with it. The request handle passed to a service-dispatch routine contains the structure `rq_cred`. It is opaque, except for one field: the flavor of the authentication credentials.

```
/*
 * Authentication data
 */
struct opaque_auth {
    enum_t    oa_flavor;          /* style of credentials */
```

```
    caddr_t    oa_base;            /* address of more auth stuff */
    u_int      oa_length;          /* not to exceed MAX_AUTH_BYTES */
};
```

The RPC package guarantees the following to the service-dispatch routine:

- The `rq_cred` field in the `svc_req` structure is well formed. So, you can check `rq_cred.oa_flavor` to get the flavor of authentication. You can also check the other fields of `rq_cred` if the flavor is not supported by RPC.
- The `rq_clntcred` field that is passed to service procedures is either `NULL` or points to a well-formed structure that corresponds to a supported flavor of authentication credential. There is no authentication data for the `AUTH_NONE` flavor. `rq_clntcred` can be cast only as a pointer to an `authsys_parms`, `short_hand_verf`, `authkerb_cred`, or `authdes_cred` structure.

`AUTH_SYS` *Authentication*

The client can use `AUTH_SYS` (called `AUTH_UNIX` in previous releases) style authentication by setting `clnt->cl_auth` after creating the RPC client handle:

```
clnt->cl_auth = authsys_create_default();
```

This causes each RPC call associated with `clnt` to carry with it the following credentials-authentication structure shown in Code Example 4-27.

Code Example 4-27 `AUTH_SYS` Credential Structure

```
/*
 * AUTH_SYS flavor credentials.
 */
struct authsys_parms {
    u_long aup_time;                /* credentials creation time */
    char *aup_machname;             /* client's host name */
    uid_t aup_uid;                  /* client's effective uid */
    gid_t aup_gid;                  /* client's current group id */
    u_int aup_len;                  /* element length of aup_gids*/
    gid_t *aup_gids;               /* array of groups user is in */
};
```

`rpc.broadcast` defaults to `AUTH_SYS` authentication.

Code Example 4-28 shows a server, with procedure `RUSERPROC_1`, that returns the number of users on the network. As an example of authentication, it checks `AUTH_SYS` credentials and does not service requests from callers whose `uid` is `16`.

Code Example 4-28 Authentication Server

```
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    struct authsys_parms *sys_cred;
    uid_t uid;
    unsigned long nusers;

    /* NULLPROC should never be authenticated */
    if (rqstp->rq_proc == NULLPROC) {
        if (!svc_sendreply( transp, xdr_void, (caddr_t) NULL))
            fprintf(stderr, "can't reply to RPC call\n");
        return;
    }

    /* now get the uid */
    switch(rqstp->rq_cred.oa_flavor) {
        case AUTH_SYS:
            sys_cred = (struct authsys_parms *) rqstp->rq_clntcred;
            uid = sys_cred->aup_uid;
            break;
        default:
            svcerr_weakauth(transp);
            return;
    }
    switch(rqstp->rq_proc) {
        case RUSERSPROC_1:
            /* make sure caller is allowed to call this proc */
            if (uid == 16) {
                svcerr_systemerr(transp);

                return;
            }
            /*
             * Code here to compute the number of users and assign it
             * to the variable nusers
             */
            if (!svc_sendreply( transp, xdr_u_long, &nusers))
                fprintf(stderr, "can't reply to RPC call\n");
```

```
            return;
        default:
            svcerr_noproc(transp);
            return;
    }
}
```

Note the following:

- The authentication parameters associated with the `NULLPROC` (procedure number zero) are usually not checked.
- The server calls `svcerr_weakauth()` if the authentication parameter's flavor is too weak; there is no way to get the list of authentication flavors the server requires.
- The service protocol should return status for access denied; in Code Example 4-28, the protocol calls the service primitive `svcerr_systemerr()`, instead.

The last point underscores the relation between the RPC authentication package and the services: RPC deals only with *authentication* and not with an individual service's *access control*. The services themselves must establish access-control policies and reflect these policies as return statuses in their protocols.

`AUTH_DES` *Authentication*

Use `AUTH_DES` authentication for programs that require more security than `AUTH_SYS` provides. `AUTH_SYS` authentication is easy to defeat. For example, instead of using `authsys_create_default()`, a program can call `authsys_create()` and change the RPC authentication handle to give itself any desired user ID and hostname.

`AUTH_DES` authentication requires that `keyserv`() daemons are running on both the server and client hosts. The NIS or NIS+ naming service must also be running. Users on these hosts need public/secret key pairs assigned by the network administrator in the `publickey`() database. They must also have decrypted their secret keys with the `keylogin`() command (normally done by `login`() unless the login password and secure-RPC password differ).

To use `AUTH_DES` authentication, a client must set its authentication handle appropriately. For example:

```
cl->cl_auth = authdes_seccreate(servername, 60, server,
        (char *)NULL);
```

The first argument is the network name or "netname" of the owner of the server process. Server processes are usually root processes, and you can get their netnames with the following call:

```
char servername[MAXNETNAMELEN];
host2netname(servername, server, (char *)NULL);
```

`servername` points to the receiving string and `server` is the name of the host the server process is running on. If the server process was run by a non-root user, use the call `user2netname()` as follows:

```
char servername[MAXNETNAMELEN];
user2netname(servername, serveruid(), (char *)NULL);
```

`serveruid()` is the user id of the server process. The last argument of both functions is the name of the domain that contains the server. `NULL` means "use the local domain name."

The second argument of `authdes_seccreate()` is the lifetime (known also as the window) of the client's credential, here, 60 seconds. A credential will expire 60 seconds after the client makes an RPC call. If a program tries to reuse the credential, the server RPC subsystem recognizes that it has expired and does not service the request carrying the expired credential. If any program tries to reuse a credential within its lifetime, it is rejected, because the server RPC subsystem saves credentials it has seen in the near past and does not serve duplicates.

The third argument of `authdes_seccreate()` is the name of the *timehost* used to synchronize clocks. `AUTH_DES` authentication requires that server and client agree on the time. The example specifies to synchronize with the server. A `(char *)NULL` says not to synchronize. Do this only when you are sure that the client and server are already synchronized.

The fourth argument of `authdes_seccreate()` points to a DES encryption key to encrypt time stamps and data. If this argument is `(char *)NULL`, as it is in this example, a random key is chosen. The `ah_key` field of the authentication handle contains the key.

The server side is simpler than the client. Code Example 4-29 shows the server in Code Example 4-28 changed to use `AUTH_DES`.

Code Example 4-29 AUTH_DES Server

```
#include <rpc/rpc.h>
    ...
    ...
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    struct authdes_cred *des_cred;
    uid_t uid;
    gid_t gid;
    int gidlen;
    gid_t gidlist[10];

    /* NULLPROC should never be authenticated */
    if (rqstp->rq_proc == NULLPROC) {
        /* same as before */
    }
    /* now get the uid */
    switch(rqstp->rq_cred.oa_flavor) {
        case AUTH_DES:
            des_cred = (struct authdes_cred *) rqstp->rq_clntcred;
            if (! netname2user( des_cred->adc_fullname.name, &uid,
                                &gid, &gidlen, gidlist)) {
                fprintf(stderr, "unknown user: %s\n",
                         des_cred->adc_fullname.name);
                svcerr_systemerr(transp);
                return;
            }
            break;
        default:
            svcerr_weakauth(transp);
            return;
    }
    /* The rest is the same as before */
```

Note the routine `netname2user()` converts a network name (or “netname” of a user) to a local system ID. It also supplies group IDs (not used in this example).

AUTH_KERB *Authentication*

SunOS 5.x includes support for most client-side features of Kerberos 4.0, except `klogin`. `AUTH_KERB` is conceptually similar to `AUTH_DES`; the essential difference is that `DES` passes a network name and DES-encrypted session key, while Kerberos passes the encrypted service ticket. The other factors that affect implementation and interoperability are given in the following subsections.

For more information, see the `kerberos(3N)` man page and the Steiner-Neuman-Shiller paper[1] on the MIT Project Athena implementation of Kerberos. You may access MIT documentation through the FTP directory `/pub/kerberos/doc` on `athena-dist.mit.edu`, or through Mosaic, using the document URL, `ftp://athena-dist.mit.edu/pub/kerberos/doc`.

Time Synchronization

Kerberos uses the concept of a time window in which its credentials are valid. It does not place restrictions on the clocks of the client or server. The client is required to determine the time bias between itself and the server and compensate for the difference by adjusting the window time specified to the server. Specifically, the *window* is passed as an argument to `authkerb_seccreate()`; the window does not change. If a *timehost* is specified as an argument, the client side gets the time from the *timehost* and alters its timestamp by the difference in time. Various methods of time synchronization are available. See the `kerberos_rpc(3N)` man page for more information.

Well-Known Names

Kerberos users are identified by a primary name, instance, and realm. The RPC authentication code ignores the realm and instance, while the Kerberos library code does not. The assumption is that user names are the same between client and server. This enables a server to translate a primary name into user identification information. Two forms of well-known names are used (omitting the realm):

- `root`.*host* represents a privileged user on client *host*.

1. Steiner, Jennifer G., Neuman, Clifford, and Schiller, Jeffrey J. "Kerberos: An Authentication Service for Open Network Systems." *USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, June 1988.

- *user*.`ignored` represents the user whose user name is *user*. The instance is ignored.

Encryption

Kerberos uses cipher block chaining (CBC) mode when sending a full name credential (one that includes the ticket and window), and electronic code book (ECB) mode otherwise. CBC and ECB are two methods of DES encryption. See the `des_crypt(3)` man page for more information. The session key is used as the initial input vector for CBC mode. The notation

```
xdr_type(object)
```

means that XDR is used on `object` as a `type`. The length in the next code section is the size, in bytes of the credential or verifier, rounded up to 4-byte units. The full name credential and verifier are obtained as follows:

```
xdr_long(timestamp.seconds)
xdr_long(timestamp.useconds)
xdr_long(window)
xdr_long(window - 1)
```

After encryption with CBC with input vector equal to the session key, the output is two DES cipher blocks:

```
CB0
CB1.low
CB1.high
```

The credential is:

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_enum(AKN_FULLNAME)
xdr_bytes(ticket)
xdr_opaque(CB1.high)
```

The verifier is:

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_opaque(CB0)
xdr_opaque(CB1.low)
```

The nickname exchange yields:

```
xdr_long(timestamp.seconds)
xdr_long(timestamp.useconds)
```

The nickname is encrypted with ECB to obtain `ECB0`, and the credential is:

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_enum(AKN_NICKNAME)
xdr_opaque(akc_nickname)
```

The verifier is:

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_opaque(ECB0)
xdr_opaque(0)
```

Using Port Monitors

RPC servers can be started by port monitors such as `inetd` and `listen`. Port monitors listen for requests and spawn servers in response. The forked server process is passed file descriptor `0` on which the request has been accepted. For `inetd`, when the server is done, it may exit immediately or wait a given interval for another service request.

For `listen`, servers should exit immediately after replying because `listen` always spawns a new process. The following function call creates a `SVCXPRT` handle to be used by the services started by port monitors:

```
transp = svc_tli_create(0, nconf, (struct t_bind *)NULL, 0, 0)
```

`nconf` is the `netconfig` structure of the transport from which the request is received.

Because the port monitors have already registered the service with `rpcbind`, there is no need for the service to register with `rpcbind`. But it must call `svc_reg()` to register the service procedure:

```
svc_reg(transp, PROGNUM, VERSNUM, dispatch,(struct netconfig *)NULL)
```

The `netconfig` structure here is `NULL` to prevent `svc_reg` from registering the service with `rpcbind`.

Note – Study `rpcgen`-generated server stubs to see the sequence in which these routines are called.

For connection-oriented transports, the following routine provides a lower level interface:

```
transp = svc_fd_create(0, recvsize, sendsize);
```

A `0` file descriptor is the first argument. You can set the value of `recvsize` and `sendsize` to any appropriate buffer size. A `0` for either argument causes a system default size to be chosen. Application servers that do not do any listening of their own use `svc_fd_create()`.

Using `inetd`

Entries in `/etc/inet/inetd.conf` have different formats for socket-based, TLI-based, and RPC services. The format of `inetd.conf` entries for RPC services is:

```
rpc_prog/vers endpoint_type rpc/proto flags user pathname args
```

where:

Table 4-7 RPC `inetd` Services

rpc_prog/vers	The name of an RPC program followed by a `/` and the version number or a range of version numbers.
endpoint_type	One of `dgram` (for connectionless sockets), `stream` (for connection mode sockets), or `tli` (for TLI endpoints).
proto	May be * (for all supported transports), a nettype, a netid, or a comma separated list of nettype and netid.
flags	Either `wait` or `nowait`.
user	Must exist in the effective passwd database.
pathname	Full path name of the server daemon.
args	Arguments to be passed to the daemon on invocation.

For example:

```
rquotad/1 tli rpc/udp wait root /usr/lib/nfs/rquotad rquotad
```

For more information, see the `inetd.conf(4)` man page.

Using the Listener

Use `pmadm` to add RPC services:

```
pmadm -a -p pm_tag -s svctag -i id -v ver \
    -m `nlsadmin -c command -D -R prog:vers`
```

The arguments are: -a means to add a service, -p *pm_tag* specifies a tag associated with the port monitor providing access to the service, -s *svctag* is the server's identifying code, -i *id* is the /etc/passwd user name assigned to service *svctag*, -v *ver* is the version number for the port monitor's data base file, and -m specifies the nlsadmin command to invoke the service. nlsadmin can have additional arguments. For example, to add version 1 of a remote program server named rusersd, a pmadm command is:

```
# pmadm -a -p tcp -s rusers -i root -v 4 \
    -m `nlsadmin -c /usr/sbin/rpc.ruserd -D -R 100002:1`
```

The command is given root permissions, installed in version 4 of the listener data base file, and is made available over TCP transports. Because of the complexity of the arguments and options to pmadm, use a command script or the menu system to add RPC services. To use the menu system, enter sysadm ports and choose the port_services option.

After adding a service, the listener must be re-initialized before the service is available. To do this, stop and restart the listener, as follows (note that rpcbind must be running):

```
# sacadm -k -p pmtag
# sacadm -s -p pmtag
```

For more information, such as how to set up the listener process, see the listen(1M), pmadm(1M), sacadm(1M) and sysadm(1M) man pages and the *TCP/IP and Data Communications Administration Guide*.

Multiple Server Versions

By convention, the first version number of a program, PROG, is named PROGVERS_ORIG and the most recent version is named PROGVERS. Program version numbers must be assigned consecutively. Leaving a gap in the program version sequence can cause the search algorithm to not find a matching program version number that is defined.

Version numbers should never be changed by anyone other than the owner of a program. Adding a version number to a program that you do not own can cause severe problems when the owner increments the version number. Sun registers version numbers and answers questions about them (rpc@Sun.com).

Suppose a new version of the `ruser` program returns an `unsigned short` rather than a `long`. If you name this version `RUSERSVERS_SHORT`, a server that wants to support both versions would do a double register. The same server handle is used for both registrations.

Code Example 4-30 Server Handle for Two Versions of Single Routine

```
if (!svc_reg(transp, RUSERSPROG, RUSERSVERS_ORIG, nuser, nconf)) {
    fprintf(stderr, "can't register RUSER service\n");
    exit(1);
}
if (!svc_reg(transp, RUSERSPROG, RUSERSVERS_SHORT, nuser, nconf)) {
    fprintf(stderr, "can't register RUSER service\n");
    exit(1);
}
```

Both versions can be performed by a single procedure.

Code Example 4-31 Procedure for Two Versions of Single Routine

```
void
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;
    unsigned short nusers2;
    switch(rqstp->rq_proc) {
        case NULLPROC:
            if (!svc_sendreply( transp, xdr_void, 0))
                fprintf(stderr, "can't reply to RPC call\n");
            return;
        case RUSERSPROC_NUM:
            /*
             * Code here to compute the number of users
             * and assign it to the variable nusers
             */
        switch(rqstp->rq_vers) {
            case RUSERSVERS_ORIG:
                if (! svc_sendreply( transp, xdr_u_long, &nusers))
                    fprintf(stderr, "can't reply to RPC call\n");
                break;
            case RUSERSVERS_SHORT:
                nusers2 = nusers;
                if (! svc_sendreply( transp, xdr_u_short, &nusers2))
                    fprintf(stderr, "can't reply to RPC call\n");
```

```
                break;
            }
            default:
                svcerr_noproc(transp);
                return;
        }
        return;
}
```

Multiple Client Versions

Since different hosts may run different versions of RPC servers, a client should be capable of accommodating the variations. For example, one server may run the old version of `RUSERSPROG(RUSERSVERS_ORIG)` while another server runs the newer version `(RUSERSVERS_SHORT)`.

If the version on a server does not match the version number in the client creation call, `clnt_call` fails with `an RPCPROGVERSMISMATCH` error. You can get the version numbers supported by a server and then create a client handle with the appropriate version number. Use either the routine in Code Example 4-32, or `clnt_create_vers()`. See the `rpc(3N)` man page for more details.

Code Example 4-32 RPC Versions on Client Side

```
main()
{
    enum clnt_stat status;
    u_short num_s;
    u_int num_l;
    struct rpc_err rpcerr;
    int maxvers, minvers;
    CLIENT *clnt;

    clnt = clnt_create("remote", RUSERSPROG, RUSERSVERS_SHORT,
                       "datagram_v");
    if (clnt == (CLIENT *) NULL) {
        clnt_pcreateerror("unable to create client handle");
        exit(1);
    }
    to.tv_sec = 10;                          /* set the time outs */
    to.tv_usec = 0;

    status = clnt_call(clnt, RUSERSPROC_NUM, xdr_void,
```

```
                (caddr_t) NULL, xdr_u_short, (caddr_t)&num_s, to);
    if (status == RPC_SUCCESS) {    /* Found latest version number */
        printf("num = %d\n", num_s);
        exit(0);
    }
    if (status != RPC_PROGVERSMISMATCH) {      /* Some other error */
        clnt_perror(clnt, "rusers");
        exit(1);
    }
    /* This version not supported */
    clnt_geterr(clnt, &rpcerr);
    maxvers = rpcerr.re_vers.high;     /*highest version supported */
    minvers = rpcerr.re_vers.low;       /*lowest version supported */
    if (RUSERSVERS_SHORT < minvers || RUSERSVERS_SHORT > maxvers) {
                                /* doesn't meet minimum standards */
        clnt_perror(clnt, "version mismatch");
        exit(1);
    }
    (void) clnt_control(clnt, CLSET_VERSION, RUSERSVERS_ORIG);
    status = clnt_call(clnt, RUSERSPROC_NUM, xdr_void,
             (caddr_t) NULL, xdr_u_long, (caddr_t)&num_l, to);
    if (status == RPC_SUCCESS)
                        /* We found a version number we recognize */
        printf("num = %d\n", num_l);
    else {
        clnt_perror(clnt, "rusers");
        exit(1);
    }
}
```

Using Transient RPC Program Numbers

Occasionally, it is useful for an application to use RPC program numbers that are generated dynamically. This could be used for implementing callback procedures, for example. In the callback example, a client program typically registers an RPC service using a dynamically generated, or transient, RPC program number and passes this on to a server along with a request. The server will then call back the client program using the transient RPC program number in order to supply the results. Such a mechanism may be necessary if processing the client's request will take a huge amount of time and the client cannot block (assuming it is single-threaded); in this case, the server will acknowledge the client's request, and call back later with the results. Another use of callbacks is to generate periodic reports from a server; the client makes

an RPC call to start the reporting, and the server periodically calls back the client with reports using the transient RPC program number supplied by the client program.

Dynamically generated, or transient, RPC program numbers are in the transient range, `0x40000000 - 0x5fffffff`. The following routine creates a service based on a transient RPC program for a given transport type. The service handle and the transient rpc program number are returned. The caller supplies the service dispatch routine, the version, and the transport type.

Code Example 4-33 Transient RPC Program—Server Side

```
SVCXPRT *
register_transient_prog(dispatch, program, version, netid)
    void (*dispatch)(); /* service dispatch routine */
    u_long *program;    /* returned transient RPC number */
    u_long version;     /* program version */
    char *netid;        /* transport id */
{
    SVCXPRT  *transp;
    struct netconfig *nconf;
    u_long prognum;

    if ((nconf = getnetconfigent(netid)) == (struct netconfig *)NULL)
        return ((SVCXPRT *)NULL);

    if ((transp = svc_tli_create(RPC_ANYFD, nconf,
                (struct t_bind *)NULL, 0, 0)) == (SVCXPRT *)NULL) {
        freenetconfigent(nconf);
        return ((SVCXPRT *)NULL);
    }

    prognum = 0x40000000;
    while (prognum < 0x60000000 && svc_reg(transp, prognum, version,
                                dispatch, nconf) == 0) {
        prognum++;
    }

    freenetconfigent(nconf);
    if (prognum >= 0x60000000) {
        svc_destroy(transp);
        return ((SVCXPRT *)NULL);
    }
    *program = prognum;
    return (transp);

}
```

Multithreaded RPC Programming

This manual does not cover basic topics and code examples for the Solaris implementation of multithread programming. Instead, refer to the *Multithreaded Programming Guide* for background on the following topics:

- Thread creation
- Scheduling
- Synchronization
- Signals
- Process resources
- Light-weight processes (lwp)
- Concurrency
- Data locking strategies

TI-RPC supports multithreaded RPC servers in Solaris 2.4 and higher. The difference between a multithreaded server and a single-threaded server is that a multithreaded server uses threading technology to process incoming client requests concurrently. Multithreaded servers can have higher performance and availability compared with single-threaded servers.

The section “MT Server Overview” on page 116 is a good place to start reading about the interfaces available in this release.

MT Client Overview

In a multithread client program, a thread can be created to issue each RPC request. When multiple threads share the same client handle, only one thread at a time will be able to make an RPC request. All other threads will wait until the outstanding request is complete. On the other hand, when multiple threads make RPC requests using different client handles, the requests are carried out concurrently. Figure 4-1 illustrates a possible timing of a multithreaded client implementation consisting of two client threads using different client handles.

Code Example 4-34 shows the client side implementation of a multithreaded `rstat` program. The client program creates a thread for each host. Each thread creates its own client handle and makes various RPC calls to the given host. Because the client threads are using different handles to make the RPC calls, they can carry out the RPC calls concurrently.

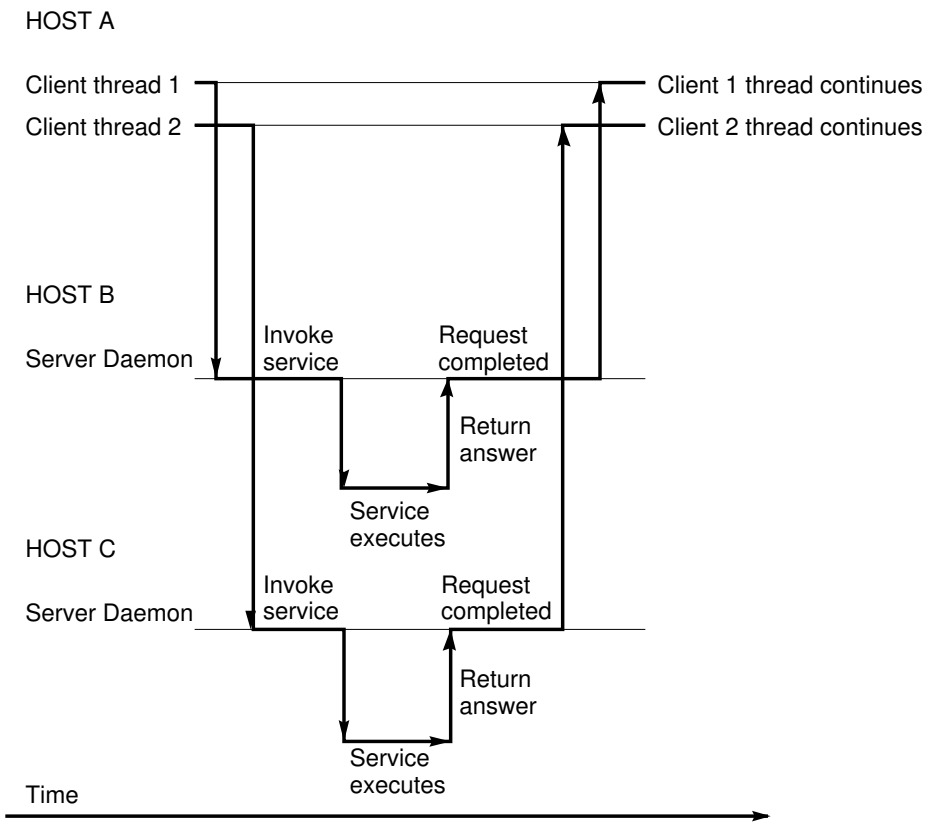

Figure 4-1 Two Client Threads Using Different Client Handles (Real time)

Note – You must link in the thread library when writing any RPC multi-threaded-safe application. The thread library must be the last named library on the link line. To do this, specify the `-lthread` option in the compile command.

Compile the program in Code Example 4-34 by typing:

```
$ cc rstat.c -lnsl -lthread
```

Code Example 4-34 Client for MT `rstat`

```
/* @(#)rstat.c 2.3 93/11/30 4.0 RPCSRC */
/*
 * Simple program that prints the status of a remote host, in a
 * format similar to that used by the 'w' command.
 */

#include <thread.h>/* thread interfaces defined */
#include <synch.h>/* mutual exclusion locks defined */
#include <stdio.h>
#include <sys/param.h>
#include <rpc/rpc.h>
#include <rpcsvc/rstat.h>
#include <errno.h>

mutex_t tty;       /* control of tty for printf's */
cond_t cv_finish;
int count = 0;

main(argc, argv)
int argc;
char **argv;
{
    int i;
    thread_t tid;
    void *do_rstat();

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

    mutex_lock(&tty);

    for (i = 1; i < argc; i++) {
        if (thr_create(NULL, 0, do_rstat, argv[i], 0, &tid) < 0) {
            fprintf(stderr, "thr_create failed: %d\n", i);
            exit(1);
        } else
            fprintf(stderr, "tid: %d\n", tid);
    }

    while (count < argc-1) {
        printf("argc = %d, count = %d\n", argc-1, count);
        cond_wait(&cv_finish, &tty);
```

```
    }

    exit(0);
}

bool_t rstatproc_stats();

void *
do_rstat(host)
char *host;
{
    CLIENT *rstat_clnt;
    statstime host_stat;
    bool_t rval;
    struct tm *tmp_time;
    struct tm host_time;
    struct tm host_uptime;
    char days_buf[16];
    char hours_buf[16];

    mutex_lock(&tty);
    printf("%s: starting\n", host);
    mutex_unlock(&tty);

    /* client handle to rstat */
    rstat_clnt = clnt_create(host, RSTATPROG, RSTATVERS_TIME,
                             "udp");
    if (rstat_clnt == NULL) {
        mutex_lock(&tty);      /* get control of tty */
        clnt_pcreateerror(host);
        count++;
        cond_signal(&cv_finish);
        mutex_unlock(&tty);/* release control of tty */

        thr_exit(0);

    }

    rval = rstatproc_stats(NULL, &host_stat, rstat_clnt);
    if (!rval) {
        mutex_lock(&tty);/* get control of tty */
        clnt_perror(rstat_clnt, host);
        count++;
        cond_signal(&cv_finish);
        mutex_unlock(&tty);/* release control of tty */
```

```
    thr_exit(0);

}

tmp_time = localtime_r(&host_stat.curtime.tv_sec, &host_time);

host_stat.curtime.tv_sec -= host_stat.boottime.tv_sec;

tmp_time = gmtime_r(&host_stat.curtime.tv_sec, &host_uptime);

if (host_uptime.tm_yday != 0)
    sprintf(days_buf, "%d day%s, ", host_uptime.tm_yday,
    (host_uptime.tm_yday > 1) ? "s" : "");
else
    days_buf[0] = '\0';

if (host_uptime.tm_hour != 0)
    sprintf(hours_buf, "%2d:%02d,",
        host_uptime.tm_hour, host_uptime.tm_min);

else if (host_uptime.tm_min != 0)
    sprintf(hours_buf, "%2d mins,", host_uptime.tm_min);
else

    hours_buf[0] = '\0';

mutex_lock(&tty);/* get control of tty */
printf("%s: ", host);
printf(" %2d:%02d%cm up %s%s load average: %.2f %.2f %.2f\n",
    (host_time.tm_hour > 12)  ? host_time.tm_hour - 12

    : host_time.tm_hour,
    host_time.tm_min,
    (host_time.tm_hour >= 12) ? 'p'
    : 'a',
    days_buf,
    hours_buf,
    (double)host_stat.avenrun[0]/FSCALE,
    (double)host_stat.avenrun[1]/FSCALE,
    (double)host_stat.avenrun[2]/FSCALE);
count++;
cond_signal(&cv_finish);
mutex_unlock(&tty);/* release control of tty */
clnt_destroy(rstat_clnt);
```

```
    sleep(10);
    thr_exit(0);
}

/*
Client side implementation of MT rstat program
*/

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

bool_t
rstatproc_stats(argp, clnt_resp, clnt)
    void *argp;
    statstime *clnt_resp;
    CLIENT *clnt;
{

    memset((char *)clnt_resp, 0, sizeof (statstime));
    if (clnt_call(clnt, RSTATPROC_STATS,
        (xdrproc_t) xdr_void, (caddr_t) argp,
        (xdrproc_t) xdr_statstime, (caddr_t) clnt_resp,
        TIMEOUT) != RPC_SUCCESS) {
        return (FALSE);

    }
    return (TRUE);
}
```

MT Server Overview

Prior to Solaris 2.4, RPC servers were single threaded. That is, they process client requests sequentially, as the requests come in. For example, if two requests come in, and the first takes 30 seconds to process, and the second takes only 1 second to process, the client that made the second request will still have to wait for the first request to complete before it receives a response. This is not desirable, especially in a multiprocessor server environment, where each CPU could be processing a different request simultaneously; or in a situation where one request is waiting for I/O to complete, other requests could be processed by the server.

Solaris 2.4 and higher provides facilities in the RPC library for service developers to create multithreaded servers that deliver better performance to end users. Two modes of server multithreading are supported in TI-RPC: the Automatic MT mode and the User MT mode.

In the Auto mode, the server automatically creates a new thread for every incoming client request. This thread processes the request, sends a response, and exits. In the User mode, the service developer decides how to create and manage threads for concurrently processing the incoming client requests. The Auto mode is much easier to use than the User mode, but the User mode offers more flexibility for service developers with special requirements.

Note – You must link in the thread library when writing RPC multithreaded-safe applications. The thread library must be the last named library on the link line. To do this, specify the `-lthread` option in the compile command.

The two calls that support server multithreading are `rpc_control()` and `svc_done()`. The `rpc_control()` call is used to set the MT mode, either Auto or User mode. If the server uses Auto mode, it does not need to invoke `svc_done()` at all. In User mode, `svc_done()` must be invoked after each client request is processed, so that the server can reclaim the resources from processing the request. In addition, multithreaded RPC servers must call on `svc_run()`. Note that `svc_getreqpoll()` and `svc_getreqset()` are unsafe in MT applications.

Note – If the server program does not invoke any of the MT interface calls, it remains in single-threaded mode, which is the default mode.

You are required to make RPC server procedures multithreaded safe regardless of which mode the server is using. Usually, this means that all static and global variables need to be protected with mutex locks. Mutual exclusion and other synchronization APIs are defined in `synch.h`. See the `condition(3T)`, `rwlock(3T)`, and `mutex(3T)` man pages for a list of the various synchronization interfaces.

Figure 4-2 illustrates a possible timing of a server implemented in one of the MT modes of operation.

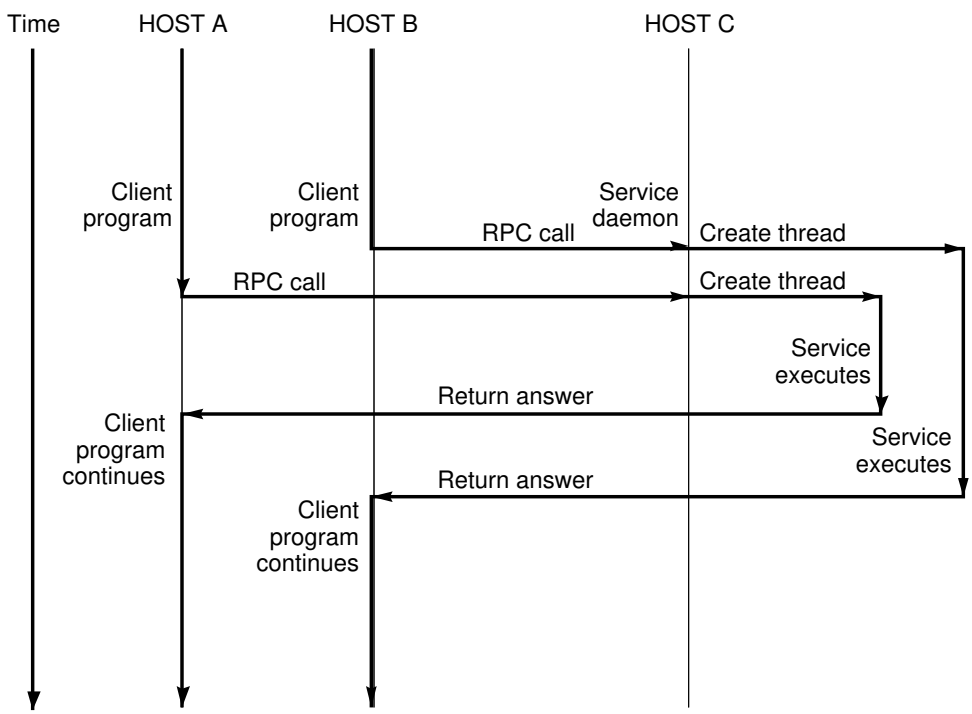

Figure 4-2 MT RPC Server Timing Diagram

Sharing the Service Transport Handle

The service transport handle, `SVCXPRT`, contains a single data area for decoding arguments and encoding results. Therefore, in the default, single-threaded mode, this structure cannot be freely shared between threads that call functions that perform these operations. However, when a server is operating in the MT Auto or User modes, a copy of this structure is passed to the service dispatch procedure in order to enable concurrent request processing. Under these circumstances, some routines which would otherwise be unsafe, become safe. Unless otherwise noted, the server interfaces are generally MT safe. See the `rpc_svc_calls(3N)` man page for more details on safety for server-side interfaces.

MT Auto Mode

In the Automatic mode, the RPC library creates and manages threads. The service developer invokes a new interface call, `rpc_control()`, to put the server into MT Auto mode before invoking the `svc_run()` call. In this mode, the programmer needs only to ensure that service procedures are MT safe.

`rpc_control()` allows applications to set and modify global RPC attributes. At present, it supports only server-side operations. Table 4-8 shows the `rpc_control()` operations defined for Auto mode. See also the `rpc_control(3N)` man page for additional information.

Table 4-8 `rpc_control()` Library Routines

`RPC_SVC_MTMODE_SET`	Set multithread mode
`RPC_SVC_MTMODE_GET`	Get multithread mode
`RPC_SVC_THRMAX_SET`	Set Maximum number of threads
`RPC_SVC_THRMAX_GET`	Get Maximum number of threads
`RPC_SVC_THRTOTAL_GET`	Total number of threads currently active
`RPC_SVC_THRCREATES_GET`	Cumulative total number of threads created by the RPC library
`RPC_SVC_THRERRORS_GET`	Number of thr_create errors within RPC library

Note – All of the get operations in Table 4-8, except `RPC_SVC_MTMODE_GET`, apply only to the Auto MT mode. If used in MT User mode or the single-threaded default mode, the results of the operations may be undefined.

By default, the maximum number of threads that the RPC server library creates at any time is 16. If a server needs to process more than 16 client requests concurrently, the maximum number of threads must be set to the desired number. This parameter may be set at any time by the server, and it allows the service developer to put an upper bound on the thread resources consumed by the server. Code Example 4-35 is an example RPC program written in MT Auto mode. In this case, the maximum number of threads is set at 20.

MT performance is enhanced if the function `svc_getargs()` is called by every procedure other than `NULLPROCS`, even if there are no arguments (`xdr_void` may be used in this case). This is true for both the MT Auto and MT User modes. For more information on this call, see the `rpc_svc_calls(3N)` man page.

Code Example 4-35 illustrates the server in MT Auto mode.

Note – You must link in the thread library when writing RPC multithreaded-safe applications. The thread library must be the last named library on the link line. To do this, specify the `-lthread` option in the compile command.

Compile the program in Code Example 4-35 by typing:

```
$ cc time_svc.c -lnsl -lthread
```

Code Example 4-35 Server for MT Auto Mode

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <synch.h>
#include <thread.h>
#include "time_prot.h"

void time_prog();

main(argc, argv)
int argc;
char *argv[];
{
int transpnum;
char *nettype;
int mode = RPC_SVC_MT_AUTO;
int max = 20;      /* Set maximum number of threads to 20 */

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

```
    if (argc == 2)
        nettype = argv[1];
    else
        nettype = "netpath";

    if (!rpc_control(RPC_SVC_MTMODE_SET, &mode)) {
        printf("RPC_SVC_MTMODE_SET: failed\n");
        exit(1);
    }
    if (!rpc_control(RPC_SVC_THRMAX_SET, &max)) {
        printf("RPC_SVC_THRMAX_SET: failed\n");
        exit(1);
    }
    transpnum = svc_create( time_prog, TIME_PROG, TIME_VERS,
        nettype);

    if (transpnum == 0) {
        fprintf(stderr, "%s: cannot create %s service.\n",
        argv[0], nettype);
        exit(1);
    }
    svc_run();
}

/*
 * The server dispatch function.
 * The RPC server library creates a thread which executes
 * the server dispatcher routine time_prog().  After which
 * the RPC library will take care of destroying the thread.
 */

static void
time_prog(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{

    switch (rqstp->rq_proc) {
        case NULLPROC:
            svc_sendreply(transp, xdr_void, NULL);
            return;
        case TIME_GET:
            dotime(transp);
```

```
            break;
        default:
            svcerr_noproc(transp);
            return;
    }
}
dotime(transp)
SVCXPRT *transp;
{

    struct timev rslt;
    time_t thetime;

    thetime = time((time_t *)0);
    rslt.second = thetime % 60;
    thetime /= 60;
    rslt.minute = thetime % 60;
    thetime /= 60;
    rslt.hour = thetime % 24;
    if (!svc_sendreply(transp, xdr_timev,(caddr_t) &rslt)) {
        svcerr_systemerr(transp);
    }
}
```

Code Example 4-36 shows the `time_prot.h` header file for the server.

Code Example 4-36 MT Auto Mode:`time_prot.h`

```
include <rpc/types.h>

struct timev {
    int second;
    int minute;
    int hour;
};

typedef struct timev timev;
bool_t xdr_timev();

#define TIME_PROG ((u_long)0x40000001)
#define TIME_VERS ((u_long) 1)
#define TIME_GET ((u_long) 1)
```

MT User Mode

In MT User mode, the RPC library will not create any threads. This mode works, in principle, like the single-threaded, or default mode. The only difference is that it passes copies of data structures (such as the transport service handle to the service dispatch routine) to be MT safe.

The RPC server developer takes the responsibility for creating and managing threads through the thread library. In the dispatch routine, the service developer can assign the task of procedure execution to newly created or existing threads. The `thr_create()` API is used to create threads having various attributes. All thread library interfaces are defined in `thread.h`. See the `pthread_create(3T)` man page for more details.

There is a lot of flexibility available to the service developer in this mode. Threads can now have different stack sizes based on service requirements. Threads may be bound. Different procedures may be executed by threads with different characteristics. The service developer may choose to run some services single threaded. The service developer may choose to do special thread-specific signal processing.

As in the Auto mode, the `rpc_control()` library call is used to turn on User mode. Note that the `rpc_control()` operations shown in Table 4-8 on page 119 (except for `RPC_SVC_MTMODE_GET`) apply only to MT Auto mode. If used in MT User mode or the single-threaded default mode, the results of the operations may be undefined.

Freeing Library Resources in User Mode

In the MT User mode, service procedures must invoke `svc_done()` before returning. `svc_done()` frees resources allocated to service a client request directed to the specified service transport handle. This function is invoked after a client request has been serviced, or after an error or abnormal condition that prevents a reply from being sent. After `svc_done()` is invoked, the service transport handle should not be referenced by the service procedure.
Code Example 4-37 shows a server in MT User mode.

Note – `svc_done()` must only be called within MT User mode. For more information on this call, see the `rpc_svc_calls(3N)` man page.

Code Example 4-37 MT User Mode: `rpc_test.h`

```
#defineSVC2_PROG 0x30000002
#defineSVC2_VERS ((u_long) 1)
#define SVC2_PROC_ADD ((u_long) 1)
#define SVC2_PROC_MULT ((u_long) 2)

struct intpair {
    u_shorta;
    u_short b;
};

typedef struct intpair intpair;

struct svc2_add_args {
    long argument;
    SVCXPRT *transp;
};

struct svc2_mult_args {
    intpair mult_argument;
    SVCXPRT *transp;
};

extern bool_t xdr_intpair();

#define NTHREADS_CONST 500
```

Code Example 4-38 is the client for MT User mode.

Code Example 4-38 Client for MT User Mode

```
#define _REENTRANT
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/uio.h>
#include <netconfig.h>
#include <netdb.h>
#include <rpc/nettype.h>
#include <thread.h>
#include "rpc_test.h"
```

```
void *doclient();
int NTHREADS;

struct thread_info {
    thread_t client_id;
    int client_status;
};

struct thread_info save_thread[NTHREADS_CONST];

main(argc, argv)
    int argc;
    char *argv[];
{
    int index, ret;
    int thread_status;
    thread_t departedid, client_id;
    char *hosts;
    if (argc < 3) {
        printf("Usage: do_operation [n] host\n");
        printf("\twhere n is the number of threads\n");
        exit(1);
    } else
        if (argc == 3) {
            NTHREADS = NTHREADS_CONST;
            hosts = argv[1];  /* live_host */
        } else {
            NTHREADS = atoi(argv[1]);
            hosts = argv[2];
        }
    for (index = 0; index < NTHREADS; index++){
        if (ret = thr_create(NULL, NULL, doclient,
        (void *)  hosts, THR_BOUND, &client_id)){
            printf("thr_create failed: return value %d", ret);
            printf(" for %dth thread\n", index);
            exit(1);
        }
        save_thread[index].client_id = client_id;
    }
    for (index = 0; index < NTHREADS; index++){
        if (thr_join(save_thread[index].client_id, &departedid,
        (void *)
        &thread_status)){
            printf("thr_join failed for thread %d\n",
            save_thread[index].client_id);
            exit(1);
        }
```

```
        save_thread[index].client_status = thread_status;
    }
}
    void *doclient(host)
    char *host;
{
    struct timeval tout;
    enum clnt_stat test;
    long result = 0;
    u_short mult_result = 0;
    long add_arg;
    long EXP_RSLT;
    intpair pair;
    CLIENT *clnt;
    if ((clnt = clnt_create(host, SVC2_PROG, SVC2_VERS, "udp" ==NULL)
{
        clnt_pcreateerror("clnt_create error: ");
        thr_exit((void *) -1);
    }
    tout.tv_sec = 25;
    tout.tv_usec = 0;
    memset((char *) &result, 0, sizeof (result));
    memset((char *) &mult_result, 0, sizeof (mult_result));
    if (thr_self() % 2){
        EXP_RSLT = thr_self() + 1;
        add_arg = thr_self();
        test = clnt_call(clnt, SVC2_PROC_ADD, (xdrproc_t) xdr_long,
        (caddr_t) &add_arg, (xdrproc_t) xdr_long, (caddr_t) &result,
        tout);
    } else {
        pair.a = (u_short) thr_self();
        pair.b = (u_short) 1;
        EXP_RSLT = (long) pair.a * pair.b;
        test = clnt_call(clnt, SVC2_PROC_MULT, (xdrproc_t)
        xdr_intpair,
        (caddr_t) &pair, (xdrproc_t) xdr_u_short,
        (caddr_t) &mult_result, tout);
        result = (long) mult_result;
    }
    if (test != RPC_SUCCESS) {
        printf("THREAD: %d clnt_call hav
        thr_exit((void *) -1);
    };
    thr_exit((void *) 0);
}
```

Code Example 4-39 shows the server side in MT User mode. MT performance is enhanced if the function `svc_getargs()` is called by every procedure other than `NULLPROC`, even if there are no arguments (`xdr_void` may be used in this case). This is true for both the MT Auto and MT User modes. For more information on this call, see the `rpc_svc_calls(3N)` man page.

Note – You must link in the thread library when writing RPC multithreaded-safe applications. The thread library must be the last named library on the link line. To do this, specify the `-lthread` option in the compile command.

Code Example 4-39 Server for MT User Mode

```
#define _REENTRANT
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/uio.h>
#include <signal.h>
#include <thread.h>
#include "operations.h"

SVCXPRT *xprt;
void add_mult_prog();
void *svc2_add_worker();
void *svc2_mult_worker();
main(argc, argv)
    int argc;
    char **argv;
{
    int transpnum;
    char *nettype;
    int mode = RPC_SVC_MT_USER;
       if(rpc_control(RPC_SVC_MTMODE_SET,&mode) == FALSE){
               printf(" rpc_control is failed to set AUTO mode\n");
                exit(0);
        }
    if (argc > 2) {
        fprintf(stderr, "usage: %s [nettype]\n", argv[0]);
        exit(1);
    }
    if (argc == 2)
        nettype = argv[1];
    else
        nettype = "netpath";
```

```
    transpnum = svc_create(add_mult_prog, SVC2_PROG,
    SVC2_VERS, nettype);
    if (transpnum == 0) {
        fprintf(stderr, "%s: cannot create %s service.\n", argv[0],
        nettype);
        exit(1);
    }
    svc_run();
}
void add_mult_prog (rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    long argument;
    u_short mult_arg();
    intpair mult_argument;
    bool_t (*xdr_argument)();
    struct svc2_mult_args *sw_mult_data;
    struct svc2_add_args *sw_add_data;
    int ret;
    thread_t worker_id;
    switch ((long) rqstp->rq_proc){
        case NULLPROC:
            svc_sendreply(transp, xdr_void, (char *) 0);
            svc_done(transp);
            break;
        case SVC2_PROC_ADD:
            xdr_argument = xdr_long;
            (void) memset((char *) &argument, 0, sizeof (argument));
            if (!svc_getargs(transp, xdr_argument,
            (char *) &argument)){
                printf("problem with getargs\n");
                svcerr_decode(transp);
                exit(1);
            }
            sw_add_data = (struct svc2_add_args *)
            malloc(sizeof (struct svc2_add_args));
            sw_add_data->transp = transp;
            sw_add_data->argument = argument;
            if (ret = thr_create(NULL, THR_MIN_STACK + 16 * 1024,
            svc2_add_worker, (void *) sw_add_data, THR_DETACHED,
                printf("SERVER: thr_create failed:");
                printf(" return value %d", ret);
```

```
                printf(" for add thread\n");
                exit(1);
            }
            break;
        case SVC2_PROC_MULT:
            xdr_argument = xdr_intpair;
            (void) memset((char *) &mult_argument, 0,
            sizeof (mult_argument));
            if (!svc_getargs(transp, xdr_argument,
            (char *) &mult_argument)){
                printf("problem with getargs\n");
                svcerr_decode(transp);
                exit(1);
            }
            sw_mult_data = (struct svc2_mult_args *)
            malloc(sizeof (struct svc2_mult_args));
            sw_mult_data->transp = transp;
            sw_mult_data->mult_argument.a = mult_argument.a;
            sw_mult_data->mult_argument.b = mult_argument.b;
            if (ret = thr_create(NULL, THR_MIN_STACK + 16 * 1024,
            svc2_mult_worker, (void *) sw_mult_data, THR_DETACHED,
            &worker_id)){
                printf("SERVER: thr_create failed:");
                printf("return value %d", ret);
                printf("for multiply thread\n");
                exit(1);
            }
            break;
        default:
            svcerr_noproc(transp);
            svc_done(transp);
            break;
    }
}
u_short mult_arg();
long add_one();
void *svc2_add_worker(add_arg)
struct svc2_add_args *add_arg;
{
```

```
    long *result;
    bool_t (*xdr_result)();
    xdr_result = xdr_long;
    result = (long *) malloc(sizeof (long));
    *result = add_one(add_arg->argument);
    if (!svc_sendreply(add_arg->transp, xdr_result,
    (caddr_t) result)){
        printf("sendreply failed\n");
        svcerr_systemerr(add_arg->transp);
        svc_done(add_arg->transp);
        thr_exit((void *) -1);
    }
    svc_done(add_arg->transp);
    thr_exit((void *) 0);
}
void *svc2_mult_worker(m_arg)
struct svc2_mult_args *m_arg;
{
    u_short *result;
    bool_t (*xdr_result)();
    xdr_result = xdr_u_short;
    result = (u_short *) malloc(sizeof (u_short));
    *result = mult_arg(&m_arg->mult_argument);
    if (!svc_sendreply(m_arg->transp, xdr_result,
    (caddr_t) result)){
        printf("sendreply failed\n");
        svcerr_systemerr(m_arg->transp);
        svc_done(m_arg->transp);
        thr_exit((void *) -1);
    }
    svc_done(m_arg->transp);
    thr_exit((void *) 0);
}
u_short mult_arg(pair)
    intpair *pair;
{
    u_short result;
    result = pair->a * pair->b;
    return (result);}
long add_one(arg)
    long arg;
{
    return (++arg);
}
```

Connection-Oriented Transports

Code Example 4-40 copies a file from one host to another. The RPC `send` call reads standard input and sends the data to the server `receive`, which writes the data to standard output. This also illustrates an XDR procedure that behaves differently on serialization and on deserialization. A connection-oriented transport is used.

Code Example 4-40 Remote Copy (Two-Way XDR Routine)

```
/*
 * The xdr routine:
 *      on decode, read wire, write to fp
 *      on encode, read fp, write to wire
 */
#include <stdio.h>
#include <rpc/rpc.h>

bool_t
xdr_rcp(xdrs, fp)
    XDR *xdrs;
    FILE *fp;
{
    unsigned long size;
    char buf[BUFSIZ], *p;

    if (xdrs->x_op == XDR_FREE)                /* nothing to free */
        return(TRUE);
    while (TRUE) {
        if (xdrs->x_op == XDR_ENCODE) {
            if ((size = fread( buf, sizeof( char ), BUFSIZ, fp))
                   == 0 && ferror(fp)) {
                fprintf(stderr, "can't fread\n");
                return(FALSE);
            } else
                return(TRUE);
        }
        p = buf;
        if (! xdr_bytes( xdrs, &p, &size, BUFSIZ))
            return(0);
        if (size == 0)
            return(1);
        if (xdrs->x_op == XDR_DECODE) {
            if (fwrite( buf, sizeof(char), size, fp) != size) {
                fprintf(stderr, "can't fwrite\n");
```

```
                return(FALSE);
            } else
                return(TRUE);
        }
    }
}
```

In Code Example 4-41 and Code Example 4-42, the serializing and deserializing are done only by the `xdr_rcp()` routine shown in Code Example 4-40.

Code Example 4-41 Remote Copy Client Routines

```
/* The sender routines */
#include <stdio.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>
#include "rcp.h"

main(argc, argv)
    int argc;
    char **argv;
{
    int xdr_rcp();

    if (argc != 2 7) {
        fprintf(stderr, "usage: %s servername\n", argv[0]);
        exit(1);
    }
    if( callcots( argv[1], RCPPROG, RCPPROC, RCPVERS, xdr_rcp, stdin,
          xdr_void, 0 ) != 0 )
        exit(1);
    exit(0);
}

callcots(host, prognum, procnum, versnum, inproc, in, outproc, out)
    char *host, *in, *out;
    xdrproc_t inproc, outproc;
{
    enum clnt_stat clnt_stat;
    register CLIENT *client;
    struct timeval total_timeout;

    if ((client = clnt_create( host, prognum, versnum, "circuit_v")
       == (CLIENT *) NULL)) {
```

```
        clnt_pcreateerror("clnt_create");
        return(-1);
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, procnum, inproc, in, outproc, out,
                          total_timeout);
    clnt_destroy(client);
    if (clnt_stat != RPC_SUCCESS)
        clnt_perror("callcots");
    return((int)clnt_stat);
}
```

The receiving routines are defined in Code Example 4-42. Note that in the server, `xdr_rcp()` did all the work automatically.

Code Example 4-42 Remote Copy Server Routines

```
/*
 * The receiving routines
 */
#include <stdio.h>
#include <rpc/rpc.h
#include "rcp.h"

main()
{
    void  rcp_service();
    if (svc_create(rpc_service,RCPPROG,RCPVERS,"circuit_v") == 0) {
        fprintf(stderr, "svc_create: errpr\n");
        exit(1);
    }
    svc_run();                      /* never returns */
    fprintf(stderr, "svc_run should never return\n");
}

void
rcp_service(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch(rqstp->rq_proc) {
        case NULLPROC:
            if (svc_sendreply(transp, xdr_void, (caddr_t) NULL) ==
                FALSE)
                fprintf(stderr, "err: rcp_service");
```

```
            return;
        case RCPPROC:
            if (!svc_getargs( transp, xdr_rcp, stdout)) {
                svcerr_decode(transp);
                return();
            }
            if(!svc_sendreply(transp, xdr_void, (caddr_t) NULL)) {
                fprintf(stderr, "can't reply\n");
                return();
            }
            return();
        default:
            svcerr_noproc(transp);
            return();
    }
}
```

Memory Allocation With XDR

XDR routines normally serialize and deserialize data. XDR routines often automatically allocate memory and free automatically allocated memory. The convention is to use a `NULL` pointer to an array or structure to indicate that an XDR function must allocate memory when deserializing. The next example, `xdr_chararr1()`, processes a fixed array of bytes with length `SIZE` and cannot allocate memory if needed:

```
xdr_chararr1(xdrsp, chararr)
    XDR *xdrsp;
    char chararr[];
{
    char *p;
    int len;

    p = chararr;
    len = SIZE;
    return (xdr_bytes(xdrsp, &p, &len, SIZE));
}
```

If space has already been allocated in `chararr`, it can be called from a server like this:

```
char chararr[SIZE];
svc_getargs(transp, xdr_chararr1, chararr);
```

Any structure through which data is passed to XDR or RPC routines must be allocated so that its base address is at an architecture-dependent boundary. An XDR routine that does the allocation must be written so that it can:

- Allocate memory when a caller requests
- Return the pointer to any memory it allocates

In the following example, the second argument is a `NULL` pointer, meaning that memory should be allocated to hold the data being deserialized.

```
xdr_chararr2(xdrsp, chararrp)
    XDR *xdrsp;
    char **chararrp;
{
    int len;

    len = SIZE;
    return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

The corresponding RPC call is:

```
char *arrptr;
arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);
/*
 * Use the result here
 */
svc_freeargs(transp, xdr_chararr2, &arrptr);
```

After use, the character array should be freed through `svc_freeargs()`. `svc_freeargs()` does nothing if passed a `NULL` pointer as its second argument.

To summarize:

- An XDR routine normally serializes, deserializes, and frees memory.
- `svc_getargs()` calls the XDR routine to deserialize.
- `svc_freeargs()` calls the XDR routine to free memory.

4

Porting From TS-RPC to TI-RPC

The transport-independent RPC (TI-RPC) routines allow the developer stratified levels of access to the transport layer. The highest-level routines provide complete abstraction from the transport and provide true transport-independence. Lower levels provide access levels similar to the TI-RPC of previous releases.

This section is an informal guide to porting transport-specific RPC (TS-RPC) applications to TI-RPC. Table 4-9 shows the differences between selected routines and their counterparts. For information on porting issues concerning sockets and transport layer interface (TLI), see the *Transport Interfaces Programming Guide*.

Porting an Application

An application based on either TCP or UDP can run in binary-compatibility mode. For some applications you only recompile and relink all source files. This may be true of applications that use simple RPC calls and use no socket or TCP or UDP specifics.

Some editing and new code may be needed if an application depends on socket semantics or features specific to TCP or UDP. Examples use the format of host addresses or rely on the Berkeley UNIX concept of privileged ports.

Applications that are dependent on the internals of the library or the socket implementation, or depend on specific transport addressing probably require more effort to port and may require substantial modification.

Benefits of Porting

Some of the benefits of porting are:

- Applications transport independence means they operate over more transports than before.
- Use of new interfaces make your application more efficient.
- Binary compatibility is less efficient than native mode.
- Old interfaces could removed from future releases.

Porting Issues

`libnsl` *Library*

`libc` no longer includes networking functions. `libnsl` must be explicitly specified at compile time to link the network services routines.

Old Interfaces

Many old interfaces are supported in the `libnsl` library, but they work only with TCP or UDP transports. To take advantage of new transports, you must use the new interfaces.

Name-to-Address Mapping

Transport independence requires opaque addressing. This has implications for applications that interpret addresses.

4

Differences Between TI-RPC and TS-RPC

The major differences between transport-independent RPC and transport-specific RPC are illustrated in Table 4-9. Also see section "Comparison Examples" on page 142 for code examples comparing TS-RPC with TI-RPC.

Table 4-9 Differences Between TI-RPC and TS-RPC

Category	TI-RPC	TS- RPC
Default Transport Selection	TI-RPC uses the TLI interface.	TS-RPC uses the socket interface.
RPC Address Binding	TI-RPC uses `rpcbind` for service binding. `rpcbind` keeps address in universal address format.	TS-RPC uses `portmap` for service binding.
Transport Information	Transport information is kept in a local file, `/etc/netconfig`. Any transport identified in `netconfig` is accessible.	Only TCP and UDP transports are supported.
Loopback Transports	`rpcbind` service requires a secure loopback transport for server registration	TS-RPC services do not require a loopback transport.
Host Name Resolution	The order of host name resolution in TI-RPC depends on the order of dynamic libraries identified by entries in `/etc/netconfig`.	Host name resolution is done by name services. The order is set by the state of the `hosts` database.
File Descriptors	Descriptors are assumed to be TLI endpoints.	Descriptors are assumed to be sockets.
`rpcgen`	The TI-RPC `rpcgen` tool adds support for multiple arguments, pass-by values, sample client files, and sample server files.	`rpcgen` in SunOS 4.1 and previous releases do not support the features listed for TI-RPC `rpcgen`.
Libraries	TI-RPC requires that applications be linked to the `libnsl` library.	All TS-RPC functionality is provided in `libc`.
MT Support	Multithreaded RPC clients and servers are supported.	Multithreaded RPC is not supported.

Function Compatibility Lists

The RPC library functions are listed in this section and grouped into functional areas. Each section includes lists of functions that are unchanged, have added functionality, and are new relative to previous releases.

Note – Functions marked with an asterisk are retained for ease of porting and may be not be supported in future releases of Solaris.

Creating Client Handles

The following functions are unchanged from the previous release and available in the current SunOS release:

```
clnt_destroy
clnt_pcreateerror
*clntraw_create
clnt_spcreateerror
*clnttcp_create
*clntudp_bufcreate
*clntudp_create
clnt_control
clnt_create
clnt_create_timed
clnt_create_vers
clnt_dg_create
clnt_raw_create
clnt_tli_create
clnt_tp_create
clnt_tp_create_timed
clnt_vc_create
```

Creating and Destroying Services

The following functions are unchanged from the previous releases and available in the current SunOS release:

```
svc_destroy
svcfd_create
*svc_raw_create
*svc_tp_create
*svcudp_create
*svc_udp_bufcreate
```

```
svc_create
svc_dg_create
svc_fd_create
svc_raw_create
svc_tli_create
svc_tp_create
svc_vc_create
```

Registering and Unregistering Services

The following functions are unchanged from the previous releases and available in the current SunOS release:

```
*registerrpc
*svc_register
*svc_unregister
xprt_register
xprt_unregister
rpc_reg
svc_reg
svc_unreg
```

SunOS 4.x Compatibility Calls

The following functions are unchanged from previous releases and available in the current SunOS release:

```
*callrpc
clnt_call
*svc_getcaller - works only with IP-based transports
rpc_call
svc_getrpccaller
```

Broadcasting

The following call has the same functionality as in previous releases, although it is supported for backward compatibility only:

```
*clnt_broadcast
```

`clnt_broadcast` can broadcast only to the `portmap` service. It does not support `rpcbind`.

The following function that broadcasts to both `portmap` and `rpcbind` is also available in the current release of SunOS:

`rpc_broadcast`

Address Management Functions

The TI-RPC library functions interface with either `portmap` or `rpcbind`. Since the services of the programs differ, there are two sets of functions, one for each service.

The following functions work with `portmap`:

```
pmap_set
pmap_unset
pmap_getport
pmap_getmaps
pmap_rmtcall
```

The following functions work with `rpcbind`:

```
rpcb_set
rpcb_unset
rpcb_getaddr
rpcb_getmaps
rpcb_rmtcall
```

Authentication Functions

The following calls have the same functionality as in previous releases. They are supported for backward compatibility only:

```
authdes_create
authunix_create
authunix_create_default
authdes_seccreate
authsys_create
authsys_create_default
```

Other Functions

`rpcbind` provides a time service (primarily for use by secure RPC client-server time synchronization), available through the `rpcb_gettime` function. `pmap_getport` and `rpcb_getaddr` can be used to get the port number of a

registered service. `rpcb_getaddr` communicates with any server running version 2, 3, or 4 of `rcpbind`. `pmap_getport` can only communicate with version 2.

Comparison Examples

The changes in client creation from TS-RPC to TI-RPC are illustrated in Code Example 4-43 and Code Example 4-44. Each example

- Creates a UDP descriptor.
- Contacts the remote host's RPC binding process to get the services address.
- Binds the remote service's address to the descriptor.
- Creates the client handle and set its time out.

Code Example 4-43 Client Creation in TS-RPC

```
struct hostent *h;
struct sockaddr_in sin;
int sock = RPC_ANYSOCK;
u_short port;
struct timeval wait;

if ((h = gethostbyname( "host" )) == (struct hostent *) NULL) {
    syslog(LOG_ERR, "gethostbyname failed");
    exit(1);
}
sin.sin_addr.s_addr = *(u_long *) hp->h_addr;
if ((port = pmap_getport(&sin, PROGRAM, VERSION, "udp")) == 0) {
    syslog (LOG_ERR, "pmap_getport failed");
    exit(1);
} else
    sin.sin_port = htons(port);
wait.tv_sec = 25;
wait.tv_usec = 0;
clntudp_create(&sin, PROGRAM, VERSION, wait, &sock);
```

The TI-RPC version assumes that the UDP transport has the `netid` *udp*. A `netid` is not necessarily a well-known name.

Code Example 4-44 Client Creation in TI-RPC

```
struct netconfig *nconf;
struct netconfig *getnetconfigent();
struct t_bind *tbind;
struct timeval wait;
```

```
    nconf = getnetconfigent("udp");
    if (nconf == (struct netconfig *) NULL) {
        syslog(LOG_ERR, "getnetconfigent for udp failed");
        exit(1);
    }
    fd = t_open(nconf->nc_device, O_RDWR, (struct t_info *) NULL);
    if (fd == -1) {
        syslog(LOG_ERR, "t_open failed");
        exit(1);
    }
    tbind = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR);
    if (tbind == (struct t_bind *) NULL) {
        syslog(LOG_ERR, "t_bind failed");
        exit(1);
    }
    if (rpcb_getaddr( PROGRAM, VERSION, nconf, &tbind->addr, "host")
         == FALSE) {
        syslog(LOG_ERR, "rpcb_getaddr failed");
        exit(1);
    }
    cl = clnt_tli_create(fd, nconf, &tbind->addr, PROGRAM, VERSION,
                         0, 0);
    (void) t_free((char *) tbind, T_BIND);
    if (cl == (CLIENT *) NULL) {
        syslog(LOG_ERR, "clnt_tli_create failed");
        exit(1);
    }
    wait.tv_sec = 25;
    wait.tv_usec = 0;
    clnt_control(cl, CLSET_TIMEOUT, (char *) &wait);
```

Code Example 4-45 and Code Example 4-46 show the differences between broadcast in TS-RPC and TI-RPC. The older `clnt_broadcast` is similar to the newer `rpc_broadcast`. The primary difference is in the `collectnames()` function: deletes duplicate addresses and displays the names of hosts that reply to the broadcast.

Code Example 4-45 Broadcast in TS-RPC

```
statstime sw;
extern int collectnames();

clnt_broadcast(RSTATPROG, RSTATVERS_TIME, RSTATPROC_STATS,
    xdr_void, NULL, xdr_statstime, &sw, collectnames);
    ...
```

```
collectnames(resultsp, raddrp)
    char *resultsp;
    struct sockaddr_in *raddrp;
{
    u_long addr;
    struct entry *entryp, *lim;
    struct hostent *hp;
    extern int curentry;

    /* weed out duplicates */
    addr = raddrp->sin_addr.s_addr;
    lim = entry + curentry;
    for (entryp = entry; entryp < lim; entryp++)
        if (addr == entryp->addr)
            return (0);

    ...
    /* print the host's name (if possible) or address */
    hp = gethostbyaddr(&raddrp->sin_addr.s_addr, sizeof(u_long),
        AF_INET);
    if( hp == (struct hostent *) NULL)
        printf("0x%x", addr);
    else
        printf("%s", hp->h_name);
}
```

Code Example 4-46 shows the Broadcast for TI-RPC:

Code Example 4-46 Broadcast in TI-RPC

```
statstime sw;
extern int collectnames();

rpc_broadcast(RSTATPROG, RSTATVERS_TIME, RSTATPROC_STATS,
     xdr_void, NULL, xdr_statstime, &sw, collectnames, (char *) 0);
    ...

collectnames(resultsp, taddr, nconf)
    char *resultsp;
    struct t_bind *taddr;
    struct netconfig *nconf;
{
    struct entry *entryp, *lim;
    struct nd_hostservlist *hs;
    extern int curentry;
    extern int netbufeq();
```

```
    /* weed out duplicates */
    lim = entry + curentry;
    for (entryp = entry; entryp < lim; entryp++)
        if (netbufeq( &taddr->addr, entryp->addr))
            return (0);
    ...
    /* print the host's name (if possible) or address */
    if (netdir_getbyaddr( nconf, &hs, &taddr->addr ) == ND_OK)
        printf("%s", hs->h_hostservs->h_host);
    else {
        char *uaddr = taddr2uaddr(nconf, &taddr->addr);
        if (uaddr) {
            printf("%s\n", uaddr);
            (void) free(uaddr);
        } else
            printf("unknown");
    }
}

netbufeq(a, b)
    struct netbuf *a, *b;
{
    return(a->len == b->len && !memcmp( a->buf, b->buf, a->len));
}
```

4