where the *addr* field contains the address of a client transport end point that initiates the connection request. The *opt* field contains any protocol-specific parameters. The *udata* field contains any optional user data to be sent along with the connection request. The *sequence* field contains an integer ID that is used to uniquely identify each connection.

The following code checks for a connection request in nonblocking mode. Notice that *t_alloc* is used to allocate the storage for the call variable. This is to ensure that the *maxlen* fields in the *addr, opt,* and *udata* are set to reflect the size of their *buf* member fields.

```
struct t_call *call = (struct t_call*)t_alloc (fd, T_CALL,T_ALL);
if (!call)
   t_error("t_alloc fails for T_CALL");
else do {
   if (t_listen(fd, call) == 0) break;          // got a connect request
   if (t_errmo!=TNODATA) {
      t_error("t_listen fails")
      exit(1);
   }
   /* do something else */
} while (1);
```

The above example first calls *t_alloc* to allocate dynamic storage for a *struct t_call* object and uses *call* to point to it. After that, the program goes into a loop where it calls *t_listen* to check for connection requests iteratively. If *t_listen* returns a zero value, a connection request is detected, and the program breaks out of the loop. However, if *t_listen* returns a nonzero value the *t_errno* global variable is checked. If *t_errno* is not set to TNODATA, some other error condition has occurred, and the program calls *t_error* to report a diagnostic message, then quits. Otherwise, a connection request is not yet available, and the program does something else, resuming the *t_listen* call afterward.

11.4.5 t_accept

The function prototype of the *t_accept* API is:

```
#include <tiuser.h>

int      t_accept ( int  fd, int newfd. struct t_call* call );
```

This function accepts a client connection request that is received via a *t_listen* call.

The *fd* value designates which server transport end point accepts the connection request. The *newfd* value designates a transport end point to be connected to the client transport end point. The *newfd* value may or may not be the same as *fd*. If they are the same, the transport end point must not have other connection requests pending, or this function will fail. On the other hand, if they are not the same value, the *newfd* must be allocated via a *t_open* call prior to its use in this function call. Then, after the *t_accept* call returns successfully, the *fd* can be used to accept more connection requests from other clients, and the *newfd* is used exclusively to communicate with the client whose transport address is specified in the *call* value. The *call* value is obtained from a *t_listen* call.

The function returns a -1 if it fails, a 0 value if it succeeds.

The following code spawns a child process to accept connection requests from one or more client processes. Note that the transport end point designated by *fd* is assumed to operate in blocking mode:

```
struct t_call *call = (struct t_call*)t_alloc (fd, T_CALL,T_ALL);
if (!call)
   t_error("t_alloc fails for T_CALL");
else while (t_listen(fd, call) == 0)          // got one connect request
   switch (fork())      {
      case -1: perror("fork"); break;         // parent process. fork fails
      default: break;                         // parent process. fork succeeds
      case 0:                                 // child process to talk to client
         if ((newfd=t_open("/dev/ticotsord",O_RDWR,0))==-1 ||
                  t_bind(newfd,0,0)==-1)
            t_error("t_open or t_bind fails");
         else if (t_accept(fd, newfd, call)==-1)
            t_error("t_accept fails\n");
         else {
            t_close(fd);// don't need this anymore
            /* now communicate with a client via the newfd */
         }
   }
```

In the above example, *t_alloc* is called to allocate a dynamic storage for a *struct t_call* object to store a client transport address. The *struct t_call* object is referenced by the *call* variable. If the *t_alloc* call succeeds, the program goes into a loop, executing the *t_listen* function to wait for client connection requests to arrive. For each connection request received, the program creates a child process to deal with that client. Specifically, each child process creates a new transport end point that is of the same transport type as that designated by *fd*. The new transport end point is referenced by the *newfd* variable, and it is bound an arbitrary

name (as assigned by the transport provider). After that, the child process calls the *t_accept* function to connect the *newfd* end point to the client's end point. If the *t_accept* call succeeds, the child process closes its copy of the *fd* descriptor, as it is no longer needed in that process and begins to communicate with the client.

11.4.6 t_connect

The function prototype of the *t_connect* API is:

```
#include <tiuser.h>

int     t_connect ( int fd, struct t_call* inaddr, struct t_call* outaddr );
```

This function sends a connect request to the server transport end point. The *fd* value designates which client transport end point is to be connected to that of the server. The server transport address is specified in the *inaddr* value. The address of the actual server transport end point bound is returned via the *outaddr* value.

The *inaddr* value must not be NULL, but the *outaddr* value may be NULL if the bound server transport address is a don't-care.

By default this function blocks the calling process until a server transport end point is connected or until a system error occurs. However, if *fd* is specified to be nonblocking, this function initiates a connection request and returns immediately (if a server transport end point is not connected right away). The *t_errno* global variable is set to TNODATA. The client can later call the *t_rcvconnect* function to check on the completion of the connect request. The function prototype of the *t_rcvconnect* function is:

```
#include <tiuser.h>

int     t_rcvconnect ( int fd, struct t_call* outaddr );
```

This function returns a -1 if it fails, a 0 value if it succeeds.

The following code sets a transport end point descriptor to be nonblocking via a *fcntl* call. It then attempts to connect to a server in a nonblocking manner. The server transport address is assumed to be 3.

```
struct t_call *call = (struct t_call*)t_alloc (fd, T_CALL, T_ALL);
if (!call) {
   t_error("t_alloc fails for T_CALL");
   exit(1);
}
call->addr.len = sizeof(int);                    // set the server's transport address
*(int*)call->addr.buf = 3;
/* set the fd to be nonblocking. This can also be set at the t_open call */
if ((flg=fcntl(fd, F_GETFL,0)==-1 ||
        fcntl(fd,F_SETFL,flg|O_NONBLOCK)==-1) {
   perror("fcntl");
   exit(2);
}
if (t_connect(fd, call, call)==-1) {
   while (t_errno==TNODATA) {    // poll for connect request to complete
      /* do something else */
      if (t_rcvconnect(fd, call)==0) break;
   }
   if (t_errno!=TNODATA)      {
      t_error("t_connect or t_rcvconnect fails");
      exit(4);
   }
} /* t_connect */
/* start communicating with a server transport end point */
```

In the above example, *fcntl* is called to set the transport end point designated by *fd* to be nonblocking. The program then calls *t_connect* to establish a connection with a server transport end point. The server transport address is assumed to be 3 (for local connection only). If *t_connect* does not succeed, the program goes into a loop where it does something else then calls *t_rcvconnect* to check on the completion of the connection request. The loop terminates when either *t_rcvconnect* returns a success status (return value is zero) or an error occurs, and *t_errno* is not set to TNODATA. If either the *t_connect* or the *t_rcvconnect* call succeeds, the program begins communicating with the server process.

11.4.7 t_snd, t_sndudata

The function prototypes of *t_snd* and *t_sndudata* APIs are:

```
#include <tiuser.h>

int     t_snd ( int fd, char* buf, unsigned len, int flags );
int     t_sndudata ( int fd, struct t_unitdata* udata );
```

The *t_snd* function sends a message of *len* bytes (contained in *buf*) to another process that is connected via the transport end point designated by *fd*. The transport end point must be a virtual circuit and has been connected to another end point via a *t_connect* (for a client process) or a *t_accept* (for a server process) call.

The *flags* value may be zero, which is the default, or it may be set with one or more of the following values, which are defined in the <tiuser.h> header:

***flags* value**	**Use**
T_EXPEDITED	Tags the message as an urgent message. This is like the MSG_OOB flag in sockets. Note that the transport provider may or may not support this option. If it does not support this, the function fails, and the *t_errno* will be set to TNOTSUPPORT
T_MORE	Tells the recipient process that the message sent via the next *t_snd* call is a continuing message of the current message

The function returns a -1 if it fails or the number of characters in *buf* that were successfully sent. Note that if *fd* is specified to be nonblocking, the function returns immediately if the message in *buf* cannot be delivered to the recipient right away. The function return value is -1, and the *t_errno* is set to TFLOW.

The *t_sndudata* is used to send datagram messages via a connectionless transport end point. The *struct t_unitdata* is declared as:

```
struct t_unitdata
{
   struct netbuf    addr;
   struct netbuf    opt;
   struct netbuf    udata;
};
```

where *udata* contains the message to be sent to a peer transport end point and whose address is specified in *addr*. The *opt* value is any transport-specific options to be used in this message delivery.

Note that there are no *flags* values that can be set in a *t_sndudata* call to specify that the message sent is a T_EXPEDITED message.

The function returns a -1 if it fails or a 0 if it succeeds. Note that if *fd* is specified to be nonblocking, the function returns immediately if the message specified in *udata* cannot be delivered to the recipient right away. In this case, the function return value is -1, and the *t_errno* is set to TFLOW.

The following example sends an urgent message (MSG1) to a connected transport end point:

```
char* MSG1 = "Hello World";
if (t_snd( MSG1, strlen(MSG1)+1,T_EXPEDITED) < 0) t_error("t_snd");
```

The following example sends a datagram message (MSG1) to the peer transport end point whose address is 3 (a local connection):

```
char* MSG1 = "Hello World";
struct t_unitdata *t_ud =
                    (struct t_unitdata*)t_alloc(fd,T_UNITDATA,T_ALL);
if (!t_ud) {
   t_error("t_alloc for T_UNITDATA fails");
   exit(1);
}
t_ud->addr.len = sizeof(int);              // set recipient transport address
*(int*)t_ud->addr.buf = 3;
t_ud->udata.len = strlen(MSG1) + 1;  // set the message to be sent
t_ud->udata.buf = MSG1;
if (t_sndudata(fd, t_ud) < 0) t_error("t_sndudata");
```

11.4.8 t_rcv, t_rcvudata, t_rcvuderr

The function prototypes of *t_rcv*, *t_rcvudata* and *t_rcvuderr* APIs are:

```
#include <tiuser.h>

int     t_rcv ( int fd, char* buf, unsigned len, int* flags);
int     t_rcvudata ( int fd, struct t_unitdata* udata, int* flags);
int     t_rcvuderr ( int fd, struct t_uderr* uderr );
```

The *t_rcv* function receives a message that is put into *buf* by another process that is connected to the transport end point as designated by *fd*. The transport end point must be a virtual circuit and has been connected to another end point via a *t_connect* (for a client process) or a *t_accept* (for a server process) call.

The *len* value specifies the maximum size of the *buf* argument. The *flags* value is an address of an integer-typed variable. This variable holds the *flags* value that is sent along with the message via the *t_snd* function call. The possible values that may be returned in *flags* are 0, T_MORE, and/or T_EXPEDITED. These values are described in the last section.

The *t_rcv* function returns -1 if it fails or the number of data bytes that were put into *buf*. Note that if *fd* is specified to be nonblocking, the function returns immediately if no message can be received right away. In this case, the function return value is -1 and the *t_errno* is set to TNODATA.

The *t_rcvudata* is used to receive datagram messages via a connectionless transport end point. The *udata* value is an address of a *struct unitdata* typed variable and holds the datagram message received, as well as the sender's address.

The *flags* value is an address of an integer-typed variable. This variable is normally assigned a return value of 0. However, if the receiving buffer *udata->udata.buf* is too small to receive the entire incoming message, the returned *flags* value is set to T_MORE and the kernel copies only enough message text to fill up the *udata->udata.buf* buffer. The process should call *t_rcvudata* again to receive the rest of the message.

The *t_rcvudata* function returns -1 if it fails or 0 if it succeeds. Note that if *fd* is specified to be nonblocking, the function returns immediately if no message can be received right away. The function return value is -1, and the *t_errno* is set to TNODATA.

The *t_rcvuderr* is used to receive error diagnostics associated with a datagram message. This should be called only if a *t_rcvudata* call returns a failure status. The *struct t_uderr* is declared as:

```
struct t_uderr
{
    struct netbuf     addr;
    struct netbuf     opt;
    long              error;
};
```

where *addr* contains the destination transport address of the erroneous message, *opt* contains any transport-specific parameters to be used with the message delivery and *error* contains an error code.

The *uderr* value may be specified as NULL, which means no error diagnostic is wanted and the function simply clears the internal error status flag.

The *t_rcvuderr* function return -1 if it fails, or 0 if it succeeds.

The following example receives a message from a connected transport end point:

```
int       flags;
char      buf[80];
if (t_rcv( fd, buf, sizeof buf, &flags ) < 0) t_error("t_rcv");
```

The following example receives a datagram message from a peer transport end point and prints an error diagnostic if the *t_rcvudata* call fails:

```
struct t_unitdata *t_ud =
                    (struct t_unitdata*)t_alloc(fd,T_UNITDATA,T_ALL);
if (!t_ud) {
    t_error("t_alloc for T_UNITDATA fails");
    exit(1);
}
    int             flags;
char    buf[80];
t_ud->udata.len = sizeof (buf);          // setup a buffer to receive msg
t_ud->udata.buf = buf;

if (t_rcvudata(fd, t_ud, &flags)< 0) {    // receive a datagram message
    if (t_errno==TLOOK)    {
        struct t_uderr *uderr=
                    (struct t_uderr*)t_alloc(fd, T_UDERROR, T_ALL);
```

```
        if (!uderr)     {
            t_error("t_alloc for T_UDERROR fails");
            exit(2);
        }
        if (t_rcvuderr(fd, uderr) < 0)      // get error code
            t_error("t_uderr");
        else cerr << "Error code is: << uderr->error << endl;
        t_free(uderr, T_UDERROR);      // free error data record
    }
    else t_error("t_rcvudata");
} else cout << "receive msg: '" << buf << "'\n";
```

11.4.9 t_sndrel , t_rcvrel

The function prototypes of *t_sndrel* and *t_rcvrel* APIs are:

```
#include <tiuser.h>

int     t_sndrel (int fd );
int     t_rcvrel (int fd );
```

The *t_sndrel* function sends an orderly connection release request to the underlying transport provider. The process cannot send any further messages to the transport end point designated by *fd*, but it can continue to receive messages via *fd* until an orderly connection release indication is received.

The *t_sndrel* function returns -1 if it fails, 0 if it succeeds. If *fd* is set to be nonblocking and the orderly release request cannot be sent to the underlying transport provider immediately, the function returns a -1 value, and *t_errno* is set to TFLOW.

The *t_rcvrel* acknowledges the receipt of an orderly connection release indication. After this function is called, the process should not attempt to receive more messages via the transport end point designated by *fd*.

The *t_rcvrel* function returns -1 if it fails, 0 if it succeeds.

Note that not all transport providers support the *t_sndrel* and *t_rcvrel* functions. If a transport provider does not support these functions and they are called, they return a -1 value, and *t_errno* is set to TNOTSUPPORT.

The following example sends an orderly release request to a connected transport end point and waits for acknowledgment of the connection release indication:

```
if (sndrel(fd) < 0)
   t_error("sndrel");
else if (t_rcvrel(fd) < 0) t_error("rcvrel");
```

11.4.10 t_snddis, t_rcvdis

The function prototypes of *t_snddis* and *t_rcvdis* APIs are:

```
#include <tiuser.h>

int     t_snddis ( int fd, struct t_call* call );
int     t_rcvdis ( int fd, struct t_discon* conn );
```

The *t_snddis* function is used to abort an established transport connection or to reject a connection request by a client. When *t_snddis* is used to reject a connection request, the *call->sequence* field specifies which connection request to reject. When *t_snddis* is used to initiate an abortive release of a transport connection, *call* value may be NULL; otherwise, only the *call->udata* field is used. This field contains user data sent to the connected transport end point along with the abortive release indication.

The *t_snddis* function returns -1 if it fails, 0 if it succeeds.

The *t_rcvdis* function is used to retrieve an abortive release indication and any user data sent along with the indication. The *conn* value may be NULL if the process does not care about the user data or the reason for the abortive release. Otherwise, the *conn* value is the address of a *struct t_discon* typed variable. The *struct t_discon* is declared as:

```
struct t_discon
{
   struct netbuf    udata;
   int              reason;
   int              sequence;
};
```

The *conn->udata* contains any user data that are sent via a *t_snddis* call. The *conn->reason* contains a transport-specific reason code for the disconnection. The *conn->sequence*

is meaningful only in a server process that has performed multiple *t_listen* calls and is used to determine which client process has initiated a *t_connect* and, finally, a *t_snddis* function call.

The *t_rcvdis* function returns -1 if it fails, 0 if it succeeds.

The following example sends an abortive release request to a connected transport end point:

```
if (t_snddis(fd) < 0) t_error("t_snddis");
```

The following example uses *t_rcvdis* to obtain a reason code for a rejection of a connection request:

```
if (t_connect(fd, call, call) < 0 && t_errno==TLOOK)
   if (t_look((fd)==T_DISCONNECT) {
      struct t_discon *conn = (struct t_discon*)t_alloc(fd,T_DIS,T_ALL);
      if (!conn)
         t_error("t_alloc for T_DIS fails");
      else if (t_rcvdis(fd,conn) < 0)
         t_error("t_rcvdis");
      else cout << "Disconnect reason code: " << conn->reason << endl;
   }
```

11.4.11 t_close

The function prototype of *t_close* APIs is:

```
#include <tiuser.h>

int      t_close ( int  fd );
```

The *t_close* function causes the transport provider to free all system resources that are allocated for the transport end point as designated by *fd*, and closes the device file associated with the transport provider. This should be called after the transport end point connection has been terminated via either the *t_sndrel* or the *t_snddis* call.

The *t_close* function returns a -1 if it fails or a 0 if it succeeds.

The following example closes a transport end point:

```
if (t_close(fd) < 0) t_error("t_close");
```

11.5 TLI Class

This section depicts the TLI class which performs functions similar to those in the sock class (see Section 11.2). Specifically, the TLI class encapsulates all low-level TLI system call interfaces and takes care of the dynamic memory management of the TLI-specific data structure (e.g., *struct t_call, struct t_bind,* etc.). These reduce the learning and programming time of users who wish to use TLI for IPC. Furthermore, the TLI class fosters maximum code reuse among all applications.

The TLI class is defined in the *tli.h* header as:

```
#ifndef TLI_H
#define TLI_H

/* TLI class definition */
#include <iostream.h>
#include <unistd.h>
#include <string.h>
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <netdir.h>
#include <netconfig.h>
#define   UDP_TRANS      "/dev/ticlts"
#define   TCP_TRANS      "/dev/ticotsord"
#define   DISCONNECT     -1

class tli
{
  private:
    int              tid;          // transport descriptor
    int              local_addr;   // transport address for local IPC
    struct nd_addrlist *addr;      // transport address for Internet IPC
    struct netconfig   *nconf;     // transport provider device file
    int              rc;           // TLI functions return status code

    /* Allocate a structure to send a datagram message to an
```

```
   end point in Internet */
struct t_unitdata* alloc_ud ( int nsid, char* service, char* host )
{
   struct t_unitdata* ud=(struct t_unitdata*)t_alloc(
            nsid==-1 ? tid : nsid, T_UNITDATA, T_ALL);
   if (!ud)     {
      t_error("t_alloc of t_unitdata");
      return 0;
   }
   struct nd_hostserv    hostserv;
   struct netconfig      *ncf;
   struct nd_addrlist    *Addr;
   void                  *hp;
   if ((hp=setnetpath()) == 0)     {
      perror("Can't init network");
      return 0;
   }
   hostserv.h_host = host;
   hostserv.h_serv = service;
   while ((ncf=getnetpath(hp)) != 0)
      if (ncf->nc_semantics == NC_TPI_CLTS
            && netdir_getbyname(ncf, &hostserv, &Addr)==0)
         break;
   endnetpath(hp);
    if (!ncf)    {
      cerr << "Can't find transport for '" << service << "'\n";
      return 0;
   }
   ud->addr = *(Addr->n_addrs);
   return ud;
};

/* Allocate a structure to send a datagram message to an end point */
struct t_unitdata* alloc_ud ( int nsid, int port_no )
{
   struct t_unitdata* ud=(struct t_unitdata*)t_alloc(
                        nsid==-1 ? tid : nsid, (T_UNITDATA), (T_ALL));
   if (!ud)        {
      t_error("t_alloc of t_unitdata");
      return 0;
   }
   ud->addr.len = sizeof(int);
   *(int*)ud->addr.buf = port_no;
   return ud;
};
```

```
    /* Report a datagram message receive error */
    void report_uderr( int nsid )
    {
       if ((t_errno) == (TLOOK))
       {
          struct t_uderr   *uderr;
          if ((uderr=(struct t_uderr*)t_alloc(nsid==-1 ? tid : nsid,
                                T_UDERROR,T_ALL))==0) {
             t_error("t_alloc of t_uderr");
             return;
          }
          if ((rc=t_rcvuderr(nsid==-1 ? tid : nsid, uderr)) < 0)
             t_error("t_rcvuderr");
          else cerr << "bad datagram. error=" << uderr->error << endl;
          t_free((char*)uderr,T_UDERROR);
       }
       else t_error("t_rcvudata");
    };
public:
    /* Constructor to create a transport end point for local IPC */
    tli( int srv_addr, int connless = 0 )
    {
       local_addr = srv_addr;
       nconf = 0;
       addr  = 0;
       if ((tid=t_open(connless ? UDP_TRANS : TCP_TRANS,
                                O_RDWR, 0)) < 0)
          t_error("t_open fails");
    };

    /* Constructor to create a transport end point for Internet IPC */
    tli( char* hostname, char* service, int connless=0 )
    {
       struct nd_hostserv  hostserv;
       void  *hp;
       int type = connless ? NC_TPI_CLTS : NC_TPI_COTS_ORD;
       local_addr = 0;
       // find the transport provider for the specified host/service
       if ((hp=setnetpath()) == 0)      {
          perror("Can't init network");
          exit(1);
       }
       hostserv.h_host = hostname;
       hostserv.h_serv = service;
       while ((nconf=getnetpath(hp)) != 0)
          if (nconf->nc_semantics == type
```

```
                && netdir_getbyname(nconf, &hostserv, &addr)==0)
            break;
        endnetpath(hp);
        if (nconf == 0 )
            cerr << "No transport found for service: '" << service << "'\n";
        else if ((tid=t_open(nconf->nc_device, O_RDWR, 0)) < 0)
            t_error("t_open fails");
};

/* Destructor function */
~tli()                      { shutdown(); close(tid); };

/* Check constructor success status */
int good()                  { return tid >= 0; };

/* Let the transport provider  to bind a name to an end point */
int Bind_anonymous( )   { return t_bind(tid, 0, 0); };

/* Bind a name to a transport end point */
int Bind()
{
    struct t_bind  *bind;
    if ((bind= (struct t_bind*)t_alloc(tid, T_BIND, T_ALL))==0)
        t_error("t_alloc for t_bind");
        return -1;
    }
    bind->qlen = 1;         // max no of pending connect request
    if (nconf)  {           // Internet address
        bind->addr = *(addr->n_addrs);
    }
    else  {                 // Local address
        ind->addr.len = sizeof(int);
        *(int*)bind->addr.buf = local_addr;
    }
    if ((rc = t_bind(tid, bind, bind)) < 0)
        t_error("t_bind");
    else                    // echo the actually bound address
        cerr << "server: t_bind: " << (*(int*)bind->addr.buf) << endl;
    return rc;
};
/* Wait for a connect request from a client transport end point */
int listen ( struct t_call*& call )
{
    if (!call && (call = (struct t_call*)t_alloc(tid, T_CALL, T_ALL))==0)
    {
        t_error("t_alloc");
```

```
        return -1;
    }
    if ((rc=t_listen(tid,call)) < 0) t_error("t_listen");
    return rc;
};

/* Accept a connect request from a client transport end point */
int accept ( struct t_call * call )
{
    /* create a new end point to communicate with client */
    int resfd;
    if (nconf)                  // Internet IPC
        resfd = t_open( nconf->nc_device, O_RDWR, 0);
    else                        // Local IPC, must be connection-based
        resfd = t_open(TCP_TRANS, O_RDWR, 0);
    if (resfd < 0)      {
        t_error("t_open for resfd");
        return -1;
    }
    // Bind an arbitrary name to the new end point
    if (t_bind(resfd,0,0) < 0)          {
        t_error("t_bind for resfd");
        return -2;
    }
    // Connect the new end point to the client
    if (t_accept(tid, resfd, call) < 0)       {
        if (t_errno == TLOOK)   {
            if (t_rcvdis(tid, 0) < 0)  {
                t_error("t_rcvdis");
                return -4;
            }
            if (t_close(resfd) < 0)     {
                t_error("t_close");
                return -5;
            }
            return DISCONNECT;
        }
        t_error("t_acept");
        return -6;
    }
    return resfd;
};

/* Initiate a connect request to a server's transport end point */
int connect()
{
```

```
    struct t_call *call;
    if ((call = (struct t_call*)t_alloc(tid, T_CALL, T_ALL))==0)    {
        t_error("t_alloc");
        return -1;
    }
    if (nconf)
        call->addr = *(addr->n_addrs);
    else {
        call->addr.len = sizeof(int);
        *(int*)call->addr.buf = local_addr;
    }
    cerr << "client: connect to addr=" << (*(int*)call->addr.buf) << endl;
    if ((rc=t_connect(tid,call,0)) < 0)    {
        t_error("client: t_connect"); return -2;
    }
    return rc;
};

/* Write a message to a connected remote transport end-pount */
int write( char* buf, int len, int nsid=-1 )
{
    if ((rc=t_snd(nsid==-1 ? tid : nsid, buf, len, 0)) < 0) t_error("t_snd");
    return rc;
};

/* Read a message from a connected remote transport end-pount */
int read( char* buf, int len, int& flags, int nsid=-1 )
{
    if ((rc=t_rcv(nsid==-1 ? tid : nsid, buf, len, &flags)) < 0)
        t_error("t_snd");
    return rc;
};

/* Write a datagram message to a remote end point in the Internet */
int writeto( char* buf, int len, int flag, char* service, char* host,
                int nsid=-1 )
{
    struct t_unitdata* ud = alloc_ud(nsid,service,host);
    ud->udata.len = len;
    ud->udata.buf = buf;
    if ((rc=t_sndudata(nsid==-1 ? tid : nsid, ud)) < 0)
        t_error("t_sndudata");
    t_free(ud); return rc;
};
```

```
        /* Write a datagram message to a remote end point on the
           same machine */
        int writeto( char* buf, int len, int flag, int port_no, int nsid=-1 )
        {
           struct t_unitdata* ud = alloc_ud(nsid,port_no);
           ud->udata.len = len;
           ud->udata.buf = buf;
           if ((rc=t_sndudata(nsid==-1 ? tid : nsid, ud)) < 0)
              t_error("t_sndudata");
           t_free(ud); return rc;
        };

        /* Write a datagram msg to a end point whose address is specified
           in ud */
        int writeto( char* buf, int len, int flag, struct t_unitdata* ud, int nsid=-1 )
        {
           ud->udata.len = len;
           ud->udata.buf = buf;
           if ((rc=t_sndudata(nsid==-1 ? tid : nsid, ud)) < 0)
              t_error("t_sndudata");
           return rc;
        };

        /* Receive a datagram message from a remote end point */
        int readfrom( char* buf, int len, int& flags, struct t_unitdata*& ud,
                                    int nsid =-1)
        {
           if (!ud && (ud=(struct t_unitdata*)t_alloc(nsid==-1 ? tid : nsid,
                       T_UNITDATA, T_ALL))==0)  {
              t_error("t_alloc of t_unitdata");
              return -1;
           }
           ud->udata.len = len;
           ud->udata.buf = buf;
           if ((rc=t_rcvudata(nsid==-1 ? tid : nsid, ud, &flags)) < 0)
              report_uderr(nsid);
           return rc;
        };

        /* Shutdown a transport connection using abortive release notification */
        int shutdown( int nsid = -1 )
        {
           return t_snddis( nsid==-1 ? tid : nsid, 0 );
        };
     };   /* class TLI */
     #endif
```

The public member functions of the TLI class are on an almost one-to-one correspondence with those of the sock class. This is a reflection of the one-to-one correspondence of the TLI APIs and the socket APIs.

The major difference between the TLI class and the *sock* class is the naming convention assigned to each type of object. Specifically, a *sock* object name may be a UNIX path name (for UNIX domain socket) or a host name and a port name (for an Internet domain socket). A TLI object name may be an integer number (for local transport communication) or a host name and a service name (for Internet transport communication).

Another difference between the TLI class and the *sock* class is the *listen* function. Whereas the *sock::listen* sets only the maximum number of pending connection requests allowed for a *sock* object, the *TLI::listen* function actually waits for a client connection request to arrive. In fact, the *sock::accept* function actually combines the operation of the *TLI::listen* and *TLI::accept* functions.

The above TLI class does not support nonblocking operations, but it could easily be modified by users to support nonblocking operations. The following two sections depict two sample IPC applications that make use of the TLI class.

11.6 Client/Server Message Example

The first example that makes use of the TLI class is the client/server message passing example, as shown in Section 10.3.7. This new version uses connection-based transport end points to set up a communication channel between the message server and its client processes. Furthermore, because the server is connected directly to each client process, each message sent from a client consists of a service command (e.g., LOCAL_TIME, UTC_TIME, or QUIT_CMD, etc.) encoded in a character string. The server sends the service response to its client in a character string format also.

The message server program, *tli_msg_srv.C*, is shown below:

```
#include "tli.h"
#include <sys/times.h>
#include <sys/types.h>
#define MSG1 "Invalid cmd to message server"
typedef enum { LOCAL_TIME, GMT_TIME, QUIT_CMD,
                    ILLEGAL_CMD } CMDS;

/* create a child process to handle a client's commands */
void process_cmd (tli* sp, int fd )
{
   char     buf[80];
```

```
        time_t  tim;
        char*   cptr;
        int     flags;
        switch (fork())  {
           case -1:   perror("fork"); return;
           case 0:    break;
           default:   return; // parent
        }
        /* read commands from a client until EOF or QUIT_CMD */
        while (sp->read(buf, sizeof buf, flags, fd) > 0)    {
           cerr << "server: read cmd: '" << buf << "'\n";
           int    cmd = ILLEGAL_CMD;
           (void)sscanf(buf,"%d",&cmd);
           switch (cmd)       {
              case LOCAL_TIME:
                 tim = time(0);
                 cptr = ctime(&tim);
                 sp->write(cptr, strlen(cptr)+1, fd);
                 break;
              case GMT_TIME:
                 tim = time(0);
                 cptr = asctime(gmtime(&tim));
                 sp->write(cptr, strlen(cptr)+1, fd);
                 break;
              case QUIT_CMD:
                 sp->shutdown(fd);                    // shutdown the connection
                 exit(0);
              default:
                 sp->write(MSG1, sizeof MSG1, fd);
           }
        }
        exit(0);
     }
     int main( int argc, char* argv[])
     {
        char  buf[80], socknm[80];
        int  port=-1, nsid, rc;
        fd_set  select_set;
        struct timeval timeRec;
        if (argc < 2)    {
           cerr << "usage: " << argv[0] << " <serviceIno> <host>\n";
           return 1;
        }
        /* check if integer address no. or a service name is specified */
        (void)sscanf(argv[1],"%d",&port);
```

```
    /* create a connection-based transport end point */
    tli *sp;
    if (port==-1)
       sp = new tli( argv[2], argv[1] );
    else sp = new tli (port);

    if (!sp || !sp->good())        {
       cerr << "server: create transport endpoint object fails\n";
       return 1;
    }

    /* Bind a name to the server's transport end point */
    if (sp->Bind() < 0)            {
       cerr << "server: bind fails\n";
       return 2;
    }

    for (struct t_call *call=0; sp->listen(call)==0; )     {
       /* accept a connection request from a client socket */
       if ((nsid = sp->accept(call)) < 0)     {
          cerr << "server: accept fails\n";
          return 3;
       }
       cerr << "server: got one client connection. nsid=" << nsid << "\n";

       /* create a child process to process commands */
       process_cmd(sp,nsid);

        close(nsid); /* re-cycle file descriptor */
    }
    return sp->shutdown();
}
```

The server program is invoked with either an integer address for local transport connection or a service and host name for Internet transport communication. The server begins execution by creating a transport end point and binds its name to it. It then enters a loop, where it listens for a connection request from client processes via the *tli::listen* function.

For each connection request received, the server calls the *tli::accept* function to connect with the client transport end point. The *tli::accept* function also returns a new transport descriptor (*nsid*) for the server to communicate exclusively with the connected client. The server calls the *process_cmd* function to process a connected client. The *process_cmd* functio, in turn, forks a child process to deal with the client. It returns immediately to the server process, so that the server can continue to monitor other connect requests.

Each child process created in the *process_cmd* function calls the *tli::read* function to read each command from a connected client and then replies via the *tli::write* function. If the client sends the QUIT_CMD to the child process, the child process destroys the connection via the *tli::shutdown* call and terminates itself.

The client program that communicates with the server program via TLI class objects is *tli_msg_cls.C*:

```
#include "tli.h"
#define QUIT_CMD 2

int main( int argc, char* argv[])
{
   if (argc < 2)      {
      cerr << "usage: " << argv[0] << " <serviceIno> <host>\n";
      return 1;
   }
   char buf[80];
   int  port=-1, rc, flags;

   /* check if an integer address or a service name is spceified */
   (void)sscanf(argv[1],"%d",&port);

   tli  *sp;                        // create a transport end point
   if (port==-1)
      sp = new tli( argv[2], argv[1] );       // Internet transport
   else sp = new tli (port);                  // local transport
   if (!sp || !sp->good())        {
      cerr << "client: create transport endpoint object fails\n";
      return 1;
   }

   /* bind an arbitrary name to the transport end point */
   if (sp->Bind_anonymous() < 0)      {
      cerr << "client: bind fails\n";
      return 2;
   }

   /* connect to a server transport end point */
   if (sp->connect() < 0)        {
      cerr << "client: connect fails\n";
      return 3;
   }
```

```
    /* Send cmds 0 -> 2 to server */
    for (int cmd=0; cmd < 3; cmd++)        {
       /* compose a command to server */
       sprintf(buf,"%d",cmd);
       if (sp->write(buf,strlen(buf)+1) < 0) return 4;

       /* exit the loop if QUIT_CMD */
       if (cmd==QUIT_CMD) break;

       /* read reply from server */
       if (sp->read(buf,sizeof buf, flags) < 0) return 5;
       cout << "client: recv '" << buf << "'\n";
    }

    /* shutdown the transport connection */
    return sp->shutdown();

} /* main */
```

The client program is invoked with a server address, which may be an integer address for local transport connection or a service and host name for Internet transport communication. The client begins execution by creating a transport end point and binds an anonymous name to it. A client does not need to have a well-defined name assigned to it because no other process initiates connection requests to the client process by address.

Once a transport end point is defined, the client calls the *tli::connect* function to establish a virtual circuit connection with the server transport end point. If this is accomplished successfully, the client sends a series of commands to the server via the *tli::write* function calls. The commands sent by the client, in their sending order, are LOCAL_TIME, UTC_TIME, and the QUIT_CMD. For each command sent (except the QUIT_CMD), the client waits for the server reply via the *tli::read* function and prints the server reply message to the standard output.

After the client sends the QUIT_CMD to the server, it shuts down the transport connect via the *tli::shutdown* function, then terminates itself.

The sample output of the client/server program execution is:

```
% CC -o tli_msg_srv tli_msg_srv.C -lnsl
% CC -o tli_msg_cls tli_msg_cls.C -lnsl
% tli_msg_srv 2 &
[1] 781
server: t_bind: 2
% tli_msg_cls 2
```

```
client: connect to addr=2
server: got one client connection. nsid=4
server: read cmd: '0'
client: recv 'Fri Feb 17 22:34:50 1995'
server: read cmd: '1'
client: recv 'Sat Feb 18 06:34:50 1995'
server: read cmd: '2'
[1] + Done              tli_msg_srv 2
%
```

The same two programs can also be run unmodified but using an Internet transport connection instead. To enable this feature, a new entry, as shown below, is added to the */etc/services* file:

```
test            4045/tcp
```

This new entry defines a new service call *test* which uses *TCP* as the transport provider, with the assigned port number of 4045. The following screen log shows the new run of the same client/server program, but here, the server transport address is *fruit* (host name) and *test* (service name):

```
% hostname
fruit
% tli_msg_srv test fruit &
[1] 776
server: t_bind: 135122
% tli_msg_cls test fruit
client: connect to addr=135122
server: got one client connection. nsid=4
server: read cmd: '0'
client: recv 'Fri Feb 17 22:34:04 1995'
server: read cmd: '1'
client: recv 'Sat Feb 18 06:34:04 1995'
server: read cmd: '2'
[1] + Done              tli_msg_srv test fruit
```

11.7 Datagram Example

The second example shows two peer processes communicating via two datagram transport end points. These transport end points are created via the TLI class also. The two peer processes are *tli_clts1* and *tli_clts2*, as created from the *tli_clts1.C* and *tli_clts2.C* files, respectively.

The *tli_clts1.c* program is:

```
#include <sys/systeminfo.h>
#include "tli.h"
#define MSG1 "Hello MSG1 from clts1"

/* get a host name */
int gethost( int argc, char* argv[], char host[], int len)
{
   if (argc!=3)      {
      if (sysinfo(SI_HOSTNAME,host,len)< 0)    {
         perror("sysinfo");
         return -1;
      }
   }  else strcpy(host,argv[2]);
   return 0;
}

int main( int argc, char* argv[])
{
   char  buf[80], host[80];
   int  port=-1, rc, flags=0;

   if (argc < 2)      {
      cerr << "usage: " << argv[0] << " <servicelport_no> [<hostname>]\n";
      return 1;
   }

   /* check if port no. of a socket name is specified */
   (void)sscanf(argv[1],"%d",&port);
   /* Create a transport end point */
   tli  *sp;
   if (port==-1)     {
      if (gethost(argc, argv, host, sizeof host) < 0) return 2;
      sp = new tli( host, argv[1], 1 );
   } else sp = new tli (port, 1);

   if (!sp || !sp->good())     {
      cerr << "clts1: create transport endpoint object fails\n";
      return 3;
   }

   /* Bind a name to the transport end point */
   if (sp->Bind() < 0)     {
      cerr << "clts1: bind fails\n";
```

```
        return 4;
    }
    struct t_unitdata *ud = 0;
    if (sp->readfrom( buf, sizeof buf, flags, ud) < 0)      {
        cerr << "clts1: readfrom fails\n";
        return 5;
    }
    cerr << "clts1: read msg: '" << buf << "'\n";

    if (sp->writeto(MSG1, strlen(MSG1)+1, flags, ud) < 0)    {
        cerr << "clts1: writeto fails\n";
        return 6;
    }
    if (sp->readfrom(buf, sizeof buf, flags, ud) < 0)    {
        cerr << "clts: readfrom fails\n";
        return 7;
    }
    cerr << "clts1: read msg: '" << buf << "'\n";
    return 0;
}
```

The *tli_clts1* program may be invoked with a single integer address (for local transport communication) or a service name, followed by an optional host name (for Internet transport communication). If a service name is specified, the name must also be defined in the */etc/services* file. If a host name is not specified with a service name, the host name of the machine on which the program is run is assumed.

The *tli_clts1* begins execution by creating a TLI object (a transport end point) based on the given command line arguments. If an Internet transport end point is to be created, the *gethost* function is called to retrieve the local host machine name from either the command line argument (if it is specified) or via the *sysinfo* function call.

After a TLI transport object is created, the process binds a name to the transport object, then reads a message (MSG2) to be sent by the *tli_clts2* process. Once the MSG2 message is received, the process prints that message to the standard output. It then writes a MSG1 message back to the *tli_clts2* process via the transport address contained in the *ud* variable. The *ud* variable is assigned by the *readfrom* call.

Finally, the process waits for the *tli_clts2* to send the last MSG3 message. Once that is received, the process prints that message to the standard output and terminates itself. The transport end point created by the process is discarded via the *TLI::~TLI* destructor function.

The *tli_clts2.c* program that communicates with the *tli_clts1* program is:

```
#include <sys/systeminfo.h>
#include "tli.h"
#define MSG2 "Hello MSG2 from clts2"
#define MSG3 "Hello MSG3 from clts2"

/* get a host name */
int gethost( int argc, char* argv[], char host[], int len)
{
   if (argc!=4)      {
      if (sysinfo(SI_HOSTNAME,host,len)< 0)     {
         perror("sysinfo");  return -1;
      }
   } else strcpy(host,argv[3]);
   return 0;
}

int main( int argc, char* argv[])
{
   char  buf[80], host[80];
   int  port=-1, clts1_port=-1, rc, flags=0;

   if (argc < 2)      {
      cerr << "usage: " << argv[0] <<
            " <service|port_no> <clts1_service|no> [<host>]\n";
      return 1;
   }
   /* check if port no. of a socket name is specified */
   (void)sscanf(argv[1],"%d",&port);
   (void)sscanf(argv[2],"%d",&clts1_port);

   /* create a transport end point */
   tli  *sp;
   if (port==-1)     {                          // Internet transport
      if (gethost(argc, argv, host, sizeof host) < 0) return 2;
      sp = new tli( host, argv[1], 1 );
   } else sp = new tli (port, 1);               // local transport
   if (!sp || !sp->good())  {
      cerr << "clts2: create transport endpoint object fails\n";
      return 2;
   }

   /* Bind a name to the transport end point */
   if (sp->Bind() < 0)     {
      cerr << "clts2: bind fails\n";
      return 3;
   }
```

```
    /* write MSG2 to the tli_clts1 process */
     if (port==-1)
        rc = sp->writeto(MSG2,strlen(MSG2)+1, 0, argv[2], host);
    else rc = sp->writeto(MSG2, strlen(MSG2)+1, 0, clts1_port);
    if (rc < 0)  {
        cerr << "clts2: writeto fails\n";
        return 4;
    }

    /* read MSG1 from the tli_clts1 process */
    struct t_unitdata *ud = 0;
    if (sp->readfrom(buf, sizeof buf, flags, ud) < 0)    {
        cerr << "clts2: readfrom fails\n";
        return 5;
    }
    cerr << "clts2: read msg: '" << buf << "'\n";

    /* write MSG3 to the tli_clts1 process */
    if (sp->writeto(MSG3, strlen(MSG3)+1, flags, ud) < 0) {
        cerr << "clts2: writeto fails\n";
        return 6;
    }
    return 0;
}
```

The *tli_clts2* program is very similar to the *tli_clts1* program. The difference is that it is invoked with either: (1) an assigned integer address and the integer address of the *tli_clts1* process (for local transport communication); or (2) its service name alone with that of *tli_clts1*, optionally followed by a machine host name (for Internet transport communication). If a service name is specified, the name must be defined in the */etc/services* file. If a host name is not specified with the service name, the host name is assumed.

The *tli_clts2* begins execution by creating a TLI object, based on the given command line arguments. If an Internet transport end point is created, the *gethost* function is called to retrieve the host machine name from either the command line argument (if it is specified) or via the *sysinfo* function call.

After a TLI transport object is created, the process binds a name to the transport object, then writes the MSG2 message (MSG2) to the *tli_clts1* process. Once the MSG2 message is sent, the process waits for the *tli_clts1* process to send it the MSG1 message. The process prints the MSG1 message to the standard output once it has been received.

Finally, the process writes the MSG3 message to the *tli_clts1* process before it terminates itself. The transport end point created by the process is discarded via the *TLI::~TLI* destructor function.

The following console log depicts the interaction of the *tli_clts1* and *tli_clts2* processes. The transport end points created here are based on integer addresses. The *tli_clts1*'s transport is assigned the integer address of 1, while that of the *tli_clts2* process is the address 2:

```
% CC -o tli_clts1 tli_clts1.C -lnsl
% CC -o tli_clts2 tli_clts2.C -lnsl
% tli_clts1 1 &
bind: 1
% tli_clts2 2 1
bind: 2
clts1: read msg: 'Hello MSG2 from clts2'
clts2: read msg: 'Hello MSG1 from clts1'
clts1: read msg: 'Hello MSG3 from clts2'
[1] + Done              tli_clts1 1
```

To run the same programs again using host name/service names, the following two entries are added to the */etc/services* file:

```
utst1                   4046/udp
utst2                   4047/udp
```

The *utst*" is the service name assigned to the *tli_clts1* transport. The *utst2* is the service name for the *tli_clts2* transport. Both services use the *UDP* transport provider, which provides connectionless communication.

The following console log depicts the interaction of the same *tli_clts1* and *tli_clts2* processes, using Internet transport end points:

```
% tli_clts1 utst1 &
bind: 135123
% tli_clts2 utst2 utst1
bind: 135124
clts1: read msg: 'Hello MSG2 from clts2'
clts2: read msg: 'Hello MSG1 from clts1'
clts1: read msg: 'Hello MSG3 from clts2'
[1] + Done              tli_clts1 utst1
```

Note that the output of the above session is identical to that of using local transport end points. No recompilation of the *tli_clts1.C* and *tli_clts2.C* programs is needed. The same two programs can also be run on different machines that are connected via an LAN. For example, suppose the *tli_clts1* program is run on a machine called *fruit* and the *tli_clts2* program is run on *apple*. The invocation syntax of the two programs is:

Run the *tli_clts1* program on the machine *fruit*:

```
fruit % tli_clts1 utst1 &
[1425]
```

Run the *tli_clts2* program on the machine *apple*:

```
apple % tli_clts2 utst2 utst1 fruit
...
```

Once the two programs are connected, the *tli_clts1* output messages are displayed on the machine *fruit*, while the *tli_clts2* messages are displayed on the machine *apple*.

11.8 Summary

This chapter examines BSD UNIX sockets and UNIX System V.3 and V.4 Transport Level Interface for interprocess communication. Sockets and TLI are better methods than are messages, shared memory, or semaphores, in that both sockets and TLI allow processes running on different machines to communicate. This is important for any serious client/server application, where a server is usually run on a power computer and client processes are run on end-user desktop computers.

The syntax of the sockets and TLI APIs are explained in detail in this chapter, as well as sample programs that illustrate their use. Furthermore, a *sock* class and a *tli* class are defined to encapsulate the API interface so as to reduce the learning and programming time of users who wish to use these constructs to create IPC applications.

Of the two IPC methods, TLI is more flexible than are sockets, in that it supports almost all transport protocols. Sockets support only a limited number of transport protocols for each socket type (this is controlled by hardware vendors who implement sockets on their computer systems). Also, TLI has more elaborate methods for transport-specific memory management (the *t_alloc* and *t_free* functions), transport error reporting (*t_error, t_rcvuderr,* and *t_look*), and connection release (*tli_snddis, tli_rcvdis, tli_sndrel* and *tli_rcvrel*). Thus, TLI allows users to create more sophisticated IPC applications.

However, TLI is available only in UNIX System V.3 and V.4, whereas sockets are available on all the latest UNIX systems (BSD 4.2, 4.3, 4.4, and UNIX System V.4). Furthermore, there is already a large volume of IPC applications existing today using sockets. Thus, if portability and interaction with existing socket-based applications are a concern to application developers, they should consider using sockets over TLI.

CHAPTER 12

Remote Procedure Calls

Remote procedure call (RPC) is a mechanism by which a process on one machine invokes another process on either the same or a remote machine to execute a function on its behalf. It is like calling a local function in that the process makes a function call and passes data to the function, then waits for the function to return. What is special here is that the function will be executed by a different process. The RPC process interaction is invariably in a client/server manner, such that the process making an RPC call is a client process, and the process that executes an RPC function in response is a server process. A server process provides one or more service functions that can be invoked by its clients.

Remote procedure call is used in network-based applications to tap network resources on different machines. For example, in a distributed database system, the server process is a database management process, and it manages the data retrieval and storage of the database files. The client processes are the database front-end programs that allow users to input data inquiry and update commands. These user-issued commands are converted by the client processes into RPC calls to the server process. The return values of the RPC calls are depicted to the user by the client processes.

Another example of an RPC application is when a server process is running on a high-powered machine and the client processes are running on less powerful machines. Whenever a client process needs to do compute-intensive jobs, it uses RPC to direct the server to execute those jobs on the server machine. This balances the workloads of the two machines and also maintains an acceptable performance in the client machine.

Other advantages of using RPC are:

- It hides most of the network transport details from programmers. In this way, it allows programmers to develop and maintain their RPC-based programs more easily
- It uses a well-defined data representation format (e.g., XDR or external data representation) to represent data that is transmitted between server and client processes. The data format is machine architecture-independent. It allows machines of different architectures (e.g., Intel x86-based machines and SUN SPARC workstations) to communicate via RPC
- It supports all network transport protocols (connectionless and connection-based)
- Most advanced operating systems (e.g., UNIX, VMS, and Windows-NT) support RPC and are compatible with each other. This allows users to develop network-based applications that run across platforms and operating systems

The following sections examine the RPC programming techniques in more detail.

12.1 History of RPC

There were different implementations of RPC by different companies in the 1980s. Among them were Sun Microsystems's Open Network Computing (ONC) and Apollo Computers's Network Computing Architecture (NCA). Today, most commercial UNIX systems, such as Hewlett Packadd's HP-UX, International Business Machines's AIX, Sun Microsystems's Sun OS 4.1.x, and Santa Cruz Operation's SCO UNIX, all implement RPC based on the ONC method.

However, Sun's Solaris 2.x operating system and UNIX System V.4 implement RPC based on a modified version of the ONC method. The two methods are very similar, namely, they both use external data representation (XDR) format to transmit data across networks, and provide a *rpcgen* compiler to simplify the creation of RPC applications. The two methods differ, in that the ONC-based RPC APIs are based on sockets, whereas System V.4 RPC APIs can be based on sockets or on TLI.

This chapter examines the RPC programming techniques supported by both the ONC and UNIX Systems V.4 methods. In the following sections, unless stated explicitly, most descriptions are applicable to the ONC and System V.4 methods.

12.1 RPC Programming Interface Levels

There are different levels of RPC programming interface. They range from the very top level, where users invoke system-supplied RPC functions in the same manner as calling C

library functions (e.g., *printf*), to the lowest level, where users create RPC programs using RPC APIs. These different programming interface levels are explained in detail in the rest of the chapter.

At the highest level, there are system-supplied RPC functions that users may call directly to collect remote system information. These functions can be used just like ordinary C library functions. The only special setup needed to use them are: (1) special header files that declare the function prototypes; and (2) links between the compiled programs with the *-lrpcsvc* switch. The *librpcsvc.a* library contains these RPC library function object codes.

The advantages of the RPC library functions are that they are easily used and impose little programming effort. However, there are only a few of these RPC library functions defined in a system. Thus, there is limited application for these functions.

The second level of RPC programming is to use the *rpcgen* compiler to generate RPC client and server stub routines automatically. Users write only the client *main* functions (which call the RPC functions) and the server RPC functions to create client and server programs. The *rpcgen* compiler can also generate XDR functions to convert any user-defined data types to XDR format for data transmission between client and server.

The advantage of using the *rpcgen* compiler is that users can focus on writing RPC functions and client main functions. There is no need to know the low-level RPC APIs. This saves programming effort and is less error prone. However, the drawbacks of this approach are that users have little control over any detailed attributes of the network transports used by the client and server programs created by *rpcgen*. They cannot manage the dynamic memory used by XDR functions.

The lowest level of RPC programming interface is to use the RPC APIs to create RPC client and server programs. The advantages of this are that users have direct control of the network transports used by the processes and the dynamic memory management in the XDR functions. However, this comes at the expense of more programming effort on the part of users.

12.2 RPC Library Functions

The RPC library function header is <rpcsvc.h>. Each header corresponds to a set of related RPC library functions and their XDR functions. The object code of these functions is stored in the *librpcsvc.a* library in the standard library directory (e.g., */usr/lib*).

The following are some common RPC library functions and their uses:

RPC library function	Uses
rusers	Gets the number of logged-in users on a remote system
rwall	Writes to a remote system
spray	Sends packets to a remote system
rstat	Gets performance data of a remote system

The following is an example program that makes use of the *rstat* RPC function to determine the up time of one or more remote systems. The up time is the elapsed time between system boot time and the current time. The *rstat* function also collects the average swap and paging statistics etc., of remote systems. The *rstat* function communicates with the *rc.rstatd* daemon running on a remote system via RPC:

```
/* rstat.C: get remote systems up time */
#include <iostream.h>
#include <rpcsvc/rstat.h>

extern "C" enum clnt_stat rstat(char *host, struct statstime *statb);
int main( int argc, char* argv[] )
{
      struct statstime statv;
      if (argc==1)  {
            cerr << "usage: " << argv[0] << " <host> ...\n";
            return 1;
      }
      while (--argc > 0)  {  /* do for each remote system specified */
            if (rstat(*++argv,&statv)==RPC_SUCCESS)  {
                  int delta = statv.curtime.tv_sec - statv.boottime.tv_sec;
                  int hour = delta / 3600;
                  int min  = delta % 3600;
                  cout << "'" << (*argv) << "' up " << hour << "hr. "
                        << (min/60) << " min. " << (min%60) << " sec."
                        << endl;
            }
            else perror("rstat");
      }
      return 0;
}
```

The above program accepts one or more remote system host names as command line argument. For each remote system specified, the process calls *rstat* to collect the remote system statistics. These data are put into the *statv* variable, and the up time of the remote system is computed by subtracting the *statv.boottime.tv_sec* value from that of the *statv.curtime.tv_sec*. The *struct statstime* data type is defined in the <rpcsvc/rstat.h> header.

A sample output of the program is:

```
%    CC rstat.C -o rstat -lrpcsvc -lnsl
%    rstat fruit lemon
'fruit' up 1 hr. 12 min. 31 sec.
'lemon' up 0 hr. 39 min. 24 sec.
%
```

As stated earlier, RPC functions are easy to use and provide the same programming interface as do C library functions. However, there are only a limited number of these functions provided by a system; thus, users need to use the *rpcgen* or the lowest RPC programming interface to create additional RPC functions for their own applications.

12.3 rpcgen

The *rpcgen* compiler is provided on most UNIX systems to support RPC-based application development. The input to the compiler is a user-written text file that describes the following information:

- An RPC program number
- One or more RPC program version numbers
- One or more RPC procedure numbers (in RPC, the term *function* and *procedure* are used interchangeably)
- Any user-defined data types that are used to pass data from and to RPC functions. The *rpcgen* creates XDR functions automatically for each of these data types
- Any optional C code that should be copied directly to the output files generated by the compiler

An RPC function is identified by a program number, a version number, and a procedure number.

An RPC program corresponds to one RPC server process, and the process is responsible for executing any of the defined procedures on a client's behalf. An RPC version specifies the revision level of a set of RPC functions. An RPC version number is an integer value and should start from 1. An RPC procedure number is an unique ID assigned to an RPC function. If there are multiple revisions of a function, the program and procedure numbers of that function are unchanged, only the RPC version number is different. All user-defined RPC function procedure numbers should start from 1. There is always an RPC function whose procedure number is zero in each RPC program. This function can be generated automatically by *rpcgen* or can be defined by users. This function takes no argument and returns nothing. Its purpose is for a client to "ping" the server to confirm the existence of the server process.

For example, given the following program, *print.c*:

```
/* print.C */
#include <iostream.h>
#include <fstream.h>

int print( char* msg )
{
        ofstream ofp( "/dev/console" );
        if (ofp) {
                ofp << msg << endl;
                ofp.close();
                return 1;
        }
        return 0;
}

int main( int argc, char* argv[] )
{
        while (--argc > 0)
                if (print(*++argv))
                        cout << "msg `" << (*argv) << "' delivered OK\n";
                else cout << " msg `" << (*argv) << "' delivered failed\n";
        return 0;
}
```

The program may be compiled and run as follow:

```
%    CC print.C -o print
%    print "Hello world" "Good-bye"
msg `Hello world` delivered OK
msg `Good-bye` delivered OK
```

the messages *Hello world* and *Good-bye* are displayed on the system console window of the machine where the *print* program is run.

To convert the *print* function to a remote procedure, a *print.x* file is created manually for it, as in the following:

```
/* print.x file: this is the input file for rpcgen */
program PRINTPROG
{
        version PRINTVER
```

```
        {
            int PRINT ( string ) = 1;
        } = 1;
} = 0x20000001;
```

In the *print.x* file, the assigned RPC program number, version number, and procedure number of the *print* function are 0x20000001, 1, and 1, respectively. The "program" and "version" are reserved key words for the *rpcgen*, and the PRINTPROG, PRINTVER, and PTINT are user-defined manifested constants for these assigned numbers in relation to the *print* function. By convention, these constants are specified in upper case, but they can be specified in lower case also.

Note the *print* function prototype declaration in *print.x*: the formal argument type of *print* is defined to be *string*, which is an RPC-defined data type for a NULL-terminated character string. Remote procedure call differentiates the data type of character pointers and NULL-terminated character arrays with the introduction of the *string* data type.

The *print.x* file is processed by *rpcgen* as:

```
%   ls
print.x
%   rpcgen print.x
%   ls
print.h   print.x    print_clnt.c   print_svc.c
```

There are three files generated by *rpcgen* from the *print.x* files. These files and their uses are:

File from *rpcgen*	**Uses**
print.h	Header file for the client and server program
print_svc.c	The server program without the RPC function definition
print_clnt.c	The client program stub. It contains all the interface functions to call the RPC server.

As a rule, if the input file to *rpcgen* is called *<name>.x*, the three corresponding output files generated by *rpcgen* are called: *<name>.h*, *<name>_svc.c*, and *<name>_clnt.c*.

The *print.h* header contains the declaration of the PRINTPROG, PRINTVER, and PRINT manifested constants, and the *print* function prototype. The *print.h* file for the above *print.x* file, as generated by *rpcgen* is:

```
#ifndef    _PRINT_H_RPCGEN
#define    _PRINT_H_RPCGEN
#include   <rpc/rpc.h>
#define    PRINTPROG          ((unsigned long)(0x20000001))
#define    PRINTVER           ((unsigned long)(1))
#define    PRINT              ((unsigned long)(1))
extern int * print_1( char**, CLIENT* );
#endif     /*!_PRINT_H_RPCGEN */
```

Note the *print* function declaration in *print.h*: the function name is the original followed by an underscore ("_") character and a procedure number. Thus, the RPC function name for the *print* function of version 1 is *print_1*. Furthermore, the return value of *print_1* is specified as *int** instead of *int*. This is typical of RPC functions: The argument and return value of each RPC function are passed by address, so if a local function accepts a *char**-typed argument, its RPC counterpart accepts a *char***-typed argument. The same is also true for return values: If a local function returns an *int*-typed value, its RPC counterpart returns an *int**-typed value.

The *print_1* function is created manually by a user from the *print* function. Its definition is specified in a separated *print_1.c* file as:

```
/* print_1.c file: server print function definition */
#include <stdio.h>
#include "print.h"

int* print_1( char** msg )
{
      static int result;
      FILE *ofp = fopen ( "/dev/console", "w"
      if (ofp)  {
            fprintf( ofp, "%s\n", *msg );
            fclose( ofp );
            result = 1;
      }
      else result = 0;
      return &result;
}
```

The major differences between the *print* and *print_1* definitions are that the argument and return values are pointer types. Thus, the *result* variable is defined as static in *print_1*, so that the function can return its address as a return value.

Once the *print_1.c* is defined, the server program can be compiled by a C compiler and run as follows:

```
%   cc -o print_server print_1.C print_svc.c -lnsl
%   print_server
```

There is no need to specify the ampersand ("&") symbol when running the server program, as it is specified in *print_svc.c* to be executed in the background automatically. The *-lnsl* option says that the server program requires linking with the *libnsl.so* or *libnsl.a* library to resolve all the RPC external library function references. This option is Sun Solaris-specific and may be replaced by a different option on a different platform.

The client program main function requires definition by the user. The main function calls the remote *print_1* function. The following *print_main.c* is modified from the *main* function in *print.C* to construct the main client program:

```
/* print_main.c: client main function */
#include <stdio.h>
#include "print.h"
int main( int argc, char* argv[] )
{
    int   *res, i;
    CLIENT *cl;
    if (argc<3)    {
        fprintf( stderr, "usage: %s <svc_host > msg ...\n" , argv[0] );
        return 1;
    }
    if (!(cl = clnt_create( argv[1], PRINTPROG, PRINTVER, "tcp"))) {
        clnt_pcreateerror( arhv[1] );
        return 2;
    }
    for (i=argc-1; i > 1; i--)    {
        if (!(res = print_1(&argv[i], cl)))   {
            clnt_perror(cl, argv[1] );
            return 3;
        }
        else if (*res==0)    {
            fprintf( stderr, "print_1 fails\n" );
            return 4;
        }
        else printf( "print_1 succeeds for %s\n", argv[i] );
    }
    return 0;
}
```

The command line arguments to the client program are: the host name of a *print* server and one or more messages to be sent to the server. The client program calls the *clnt_create* function to obtain a handle to communicate with the *print* server. The arguments to the *clnt_create* function are the server host machine name, the server program number, and version number. The last argument, *tcp*, specifies that the client and server processes will communicate via the TCP/IP transport protocol.

If the *clnt_create* call fails, it returns a NULL pointer, and the *clnt_pcreateerror* function is called to depict a diagnostic message for the error. On the other hand, if the *clnt_create* function succeeds, it returns a *CLIENT** handle, which is used as the second argument value in the subsequent *print_1* function call. The *print_1* function definition for the client program is defined in the *print_clnt.c* file. It is a stub, which in turn, calls the *print_1* function in the server program.

If the client's *print_1* function returns a NULL pointer, it means that the remote function call failed. In this case, the specified transport is not available or not working somehow, and the *clnt_perror* function is called to depict the reason of the failure. If *print_1* returns a non-NULL pointer value, the *res* pointer is consulted to obtain the returned status code of the *print_1* function. The possible values of the status code are application-defined and vary for different RPC functions.

The client program is generated by compiling the *print_main.c* and *print_clnt.c* modules together:

```
%    cc -o print_client print_clnt.c print_main.c -lnsl
```

Assume that the *print_server* program is running in the background on a machine called *fruit*. The *print_client* program can then be run as follows on any machine that is connected to *fruit* via a local or wide area network:

```
%   print_client   fruit  "Hello world"    "Good-bye"
print_1 succeeds for 'Hello world'
print_1 succeeds for 'Good-bye'
```

The system console window on *fruit* should display the following messages:

```
Hello world
Good-bye
```

12.3.1 clnt_create

The formal syntax of the *clnt_create* function is:

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

CLIENT* clnt_create ( const char* hostname, const u_long prognum,
                      const u_long versnum, const char* nettype );
```

The *hostname* value is a NULL-terminated character string and specifies the name of a remote machine where the server process is run.

The *prognum* and *versnum* values are the program number and version number, respectively, of the remote function to be called.

The *nettype* value is a NULL-terminated character string that specifies what transport to use for communication between client and server. The possible values of *nettype* and their meanings are:

***nettype* value**	**Meaning**
"netpath"	Choose a transport in the order specified in the NETPATH environment variable. If the NETPATH environment is not set, choose a "*visible*" transport, in the order specified in the */etc/netconfig* file
""	Same interpretation as "*netpath*"
"visible"	Choose a transport in the order specified in the */etc/netconfig* file, which has the "*v*" (visible) flag set
"circuit_v"	Same as "*visible*", but choose only connection-based transport
"datagram_v"	Same as "*visible*", but choose only connectionless transport
"circuit_n"	Same as "*netpath*", but choose only connection-based transport
"datagram_n"	Same as "*netpath*", but choose only connectionless transport
"udp"	Use the UDP transport
"tcp"	Use the TCP transport

If a *nettype* value is "*netpath*", "", "*circuit_n*", or "*datagram_n*", the client and server processes consult the NETPATH environment variable to determine which transport to use

for RPC communication. The NETPATH environment variable is user-defined and contains a colon-delimited list of transports. The following is an example of a shell command defining the NETPATH environment variable:

```
% setenv NETPATH tcp:udp
```

The return value of the function is NULL if it fails or a *CLIENT** pointer that is the handler for communication with a server process.

Note the *clnt_create* function is UNIX System V. 4-specific. The ONC functions to create RPC client handles are:

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

CLIENT* clnttcp_create (struct sockaddr_in* svr_addr,
        const u_long prognum, const u_long versnum, int* sock_p,
        const u_long sendbuf_size, const u_long recvbuf_size);

CLIENT* clntudp_create (struct sockaddr_in* svr_addr,
        const u_long prognum, const u_long versnum,
        struct timeval retry_timeout, int* sock_p);
```

The *clnttcp_create* and the *clntudp_create* functions are the TCP and UDP versions of the *clnt_create* function, respectively. These two functions use sockets as their underlying communication method, and the NETPATH environment variable is not used. Specifically, the *svc_addr* argument value is a pointer to the socket address of a server host name, and the *sock_p* argument value is the pointer to a socket port number of an RPC server. The socket port number may be specified as RPC_ANYSOCK, which means that it can be whatever port actually in use by the server.

Finally, the *retry_timeout* argument value specifies how long a client process should wait for a server response before it sends its request to the server again.

12.3.2 The rpcgen Program

The invocation syntax of the *rpcgen* program is:

```
rpcgen [<options>] <input_file>
```

The *input_file* argument is the path name of *a.x* file created by a user. This file specifies an RPC program number, a version number(s), and a procedure number(s). Furthermore, any user-defined data type used as an input argument and/or return value for RPC functions is also defined in this file.

The *rpcgen* program may take many options, but the following options are of most significance:

***rpcgen* option**	**Meaning**
-K <time>	Specifies when a server process should exist after it has serviced a client request. If this option is not specified, the default *<time>* is 120 seconds. If *<time>* is set to -1, the server process will never terminate
-s <transport>	Specifies a transport to be used for the server process. The possible *<transport>* values are the same as those of the *nettype* values in a *clnt_create* call

For example. the following command invokes *rpcgen* to compile the *msg.x* file. The server program derived from the corresponding *msg_svc.c* file will exist 60 seconds after it has serviced a client request. It will use one of the connection-based and "*visible*" transports, as specified in the */etc/netconfig* file to communicate with its client process.

```
%    rpcgen -K 60 -s circuit_v msg.x
```

12.3.3 A Directory Listing Example Using rpcgen

This section depicts another example of an RPC program generated via the *rpcgen*. The RPC function *scandir* gets a directory path name and an integer flag as input arguments. The function returns a linked list of file names that exist in the named directory. Furthermore, if the input flag value is nonzero, it also returns the UID and the last modification time stamp of each file found.

The *scan.x* input file to *rpcgen* is:

```
/* scan.x: directory listing service */
const MAXNLEN = 255;
typedef string name_t<MAXNLEN>;
```

```
/* input argument data type to scandir()
typedef struct arg_rec *argPtr;
struct arg_rec
{
        name_t          dir_name;
        int             lflag;
};

/* The linked list record structure for one file info */
typedef struct dirinfo *infolist;
struct dirinfo
{
        name_t          name;   /* file name */
        u_int           uid;    /* UID       */
        u_long          modtime; /* last modification time */
        infolist        next;   /* linked-list ptr */
};

/* return data type of scan() */
union res switch (int errno)
{
        case 0:         infolist  list;
        default:        void;
};

program SCANPROG
{
        version SCANVER
        {
                res SCANDIR(argPtr) = 1;
        } = 1;
} = 0x20000100;
```

The input argument type to the *scandir_1* RPC function is the address of a pointer to a *struct arg_rec* typed variable, which specifies a directory path name (the *struct arg_rec::dir_name* field) and an integer flag (the *struct arg_rec::lflag* field). The return data type of the function is nothing if the function fails. Otherwise, a linked list of records is produced, with each record of the type *struct dirinfo* specifying file information. The *name_t* definition states that it is a NULL-terminated character string with, at most, MAXNLEN characters.

The RPC program, version, and procedure numbers are specified as 0x20000100, 1, and 1, respectively.

The *scan.x* is compiled by the *rpcgen* as:

```
%    rpcgen scan.x
```

There are four files generated from this compilation: *scan.h, scan_svc.c, scan_clnt.c,* and *scan_xdr.c*. The *scan_xdr.c* file contains the XDR conversion functions for the *struct arg_rec* and the *struct dirinfo* data values.

The *scandir_1* function for the server program is defined in the *scan_1.c* file:

```
/* scan_1.c: server's scandir function definition */
#include <dirent.h>
#include <string.h>
#include <malloc.h>
#include <sys/stat.h>
#include "scan.h"

res* scandir_1(argPtr* darg)
{
   DIR *dirp;
   struct dirent *d;
   infolist  nl, *nlp;
   struct stat statv;
   static res res;

   if !(dirp = opendir((*darg)->dir_name)))  {
      res.errno = errno;
      return &res;
   }
   xdr_free(xdr_res, &res);
   nlp = &res.res_u.list;
   while (d=readdir(dirp))      {
      nl = *nlp = (infolist)malloc(sizeof(struct dirinfo));
      nl->name = strdup(d->d_name);
      nlp = &nl->next;
      if ((*darg)->lflag)      {
         char pathnm[256];
         sprintf(pathnm,"%s/%s",(*darg)->dir_name,d->d_name);
         if (!stat(pathnm,&statv))    {
            nl->uid = statv.st_uid;
            nl->modtime = statv.st_mtime;
         }
      }
   }
   *nlp = NULL;
```

```
    res.errno = 0;
    closedir(dirp);
    return &res;
}
```

The *scandir_1* function calls the *opendir* API to get a handler for scanning a directory file whose name is specified in the function input argument *(*darg)->dir_name*. If *opendir* fails, the *scandir_1* function returns the *errno* value via the *res.errno* field.

If the *opendir* call returns successfully, the function calls *readdir* repeatedly to get all files in the named directory. For each file obtained, a *struct dirinfo* record is allocated dynamically. The function stores the newly obtained file name and, possibly, the file UID and last modification time (if the *(*darg)->lflag* value is nonzero) in that record. The *struct dirinfo* records are chained together into a linked list, returned via the *res.res_u.list* field.

The server program is compiled and run as follows:

```
%    cc scan_1.c scan_xdr.c scan_svc.c -o scan_svc -lnsl
%    scan_svc
```

The client main program is specified in the *scan_main.c*:

```
/* scan_main.c: main function for the client program */
#include <stdio.h>
#include "scan.h"

int main( int argc, char* argv[])
{
    struct arg_rec  *iarg = (struct arg_rec*)malloc(sizeof(struct arg_rec));
    res   *result;
    infolist nl;

    if (argc!=4)       {
        fprintf( stderr, "usage: %s host directory <long>\n", argv[0] );
        return 1;
    }
    char *server = argv[1];
    iarg->dir_name = argv[2];
    iarg->lflag = 0;
    if (sscanf(argv[3],"%u",&(iarg->lflag))!=1) {
        fprintf( stderr, "Invalid argument: '%s'\n", argv[3] );
        return 2;
```

```
    CLIENT *cl = clnt_create(argv[1], SCANPROG, SCANVER, "visible");
    if (!cl)  {
       clnt_pcreateerror(server);
       return 3;
    }

    if (!(result = scandir_1(&iarg, cl)))  {          // RPC call fails
       clnt_perror(cl, server);
       return 4;
    }

     if (result->errno)  {                  // function returns failure code
       errno = result->errno;
       perror(iarg->dir_name);
       return 5;
    }

    for (nl=result->res_u.list; nl; nl=nl->next)    { // function succeeds
       if (iarg->lflag)
          printf( "...%s, uid=%d, mtime=%s\n" , nl->name, nl->uid,
                ctime(&nl->modtime) );
       else printf ( "...%s\n", nl->name);
    }
    return 0;
}
```

The client program is invoked with the host name of the server process, a remote directory name, and an integer flag that specifies whether the UID and last modification time stamp of files in the specified directory are wanted (*lflag* value is nonzero) or not (*lflag* value is zero).

The client *main* function calls the *clnt_create* to get a handler of the transport end point that connects the specified server process. It then packs all the input argument data to the dynamic memory (pointed to by the *iarg* variable) before it calls the *scandir_1* RPC function. After the RPC function returns, the return value is checked to see whether the RPC call succeeded. If the call succeeded, the remote file information is printed to the standard output accordingly. Otherwise, an appropriate error diagnostic is depicted to the user.

The client program is compiled and run as follows. It is assumed that the server is running on a machine called *fruit*.

```
%    cc scan_main.c scan_xdr.c scan_clnt.c -o scan_cls -lnsl
%    scan_cls  fruit     /etc     1
...magic, uid=2, mtime=Wed Aug  3 11:32:33 1994
...protocols, uid=10, mtime=Wed Aug  3 11:32:30 1994
```

12.3.4 rpcgen Limitations

The fact that *rpcgen* hides the low-level RPC APIs from users has both advantages and disadvantages.

rpcgen has the advantages of reducing programming effort and of being less error prone. Furthermore, users can concentrate more in coding RPC functions and client *main* functions, rather than RPC transport interface functions.

The disadvantages of *rpcgen*, however, are the following:

- Users have no direct control of the transport used by the server and client programs generated by *rpcgen*
- Users cannot manage the dynamic memory used by the XDR functions generated by *rpcgen*
- Most *rpcgen* compilers do not generate C++-compatible client and server stub functions. This may require manual modification of those stubs to make them acceptable for the C++ compiler (note that the *rpcgen* on Sun Microsystems workstations provides the -C option for generating C++-compatible files).

Given these limitations of *rpcgen*, users need to learn the lower level RPC APIs. This would allow users to work around the above *rpcgen* limitations if they become significant obstacles to application development.

12.4 Low-Level RPC Programming Interface

The low-level RPC APIs are declared in the <rpc/rpc.h> header. These APIs include the creation of client and server handles with user-specified transports, the registration of RPC functions to the *rpcbind* daemon, and the calling of remote RPC functions from client processes. Furthermore, client processes can specify authentication methods via these APIs to establish secure connections with server processes.

Before the low-level RPC APIs are presented, the methods to create XDR functions for user-defined data types are covered. This is because some lower level RPC APIs require users to specify the actual data and their corresponding XDR functions for RPC function arguments and return values. By writing their own XDR functions, users also have direct control over dynamic memory allocation and deallocation in those functions.

12.4.1 XDR Conversion Functions

An XDR function is called twice whenever a piece of data is passed between a client process and a server process. For example, when a client passes some data to an RPC func-

tion, the data are converted to XDR format before being transmitted to the network. This conversion process is called *serializing*. Then, before the target RPC function receives the data, the same XDR function is called on the server side to convert the data from XDR format to the data format of the host machine. This process is called *deserializing*. There are built-in basic XDR conversion functions for most RPC basic data types. These basic functions are capable of performing both serializing and deserializing. Furthermore, user-defined XDR functions (which, in turn, call the basic XDR functions) automatically inherit the serializing and deserializing capability. These built-in basic XDR functions are:

RPC data type	**XDR function**
int	xdr_int
long	xdr_long
short	xdr_short
char	xdr_char
u_int	xdr_u_int
u_long	xdr_u_long
u_short	xdr_u_short
u_char	xdr_u_char
float	xdr_float
double	xdr_double
enum	xdr_enum
bool	xdr_bool
string	xdr_string
union	xdr_union
opaque	xdr_opaque

The *u_int, u_long*, etc. data types are the unsigned counterparts of the data types *int, long*, etc. The *bool* data type is converted to the *bool_t* data type by the *rpcgen*, and the *boot_t* data type is defined in C as:

```
typedef enum { TRUE=1, FALSE=0 }    bool_t;
```

The *opaque* data type stands for a sequence of arbitrary bytes. It may be used to declare fixed-size arrays or variable-size arrays, such as the following:

```
opaque   x[56];
opaque   vx<56>;
```

The above definitions are converted to C definitions by *rpcgen* as:

```
char x[56];
struct
{
    u_int   xv_len;          /* actual length of the xv_val array */
    char    *xv_val;         /* dynamic array */
} xv;
```

If users define their own data types, they can use either *rpcgen* to generate the XDR functions for these data types or write their own XDR conversion functions. For example, if the user defines the following data type *struct complex*:

```
struct complex
{
    unsigned        uval;
    char            ary[80];
    int             *ptr;
    long            lval;
    double          dval;
};
```

the XDR function for the *struct complex* is:

```
bool_t xdr_complex ( XDR *xdrs, struct complex* objp)
{
    if (!xdr_u_int(xdrs, &objp->uval)) return FALSE;
    if (!xdr_vector(xdrs, objp->ary, 80,
       sizeof(char), (xdrproc_t)xdr_char))
            return FALSE;
    if (!xdr_pointer(xdrs, &objp->ptr, sizeof(int), (xdrproc_t)xdr_int))
            return FALSE;
    if (!xdr_long(xdrs, &objp->lval)) return FALSE;
    if (!xdr_double(xdrs,&objp->dval)) return FALSE;
    return TRUE;
}
```

All XDR function return values are of type *bool_t*, which is TRUE if a function succeeds, FALSE otherwise. The XDR function for *complex* consists of calling basic XDR functions to convert each field member of the *complex* record. Note that the fixed-size array *complex::ary* is converted by the RPC built-in function *xdr_vector*. The arguments of the *xdr_vector* function are: an XDR pointer that points to a buffer holding the converted data,

the fixed-size array address, the number of elements in the array, the size of each array element, and the XDR function converting each array element.

The *complex::ptr* member is converted by another RPC built-in XDR function *xdr_pointer*. The arguments to the *xdr_pointer* function are: an XDR pointer that points to a buffer holding the converted data, the address of the pointer, the size of the data that the pointer points to, and the XDR function for those same data.

12.4.2 Lower Level RPC APIs

To create RPC-based client and server programs, there are two sets of RPC APIs (one for each). These APIs and their calling sequences in the client and server processes are shown in Figure 12.1.

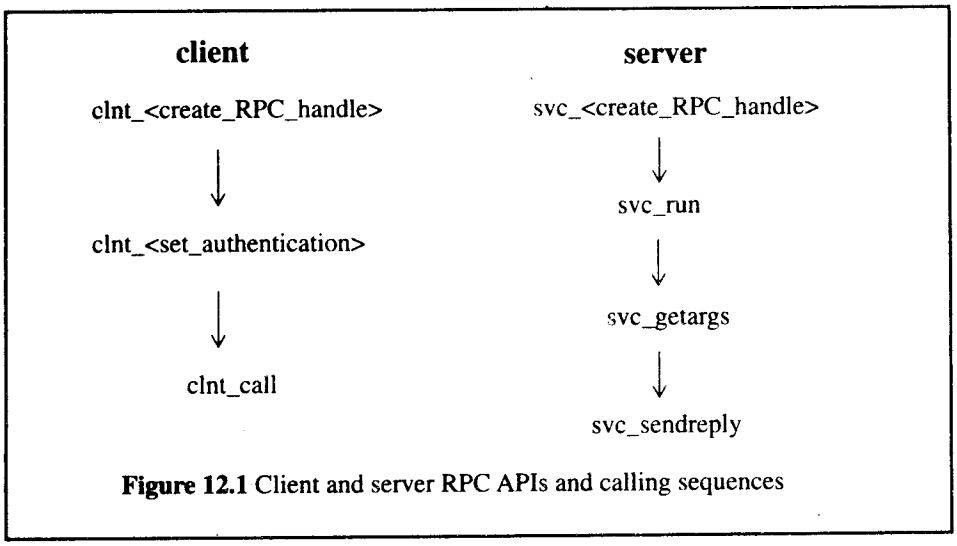

Figure 12.1 Client and server RPC APIs and calling sequences

The *clnt_<create_RPC_handle>* stands for a set of RPC APIs, each of which creates a client handle that can be used to communicate with an RPC server for specified RPC program and a version numbers. These APIs differ in their level of detail in specifying the network transport protocol used in communicating with an RPC server. These APIs are:

Client API	Use
clnt_create	Specifies a generic class of transport to be selected at run time
clnt_tp_create	Specifies a specific transport to be used

Client API	Use
clnt_tli_create	Specifies a TLI transport end point to be used, and the sending and receiving buffer size for RPC communication. The TLI handle is created in the client program

The *clnt_<create_authentication>* stands for a set of RPC APIs, each of which creates an authentication data record to be used by an RPC server to authenticate a client. These APIs are optional and are needed only if an RPC server requires security control. Furthermore, users may create their own RPC client/server authentication schemes and their own RPC authentication functions for their programs. The standard RPC authentication functions for client programs are:

Client API	Use
authnone_create	Creates a NULL authentication data record. The calling RPC server should not require client authentication
authsys_create_default	Creates an authentication record based on System V process access control method
authdes_seccreate	Creates an authentication record whose data is encrypted using the DES encryption method

The *clnt_call* API calls an RPC server to execute an RPC program of a given procedure number. The function call includes the input argument and its XDR function, as well as the address of the variable that receives the return value and its XDR function.

On the server side, the *svc_<create_RPC_handle>* stands for a set of RPC APIs, each of which creates a server handle that can be used to respond to client RPC requests. These APIs differ in their level of detail specifying the network transport protocol used in communicating with RPC clients. These APIs are:

Server API	Use
svc_create	Specifies a generic class of transport to be selected at run time
svc_tp_create	Specifies a specific transport to be used
svc_tli_create	Specifies a TLI transport end point to be used, and the sending and receiving buffer size for RPC communication. The TLI handle is created in the server program

The *svc_run* API is called after a server creates an RPC handle. This function goes into an infinite loop waiting for client RPC requests to arrive and calls a user-defined dispatcher function to service each call. This function may be replaced by a user-defined function, particularly if users want the server to do something else whenever it is not servicing requests.

The dispatcher function calls the *svc_getargs* API to extract any RPC function arguments sent from a client process. It then calls the requested RPC function and uses the *svc_sendreply* API to send the function return value to the client.

The syntax of these APIs is discussed in later sections. The next section introduces two RPC classes that encapsulate low-level RPC API interfacing. These RPC classes provide a simplified RPC programming interface for users, while allowing user control over RPC transport specification and memory management of XDR function data.

12.5 RPC Classes

The section defines two RPC classes: One for constructing a RPC server, and one for construction a RPC client. The main use of these classes is to provide a high-level RPC interface to application developers, so that they don't have to know the details of the ONC or UNIX System V RPC APIs. Furthermore, users may derive their own subclasses from these RPC classes, such that their subclass objects store more data than do the RPC server or client handles, as well as provide additional functions (for example, predefined callback and broadcast functions).

In summary, the advantages of the RPC classes are:

- They hide the differences in APIs between the ONC and System V.4 methods. This makes applications that use these classes portable on most commercial UNIX systems
- They reduce the learning time and programming effort of users in creating RPC applications
- These classes enable users to focus their programming efforts in developing their core client and server functions. This provides the same advantage as does *rpcgen*
- The RPC classes can be modified by users to control RPC transports and the authentication methods used, as well as any dynamic memory management of XDR function data

The following *RPC.h* header defines two RPC classes that encapsulate the RPC client and server APIs.

```
#ifndef RPC_H
#define RPC_H
```

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <rpc/rpc.h>
#include <utmp.h>
#define ADD_FUNC  ADD_PROC
#ifdef   SYSV4
#include <rpc/svc_soc.h>
#include <rpc/pmap_clnt.h>
#include <netconfig.h>

#else                   /* ONC */
#include <rpc/pmap_clnt.h>
#include <sys/socket.h>
#include <netdb.h>
#define AUTH_SYS  AUTH_UNIX
#endif

#define UNDEF_PROGNUM     0x0
#define TCP_BUFSIZ        4098
typedef int (*rpcprog)(SVCXPRT*);

typedef struct
{
     unsigned       prgnum;
     unsigned       vernum;
     unsigned       prcnum;
     rpcprog        func;
} RPCPROG_INFO;

/* RPC server class                                    */
class RPC_svc
{
        int                 rc;
        unsigned            prgnum, vernum;
        static RPCPROG_INFO* progList;
        static int          numProg;
        SVCXPRT             *svcp;
      public:
        /* Dispatch routine */
        static void dispatch( struct svc_req* rqstp, SVCXPRT  *xport )
        {
```

```
        if (rqstp->rq_proc==NULLPROC) {
                svc_sendreply(xport, (xdrproc_t)xdr_void, 0);
                return ;
        }
        uid_t  uid = 0;
        gid_t  gid = 0, gids[20];
        short  len = 0;
        switch (rqstp->rq_cred.oa_flavor)   {
          case AUTH_NONE:
                break;
          case AUTH_SYS: {
#ifdef SYSV4
                struct authsys_parms* authp;
                authp = (struct authsys_parms*)rqstp->rq_clntcred;
#else
                struct authunix_parms* authp;
                authp = (struct authunix_parms*)rqstp->rq_clntcred;
#endif
                uid   = authp->aup_uid;
                gid   = authp->aup_gid;
           }   break;
#ifdef SYSV4
           case AUTH_DES: {
                if (!authdes_getucred(
                        (struct authdes_cred*)rqstp->rq_clntcred,
                        &uid, &gid, &len, (int*)gids)) {
                        svcerr_systemerr(xport);
                        return;
                }
           }   break;
  #endif
           default:
                svcerr_weakauth(xport);
                return;
        }
        /*Example authentication checking*/
        if (uid != getuid() && uid!=(uid_t)0) {
          svcerr_weakauth(xport);
          return;
        }

        for ( int i=0; i < RPC_svc::numProg; i++ )
          if (RPC_svc::progList[i].prcnum==rqstp->rq_proc &&
                RPC_svc::progList[i].vernum==rqstp->rq_vers &&
                RPC_svc::progList[i].prgnum==rqstp->rq_prog)
          {
```

```
                    if ((*RPC_svc::progList[i].func)(xport)
                            !=RPC_SUCCESS)
                        cerr << "rpc server execute prog "
                            << rqstp->rq_proc << " fails\n";
                    break;
                }
            if (i >= RPC_svc::numProg) svcerr_noproc(xport);
        };

        /* Constructor function. Create an RPC server object for the
           given prognum/version
        */
        RPC_svc( unsigned prognum, unsigned versnum,
                    const char* nettype)
        {
#ifdef SYSV4
            rc=svc_create(dispatch, prognum, versnum, nettype);
            if (!rc)
                cerr << "Can't create RPC server for prog: "
                    << prognum << endl;
            else prgnum = prognum, vernum = versnum;
            svcp = 0;
#else
            int proto = 0;
            if (nettype && !strcmp(nettype,"tcp")) {
                svcp=svctcp_create(RPC_ANYSOCK, TCP_BUFSIZ,
                                    TCP_BUFSIZ);
                proto = IPPROTO_TCP;
            } else {
                svcp=svcudp_create(RPC_ANYSOCK);
                proto = IPPROTO_UDP;
            }

            if (!svcp) {
                rc = 0;
                cerr << "Can't create RPC server for prog: "
                    << prognum << endl;
            } else {
                rc = 1;
                prgnum = prognum, vernum = versnum;
            }
            pmap_unset( prognum, versnum );
            if (!svc_register(svcp, prognum, versnum, dispatch, proto))
            {
                cerr << "could not register RPC program/ver: "
                    <<  prognum << '/' <<  versnum << endl;
```

```
                rc = 0:
            }
#endif
    };

    /* create a server handle for call back */
    RPC_svc( int fd, char* transport, u_long progno, u_long versno )
    {
#ifdef SYSV4
        struct netconfig *nconf = getnetconfigent(transport),
        if (!nconf) {
            cerr << "invalid transport: " << transport << endl;
            rc = 0;
            return;
        }
        svcp = svc_tli_create( fd, nconf, 0, 0, 0);
        if (!svcp) {
            cerr << "create server handle fails\n";
            rc = 0;
            return;
        }
        if (progno == UNDEF_PROGNUM)
            progno = gen_progNum( versno, nconf,
                                  &svcp->xp_ltaddr);
        if (svc_reg(svcp, progno, versno, dispatch, nconf)==FALSE)
        {
            cerr << "register prognum failed\n";
            rc = 0;
        }
        freenetconfigent( nconf );
#else
        /* fd should be a socket desc. which may be
            RPC_ANYSOCK */
        if (progno == UNDEF_PROGNUM) {
            progno = gen_progNum ( versno, &fd, transport );
        }
        int     proto = 0;
        if (!strcmp(transport,"tcp")) {
            svcp = svctcp_create( fd, TCP_BUFSIZ, TCP_BUFSIZ );
            if (fd) proto = IPPROTO_TCP;
        }
        else {
            svcp = svcudp_create ( fd );
            if (fd) proto = IPPROTO_UDP;
        }
        if (!svcp) {
```

```
                cerr << "create server handle fails\n";
                rc = 0;
                return;
            }
            if (fd) pmap_unset( progno, versno );
            if (!svc_register(svcp, progno, versno, dispatch, proto)) {
                cerr << "could not register RPC program/ver: "
                     << progno << '/' << versno << endl;
                rc = 0;
                return;
            }
#endif
            prgnum = progno, vernum = versno;
            rc = 1;
    };

    /* return program number */
    u_long progno()   { return prgnum;   };

    /* destructor function */
     RPC_svc()
    {
            pmap_unset( prgnum, vernum );
            svc_unregister( prgnum, vernum );
            if (svcp) svc_destroy(svcp);
    };

    /* Check if a server object is created successfully */
    int good()      { return rc;     };

    /* server pool RPC request from clients */
    static void run() { svc_run();     };


    /* poll for RPC requests. This is for asynchornous RPC call-back */
    static int poll( time_t timeout )
    {
            int read_fds = svc_fds;
            struct timeval stry;
            stry.tv_sec = timeout;
            stry.tv_usec = 0;
            switch (select(32, &read_fds, 0, 0, &stry)) {
               case -1:   return -1;
               case 0:    return 0;  /* no event */
               default:   svc_getreq( read_fds );
            }
```

```
        return 1;
};

/* register an RPC function and start servicing RPC requests */
int run_func( int procnum, rpcprog func )
{
        if (good()) {
           if (func) add_proc( procnum, func );
           run();       /* this willl never return */
        }
        return -1;
};

/* register a new RPC function */
void add_proc( unsigned procnum, rpcprog func )
{
        for (int i=0; i < numProg; i++)
        progList[numProg-1].func = func &&;
        progList[numProg-1].prgnum = prgnum &&;
        progList[numProg-1].vernum = vernum;
        if (++numProg == 1)
           progList = (RPCPROG_INFO*)malloc(
                                   sizeof(RPCPROG_INFO));
        else
           progList = (RPCPROG_INFO*)realloc((void*)progList,
                                sizeof(RPCPROG_INFO)*numProg);
        progList[numProg-1].func = func;
        progList[numProg-1].prgnum = prgnum;
        progList[numProg-1].vernum = vernum;
        progList[numProg-1].prcnum = procnum;
};

/* Called by an RPC function to get argument value from a client */
int getargs( SVCXPRT* transp, xdrproc_t func, caddr_t argp )
{
        if (!svc_getargs( transp, func, argp))  {
           svcerr_decode(transp);
           return -1;
        } else return RPC_SUCCESS;
};

/* Called by an RPC function to send reply to a client */
int reply( SVCXPRT* transp, xdrproc_t func, caddr_t argp )
{
        if (!svc_sendreply(transp, func, argp)) {
           svcerr_systemerr(transp);
           return -1;
        }
```

```
            else return RPC_SUCCESS;
        };

#ifdef SYSV4
        /* Generate a transient RPC program no. */
        static unsigned long gen_progNum( unsigned long versnum,
                                struct netconfig* nconf, struct netbuf* addr)
        {
            static unsigned long transient_prognum = 0x5FFFFFFF;
            while (!rpcb_set( transient_prognum--, versnum,
                    nconf, addr))
                continue;
            return transient_prognum +1;
        };
#endif

        static unsigned long gen_progNum ( unsigned long versnum,
                                            int* sockp, char* nettype )
        {
            static unsigned long transient_prognum = 0x5FFFFFFF;
            int       s, len, proto = IPPROTO_UDP;
            int       socktype = SOCK_DGRAM;
            struct sockaddr_in  addr;

            if (!strcmp(nettype,"tcp")) {
                socktype = SOCK_STREAM;
                proto    = IPPROTO_TCP;
            }

            if (*sockp== RPC_ANYSOCK) {
                if ((s = socket(AF_INET, socktype, 0)) < 0) {
                    perror("socket");
                    return 0;
                }
                *sockp = s;
            }
            else s = *sockp;

            addr.sin_addr.s_addr = 0;
            addr.sin_family      = AF_INET;
            addr.sin_port        = 0;
            len                  = sizeof(addr);

            (void)bind( s, (struct sockaddr*)&addr, len );
            if (getsockname( s, (struct sockaddr*)&addr, &len ) < 0)
            {
```

```
                perror("getsockname");
                return 0;
            }
            while (!pmap_set( transient_prognum--, versnum, proto,
                                addr.sin_port))
                continue;
            return transient_prognum +1;
        };

};/* RPC_svc */

/* RPC client class
class RPC_cls
{
    CLIENT  *clntp;
    char    *server;
  public:
    /*  Constructor function. Create an RPC client object for
        the given server/prognum/version */
    RPC_cls( char* hostname, unsigned prognum, unsigned vernum,
            char* nettype)
    {
#ifdef SYSV4
        if (!(clntp=clnt_create(hostname,prognum,vernum,nettype)))
            clnt_pcreateerror(hostname);
        else {
            server = new char[strlen(hostname)+1];
            strcpy(server,hostname);
        }
#else
        struct hostent* hp = gethostbyname(hostname);
        struct sockaddr_in  server_addr;
        int    addrlen, sock = RPC_ANYSOCK;

        if (!hp)
            cerr << "Invalid host name: '" << hostname << "'\n";
        else {
            addrlen = sizeof(struct sockaddr_in);
            bcopy( hp->h_addr, (caddr_t)&server_addr.sin_addr,
                            hp->h_length);

            server_addr.sin_family = AF_INET;
            server_addr.sin_port = 0;

            if (nettype && !strcmp(nettype,"tcp"))
                clntp=clnttcp_create(&server_addr, prognum, vernum,
```

```
                                        &sock, TCP_BUFSIZ, TCP_BUFSIZ);
                    else {
                        struct timeval  stry;
                        stry.tv_sec = 3;
                        stry.tv_usec = 0;
                        clntp=clntudp_create(&server_addr, prognum,
                                                vernum, stry, &sock);
                    }

                    if (!clntp)
                        clnt_pcreateerror(hostname);
                    else {
                        server = new char[strlen(hostname)+1];
                        strcpy(server,hostname);
                    }
                }
#endif
                if (clntp) set_auth ( AUTH_NONE );
        };

        /* destructor function */
        ~RPC_cls()  { (void)clnt_destroy( clntp ); };

        /* Check if a client object is created successfully */
        int  good() { return clntp ? 1 : 0; };

        /* set authentication data */
        void set_auth( int choice, unsigned timeout = 60 )
        {
                switch (choice) {
                    case AUTH_NONE:
                        clntp->cl_auth = authnone_create();
                        break;
                    case AUTH_SYS:
                    case AUTH_SHORT:
#ifdef SYSV4
                        clntp->cl_auth = authsys_create_default();
#else
                        clntp->cl_auth = authunix_create_default();
#endif
                        break;
                    case AUTH_DES: {
                        char netname[MAXNETNAMELEN+1];
                        des_block  ckey;
                        if (key_gendes(&ckey))  perror("key_gendes");
                        if (!user2netname(netname, getuid(), 0))
```

```
                clnt_perror(clntp,server);
            else clntp->cl_auth = authdes_seccreate(netname,
                                timeout, server, &ckey);
            if (!(clntp->cl_auth)) {
                cerr << "client authentication setup fails\n";
                perror("authdes_seccreate");
                clnt_perror(clntp,server);
                clntp->cl_auth = authnone_create();
            }
        } break;
        default:
            cerr << "authentication method '" << (int)choice
                << "' not yet supported\n";
            clntp->cl_auth = authnone_create();
    }
};

/* Call an RPC function */
int call( unsigned procnum, xdrproc_t xdr_ifunc, caddr_t argp,
        xdrproc_t xdr_ofunc, caddr_t rsltp, unsigned timeout = 20 )
{
        if (!clntp) return -1;
        struct timeval timv;
        timv.tv_sec  = timeout;
        timv.tv_usec = 0;
        if (clnt_call(clntp, procnum, xdr_ifunc, argp,
                xdr_ofunc, rsltp, timv)!=RPC_SUCCESS) {
            clnt_perror(clntp, server);
            return -2;
        }
        return RPC_SUCCESS;
};

/* Support RPC broadcast */
static int broadcast( unsigned prognum, unsigned versnum,
                unsigned procnum, resultproc_t callback,
                xdrproc_t xdr_ifunc, caddr_t argp,
                xdrproc_t xdr_ofunc, caddr_t rsltp,
                char* nettype = "datagram_v")
{
#ifdef SYSV4
        return rpc_broadcast(prognum, versnum, procnum,
                xdr_ifunc, argp, xdr_ofunc, rsltp, callback, nettype);
#else
        return clnt_broadcast(prognum, versnum, procnum,
                xdr_ifunc, argp, xdr_ofunc, rsltp, callback);
```

```
#endif
    };
    /* set client time-out period */
    int set_timeout( long usec )
    {
            if (!clntp) return -1;
            struct timeval timv;
            timv.tv_sec = 0;
            timv.tv_usec = usec;
            return clnt_control( clntp, CLSET_TIMEOUT, (char*)&timv);
    };
    /* get client time-out period */
    long get_timeout()
    {
            if (!clntp) return -1;
            struct timeval timv;
            if (clnt_control( clntp, CLGET_TIMEOUT, (char*)&timv)==-1)
            {
               perror("clnt_control");
               return -1;
            }
            return timv.tv_usec;
    };
};
#endif /* _RPC_H */
```

The calling sequences of these RPC member functions in a typical client and server are shown in Figure 12.2.

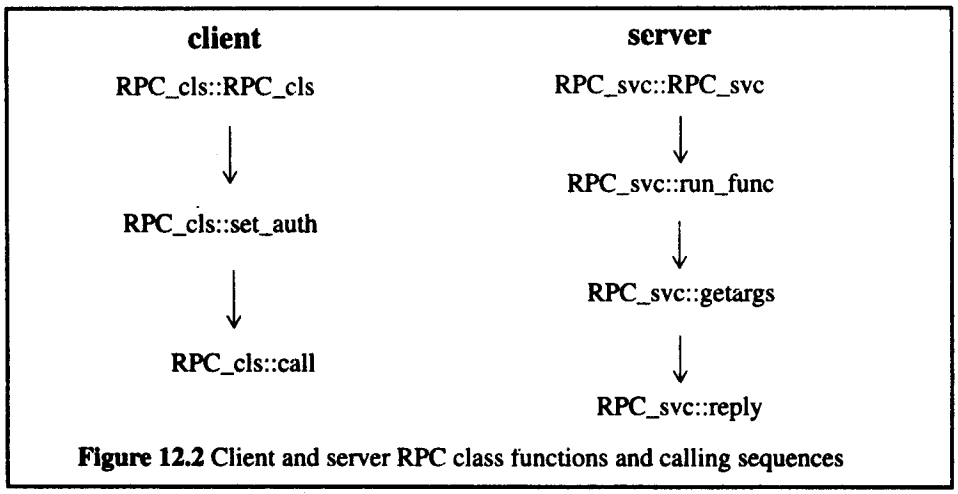

Figure 12.2 Client and server RPC class functions and calling sequences

The *RPC_svc::RPC_svc* constructor function creates a server RPC handle for the given RPC program and version numbers. Furthermore, this function registers a user-defined dispatcher function that is called when a client calls an RPC function managed by the server. The last argument to *RPC_svc::RPC_svc* is the *nettype* value, which defines the transport protocol to be used between the server and its clients. This function internally calls the *svc_create* API to create the server handle. However, users can easily modify the *RPC_svc::RPC_svc* function to call the *svc_tp_create* or *svc_tli_create* API instead.

The *RPC_svc::run_func* is called to register an RPC function and its assigned procedure number to the server, then causes the server to block and wait for client RPC calls (via the *svc_run* API). Note that if a user wishes to register more than one RPC function on a server, the server program will be changed accordingly, as follows:

```
RPC_svc *svcp = new RPC_svc(....);
if (!svcp || !svcp->good()) return 1;
svcp->add_func( <procnum1>, prog1 );
...
svcp->add_func( <procnumN-1>, progN-1);
svcp->run_func( <procnumN>, progN);
```

When a client RPC request arrives, the dispatcher function registered via the *RPC_svc::RPC_svc* function is called to service the client request. The dispatcher function does the following:

- Checks that the client-requested RPC number is 0. If it is, the client is simply pinging the server, and it simply sends a reply to the client with a NULL return value
- If the client specifies any authentication data, the function validates the data and flags an authentication error if the validation fails. Note that in the current function, the client authentication is optional. This may need to be changed for secure RPC transactions. The server should always insist that clients send authentication data in each RPC call. The RPC authentication method is discussed in a latter section
- After the client authentication succeeds, the dispatcher function calls the RPC function requested by the client

Each RPC function called by the *RPC_svc::dispatch* function should have the function prototype of:

```
int <function_name> ( SVCXPRT* );
```

where the argument is a transport handler for communication with a client's transport end point. The function return value is zero (RPC_SUCCESS) if it succeeds, nonzero other-

wise. In addition to these, the RPC function should expect a global *RPC_svc** pointer (whose variable name is application-defined) that holds the address of the RPC server handle. The function should call *RPC_svc::getargs* to extract any arguments from the calling client and use the *RPC_svc::reply* to send a return value to the client.

In a client program, the client handle for specified RPC program and version numbers is obtained via the *RPC_cls::RPC_cls* constructor function. The constructor function internally calls the *clnt_create* API to acquire the client handle. Like the *RPC_svc* constructor function, the *RPC_cls::RPC_cls* function may be changed by users to use the *clnt_tp_create* or *clnt_tli_create* API instead.

Client authentication data may be set via the *RPC_cls::set_auth* call. The *RPC_cls* class currently supports the AUTH_NONE, AUTH_SYS, and AUTH_DES methods, but users may modify the *RPC_cls::set_auth* function to implement their own authentication methods. The AUTH_NONE, AUTH_SYS, and AUTH_DES authentication methods are described in a later section. Note that ONC uses AUTH_UNIX instead of the AUTH_SYS.

The client calls an RPC function via the *RPC_cls::call* function. The arguments to the *RPC_cls::call* function are: an RPC procedure number, an XDR function to convert input arguments to the XDR format, the address of a variable that holds the input arguments, an XDR function to convert the RPC function return value (from XDR format to local machine format), and the address of a variable holding the RPC function return value. This *RPC_cls::call* returns RPC_SUCCESS if it succeeds, a nonzero value otherwise.

The static *RPC_cls::broadcast* function supports RPC broadcast requests from a client process to all server processes on a network. This function is described in detail in section 12.8.

Finally, applications that use the RPC classes should be compiled with the following options on different commercial UNIX systems:

UNIX system	CC compile options
Solaris 2.x, SCO 5.x	-DSYSV -lsocket -lnsl
Sun OS 4.1.x	-lnsl
HP-UX 9.0.x, 10.x	None
IBM AIX 3.x and 4.x	-lrpcsvc
SCO 3.x	-lsocket

The above compile options specify which RPC system libraries to be linked with user applications on the various UNIX systems. The *-DSYSV4* option is needed on Sun's Solaris 2.x system, so that UNIX System V.4 RPC APIs are used instead of the ONC APIs.

To illustrate the use of RPC classes, the remote message printing programs, as shown in Section 12.3, are rewritten as follows. Note that the new client and server programs are simpler than their previous versions, which use *rpcgen*:

The client program *msg_cls2.C* is:

```
/* client program: using low-level RPC APIs */
#include "msg2.h"
#include "RPC.h"

int main(int argc, char* argv[])
{
   int res;
   if (argc<3)    {
      cerr << "usage: " << argv[0] << " host msg <nettype>\n";
      return 1;
   }

   /* create a client handler to an RPC server */
   RPC_cls cl( argv[1], MSGPROG, MSGVER,
                        argc>=4 ? argv[3] : "netpath");
   if (!cl.good()) return 1;

   /* call the printmsg RPC function. return value is set to res */
   if (cl.call( PRINTMSG, (xdrproc_t)xdr_string, (caddr_t)&argv[2],
               (xdrproc_t)xdr_int, (caddr_t)&res) != RPC_SUCCESS
      return 3;

   /* check RPC function's return value */
   if (res!=0)
      cerr << "clnt: call printmsg fails\n";
   else cout << "clnt: call printmsg succeeds\n";

   return 0;
}
```

The client program is invoked with two or three arguments. The first argument is the host name of the machine where the RPC server is running. This may be a local machine name if the client and server are both running on the same machine. The second argument to the client program is a character string message sent to the RPC server for printout. The optional third argument is the transport to be used by the client to communicate with the server. If the third argument is not specified, the default is "*netpath*". The legal values for the third argument and their meanings are the same as that for the *nettype* argument of the *clnt_create* API, as described in Section 12.3.1.

The *msg2.h* is user-defined and contains only the declaration of the *printmsg* function RPC program, version, and procedure numbers:

```
/* msg2.h */
#ifndef MSG2_H
#define MSG2_H

#include <rpc/rpc.h>

#define MSGPROG ((unsigned long)(0x20000001))
#define MSGVER ((unsigned long)(1))
#define PRINTMSG ((unsigned long)(1))

#endif /* !MSG_H */
```

The RPC server program that corresponds to the client program is *msg_svc2.C*:

```
/* server program: low-level RPC APIs */
/*  usage: msg_svc2  <transport> */
#include <iostream.h>
#include <fstream.h>
#include "msg2.h"
#include "RPC.h"
static RPC_svc *svcp = 0;              // RPC server handle
/* the RPC function */
int printmsg( SVCXPRT* xtrp )
{
   int      res  = 0;                  // holds the return status code
   char     *msg = 0;                  // hold client's function argument

   /* get function argument from client */
   if (svcp->getargs( xtrp, (xdrproc_t)xdr_wrapstring,(caddr_t)&msg) !=
                RPC_SUCCESS)
      return -1;

   /* get argument successful. Open the system console for output */
   ofstream ofs ("/dev/console");
   if (ofs)
      ofs << "server:' " << msg << "'\n";
   else res = -1;

   /* send return status code to client */
   if (svcp->reply(xtrp, (xdrproc_t)xdr_int, (caddr_t)&res)
          !=RPC_SUCCESS)
      res = -1;
```

```
    /* RPC function completes. Send a success return code to dispatch */
    return res;
}

/* main server function */
int main(int argc, char* argv[])
{
    /* create a server for the given program number and version number */
    svcp = new RPC_svc( MSGPROG, MSGVER,
                        argc==2 ? argv[1] : "netpath");

    /* register the given RPC function and then waits for clients' RPC
       requests */
    if (svcp && svcp->run_func( PRINTMSG, printmsg )) ;

    return 0; /* shouldn't get here unless the server handle creation failed */
}
```

The server program is invoked with either no argument or a *nettype* specification. If no *nettype* argument is specified, the "*netpath*" value is used as the default.

The server process calls the *RPC_svc::RPC_svc* function to create a server handle for the given RPC program number, version number, and *nettype* value. After that, the server calls the *RPC_svc::run_func* to register the *printmsg* function as a client-callable RPC function, then calls the *svc_run* to poll client RPC requests.

When a client's RPC request arrives, the *svc_run* function calls the *RPC_svc::dispatch* function to handle the request. The *RPC_svc::dispatch* function is responsible for checking that the client RPC procedure number is correct and that the client authentication, if specified, is valid. The RPC function is then called.

The RPC function, as invoked by the *RPC_svc::dispatch* function, calls the *RPC_svc::getargs* function to get the client's argument data. When the RPC function returns, it sends its return value back to the client via the *RPC_svc::reply* function.

The final piece of the source code needed for this example is a separate C file. *RPC.C* contains definitions of the *RPC_svc::numProg* and *RPC_svc::progList* static variables. The *RPC_svc::progList* is a pointer to a dynamic array that keeps track of each RPC function corresponding to a unique combination of a program number, version number, and procedure number. The *RPC_svc::numProg* contains the number of valid entries in the *RPC_svc::progList* array.

The *RPC.C* file content is:

```
#include "RPC.h"
int         RPC_svc::numProg = 0;
RPCPROG_INFO *RPC_svc::progList = 0;
```

The *printmsg* client and server programs are compiled (on a Sun's Solaris 2.x system) and run as follows:

```
%    CC -DSYSV4 -c RPC.C
%    CC -DSYSV4 msg_cls2.C RPC.o -o msg_cls2 -lsocket -lnsl
%    CC -DSYSV4 msg_svc2.C RPC.o -o msg_svc2 -lsocket -lnsl
%    msg_svc2 &
[135]
%    msg_cls2 fruit "Hello RPC world"
clnt: call printmsg succeeds
```

In the above sample execution, both the client and server processes are run on a machine called *fruit*. The server is run explicitly in the background, and the client is invoked with the message string *Hello RPC world*. After the client runs, the server prints the message *server: 'Hello RPC world'* to the system console of *fruit*.

To aid users in better understanding the operation of RPC classes, the low-level RPC APIs are presented in the next few sections.

12.5.1 svc_create

The syntax of the *svc_create* function is:

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

int     svc_create (void (*dispatch)(struct svc_req*, SVCXPRT *),
                    u_long prognum, u_long versnum, char* nettype);
```

The *svc_create* function creates a transport end point for the given *nettype* value in a server process. The server monitors all RPC calls to the given program and version numbers. Furthermore, for each of these RPC requests, the *dispatch* function is called to respond to it.

The possible values and their meanings for the *nettype* argument are shown in Section 12.3.1.

The *dispatch* function is user-defined and takes two arguments. The first argument contains client RPC call information that is useful when the server responds to the call. Specifically, the *struct svc_req* data type is defined by the <rpc/svc.h> header as:

```
struct svc_req
{
   u_long              rq_prog;      /* service program number */
   u_long              rq_vers;      /* service protocol version */
   u_long              rq_proc;      /* the desired procedure */
   struct opaque_auth rq_cred;       /* raw cred. from the wire */
   caddr_t             rq_clntcred;  /* read only cooked cred. */
   struct __svcxprt *  rq_xprt;      /* associated transport */
};
```

where the *rq_prog*, *rq_vers*, and *rq_proc* fields contain the RPC function's program, version, and procedure numbers, respectively, that a client wishes to invoke. The *rq_cred* and *rq_clntcred* fields contain client authentication data accessible by the *dispatch* function to authenticate the client. The *rq_xprt* field contains the client transport information and is generally ignored by the *dispatch* function.

The second argument of the *svc_create* is the transport end-point handle. It is passed to the RPC function, which then uses it to get the function argument values from a client. It is also used to send return values to the client.

The return value of the function is a nonzero server handle if it succeeds, zero if it fails.

The ONC functions to create RPC server handles are:

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

SVCXPRT* svctcp_create ( int svr_addr, const u_long sendbuf_size,
                                       const u_long recvbuf_size);

SVCXPRT* svcudp_create ( int svr_addr );
```

The *svctcp_create* and the *svcudp_create* functions are the TCP and UDP versions of the *svc_create* function, respectively. Furthermore, these two functions use sockets as their underlying communication method. Specifically, the *svc_addr* argument value is a socket port number used by an RPC server to communicate with its clients. The socket port number may be specified as RPC_ANYSOCK, which means it can be any port number assigned by the host system.

Finally, the *sendbuf_size* and *recvbuf_size* argument values specify buffer sizes to send and receive data between a server and its clients.

12.5.2 svc_run

The syntax of the *svc_run* function is:

```
#include <rpc/rpc.h>
void    svc_run (void );
```

This function is called by an RPC server to wait for client RPC calls to arrive. When an RPC call arrives, the function calls a *dispatch* function that was registered via the *svc_create, svc_tp_create,* or *svc_tli_create* APIs to service the request.

This function does not return.

12.5.3 svc_getargs

The syntax of the *svc_getargs* function is:

```
#include <rpc/rpc.h>
boot_t  svc_getargs (SVCXPRT* xprt, xdrproc_t* func, caddr_t argp);
```

This function is called by the RPC function in a server process. It is called to retrieve function arguments that are sent by a client process. The *xprt* argument is a transport handle that is connected to a client process. The *argp* argument holds the address of a variable where client argument data are placed. Finally, the *func* argument is a pointer to an XDR function that is used to deserialize client argument data to the server's host machine data format.

This function returns TRUE if it succeeds, FALSE otherwise.

12.5.4 svc_sendreply

The syntax of the *svc_sendreply* function is:

```
#include <rpc/rpc.h>
boot_t  svc_sendreply (SVCXPRT* xprt, xdrproc_t* func, caddr_t resultp);
```

This function is called by an RPC function in a server process. It is called to send return values to a client process. The *xprt* argument is a transport handle that connected to a client process. The *resultp* argument holds the address of a variable where the function return values are placed. Finally, the *func* argument is a pointer to an XDR function used to serialize the return value to XDR format.

This function returns TRUE if it succeeds, or FALSE otherwise.

12.5.5 clnt_create

The syntax of the *clnt_create* function is:

```
#include <rpc/rpc.h>
CLIENT*  clnt_create (char* hostnm, u_long prognum, u_long versnum,
                      const char* nettype);
```

This function creates a handle to communicate with an RPC server. The *hostnm* argument is the name of the machine where the RPC server is running. The *prognum* and *versnum* arguments identify the RPC server by the RPC program and version numbers. The *nettype* argument specifies the transport used in connecting to the server process.

The possible values and meanings of the *nettype* argument are shown in Section 12.3.1.

The function return value is a nonzero client handle if it succeeds, NULL if it fails. If the function fails, users may call the *clnt_pcreateerror* API to print a more detailed error diagnostic message to the standard output. The function prototype of the *clnt_pcreateerror* API is:

```
void clnt_pcreateerror( const char* msg_prefix_string );
```

The *msg_prefix_string* argument is a user-defined message string that is depicted, along with the diagnostic message from the *clnt_pcreateerror* function.

12.5.6 clnt_call

The syntax of the *clnt_call* function is:

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

enum clnt_stat
    clnt_call (CLIENT* clntp, u_long funcnum, xdrproc_t argfunc,
               caddr_t argp, xdrproc_t resfunc, caddr_t resp,
               struct timeval timv );
```

This function is called in a client process to invoke an RPC function. The *clntp* argument is the client handle obtained from a *clnt_create, clnt_tp_create,* or *clnt_tli_create* API. The *funcnum* argument is the RPC function procedure number. The *argfunc* argument is the address of an XDR function used to serialize the client input argument data to XDR format before they are sent to the RPC function. The *resfunc* argument is the address of an XDR function used to deserialize RPC function return values to the client's data format. Finally, the *timv* argument specifies the time-out limit (in CPU seconds or microseconds) for this call.

The function return value is RPC_SUCCESS if it succeeds, a nonzero return code if it fails. If it fails, the client process may call *clnt_perror* to print a more detailed error diagnostic message to the standard output. The function prototype of the *clnt_perror* API is:

```
void clnt_perror( const CLIENT* clntp, const char* msg_prefix );
```

The *prefix_string* argument is a user-defined character string that is depicted along witn the diagnostic message from the *clnt_pcreateerror* function. The *clntp* argument is the client handle identifying the calling process.

12.6 Managing Multiple RPC Programs and Versions

The RPC classes can be used to create a server process that manages multiple RPC programs. Each program may contain one or more versions of a set of RPC functions. The following example illustrates how this is done.

The server program in this example maintains two RPC programs: The first program number is PROG1NUM, the second is PROG2NUM. PROG1NUM contains two versions (VERS1NUM and VERS2NUM) of an RPC function whose procedure number is FUNC1NUM. The program also contains another RPC function whose version and program numbers are VERS1NUM and FUNC2NUM, respectively. The second program

(PROG2NUM) consists of one RPC function whose version and procedure numbers are VERS1NUM and FUNC2NUM, respectively. The declarations of these RPC program, version, and procedure numbers are contained in the *test.h* header:

```
#ifndef TEST_H
#define TEST_H

#define PROG1NUM  0x20000010
#define PROG2NUM  0x20000015

#define VERS1NUM  0x1
#define VERS2NUM  0x2

#define FUNC1NUM  0x1
#define FUNC2NUM  0x2

#endif
```

The RPC server program is *test_svc.C*:

```
#include "RPC.h"
#include "test.h"

RPC_svc *svc1p, *svc2p, *svc3p;

/* RPC function: prog_no=1, vers_no=1, proc_no=1 */
int func1_1_1 (SVCXPRT* xprt)
{
   cerr << "*** func1_1_1 called\n";
   svc1p->reply(xprt, (xdrproc_t)xdr_void, 0);
   return RPC_SUCCESS;
}

/* RPC function: prog_no=1, vers_no=1, proc_no=2 */
int func1_1_2 (SVCXPRT* xprt)
{
   cerr << "*** func1_1_2 called\n";
   svc1p->reply(xprt, (xdrproc_t)xdr_void, 0);
   return RPC_SUCCESS;
}

/* RPC function: prog_no=1, vers_no=2, proc_no=1 */
int func1_2_1 (SVCXPRT* xprt)
{
   cerr << "*** func1_2_1 called\n";
```

```
    svc2p->reply(xprt, (xdrproc_t)xdr_void, 0);
    return RPC_SUCCESS;
}

/* RPC function: prog_no=2, vers_no=1, proc_no=1 */
int func2_1_1 (SVCXPRT* xprt)
{
    cerr << "*** func2_1_1 called\n";
    svc3p->reply(xprt, (xdrproc_t)xdr_void, 0);
    return RPC_SUCCESS;
}

/* server main function */
int main(int argc, char* argv[])
{
    char* nettype = (argc>1) ? argv[1] : "netpath";

    /* create server handle for prog_no=1, vers=1 */
    svc1p = new RPC_svc ( PROG1NUM, VERS1NUM, nettype );

    /* create server handle for prog_no=1, vers=2 */
    svc2p = new RPC_svc ( PROG1NUM, VERS2NUM, nettype );

    /* create server handle for prog_no=2, vers=1 */
    svc3p = new RPC_svc ( PROG2NUM, VERS1NUM, nettype );

    if (!svc1p->good() || !svc2p->good() || !svc3p->good())    {
        cerr << "create server handle(s) failed\n";
        return 1;
    }
    /* register a function: prog_no=1, vers_no=1, proc_no=1,
           func=func1_1_1*/
    svc1p->add_func( FUNC1NUM, func1_1_1 );

    /* register a function: prog_no=1, vers_no=1, proc_no=2,
           func=func1_1_2*/
    svc1p->add_func( FUNC2NUM, func1_1_2 );

    /* register a function: prog_no=1, vers_no=2, proc_no=1,
           func=func1_2_1*/
    svc2p->add_func( FUNC1NUM, func1_2_1 );

    /* register a function: prog_no=2, vers_no=1, proc_no=1,
           func=func2_1_1*/
    svc3p->add_func( FUNC1NUM, func2_1_1 );
```

```
    /* wait for clients' RPC requests for all servers */
    RPC_svc::run();
    return 0;
}
```

The server program takes an optional argument from the command line, which specifies the correct transport type to use. If this is not specified, the default *nettype* value is "*netpath*".

The server creates three *RPC_svc* objects, one for each version of the RPC program it manages:

RPC_svc	Program Managed	Version Managed
svc1p	PROG1NUM	VERS1NUM
svc2p	PROG1NUM	VERS2NUM
svc3p	PROG2NUM	VERS1NUM

Once all three *RPC_svc* objects are created successfully, the server registers the RPC functions via *RPC_svc* objects. The name of each RPC function is constructed as: the prefix string *func*, followed by a program number, an underscore, a version number, another underscore, and finally, a procedure number. Thus, a function named *func1_2_1* means the function is version 2 of procedure 1 in RPC program 1.

After all the RPC functions are registered, the server calls the *RPC_svc::run* function to wait for client RPC requests to arrive. When any one of these requests arrives, the *RPC_svc::dispatch* function is called, which, in turn, calls one of the registered RPC functions (according to the client's specified program, version, and procedure numbers).

The client program for this example is *test_cls.C*:

```
#include "RPC.h"
#include "test.h"

int main(int argc, char* argv[])
{
   if (argc < 2)  {
      cerr << "usage: " << argv[0] << " <server-host> [<nettype>]\n";
      return 1;
   }

   char* nettype = (argc > 2) ? argv[2] : "netpath";

   while (1)  {
```

```
    unsigned progid, progno, verno, procno;
    /* get desire RPC program no, version no, and function no */
    do {
       cout << "Enter prog#, ver#, func#: " << flush;
       cin >> progno >> verno >> procno;
       if (cin.eof()) return 0;
    } while (!cin.good());

     /* translate user program no to internal number */
    progid = (progno==1) ? PROG1NUM : PROG2NUM;

    /* create a client handle to the requested RPC server */
    RPC_cls *clsp = new RPC_cls ( argv[1], progid, verno, nettype);
    if (!clsp->good())   {
       cerr << "create client handle(s) failed\n";
       return 2;
    }

    /* call the user-requested RPC function */
    if (clsp->call( procno, (xdrproc_t)xdr_void, 0, (xdrproc_t)xdr_void, 0 )
              != RPC_SUCCESS)
       cerr << "client call RPC function fails\n";

    delete clsp;
  }
  return 0;
}
```

The client program is invoked with the server host machine name, and optionally, a *nettype* value. If no *nettype* value is specified, it defaults to "*netpath*".

The client program is an interactive program and prompts a user to enter the program, version, and procedure numbers for each RPC function called. For each set of numbers obtained the client process creates a *RPC_cls* object and calls the requested function via that object. The client process terminates when EOF is encountered in the input stream.

The server and client programs are compiled and run as shown below. In the example, both the server and client are run on a machine called *fruit*.

```
%    CC -DSYSV4 test_cls.C RPC.C -o test_cls -lsocket -lnsl
%    CC -DSYSV4 test_svc.C RPC.C -o test_svc -lsocket -lnsl
%    test_svc &
[1235]
%    test_cls fruit
Enter prog#, ver#, func#: 1 1 1
```

```
***f unc1_1_1 called
Enter prog#, ver#, func#: 1 1 2
*** func1_1_2 called
Enter prog#, ver#, func#: 1 2 1
*** func1_2_1 called
Enter prog#, ver#, func#: 2 1 1
*** func2_1_1 called
Enter prog#, ver#, func#: 1 1 0
Enter prog#, ver#, func#: 4 1 2
fruit: RPC: Procedure unavailable
client call RPC function fails
Enter prog#, ver#, func#: ^D
```

In the above sample run, the RPC functions were called in this order: *func1_1_1*, *func1_1_2*, *func1_2_1*, and *func2_1_1*. The user input *1 1 0* causes the client to ping the RPC server for the program PROG1NUM (version VERS1NUM). There is no response message depicted for this ping operation. Finally, the user inputs *4 1 2* in an attempt to call a nonexistent RPC function, and error messages are flagged from both the *RPC_cls::call* function and the client *test_cls.C* program.

12.7 Authentication

Some RPC services are restricted to designated classes of users who can make use of them. This requires client processes to authenticate themselves to the servers before requested RPC functions can be called. UNIX systems provide a few basic authentication methods for users and allow users to define their own authentication methods.

The UNIX System V.4 RPC built-in authentication methods are: AUTH_NONE, AUTH_SYS, AUTH_SHORT, and AUTH_DES. The ONC RPC authentication methods are: AUTH_NONE, AUTH_UNIX (equivalent to AUTH_SYS), and AUTH_DES. These authentication methods are described in more detail in the following sections.

To support authentication, (whether it is a system-supplied or user-defined method) the *struct svc_req* argument data passed from a client to an RPC server dispatch function specifies the target function's numbers (program, version, and procedure) and client authentication data. Specifically, the *struct svc_req* type is declared as:

```
struct svc_req
{
   u_long              rq_prog;    /* service program number */
   u_long              rq_vers;    /* service protocol version */
   u_long              rq_proc;    /* the desired procedure */
   struct opaque_auth  rq_cred;    /* raw cred. from the wire */
```

```
    caddr_t             rq_clntcred;    /* read only cooked cred */
    struct __svcxprt *  rq_xprt;        /* associated transport */
};
```

where the *struct opaque_auth* data type is declared in the <rpc/auth.h> header as:

```
struct opaque_auth
{
    enum_t              oa_flavor;      /* authentication method */
    caddr_t             oa_base;        /* pointer to custom auth. data */
    u_int               oa_length;      /* size of the data pt. by oa_base */
};
```

The *opaque_auth::oa_flavor* field specifies which authentication method is used by the client. If the argument value is AUTH_NONE, AUTH_SYS, AUTH_SHORT, or AUTH_DES, the *opaque_auth::oa_base* and *opaque_auth::oa_length* fields are don't-care. The *svc_req::rq_clntcred* field points to a data record that contains the corresponding authentication data.

However, if the *opaque_auth::oa_flavor* field value is not one of the AUTH_xxx, the *opaque_auth::oa_base* field points to a user-defined authentication data record, and the *opaque_auth::oa_length* field contains the size of the data record referenced by the *opaque_auth::oa_base* field.

The following three sections examine the UNIX system built-in RPC authentication methods. Users can create their own authentication methods based on these.

12.7.1 AUTH_NONE

This is the default UNIX System V RPC authentication method, which actually does not use any authentication at all. A client can explicitly set this authentication method by calling the *authnone_create* API, as follows:

```
CLIENT* clntp = clnt_create( ... );
if (clntp)   {
    clntp->cl_auth = authnone_create();
    clnt_call (clntp, ...);
}
```

After a client calls the *authnone_create* function, any RPC dispatch functions called by this client receive the *svc_req::rq_cred.oa_flavor* value as AUTH_NONE. These functions

should ignore the *svc_req::rq_cred.oa_base*, *svc_req::rq_cred.oa_length*, and the *svc_req::rq_clntcred* values.

12.7.2 AUTH_SYS (or AUTH_UNIX)

This method uses the UNIX system process access control method, which is based on process user ID and group GID to authenticate clients. A client can explicitly set this authentication method by calling the *authsys_create_default* API, as follows:

```
CLIENT* clntp = clnt_create( ... );
if (clntp) {
   clntp->cl_auth = authsys_create_default();
   clnt_call (clntp, ...);
}
```

After a client calls the *authsys_create_default* function, any RPC dispatch functions called by this client will receive the *svc_req::rq_cred.oa_flavor* value as AUTH_SYS, and the *svc_req::rq_clntcred* field will point to a data record with the following structure:

```
struct authsys_parms
{
   u_long          aup_time;      /* auth. data creation time */
   char*           *aup_machname;/* client's machine name */
   uid_t           aup_uid;       /* client's effective UID */
   gid_t           aup_guid;      /* client's effective GID */
   u_int           aup_len;       /* no. of entry in aup_gids */
   gid_t *         aup_gids;      /* client's supplemental GIDs */
};
```

An example of a server dispatch function that checks client authentication is:

```
int diaptch ( struct svc_req* rqstp, SVCXPRT* xtrp )
{
   struct authsys_parms* ptr
   switch (rqstp->oa_flavor)  {
      case AUTH_NONE:
         break;
      case AUTH_SYS:
         ptr = (struct authsys_parms*)rqstp->rq_clntcred;
         if (ptr->aup_uid != 0)  {
            svcerr_systemerr(xtrp);
```

```
            return;
        }
        break;
    case AUTH_DES:
        ...
        break;
    default:
        svcerr_weakauth( xtrp );
        return;
    }
    /* perform or call the actual RPC function */
    ...
}
```

In the above example, the RPC server skips checking client authentication if it is specified as AUTH_NONE in the *rqstp->rq_cred.oa_flavor*. However, if the client's selected authentication method is AUTH_SYS, the server checks whether the client effective UID is superuser, (via the *rqstp->rq_clntcred.aup_uid* argument). If it is not, the *svcerr_systemerr* API is called by it to print a system error message. This is just an example, and real user applications may authenticate client UIDs and/or GIDs in any way they desire.

If a client authentication is none of the system default methods, the server calls the *svcerr_weakauth* API to send an "unsupported" authentication error to the client.

The function prototypes of the *svcerr_weakauth* and *svcerr_systemerr* APIs are:

```
void    svcerr_weakauth ( const SVCXPRT* xtrp );
void    svcerr_systemerr ( const SVCXPRT* xtrp );
```

The *xtrp* argument to both of the above functions is a transport handle for communication with a client process. This argument value is passed as the second argument to an RPC server's dispatch function.

Note that ONC provides the *authunix_create_default* API instead of *authsys_create_default*, and that the AUTH_SYS and AUTH_SHORT constants are replaced by the AUTH_UNIX constant. However, the underlying authentication method based on process UIDs and GIDs are the same on all UNIX systems.

12.7.3 AUTH_DES

The AUTH_SYS authentication method is simple to use but is not secured because client identification, namely, UID and GID, are not guaranteed unique on the Internet. Furthermore, a client can easily alter the *cl->cl_auth* data to change its identity to someone else before it makes an RPC call. To remedy these defects, the AUTH_DES was created to provide a more sophisticated authentication method for RPC applications.

To use the AUTH_DES method, a client process first needs to call the *user2netname* API to get a "*netname*" that is guaranteed to be unique on the entire Internet. This *netname* is constructed by taking the domain name of the client process and prepending it with the name of the process operating system and effective UID. For example, if a process is running on a UNIX machine in the domain *TJSys.com* and the process's effective UID is 125, its *netname* is *unix.125@TJS.yscom*. This *netname* is unique because a domain name is always unique on the Internet. Furthermore, within a domain, each UID should be unique among all machines running the same operating system. Thus, if the *TJSys.com* domain contains VMS machines that also have a user with the UID of 125, the *netname* of any process created by that user is *vms.125@TJSys.com*. This differentiates it from the UNIX process with the same UID and domain.

As an alternative to the *user2netname* API, a process may call the *host2netname* API to get a *netname* for the machine on which it is running. This *netname* is guaranteed to be unique on the Internet, but it refers to a machine and not a user. In the above example, if the process is running on a UNIX machine called *fruit*, the *netname* returned by *host2netname* is *unix.fruit@TJSys.com*. The choice of which API (*user2netname* or *host2netname*) to use depends on whether users want their RPC applications to check authentication at the user level or at the machine level.

The syntax of the *user2netname* and *host2netname* APIs are:

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

int     user2netname (char netname[MAXNETNAMELEN+1],
                      uid_t eUID, const char* domain );

int     host2netname (char netname[MAXNETNAMELEN+1],
                      const char* hostnm, const char* domain );
```

The first argument of both functions is a character buffer of at least MAXNETNAME+1 size. This is to hold the returned unique name of the process. The second argument of the *user2netname* is the effective UID of the process, whereas the second argument to the *host2netname* is the host machine name. The third argument to both functions is the process

domain name. If the *domain* argument value is passed as NULL, local domain name is assumed.

These functions return 1 if they succeed, 0 if they fail.

To create an AUTH_DES data record in a client process, the *authdes_seccreate* API is called. The function prototype of this API is:

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

int     authdes_seccreate (char netname[MAXNETNAMELEN+1],
            unsigned window, const char* time_host, const des_block* ckey);
```

The *netname* argument value is either the calling process *netname* or its host machine *netname*. This specifies the identity of the client process.

The *window* argument specifies a time period, in seconds, when the client credential as established, by this call will expire. If an RPC server receives an RPC call from a client that was authenticated more than *window* seconds later, the server will reject the request.

The *time_host* argument specifies a machine name upon which the authentication time stamp is based. This is usually the target RPC server's machine name. If this argument is specified as NULL, there is no need to synchronize client and server time.

The *ckey* argument is a DES key that is used to encrypt the client credential. This key is used by the target server to decrypt the credential. If this argument is specified as NULL, the operating system generates a random DES key for it. A client can explicitly get a DES key via the *key_gendes* API.

The *authdes_seccreate* API returns an *AUTH** pointer that points to the encrypted client credential if it succeeds, NULL if it fails.

The *key_gendes* API creates a DES key for a calling process. Its function prototype is:

```
int key_gendes( des_block* ckey );
```

The argument of the *key_gendes* function is the address of a *des_block*- typed variable. This is to hold the generated DES key. The function returns 0 it succeeds, -1 if it fails.

On an RPC server side, a server can retrieve a client's DES credential via either the *authdes_getucred* or the *netname2host* API. Specifically, the *authdes_getucred* is used if the credential is a user *netname* (obtained via the *user2netname* API). This API decrypts the credential and returns the client UID and GID(s) to the server. On the other hand, if the client credential is a machine *netname* (obtained via the *host2netname* API), the *netname2host* API is called to extract the client host machine name accordingly.

The function prototypes of these APIs are:

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

int     authdes_getucred (const struct authdes_cred* adc,
          uid_t* uid_p, gid_t* gid_p, short* len_p, gid_t* gidArray);

int     netname2host (const char* netname, char* hostname, int len);
```

For the *authdes_getucred* API, the *adc* argument value is obtained via the *rqstp* argument of the server dispatch function. Specifically, the *rqstp->rq_clntcred* field is the value for the *adc* argument. This argument points to the client's encrypted credential.

The *uid_p* and *gid_p* arguments are addresses of variables that hold the returned UID and GID of a client, respectively.

The *gidArray* and *len* arguments are address of variables that hold the returned array of supplemental GIDs and the number of client entries, respectivelv.

For the *netname2host* API, the *netname* argument value is obtained via the *rqstp* argument of the server dispatch function. Specifically, the *rqstp->rq_clntcred->adc_fullname.name* field is the value for the *netname* argument.

The *hostname* argument is the address of a character buffer that holds the returned client host machine name. The *len* argument specified the maximum size of the buffer pointed to by the *hostname* argument.

Both the *authdes_getucred* and *netname2host* functions return 1 if they succeed, 0 if they fail. Note that in the RPC classes defined in Section 12.5, the *RPC_cls::set_auth* may be called by a client process to set the AUTH_SYS or AUTH_DES credential. If the AUTH_DES method is used, the client's credential is based on its effective UID and GID

On the server side, the *RPC_svc::dispatch* function checks each client credential according to the authentication method used. If a client uses the AUTH_NONE method, the dispatch function simply skips the credential check. This may not be allowed in real life

secured RPC applications. The users may change the dispatch function to flag an error and refuse to execute the requested RPC function if the client specifies the AUTH_NONE method. Furthermore, if a client uses the AUTH_DES method, the dispatch function calls the *authdes_getucred* API to extract client UID and GID(s). This is acceptable as long as the *RPC_cls::set_auth* calls the *user2netanme* API only (and not *host2netname*) to create the client's credential. However, if a user's application uses *host2netname* and/or *user2netname* to generate client credentials, the *RPC_svc::dispatch* function should be changed accordingly.

12.7.4 Directory Listing Example with Authentication

This directory listing example is shown in Section 12.3.3 is shown again below, but rewritten with the following changes:

* It uses RPC classes instead of *rpcgen*
* It illustrates how a client process pings a server process
* It illustrates RPC authentication mechanism

The *RPC.h* and *RPC.C* files are as shown in Section 12.5. The *scan2.h* file is created manually as follows:

```
#ifndef SCAN2_H
#define SCAN2_H

#include <rpc/rpc.h>
#define MAXNLEN 255
typedef char *name_t;
typedef struct arg_rec *argPtr;

struct arg_rec
{
   name_t    dir_name;
   int       lflag;
};

typedef struct arg_rec arg_rec;
typedef struct dirinfo *infolist;

struct dirinfo
{
   name_t    name;
   u_int     uid;
   long      modtime;
```

```
    infolist      next;
};
typedef struct dirinfo dirinfo;

struct res
{
    int           errno;
    union
    {
        infolist      list;
    } res_u;
};
typedef struct res res;

#define SCANPROG ((unsigned long)(0x20000100))
#define SCANVER ((unsigned long)(1))
#define SCANDIR ((unsigned long)(1))

extern "C" bool_t xdr_name_t(XDR *, name_t*);
extern "C" bool_t xdr_argPtr(XDR *, argPtr*);
extern "C" bool_t xdr_arg_rec(XDR *, arg_rec*);
extern "C" bool_t xdr_infolist(XDR *, infolist*);
extern "C" bool_t xdr_dirinfo(XDR *, dirinfo*);
extern "C" bool_t xdr_res(XDR *, res*);

#endif /* !SCAN_H */
```

The client program *scan_cls2.C* is:

```
#include <errno.h>
#include "scan2.h"
#include "RPC.h"

int main( int argc, char* argv[])
{
    static res result;
    infolist nl;

    if (argc<3) {
        cerr << "usage: " << argv[0] << " host directory [<long>]\n";
        return 1;
    }

    /* create a client RPC hanlde */
    RPC_cls cl( argv[1], SCANPROG, SCANVER, "netpath");
```

```
if (!cl.good()) return 1;

/* set authentication credential base on DES encryption method */
cl.set_auth( AUTH_DES );

/* ping the RPC server to make sure it is alive */
if (cl.call( 0, (xdrproc_t)xdr_void, 0, (xdrproc_t)xdr_void, 0 ) ==
      RPC_SUCCESS)
   cout << "Prog " << SCANPROG << " /" << SCANVER << ") is alive\n";
else {
   cerr << "Prog " << SCANPROG << "/" << SCANVER << " is dead!\n";
   return 2;
}

/* allocate memory to hold the return directory listing */
struct arg_rec *iarg = (struct arg_rec*)malloc(sizeof(struct arg_rec));

iarg->dir_name = argv[2];          // set remote directory name
iarg->lflag = 0;                   // set long listing flag
if (argc==4 && sscanf(argv[3],"%u",&(iarg->lflag))!=1)  {
   cerr << "Invalid argument: " << argv[3] << endl;
   return 3;
}

/* Call the RPC function */
if (cl.call( SCANDIR, (xdrproc_t)xdr_argPtr, (caddr_t)&iarg,
         (xdrproc_t)xdr_res, (caddr_t)&result) != RPC_SUCCESS)
{
   cerr << "client: call RPC fails\n";
   return 4;
}

/* RPC call completed. Check the function's return code */
if (result.errno)  {
   errno = result.errno;
   perror(iarg->dir_name);
   return 5;
}

/* RPC function completes successfully. Now list remote dir content */
for (nl=result.res_u.list; nl; nl=nl->next)   {
   if (iarg->lflag)
      cout << "..." << nl->name << ", uid=" << nl->uid << ",mtime="
           << ctime(&nl->modtime) << endl;
   else cout << "..." << nl->name << "\n";
}
```

```
    return 0;
}
```

The client program is invoked with the server host machine name, a remote directory name, and possibly, an integer flag. The remote directory name is the directory whose content is to be returned by the RPC function. The optional integer flag specifies whether the returned directory listing should be in detailed (*lflag*=1) format or with file names (*lflag*=0) only.

If a client program is invoked with the correct arguments, it creates a *RPC_cls* object for connection with a server via the *RPC_cls::RPC_cls* constructor function. The RPC server is identified by the SCANPROG, SCANVER, and SCANFUNC constants (the RPC program, version, and procedure numbers).

Once the client *RPC_cls* object is created, the client process calls the *RPC_cls::set_auth* function to generate a client credential using the AUTH_DES method. The *RPC_cls::set_auth* function can create authentication credentials by using AUTH_NONE, AUTH_SYS, or AUTH_DES can hide all low-level RPC authentication APIs from users.

After the client credential is set up, the client calls the RPC server with a procedure number of 0. This is to ping the RPC server to make sure it is alive. If the *RPC_cls::call* function fails, the client prints an error message to that effect and quits; otherwise, client execution continues.

The client allocates dynamic memory for the *iarg* variable to store the input argument in the RPC function: a remote directory name and a long listing flag. The client calls the RPC function via the *RPC_cls::call* function and specifies that its return value be stored in the *result* variable. Furthermore, the XDR functions for the input argument and return value are the user-defined *xdr_argPtr* and *xdr_res* functions, respectively.

If the RPC function returns a success status code, the client program prints the RPC function return value (the remote directory content listing) to the standard output. Note that if the directory listing format flag is 1, the printout for each file consists of the file name, UID, and last modification time. If, however, the directory listing format flag is zero, only the name of each file in the remote directory is shown.

The server program that provides the remote directory listing service is in the *scan_svc2.C* file:

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <string.h>
```

```
#include <malloc.h>
#include <errno.h>
#include <sys/stat.h>
#include "scan2.h"
#include "RPC.h"
static RPC_svc *svcp = 0;

/* The RPC function */
int scandir( SVCXPRT* xtrp )
{
  DIR *dirp;
  struct dirent *d;
  infolist  nl, *nlp;
  struct stat statv;
  res res;
  argPtr darg = 0;

  /* Get function argument from a client */
  if (svcp->getargs( xtrp, (xdrproc_t)xdr_argPtr,
                (caddr_t)&darg)!=RPC_SUCCESS)
    return -1;

  /* start scaning the requested directory */
  if (!(dirp = opendir(darg->dir_name)))    {
    res.errno = errno;
    (void)svcp->reply(xtrp, (xdrproc_t)xdr_res, (caddr_t)&res);
    return -2;
  }

  /* free memoryallocated from a previous RPC call */
  xdr_free((xdrproc_t)xdr_res, (char*)&res);
  /* store files' informaton to res as the return values */
  nlp = &res.res_u.list;
  while (d=readdir(dirp))  {
    nl = *nlp = (infolist)malloc(sizeof(struct dirinfo));
    nl->name = strdup(d->d_name);
    nlp = &nl->next;
    if (darg->lflag)  {
      char pathnm[256];
      sprintf(pathnm,"%s/%s",darg->dir_name,d->d_name);
      if (!stat(pathnm,&statv))    {
        nl->uid = statv.st_uid;
        nl->modtime = statv.st_mtime;
      }
    }
  }
```

```
    *nlp = 0;
    res.errno = 0;
    closedir(dirp);

    /* Send directory listing to client */
    if (svcp->reply(xtrp, (xdrproc_t)xdr_res,
                (caddr_t)&res)!=RPC_SUCCESS)
       return -2;

    return RPC_SUCCESS;
}

/* RPC server's main function */
int main(int argc, char* argv[])
{
    svcp = new RPC_svc( SCANPROG, SCANVER, "netpath");
    if (svcp->run_func( SCANDIR, scandir )) return 1;
     return 0; /* shouldn't get here */
}
```

The above program creates an RPC server that provides a directory listing service. There is no command line argument needed to invoke the program.

The process creates a *RPC_svc* object via the *RPC_svc::RPC_svc* constructor function. After this is done, the server calls the *RPC_svc::run_func* function to register the *scandir* RPC function to the *RPC_svc* object and waits for client RPC calls to arrive.

When a client RPC call arrives, the *RPC_svc::dispatch* function is called. This function first checks whether the requested RPC procedure number is zero. If the client is pinging the server, this function simply returns with a NULL reply. This completes the server response to a ping request.

If a client is not pinging the server, the dispatch function checks the client credential according to the authentication method specified. If the authentication check fails, the dispatch function raises an RPC system error and quits. The current *RPC_svc::dispatch* function accepts only clients whose UIDs are either zero (superuser) or the same as that of the server process.

After the client credential is verified as correct, the dispatch function finds the requested RPC function and invokes it. In this example, the only RPC function is *printmsg*; which performs the following operations when invoked:

- It obtains function arguments from a client process
- It frees any dynamic memory allocated for the *res* variable in a previous call

- It scans the requested directory and puts information from all files in that directory to the *res* variable
- It sends the *res* variable as return value to the calling client

The final pieces of code in this example are the XDR functions for user-defined data types (e.g., *struct arg_rec, argPtr, infolist*, etc.). These XDR functions are defined in the *scan2_xdr.c* file:

```
#include "scan2.h"

/* XDR function for the name_t data type */
bool_t xdr_name_t(register XDR *xdrs, name_t *objp)
{
   register long *buf;
   if (!xdr_string(xdrs, objp, MAXNLEN)) return (FALSE);
   return (TRUE);
}

/* XDR function for the argPtr data type */
bool_t xdr_argPtr(register XDR *xdrs, argPtr *objp)
{
   register long *buf;
   if (!xdr_pointer(xdrs, (char **)objp, sizeof (struct arg_rec), (xdrproc_t)
                    xdr_arg_rec))
      return (FALSE);
   return (TRUE);
}

/* XDR function for the arg_rec data type */
bool_t xdr_arg_rec(register XDR *xdrs, arg_rec *objp)
{
   register long *buf;
   if (!xdr_name_t(xdrs, &objp->dir_name)) return (FALSE);
   if (!xdr_int(xdrs, &objp->lflag)) return (FALSE);
   return (TRUE);
}

/* XDR function for the infolist data type */
bool_t xdr_infolist(register XDR *xdrs, infolist *objp)
{
   register long *buf;
   if (!xdr_pointer(xdrs, (char **)objp, sizeof (struct dirinfo), (xdrproc_t)
                    xdr_dirinfo))
      return (FALSE);
```

```
    return (TRUE);
}

/* XDR function for the dirinfo data type */
bool_t xdr_dirinfo(register XDR *xdrs, dirinfo *objp)
{
    register long *buf;
    if (!xdr_name_t(xdrs, &objp->name)) return (FALSE);
    if (!xdr_u_int(xdrs, &objp->uid)) return (FALSE);
    if (!xdr_long(xdrs, &objp->modtime)) return (FALSE);
    if (!xdr_infolist(xdrs, &objp->next)) return (FALSE);
    return (TRUE);
}

/* XDR function for the res data type */
bool_t xdr_res(register XDR *xdrs, res *objp)
{
    register long *buf;
    if (!xdr_int(xdrs, &objp->errno)) return (FALSE);
    if (objp->errno==0) {
        if (!xdr_infolist(xdrs, &objp->res_u.list)) return (FALSE);
    }
    return (TRUE);
}
```

The above XDR functions should be self-explanatory. These XDR functions can be generated manually or via the *rpcgen*. For the latter, users need to declare their data types in a *rpcgen* .x file and use *rpcgen*-specific data types, such as *string*, where applicable.

The above client and server programs are created by the following shell commands:

```
%   CC -c scan2_xdr.c RPC.C
%   CC -DSYSV4 -o scan_svc2 scan_svc2.C scan2_xdr.o \
            RPC.o -lsocket -lnsl
%   CC -DSYSV4 -o scan_cls2 scan_cls2.C scan2_xdr.o \
            RPC.o -lsocket -lnsl
```

The sample run of the client and server programs are shown below. The server program is run on a machine called *fruit*, whereas the client program may be run on any machine that is connected to *fruit*:

```
%   scan_svc2 &
[1] 955
```

```
% scan_cls2 fruit .
....
.....
...scan_cls2.C
...scan_svc2.C
...RPC.C
...RPC.h
...scan2_xdr.c
...scan2.h
...scan_svc2
...scan_cls2
Prog 536871168 (version 1) is alive
```

12.8 RPC Broadcast

Some RPC requests may require a response from all servers on the network that provides the requested services. For example, a client process may wish to set the system clock of all machines on the LAN. Assume there is an RPC server running on each machine and that its effective user ID is the superuser. The client process broadcasts the new clock time to all these servers with one RPC call, and each server updates its system clock accordingly.

To use RPC broadcasting, a process can use the *RPC_cls::broadcast* member function. This is a static function and does not require an *RPC_cls* object be created prior to making the call. This function, in turn, calls the *rpc_broadcast* API to implement the broadcast. The function prototype of the *rpc_broadcast* API is:

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

enum clnt_stat rpc_broadcast (unsiged prognum, unsigned versnum,
        unsigned funcnum, xdrproc_t argfunc, caddr_t argp,
        xdrproc_t resfunc, caddr_t resp, resultproc_t callme, char* nettype);
```

The *prognum, versnum,* and *funcnum* arguments are the numbers of an RPC function to be invoked.

The *argfunc* argument is the address of an XDR function used to serialize/de-serialize the RPC function argument as specified in the *argp* argument. Similarly, the *resfunc* argument is the address of an XDR function used to serialize/deserialize the RPC function return value to be placed in the *resp* argument.

The *nettype* argument specifies the transport to be used for the RPC broadcast call. This must be a connectionless transport protocol, such as UDP. The default value for *nettype* in the *RPC_cls::broadcast* function is "*datagram_v*", which can use any "*visible*" datagram transport as specified in the */etc/netconfig* file. Two other *rpc_braodcast* restrictions are: (1) a broadcast request may not exceed the MTU (maximum transfer unit) limits of its host machine (for Ethernet-based machines, the MTU limit is 1500 bytes); and (2) only servers that are registered with the *rpcbind* daemon can respond to RPC broadcasts. This is the case if a server is created via the *svc_create* or *svc_tp_create* APIs.

The *callme* argument is a user-defined function that is called for each RPC server response. The function prototype of the *callme* function is:

```
int callme ( caddr_t resp, struct netbuf* server_addr, struct netconf* nconf);
```

where the *resp* argument is the same *resp* value specified in the *rpc_broadcast* call. This is the address of a variable defined in the client process that holds the server return value. The *server_addr* argument contains a responding server address. The *nconf* argument contains the network transport information used by the server.

Once the *rpc_braodcast* function is called, it blocks the calling process to wait for server responses. For each RPC server response, the function calls the *callme* function to service the response. If the *callme* function returns a 0 value, the *rpc_broadcast* waits for another server response to arrive. On the other hand, if the *callme* function returns a nonzero value, the *rpc_braodcast* function terminates and returns control to the calling process.

The *rpc_broadcast* function returns an RPC_TIMEDOUT value if it has waited and tried the broadcast several times without getting any server response. It returns an RPC_SUCCESS value if the *callme* function returned TRUE; otherwise, it returns a nonzero value to indicate an error.

The *rpc_broadcast* function uses the AUTH_SYS method to authenticate the calling process to all RPC server processes that receive the broadcast call.

Note that in ONC, the *rpc_broadcast* API is replaced by the *clnt_broaacast* API. The two functions have almost the same signature and return value, except that the *clnt_broadcast* API does not use the *nettype* argument.

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

enum clnt_stat  clnt_broadcast (unsiged prognum, unsigned versnum,
        unsigned funcnum, xdrproc_t argfunc, caddr_t argp,
        xdrproc_t resfunc, caddr_t resp, resultproc_t callme );
```

Furthermore, the *callback* function prototype for the *clnt_broadcast* API is:

```
int callme ( caddr_t resp, struct sockaddr_in* server_addr );
```

where the *resp* argument is the same *resp* value as specified in the *clnt_broadcast* call. This is the address of a variable defined in the client process that holds the server return value. The *server_addr* argument contains a responding server address. Its data type is a pointer to a socket address.

12.8.1 RPC Broadcast Example

The *msg_cls2.C* program shown in Section 12.5. is rewritten below using RPC broadcast. Only the client program is changed. The new client program *msg_cls3.C* is:

```
/* client program: use broadcast to print msg on server's system console */
#include "msg2.h"
#include "RPC.h"

static unsigned int num_responses = 0;


/* client's broadcast call back function */
bool_t callme (caddr_t res_p, struct netbuf* addr, struct netconfig *nconf)
{
   num_responses++;                    // keep track of no.of server responded

   if (res_p==0 || *((int*)res_p)!=0)  {
      cerr << "clnt: call printmsg fails\n";
      return TRUE; /* stop broadcast due to error */
   }
   cout << "clnt: call printmsg succeeds\n";
   return FALSE; /*wait for more response */
}

/* client main function */
int main(int argc, char* argv[])
{
   int   res;
   if (argc<2)    {
      cerr << "usage: " << argv[0] << " msg <transport>\n";
```

```
    return 1;
  }
  /* client sends a broadcast request and waits for responses */
  int rc = RPC_cls::broadcast( MSGPROG, MSGVER, PRINTMSG,
       (resultproc_t)callme, (xdrproc_t)xdr_string, (caddr_t)&argv[1],
       (xdrproc_t)xdr_int, (caddr_t)&res);

  switch (rc)   {
    case RPC_SUCCESS:       break;
    case RPC_TIMEDOUT:      if (num_responses) break;
    default:                cerr << "RPC broadcast failed\n";
                            return 2;
  }
  cout << "RPC broadcast done. No reponses: " < <num_responses
       << endl;
  return 0;
}
```

The new *printmsg* client program takes one command line argument and broadcasts it as the message. It calls the *RPC_cls::broadcast* function to broadcast the message. In the *RPC_cls::broadcast* call, the client process specifies *callme* as the function to be called by the *rpc_broadcast* API for each server response. Furthermore, the *printmsg*'s argument and XDR function, as well as the variable holding the *printmsg* return value and XDR function, are set in the *RPC_cls::broadcast* function call in the same manner as in an *RPC_cls::call* function call.

The *callme* function is called for each server response to the broadcast. The function simply checks that the return value succeeds or does not. The function returns TRUE to stop the broadcast if the server return value is a failure or FALSE to continue receiving more server responses. The *callme* function increments the global variable *num_responses* to keep track of the number of servers actually responding to the broadcast.

After the *RPC_cls::broadcast* call returns, the client program checks the function return status code. If the status code is RPC_SUCCESS, the broadcast was terminated by the *callme* function and all is well. However, if the status code is RPC_TIMEDOUT, then the *num_responses* variable is checked to see whether any server responded to the broadcast. If there are none (the *num_responses* value is zero), the RPC broadcast failed, and an error message is flagged to the user. On the other hand, if the *num_responses* variable value is nonzero, it means that the RPC broadcast was successful. The *rpc_broadcast* function returns because all servers responded to the broadcast.

The sample run of the server program, *msg_svc2* (as shown in Section 12.5) and the new client program, *msg_cls3*, which runs in RPC broadcast mode, is:

```
%   CC -DSYSV4 -o msg_cls3 msg_cls3.C RPC.C -lsocket -lnsl
%   msg_cls3 "Testing RPC broadcast feature"
clnt: call printmsg succeeds
clnt: call printmsg succeeds
...
```

The system consoles on all machines running the *msg_svc2* daemon print the message *Testing RPC broadcast feature*.

12.9 RPC Call Back

In some RPC applications, it may be desirable for a server to call a client process back after some period of time. This allows the client process to do some other work in the meantime. An example of this is when a client process requests an RPC server process to execute a time-consuming function but does not wish to wait for the RPC function to finish before continuing execution. Instead, the client specifies an RPC function that the server can call when it is ready to send results back to the client. Thus, the client and server can both be doing useful work concurrently, improving overall system efficiency.

For an RPC server to call a client back, the client must define an RPC program number, a version number, and a procedure number for the callback function. In a sense, the client is acting as both an RPC client and a server.

The following example programs illustrate how this is done. In the example, the RPC server provides an alarm clock service to processes on the LAN. A client process that desires this service sends an RPC call to the server and specifies the following information:

* The client process host machine name
* The client's callback RPC function program, version, and procedure numbers
* The alarm clock period, in seconds

When the RPC server receives a request from a client, it forks a child to set up an alarm signal, which is sent to the child process after the client-specified alarm clock period expires. When the alarm period does expire, the child process makes an RPC call to the client callback function to inform it of that fact and afterward terminates itself. During all this processing, the RPC server is continuously monitoring for other client alarm service requests. The original client process is working on something else during the alarm clock period.

The header file, *aclock.h*, is shared by the client and server programs:

```
#ifndef ACLOCK_H
#define ACLOCK_H

#include <rpc/rpc.h>
#define    MAXNLEN      255

typedef char *name_t;

/* client's call-back information to the RPC server */
struct arg_rec
{
   name_t    hostname;          // client's host machine name
   u_long    prognum;           // client's RPC function program no.
   u_long    versnum;           // client's RPC function version no.
   u_long    funcnum;           // client's RPC function procedure no.
   u_long    atime;             // alarm clock time
};

/* client's call-back RPC functions' version no. and procedure no. */
#define    CLNTVERNUM          1
#define    CLNTFUNCNUM         1

/* server's RPC function's program number, version no., and function no. */
#define    ACLKPROG            ((unsigned long)(0x20000100))
#define    ACLKVER             ((unsigned long)(1))
#define    ACLKFUNC            ((unsigned long)(1))
/* XDR functions for conversion of client's call-back data */
extern  bool_t xdr_name_t(XDR *, name_t*);
extern  bool_t xdr_arg_rec(XDR *, arg_rec*);

#endif /* !ACLOCK_H */
```

The RPC server program, *aclk_svc.C*, is:

```
#include <signal.h>
#include "aclock.h"
#include "RPC.h"

RPC_svc *svcp;                  // the RPC server handle
static struct arg_rec argRec;   // contains a client's call-back info

/* make an RPC call to a client's call-back function */
void call_client( int signum )
{
   u_long timv= alarm(0); /* alarm time remaining */
```

```
    RPC_cls cls( argRec.hostname, argRec.prognum,
                                        argRec.versnum, "netpath");
    if (!cls.good())  {
       cerr << "call_client: create RPC_cls object failed\n";
       exit(1);
    }

    if (cls.call( argRec.funcnum, (xdrproc_t)xdr_u_long, (caddr_t)&timv,
                  (xdrproc_t)xdr_void, 0 )!=RPC_SUCCESS)
       cerr << "call_client: call client failed\n";

    exit(0); /* kill the child process */
}

/* server's RPC function. Invoked by a client to setup an alarm service */
int set_alarm( SVCXPRT* xtrp )
{
    /* Get client's info: host name, RPC call-back function's program no,
       version no, and procedure number
    */
    if (svcp->getargs( xtrp, (xdrproc_t)xdr_arg_rec, (caddr_t)&argRec)
              !=RPC_SUCCESS)
       return -1;

    /* send a dummy reply to client */
    if (svcp->reply( xtrp, (xdrproc_t)xdr_void, 0)!=RPC_SUCCESS) {
       cerr << "printmsg: sendreply failed\n";
       return -2;
    }

    /* create a child process to handle this client */
    switch (fork()) {
       case -1:  perror("can't fork");
                 break;
       case 0:   /* child process */
                 alarm(argRec.atime);
                 signal(SIGALRM, call_client);
                 pause();            // wait for alarm to expire
    }

    /* parent process. Return to main loop to service other clients*/
    return RPC_SUCCESS;
}

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