Networking 12

Introduction

Sun recognizes the need to integrate systems and peripheral devices from multiple vendors into a single computing environment. To support this environment, Sun promotes client-server networking and distributed computing. This chapter provides a brief history of distributed computing and discusses Sun's network products. It also discusses the future of ONC™ (Open Network Computing).

Distributed Computing

The original network model involved a mainframe processor and a series of terminals sharing the resources of the centralized computer. To meet changing user demands and take advantage of emerging technology, the model evolved into distributed access, a network including homogeneous workstations or personal computers that provided users with desktop computing power as well as access to devices throughout the network.

Distributed processing enhances this model and provides support for heterogeneous devices connected transparently to a network. You can take advantage of specific computing power by directing that processing to take place on a specific machine. The entire computing power of the network is available to meet your needs.

Solaris 2.x is a platform that supports the four requirements of distributed computing:

- *Connectivity*: Connecting new devices, including those based on proprietary and *de facto* standards, to existing architectures.
- *Distributed access*: Sharing network resources.
- *Network management*: Managing and administering the distributed work group in a multi-vendor environment.
- *Standards compliance*: Supporting and promoting open industry standards; offering a smooth migration path to emerging industry standards; providing tools for developing standards-compliant distributed applications.

Solaris 2.x supports a set of core services known as ONC (Open Network Computing). ONC is an environment for heterogeneous distributed computing.

Distributed Applications

A *distributed application* is an application where different components run on different computers in the network. In this way, individual procedures within the application can be processed on the computers best suited for the task.

Two of the more important types of distributed applications are *groupware* and *client-server systems*.

Groupware

Groupware supports collaboration among groups of users at local or remote sites. Popular examples of groupware include bulletin boards, calendars, shared documents, and shared whiteboard systems. These distributed applications let a workgroup act on common data in realtime.

Another groupware example is a distributed financial planning and analysis support system. This type of system combines a distributed financial database with spreadsheet access, report generation, and tools for financial analysis. Several analysts can work on a common spreadsheet, greatly accelerating the budgeting process. Local budgets can be consolidated into corporate budgets though the network.

The Client-Server Model

SunSoft's distributed processing solutions implement the client-server model is shown in Figure 12-1. A *server* is a machine that provides resources to the network; a *client* is a machine that uses these resources. A machine may be both a client and a server, or a server may be dedicated to honoring client requests. An example is a print server – a printer attached to a workstation that only manages print functions.

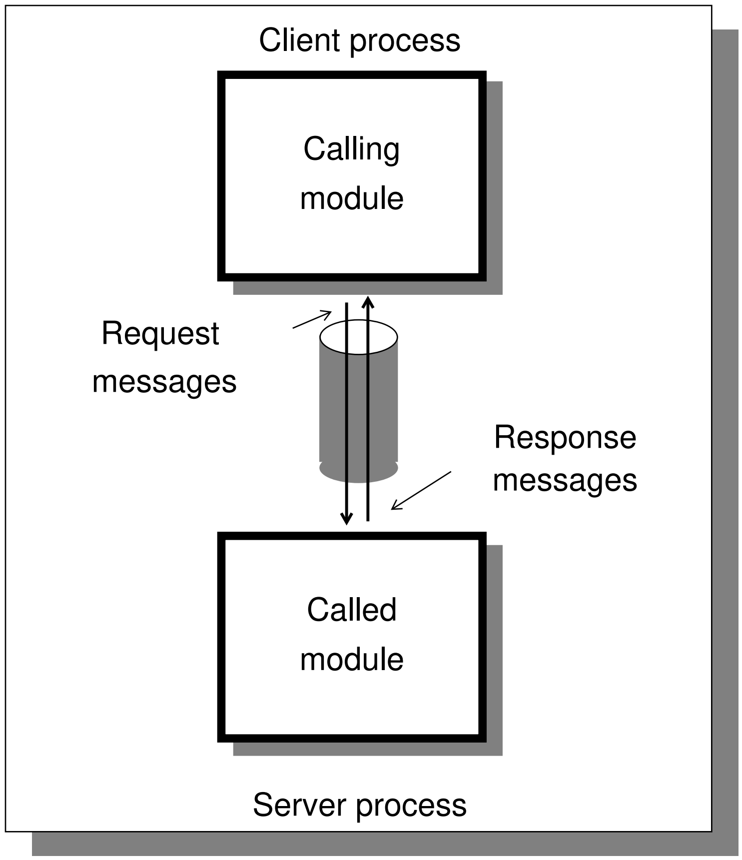

Figure 12-1 Client-Server Model

An example of a distributed application is one that requires a high-powered processing engine such as a Cray supercomputer to do intensive computations, and displays the result on a SPARC® workstation with a powerful standard graphical user interface such as OPEN LOOK®. In this example, the program running on the Cray is the client, and the workstation displaying the results is the server. Other examples of distributed applications include SunNet™ Manager and OpenWindows DeskSet™ Calendar Manager.

Another example would be an application that uses several SQL database servers, such as Sybase, to run a distributed database application.

Solaris Distributed Computing

Solaris Distributed Computing is a platform and standards-based framework for implementing a diversity of networking services. ONC is the core set of services provided with Solaris.

Open Network Computing (ONC)

The environment created by Solaris ONC products supports transparent multi-vendor networking. To increase the use of ONC as a platform, SunSoft placed the specifications for the following key ONC components in the public domain:

- NFS, which supports transparent access to files on remote systems
- Remote Procedure Call (RPC), which enables a program to call a procedure residing on a remote system
- External Data Representation (XDR), which provides a standard format for encoding data to be transferred between systems

More than 150 companies have developed ONC implementations on a variety of hardware platforms and operating systems. These include UNIX BSD 4.3/4.2, UNIX System V, DEC VAX VMS, MS-DOS®, IBM mainframes, MVS, VM, AIX, HP/UX, MacOS, and OS/2®. Almost 200 universities and research institutions have licensed ONC for training and research purposes. By early 1992, an estimated 3.1 million systems, from PCs to mainframes, were running NFS®. At least 80 percent of these systems were not Sun platform based.

For more information on ONC and related products, refer to the *ONC/NFS: A Technology Guide to Distributed Computing*.

Open Network Computing Plus (ONC+)

With Solaris 2.x, SunSoft introduced an enhanced set of services, ONC+. ONC+ includes Transport Independent Remote Procedure Call (TI-RPC), an upward-compatible enhancement of RPC and NIS+, a new name service replacing NIS. ONC+ also includes performance and security enhancements to NFS.

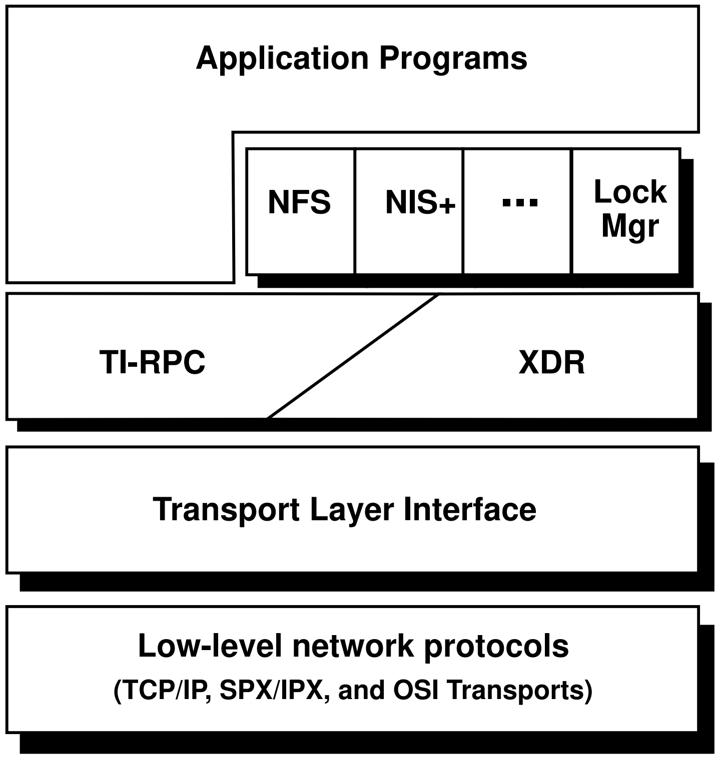

Figure 12-2 ONC (Open Network Computing)

Distributed File Systems

Solaris 2.x offers services that permit file sharing across the network. This section discusses some of these services.

Network File System (NFS)

The best known of ONC's network services, NFS, makes file systems on remote, heterogeneous machines appear as if they reside on an end user's own system.

The UNIX `automount` daemon simplifies the mounting and unmounting of NFS file systems. It automatically and transparently mounts an NFS file system whenever a file or directory within that system is referenced. The `automount` daemon traps references to the specified directory and uses a map to determine a server, exported file system, and appropriate mount options. The map may be either a local file or an NIS map. The name-to-location binding is dynamic; updates to an NIS map are transparent to users, eliminating the need to pre-mount shared file systems for applications with hard-coded file references. Dynamic binding also eliminates the need for a system to remember which hosts must be mounted for each application.

Other Distributed File Systems

Solaris Distributed Computing has provisions for supporting multiple types of distributed file systems. One provision is a Distributed File System Framework, with unified commands and configuration files for administering distributed file systems. Another provision is the Virtual File System (VFS) interface, also known as the VNODE Interface.

Besides NFS, the SVR4 Remote File System (RFS) is supported in Solaris 2.x through the unified facilities. Independent vendors have integrated other file systems such as the Andrew File System (AFS) into Solaris.

Name Services

The following sections describe name services provided with the Solaris environment.

Network Information Service (NIS)

ONC's Network Information Service (NIS) is a lookup service that manages information about the network as a whole and provides links to network resources.

An NIS map might include information about passwords, host names, and user groups. Programs can ask NIS for the value associated with a particular key without knowing the location of the requested information. For example, a program might want to know your group to verify your right to the application. The NIS server manages the location of information. NIS is tolerant of system failures on the network; if the primary server fails, another takes over.

As part of the lookup service, NIS provides functions to facilitate interaction with an NIS database. For example, `yp_match` returns the value associated with a passed key; `yp_first` returns the first key-value pair from the named map in the named domain; and `yp_master` returns the machine name of the master NIS server for a given map.

Network Information Service Plus (NIS+)

NIS+ is a complete redesign of NIS. NIS+ features the following improvements over NIS:

- NIS+ supports hierarchical name spaces similar to those of the Internet Domain Name System. Names are structured as qualified names. This kind of naming scheme easily supports populations numbering into the millions.

- NIS+ has an API suitable for rapidly changing databases. NIS+ provides a rich set of data management functions callable by system services and applications.
- The protocol that NIS+ uses to update databases is more efficient and robust. Only the actual changes are propagated. A by-product of the more efficient update protocol is a reduction in the data loss that can result from a communication failure.
- NIS+ generalizes the maps of NIS by replacing them with tables. Tables, for example, can have multiple search keys.
- NIS+ has considerably more support for security. Using Secure RPC for authentication, NIS+ supports a choice of authentication schemes. NIS+ also provides fine-grained access control on tables and their data elements.

NIS+ provides a naming or directory service with high performance and scalability, along with a lightweight database management capability.

Name Service Switch

Solaris 2.x has a Name Service Switch that tells which name services to use for each type of lookup. By using the switch, certain of the Solaris system services provide name service independence. The system calls that have this capability are mainly those of the form `getXbyY`; for example, `gethostbyame()`.

The Name Service Switch consists of a configuration file which defines a simple policy for each database. These databases are called *maps* in NIS and *tables* in NIS+. The database policy determines the order in which name services are consulted. It also indicates whether to continue the search if either a name service is not running or a lookup fails to find a name.

Domain Name Service

Solaris supports the Internet's Domain Name Service (DNS).

DNS is an application layer protocol that is part of the standard TCP/IP protocol suite. Specifically, DNS is a *naming* service; it obtains and provides information about hosts on a network.

Domain Name Service performs naming between hosts within your local administrative domain and across domain boundaries. It is distributed among a set of servers, commonly known as *name servers*, each of which implements DNS by running a daemon called `named`.

Note – The `named` daemon is also called the Berkeley Internet Name Domain service, or BIND, because it was developed at University of California at Berkeley.

On the client side, DNS is implemented through the *resolver*. The resolver is neither a daemon nor a particular program; rather, it is a library compiled into applications that need to know machine names. The resolver's function is to resolve users' queries; to do that, it queries a name server, which then returns either the requested information or a referral to another server.

For more information on DNS, see *Administering NIS+ and DNS*.

Other Network Services

Lock Manager

In conjunction with NFS, the Network Lock Manager coordinates access to files and records, preventing multiple users from modifying the same data at the same time. The Lock Manager also manages recovery from a server or client crash. If a server crashes, clients holding locks on its files must be able to recover their locks. Similarly, if a client crashes, the server must hold its locks while the client recovers.

The Network Lock Manager follows the industry standard for file and record locking defined by the System V Interface Definition (SVID) published by AT&T. All applications written to this definition will be compatible with future releases of the UNIX and SunOS operating systems.

NETdisk

With Sun's NETdisk™, a diskless workstation can boot its operating system from a server that supports ONC/NFS NETdisk protocols. The server need not be from the same vendor or architecture as the diskless workstation. When the workstation is operating, NETdisk provides diskless paging and swapping services.

Distributed Applications Services

ONC provides a platform for distributed applications which consists of Remote Procedure Calls (RPC) and External Data Representation (XDR). This section discusses XDR, the original RPC service, and the newer Transport-Independent RPC (TI-RPC) service.

XDR

The eXternal Data Representation (XDR) specification is one part of the ONC protocol platform. Computer systems do not share a standard method of internal data representation. For one computer to act on data from another, the data must conform to an expected format. XDR provides that standard format and supports all types of data, including integer, real, double, enumeration, string, structure, and union as shown in Figure 12-3.

XDR is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation.

Computers differ in basic word size, in the order in which bytes and other units are combined to form larger units, and in the use of padding and alignment.

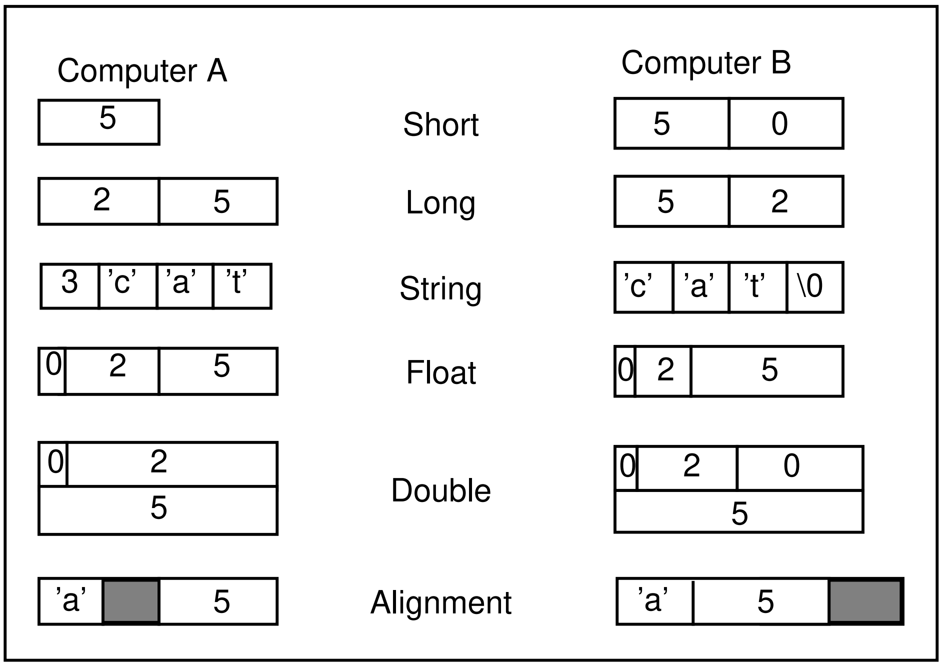

Figure 12-3 Data Representations on Different Machines

RPC

Remote Procedure Call (RPC) is another part of the ONC protocol platform. Using the RPC mechanism, a local program can call a procedure residing on a remote system. The technique used to invoke a remote procedure is similar to that for a local procedure. The interaction follows the client-server model, with the local program as client and the remote procedure as server (although these roles can be reversed). RPC programs are portable, provided that the new platform supports the RPC library, as shown in Figure 12-4.

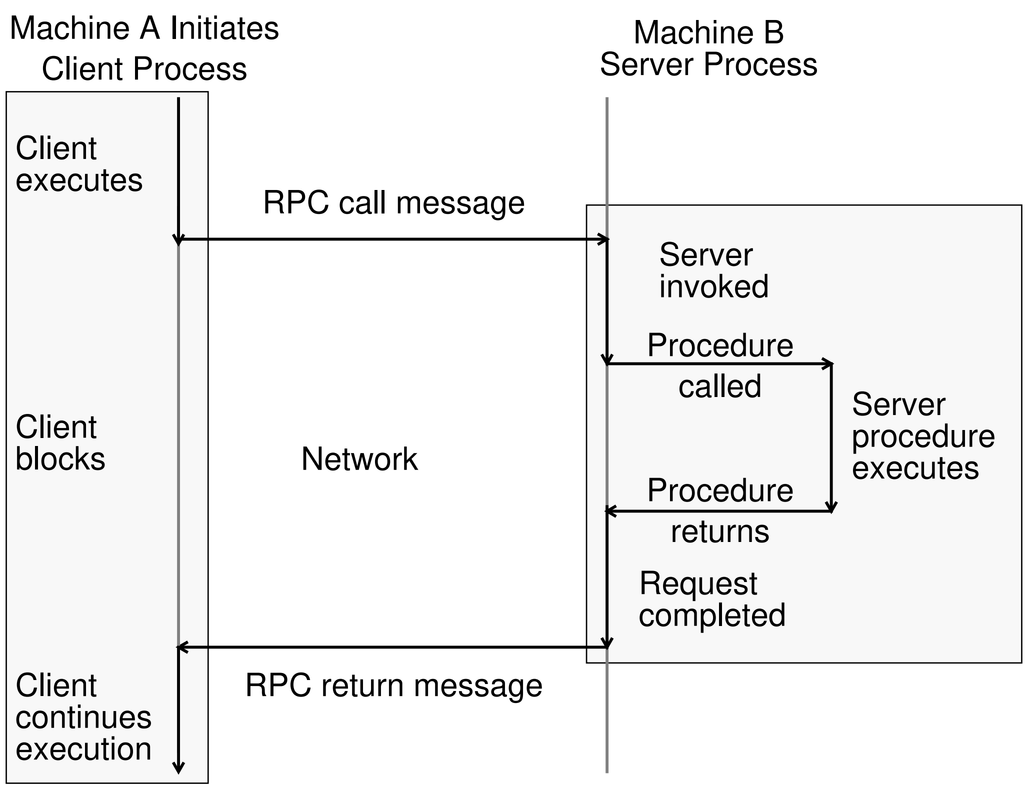

Figure 12-4 RPC Model

RPC features include:

- *Flexible event handling in the server*
  Developers have a choice of using a prepackaged main loop, `svc_run()`, or writing their own main loop and calling the `poll()` function directly. This makes it possible, for example, to develop RPC servers with graphical user interfaces.

- *Broadcast RPC*
  Users can broadcast a remote procedure call, perhaps to locate an under-utilized machine. This feature is supported only when using transports that support broadcast, such as User Datagram Protocol (UDP).

- *Batching*
  RPC batch facilities let clients continue computing after making a call request, instead of waiting for a response from the server.
- *Authentication and secure RPC*
  RPC provides several ways for clients and servers to authenticate each other. For applications where security requirements are light, lightweight authentication mechanisms such as UNIX can be used. For applications with stronger security requirements, secure RPC is available. One kind of secure RPC uses a public key encryption algorithm along with DES to provide identification as strong as most timesharing systems. Another kind, part of SunOS/SVR4, uses the Kerberos system from MIT to identify the clients.

TI-RPC

Transport-Independent RPC (TI-RPC) is the RPC service of Solaris 2.x. Developed jointly by AT&T and Sun Microsystems, TI-RPC is an upward compatible extension of the Transport-Specific RPC (TS-RPC) that shipped with previous SunOS releases. The RPC protocol itself remains unchanged.

With TI-RPC, RPC protocol data units can use any transport available on the system. TI-RPC is layered on TLI, which provides transport independence. With TI-RPC a distributed application can select a network transport protocol at runtime.

Solaris 2.x offers developers several levels of access to its RPC services. The highest level, which is recommended for most developers, is through an RPC compiler, or code generator. Two RPC compilers are available—`rpcgen` and RPC TOOL, both of which are discussed in this section.

Developers who code directly to TI-RPC can use a simplified interface or one of the lower level interfaces. Lower level interfaces give progressively more access to the details of the underlying transport services. Applications can be written to be “transport aware” or “TLI aware” if they need to exert control at these levels. Developers who use the lower level services should be careful to make full use of TLI’s support for transport independence and TI-RPC’s support for runtime transport selection.

Migrating from Transport-Specific RPC to TI-RPC.

Since TI-RPC is implemented on TLI whereas TS-RPC was implemented on sockets, applications that perform socket operations on the underlying sockets in TS-RPC require an additional library for compatibility with TI-RPC. This library, *librpcsoc*, is furnished with the SunOS/BSD Source Compatibility Package.

Applications that do not depend on the underlying sockets in TS-RPC are compatible with TI-RPC without the need for the Source Compatibility Package. However, since TI-RPC is a superset of TS-RPC, it includes some transport-specific functions that can be replaced by new transport-independent functions. To take full advantage of TI-RPC, applications should migrate away from the use of the transport-specific functions.

Porting an RPC Application to TI-RPC

To port an RPC application, you need to:

- Do nothing if your application works in binary compatibility mode and you are satisfied that it uses only UDP or TCP transports. You will eventually have to port your application to the Solaris 2.x environment; the binary compatibility package is not a permanent solution.
- Regenerate (with `rpcgen`) and recompile automatically generated RPC modules, if your application uses simple RPC calls and no BSD socket or TCP or UDP specifics.
- Do minimal editing and write some new code if the application depends on any BSD socket semantics or any features specific to TCP or UDP (examples of the latter include knowing the format of host addresses or relying on the Berkeley UNIX system concept of privileged ports).
- Do more than minimal revision if the application is dependent on the internals of the RPC library, the sockets implementation, or has ingrained assumptions about transport addressing.

Advantages of Porting over Using Binary Compatibility

Porting is recommended for the following reasons:

- Porting makes applications transport independent and thus able to run over many more transports than before.
- Binary compatibility is less efficient than native mode.
- Binary compatibility and old interfaces might be dropped in future releases.

Note – Applications that open their own sockets and use them in RPC calls, such as `clnt{tcp,udp}_create`, `clntudp_bufcreate`, `svc{tcp,udp}_create`, and `svcudp_bufcreate`, are not portable. Link these applications with the `librpcsoc` library with a command such as `cc -o filename -lrpcsoc -lnsl`.

Porting Issues

Consider the following issues when you decide to port your code to the Solaris 2.x environment:

- `libnsl` is a new library. Networking functions are no longer a part of `libc`.
- Many old interfaces are present in the new library, but they work only with TCP and UDP transports. To take advantage of new transports, use the new interfaces.
- Use the new name-to-address and network selection mechanisms for transport-independence.
- The new transports are based on TLI. TLI's form of opaque addressing can have implications for applications that previously knew how to interpret addresses. Specifically, the lowest level address format is now a `struct netbuf`; formerly, it was a `structsockaddr`.

Using `rpcgen`

The `rpcgen` compiler contains many improvements in the Solaris 2.x environment, including support for ANSI C and multiple arguments, and more client- and server-side templates.

If you port your code using `rpcgen` to generate the new support code and then compile and link the code with the new library, you will not have to change your input files.

Creating Client Handles

The following calls have the same functionality as in the SunOS 4.1.x releases, although those marked with a dagger (†) could disappear in future releases:

```
clnt_destroy
clnt_pcreateerror
clntraw_create †
clnt_spcreateerror
clnttcp_create †
clntudp_bufcreate †
clntudp_create †
```

These calls have added functionality but have the same interfaces:

`clnt_control`	added `CLGET_SVC_ADDR`, `CLSET_SVC_ADDR`, `CLGET_XID`, `CLSET_XID`, `CLGET_VERS`, `CLSET_VERS`, `CLGET_PROG`, and `CLSET_PROG`
`clnt_create`	the last argument is now a `nettype`
`clnt_create_vers`	the last argument is now a `nettype`

The following calls have new functionality:

```
clnt_dg_create
clnt_raw_create
clnt_tli_create
clnt_tp_create
clnt_vc_create
```

See the *SunOS 5.2 Network Interfaces Programmer's Guide* section on RPC programming for ideas about when to use a particular client-side interface.

Client Creation Code Examples

If you have existing client-creation code written for RPC, such as:

```
clnt_create("hostname", PROGRAM, VERSION, "udp");
```

you could rewrite it this way for TI-RPC:

```
clnt_create("hostname", PROGRAM, VERSION, "datagram_n");
```

If you have more extensive client creation code written for RPC, such as:

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

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

you could rewrite it so that it will run on TI-RPC:

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

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

Creating and Destroying Services

The following calls have the same functionality as in the SunOS 4.1.x releases, although those marked with a dagger (†) could disappear in future releases:

```
svc_destroy
svcfd_create
svcraw_create †
svctcp_create †
svcudp_create †
svcudp_bufcreate †
```

These calls have new functionality:

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

See the *SunOS 5.2 Network Interfaces Programmer's Guide* section on RPC programming for hints on when to use a particular server-side interface.

Registering and Unregistering Services

The following calls have the same functionality as in the SunOS 4.1.x releases, although those marked with a dagger (†) could disappear in future releases:

```
registerrpc †
svc_register †
svc_unregister †
xprt_register
xprt_unregister
```

These calls have new functionality:

```
rpc_reg
svc_reg
svc_unreg
```

Service Registration Code Example

If you have service registration code written for RPC, such as:

```
SVCXPRT *transp;
extern void svcfunc();

if ((transp = svcudp_create (RPC_ANYSOCK)) == (SVCXPRT *) NULL) {
    syslog(LOG_ERR, "svcudp_create failed");
    exit(1);
}
if (!svc_register(transp, PROGRAM, VERSION, svcfunc, IPPROTO_UDP)) {
    syslog(LOG_ERR, "svc_register IPPROTO_UDP failed");
    exit(1);
}
```

```
    if ((transp = svctcp_create(RPC_ANYSOCK)) == (SVCXPRT *) NULL) {
        syslog(LOG_ERR, "svctcp_create failed");
        exit(1);
    }
    if (!svc_register(transp, PROGRAM, VERSION, svcfunc, IPPROTO_TCP)) {
        syslog(LOG_ERR, "svc_register IPPROTO_TCP failed");
        exit(1);
    }
```

you could rewrite it this way for TI-RPC:

```
     extern void svcfunc();

    if (!svc_create(svcfunc, PROGRAM, VERSION, "netpath") {
        syslog(LOG_ERR, "svc_create failed");
        exit(1);
    }
```

Calls

The following calls have the same functionality as in the SunOS 4.1.x releases:

```
callrpc
clnt_call
svc_getcaller
```

These calls have new functionality:

```
rpc_call
svc_rpcgetcaller
```

Broadcasting

The following call has the same functionality as in the SunOS 4.1.x releases, although it is supported for backward compatibility only:

```
clnt_broadcast
```

The `rpc_broadcast` and `rpc_broadcast_exp` calls have new functionality; they broadcast to both `portmap` and `rpcbind`.

Broadcasting Code Example

If you have broadcasting code written for RPC, such as:

```
       statstime sw;
       extern int collectnames();
        ...
        clnt_broadcast(RSTATPROG, RSTATVERS_TIME, RSTATPROC_STATS,
            xdr_void, NULL, xdr_statstime, &sw, collectnames);
    ...
    collectnames(resultsp, raddrp)
        char *resultsp;
        struct sockaddr_in *raddrp;
    {
        u_long addr;
        struct entry *entryp, *lim;
        struct hostent *hp;
        extern int curentry;

        /*
         * weed out duplicates
         */
        addr = raddrp->sin_addr.s_addr;
        lim = entry + curentry;
        for (entryp = entry; entryp < lim; entryp++)
            if (addr == entryp->addr)
                return (0);
        /*
         *  print the host's name (if possible) or address
         */
        hp = gethostbyaddr(&raddrp->sin_addr.s_addr, sizeof(u_long), AF_INET);
        if (hp == (struct hostent *) NULL)
            printf("0x%x\n", addr);
        else
            printf("%s\n", hp->h_name);
```

you could rewrite it in this way for TI-RPC:

```
        statstime sw;
     extern int collectnames();
        ...
        rpc_broadcast(RSTATPROG, RSTATVERS_TIME, RSTATPROC_STATS,
            xdr_void, NULL, xdr_statstime, &sw, collectnames, (char *) 0);
    ...
    collectnames(resultsp, taddr, nconf)
        char *resultsp;
```

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

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

Using the rpcbind and portmap Functions

The RPC library can speak with either the previous RPC's `portmap` program, or with its replacement, `rpcbind`. Use the following functions to communicate with `portmap`:

```
pmap_set
pmap_unset
pmap_getport
pmap_getmaps
pmap_rmtcall
```

For new applications, use the following new `rpcbind` functions:

```
rpcb_set
rpcb_unset
rpcb_getaddr
rpcb_getmaps
rpcb_rmtcall
```

Note that `rpcb_getaddr` can be used to talk to `portmap` as well as `rpcbind`.

`The rpcbind` function now provides a simple time service (primarily for use by secure RPC client-server time synchronization), available through the `rpcb_gettime` function. This function returns the server's time.

Recompilation Issues

Include `-lnsl` in the link step and `-lrpcsoc` if you are using old interfaces.

When recompiling an application that uses any of the following functions or macros, use the `-DPORTMAP` preprocessor control:

```
callrpc
clntraw_create
clnttcp_create
clntudp_bufcreate
clntudp_create
svc_getcaller
svc_register
svc_unregister
svcfd_create
svcraw_create
svctcp_create
svcudp_bufcreate
svcudp_create
```

Authentication

The following calls have the same functionality as in the SunOS 4.1.x releases, although they are supported for backward compatibility only:

```
authdes_create
authunix_create
authunix_create_default
```

These calls have new functionality:

```
authdes_seccreate
authsys_create
authsys_create_default
```

Contrasting TLI and Sockets

TLI and sockets are quite different ways of doing essentially the same thing. The section “Transport-Level Programming: Sockets, TLI, and STREAMS” in this chapter provides an overview of TLI and sockets. The following issues are related to transport independence and might have some bearing on RPC applications:

- Privileged ports are an artifact of the Berkeley UNIX system implementation of TCP and UDP. They were never portable and have no place in a transport-independent environment.
- TLI addresses are opaque. There is no transport-independent way of separating the portion of an address that names a host from the portion of an address that names the service at that host. Code written with the assumption that it can discern the host address of an RPC caller must be changed.

Name-to-Address Translation

The following calls have the same functionality as in the SunOS 4.1.x releases:

```
gethostbyname
gethostbyaddr
gethostent
getrpcbyname
getrpcbynumber
getservbyname
getservbyaddr
```

The following calls have new functionality:

```
netdir_free
netdir_getbyname
netdir_getbyaddr
netdir_options
netdir_perror
netdir_sperror
taddr2uaddr
uaddr2taddr
```

For more information, see the *SunOS 5.2 Network Interfaces Programmer’s Guide*.

Differences in the TI-RPC and RPC Environments

The major differences between TI-RPC and previous releases of RPC are illustrated in the following table:

Table 12-1 Differences Between TI-RPC and RPC

	SunOS 5.x TI-RPC	SunOS 4.1.x RPC
Default Transport Selection	TI-RPC uses the TLI interface.	RPC uses the socket interface.
RPC Address Binding	TI-RPC uses `rpcbind` for service binding. `rpcbind` keeps addresses in universal address format.	RPC uses `portmap` for service binding. `portmap` maintains addresses as a list of port numbers.
Transport Information	Transport information is kept in a local file, `/etc/netconfig`. Any transport identified in `netconfig` is accessible.	Only TCP and UDP transports were supported.
Loopback Transports	`rpcbind` service requires a secure loopback transport for server registration.	RPC services do not require a loopback transport.
Host Name Resolution	Host name resolution in TI-RPC uses name services as specified in `/etc/nsswitch.conf.`	Hostname resolution is done by name services. The order is set by the state of the `hosts` database.
File Descriptors	Descriptors are assumed to be TLI endpoints.	Descriptors are assumed to be sockets.
`rpcgen` *for TI-RPC*	TI-RPC's version of the `rpcgen` tool adds support of multiple arguments, pass-by-value, sample client, and sample server files.	The SunOS 5.x `rpcgen` adds support of multiple arguments, call-by-value, and ANSI C. It also generates sample client code, sample server code, and sample makefiles.
`libnsl` *Requirement*	SunOS 5.x TI-RPC requires that applications be linked to the `libnsl` library.	All RPC functionality is provided in `libc`.

Converting Local Procedures into Remote Procedures

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

The original program follows:

Code Example 12-1 Single Computer Version of printmsg.c

```
/* printmsg.c: print a message on the console */
#include <stdio.h>

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

    if (argc != 2) {
        fprintf(stderr, "usage: %s <message>", argv[0]);
        exit(1);
    }
    message = argv[1];
    if (!printmessage(message)) {
        fprintf(stderr,"%s:couldn't print your message",argv[0]);
        exit(1);
    }
    printf("Message Delivered!\n");
    exit(0);
}
/* Print a message to the console.
 * Return a boolean indicating whether or not the
 * message actually printed.
 */

printmessage(msg)
    char *msg;
{
    FILE *f;

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

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

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

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

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

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

Remote procedures are always declared as part of remote programs. The code above declares an entire remote program that contains the single procedure `PRINTMESSAGE`. In this example, the `PRINTMESSAGE` procedure is declared to be procedure `1`, in version `1` of the remote program `MESSAGEPROG`, whose program number is `0x20000001`. Version numbers are incremented when functionality is changed in the remote program.

Changes can be made to existing procedures or new procedures can be added. More than one version of a remote program can be defined and a version can have more than one procedure defined.

Note that the program and procedure names are declared with all capital letters. This is not required, but is a good convention to follow.

Note also that the argument type is `string` and not `char *` as it would be in C. This is because a `char *` in C is ambiguous. `char *` could be used to mean an array of characters, but it could also represent a pointer to a single character. In RPC language, a null-terminated array of `char` is called a `string`.

There are two more things to write: the remote procedure itself, and the main client program that calls it.

The following remote procedure implements the PRINTMESSAGE procedure.

```
/*
 * msg_proc.c: implementation of the remote
 * procedure "printmessage"
 */
#include <stdio.h>
#include "msg.h" /* msg.h generated by rpcgen */
/*
 * Remote version of "printmessage"
 */
/*ARGSUSED1*/
int *
printmessage_1(msg, req)
    char **msg;
    struct svc_req *req; /* details of call */
{
    static int result; /* must be static! */
    FILE *f;
    f = fopen("/dev/console", "w");
    if (f == (FILE *)NULL) {
        result = 0;
        return (&result);
    }
    fprintf(f, "%s\n", *msg);
    fclose(f);
    result = 1;
    return (&result);
}
```

Notice that the declaration of the remote procedure `printmessage_1()` differs from that of the local procedure `printmessage()` in four ways:

- It takes a pointer to the pointer to the character array instead of the pointer itself. This is true of all remote procedures when the `-N` option is not used: they always take pointers to their arguments rather than the arguments themselves. Remote procedures are always called with a single argument. If more than one argument is required, the arguments must be passed in a `struct`.
- It is called with two arguments. The second argument contains information on the context of an invocation: the program, version, and procedure numbers, raw and canonical credentials, and an `SVCXPRT` structure pointer (the `SVCXPRT` structure contains transport information). This information is made available in case the invoked procedure requires it to perform the request.

- It returns a pointer to an integer instead of the integer itself. This is also true of remote procedures when the `-N` option is not used: they return pointers to the result. Always declare the result `static`. (If the result is declared local to the remote procedure, references to it by the server stub are invalid after the remote procedure returns.)
- An `n_1` is appended to its name. In general, all remote procedure calls generated by `rpcgen` are named as follows: the procedure name in the program definition (here `PRINTMESSAGE`) is converted to all lower case letters, an underbar (_) is appended to it, and the version number (here `1`) is appended. This allows multiple versions of the same procedure.

Finally, declare the main client program that calls the remote procedure, as shown in the following code example.

Code Example 12-2 Remote Version of printmsg.c

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

main(argc, argv)
    int argc;
    char *argv[];
{
    CLIENT *clnt;
    int *result;
    char *server;
    char *message;
    externintsys_nerr;
    externchar*sys_errlist[];

    if (argc != 3) {
        fprintf(stderr,usage: %s host message\n”, argv[0]);
        exit(1);
    }
    /*
     * Save values of command line arguments
     */
    server = argv[1];
    message = argv[2];
    /*
     * Create client “handle” used for calling
     * MESSAGEPROG on the server
     * designated on the command line.
```

Code Example 12-2 Remote Version of printmsg.c (Continued)

```
     */
    clnt = clnt_create(server,MESSAGEPROG,MESSAGEVERS,"visible");
    if (clnt == (CLIENT *)NULL) {
        /*
         * Couldn't establish connection with server.
         * Print error message and die.
         */
        clnt_pcreateerror(server);
        exit(1);
    }
    /*
     * Okay, we successfully called the remote procedure.
     */
    if (*result == 0) {
        /*
```

There are four things to note here:

1. A client handle is created using the RPC library routine `clnt_create()`. This client handle is passed to the *stub* routine that calls the remote procedure.
2. The last parameter to `clnt_c create()` is `visible`, which specifies that any transport noted as visible in `/etc/netconfig` can be used.
3. The remote procedure `printmessage_1()` is called exactly the same way as it is declared in `msg_proc.c` except for the inserted client handle as the second argument. It also returns a pointer to the result instead of the result.
4. The remote procedure call can fail in two ways. The RPC mechanism can fail or there can be an error in the execution of the remote procedure. In the former case, the remote procedure, in this case `printmessage_1()`, returns a `NULL`. In the latter case, the error reporting is application dependent. Here, the error is returned via `*result`.

Here is how to put all the pieces together:

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

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

In this example, no XDR routines were generated because the application uses only the basic types that are included in the RPC/XDR Library.

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

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

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

Once created, the server program is installed on a remote machine and run. If the machines are homogeneous, the server binary can be copied. If they are not, the server source files must be copied to and compiled on the remote machine. In this example, the remote machine is called remote and the local machine is called local. The server is started from the shell on the remote system:

```
remote$ msg_server
```

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

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

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

By executing rprintmsg, a user can print a message on any system console (including the local console) if the server msg_server is running on the target system.

To get an RPC program number, collect the .x file for your RPC service and submit it with your request to rpc@sun.com. A number will be assigned to your project or product and will then be returned to you.

Developing Distributed Applications

The first step in designing any application is to define what it does. This step involves partitioning the application into procedures, the building blocks for the application. In a distributed application, the designer must decide which procedures to place in servers and which to retain in the client. A procedure that uses a resource such as a printer on a remote machine naturally becomes a server. When the decision is less obvious, the designer must consider the trade-off between the network overhead incurred by a remote procedure and the memory overhead incurred for a procedure in the client.

Distributed application development must also accommodate such varied demands as:

- Communication between application procedures executing on remote systems.
- Data exchange between procedures executing on machines with different architectures and data representation formats.
- An automated, simplified, transparent development environment that frees the programmer from concerns about the specific location of network services, the operating system, and the machine architecture.
- A flexible administrative model for the application when it is delivered to the end user, including policy choices such as type and degree of security, and network resource usage.

This section describes the tools available for building distributed end-user applications.

rpcgen

The original ONC RPC compiler, `rpcgen`, uses RPCL (RPC Language), a high-level, C-like data description language that simplifies the RPC programming task. As input to the RPC compilation process, programmers compose a description of the client-server relationship as though the remote procedure were local. Output includes C language data structure definitions used by the client and server, XDR routines that generate the data structures, and boiler plate calling routine code for both client and server. The result is a set of C include files for compiling either the server or the client side of the application. In speed-critical applications, handwritten code can easily be linked with the generated output. Additionally, the generated output can be edited before it is compiled for execution.

The following features are new to Solaris 2.x:

- C-style Mode
  The `rpcgen` compiler now has two compilation modes, C-style and Default. The `-N` flag makes `rpcgen` compile RPC input in the C-style mode. The new mode lets arguments be passed by value, instead of as pointers, to a structure. The C-style mode also supports passing multiple arguments. The output of `rpcgen` without the `-N` flag is the same as in previous releases.

- Client, Server, and MakeFile Template Generation
  `rpcgen` can now generate both client- and server-side templates. These templates can be used as a base to implement your applications. Select these options with the `-Sc` (Client-side template), `-Ss` (server-side template), `-Sm` (makefile template), or `-a` (generate all files) flags.
- TI-RPC or RPC Library Selection
  `rpcgen` can now generate code that uses either the TI-RPC (transport-independent RPC) or the transport-specific RPC support libraries. The default case generates TI-RPC library calls. The `-b` flag generates calls to the RPC library routines for backward compatibility.
- ANSI C Compliant Code
  Support has been added to generate output that conforms to ANSI C standards. This code can be used in the Sun C++ 2.1 environment. Select this option with `-C`.

The ONC RPC Application Toolkit

The ONC RPC Application Toolkit consists of a simple to use but powerful RPC compiler based on RPC TOOL technology from Netwise. The ONC RPC TOOL has extensions for asynchronous client operation, callback, authentication, audit trails, data compression, and many other features.

ONC RPC TOOL accepts ANSI C like input specifications describing client/server interfaces. It provides the ability to customize an application through the use of hooks and traps.

Integrating Client/Server Applications into System Environments

Several services are available to enable clients to locate and connect to servers.

rpcbind

Transport services do not provide address-lookup services. They provide only message transfer across a network.

A client program needs a way to obtain the address of its server program. In earlier system releases this service was performed by `portmap`. In Solaris 2.x `rpcbind` replaces the `portmap` facility that was included with Transport-Specific RPC (TS-RPC).

RPC makes no assumption about the structure of a network address. It deals with universal addresses specified only as null-terminated strings of `ASCII` characters. RPC translates universal addresses into local transport addresses by using routines specific to the transport. For more details on these routines, see the `netdir(3N)` man page.

rpcbind provides the operations:

- Add a registration
- Delete a registration
- Get address of a specified program number, version number, and transport
- Get the complete registration list
- Perform a remote call for a client
- Return the time

Address Registration

Because rpcbind maps RPC services to their addresses, its address must be known. The name-to-address translation routines of all transports must reserve a known address for rpcbind. For example, in the Internet domain, rpcbind has port number 111 on both TCP and UDP. When rpcbind is started, it registers its location on each of the transports supported by the host. rpcbind is the only RPC service that must have a known address.

For each supported transport, rpcbind registers the addresses of RPC services and makes the addresses available to clients. A service makes its address available to clients by registering the address with the rpcbind daemon. The address of the service is then available to rpcinfo and to programs using library routines named in rpcbind. No client or server can assume the network address of an RPC service.

Client and server programs and client and server hosts are usually distinct but they need not be. A server program can also be a client program. When one server calls another rpcbind server, it makes the call as a client.

To find a remote program's address, a client sends an RPC message to a host's rpcbind daemon. If the service is on the host, the daemon returns the address in an RPC reply message. The client program can then send RPC messages to the server's address. A client program can minimize its calls to rpcbind by storing the network addresses of recently called remote programs.

The RPCBPROC_CALLIT procedure of rpcbind lets a client make a remote procedure call without knowing the address of the server. The client passes the target procedure's program number, version number, procedure number, and calling arguments in an RPC call message. rpcbind looks up the target procedure's address in the address map and sends an RPC call message, including the arguments received from the client, to the target procedure.

When the target procedure returns results, RPCBPROC_CALLIT passes them to the client program. It also returns the target procedure's universal address so that the client can later call it directly.

The RPC library provides an interface to all rpcbind procedures. Some of the RPC library procedures also call rpcbind automatically for client and server programs.

The rpcinfo Command

The `rpcinfo` utility reports current RPC information registered with `rpcbind`. The `rpcinfo` utility, with either `rpcbind` or the `portmap` utility, reports the universal addresses and the transports for all registered RPC services on a specified host. The `rpcinfo` utility can call a specific version of a specific program on a specific host and report whether a response is received; it can also delete registrations. For details, see the `rpcinfo` man page.

Port Monitors

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

It may be useful for services to wait for a specified interval after satisfying a service request in case another request follows. If there is no call in the specified time, the server exits, and some port monitors, such as `inetd`, continue to monitor for the server. If a later request for the service occurs, the port monitor gives the request to a waiting server process (if any), rather than spawning a new process.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`inetd` Daemon

One of the daemons provided with the system is `inetd`. The `inetd` daemon is invoked at boot time, and gets the services for which it listens from the file `/etc/inetd.conf`. `inetd` creates one socket for each service listed in `/etc/inetd.conf`, binding the appropriate port number to each socket.

`inetd` does a `select()` on each socket for readability, waiting for a connection request to the service corresponding to that socket. For `SOCK_STREAM` type sockets, `inetd` does an `accept()` on the listening socket, calls `fork()`, calls `dup()` to duplicate the new socket to file descriptors `0` and `1` (`stdin` and `stdout`), closes other open file descriptors, and calls `exec()` to start the appropriate server.

Servers that use `inetd` are simplified because `inetd` does most of the work to establish a connection. The server started by `inetd` has the socket connected to its client on file descriptors `0` and `1`, and can immediately `read()`, `write()`, `send()`, or `recv()`. Servers can use buffered I/O as provided by the `stdio` conventions, as long as they use `fflush()` when appropriate.

`getpeername()` returns the address of the peer (process) connected to a socket; it is useful in servers started by `inetd`. For example, to log the Internet address in decimal dot notation (such as 128.32.0.4) of a client, an `inetd` server could use the following:

```
struct sockaddr_in name;
int namelen = sizeof name;
 ...
if( getpeername( 0, (struct sockaddr *) &name, &namelen ) < 0 ) {
    syslog( LOG_ERR, "getpeername: %m" );
    exit(1);
} else
    syslog( LOG_INFO, "Connection from %s",
            inet_ntoa( name.sin_addr ));
 ...
```

`getpeername()` is also useful in other circumstances; see the man page for further information.

Using `inetd`

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

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

where:

`rpc_prog/vers`	is the name of an RPC program followed by a / and the version number or a range of version numbers
`endpoint_type`	is one of `dgram` (for connectionless sockets), `stream` (for connection mode sockets), or `tli` (for TLI endpoints)
`proto`	may be * (for all supported transports), a `nettype`, a `netid`, or a comma separated list of `nettype` and `netid`
`flags`	are either `wait` or `nowait`
`user`	must exist in the effective `passwd` database
`pathname`	is the full pathname of the server daemon
`args`	are the arguments to be passed to the daemon on invocation

For example:

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

For more information, see the man page for `inetd.conf`.

Using the Listener

Use `pmadm` to add RPC services, as in:

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

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

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

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

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

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

For more information, such as how to set up the listener process, see the `listen`, `pmadm`, `sacadm`, and `sysadm` man pages and the *SunOS 5.2 Administering TCP/IP and UUCP*.

Security in Solaris Distributed Computing

In most cases, the security issues that affect an individual system are the same ones that are important in securing a network. In the workplace, a number of systems connected to a server can be thought of as one large multifaceted system. Security is also multifaceted. A variety of services provide fences and checkpoints at different places in the network. One of the most important security services in a network is session-level authentication, or verifying the identity of a remote user or system. Authentication is essential because permission to access a network resource is associated with the identity of the requester. In Solaris 2.x the authentication service is provided by Secure RPC.

Because the ONC distributed services are layered on RPC, they can use the services of Secure RPC to provide their own levels of security. For example, Secure NFS is based on the underlying Secure RPC mechanism.

In Solaris 2.x, developers and administrators can choose among different technologies for authentication. Solaris 2.x and earlier versions offer DES authentication, which refers to a combination of the Data Encryption Standard and the Diffie-Hellman method of public key cryptography. Solaris 2.x also supports the Kerberos service from MIT.

See *Secure Networking in the Sun Environment* and *SunOS 5.2 Administering Security, Performance, and Accounting* for more information.

Transport-Level Programming: Sockets, TLI, and STREAMS

Many distributed applications have been written to use network services at the transport level. UNIX applications at this level commonly use the Internet TCP/IP or UDP/IP protocols. Sockets, TLI, and STREAMS are the interfaces and mechanisms that support this level of programming.

Sockets

Sockets were introduced in UNIX 4.2BSD as facilities for interprocess communication and networking. Sockets provided the basic mechanism and API for the original implementations of the TCP/IP suite in UNIX.

Sockets allow processes to communicate transparently across machine boundaries. A process uses a socket to establish a named path between itself and another process. Normally, sockets are given a type (such as stream, datagram, or raw) and a protocol family in which to operate (UDP or TCP). A communications domain (for example, UNIX, DEC, X.25, or OSI) establishes how sockets are addressed and defines protocols to implement the various socket types within that domain.

TLI

The Transport Layer Interface (TLI), introduced by AT&T, is an alternative to the socket interface. TLI is modeled on the industry standard Transport Service Definition (ISO 8072). It can be used to access both TCP and UDP, as well as a variety of other transport protocols.

TLI is implemented as a user library using the STREAMS I/O mechanism. The interface between the TLI library and the STREAMS module in the kernel is called Transport Provider Interface (TPI).

STREAMS

STREAMS is a collection of system calls, kernel resources, and kernel routines that serve as a framework for the development of device drivers and communication services. Developed by Dennis Ritchie in the AT&T Bell Laboratories ResearchVersion 8 UNIX system, STREAMS is used consistently within the Solaris 2.x kernel.

Although originally designed to deal with the inadequate character I/O mechanism within UNIX, STREAMS is a modular, flexible facility for developing all types of communication protocols.

STREAMS and Sockets in Solaris 2.x

The implementations of the internet protocols in the Solaris 2.x kernel are STREAMS-based. The socket interface is provided in Solaris 2.x by a user-level library, `libsocket`, which is supported by a special STREAMS module in the kernel.

Migrating Socket-Based Code to Solaris 2.x

The biggest change in using sockets in Solaris 2.x is the need to specify the socket library (`-lsocket`) when compiling. Sockets and the socket implementation are largely compatible with previous releases of SunOS, but there are some differences that an application programmer must be aware of. These differences are listed in Chapter 7 of the *SunOS 5.2 Network Interfaces Programmer's Guide*.

Migrating from Sockets to TLI and STREAMS

Developers seeking greater transport independence in application code can use TLI in place of the socket interface. However, a consequence of this migration is that it is more difficult to use TLI to set some options that are specific to the TCP protocol. Clearly, the preferred direction is to avoid dependency on such options.

To control TCP-specific options in TLI in Solaris 2.x you can do the equivalent of setting socket options. This is provided through the TLI option management function. Details are given in the *Solaris 2.x Release Notes*.

For more information, see the section titled "Socket-to-TLI Equivalents" in the *SunOS 5.2 Network Interfaces Programmer's Guide*.

System and Network Administration

Administration Tool

The Solaris administration tool, `admintool`, provides distributed system administration with a graphical user interface. Introduced in Solaris 2.x, `admintool` provides the following set of administrative functions:

- *Database manager*: for databases supported by the three Solaris naming services: DNS, NIS, NIS+, and the respective `/etc` files.
- *Host manager*: for setting up stand-alone, dataless, and diskless systems.
- *Printer manage*r: for setting up printers and print servers.
- *User account manager*: for setting up system users.

Network Management: SunNet Manager

SunNet Manager is a network management system that provides user and development tools for administration of distributed work groups. These tools automatically monitor and analyze systems and networks, notifying users when they need to be involved. SunNet Manager lets you manage more resources in less time and operate in larger networks. Network control is consistently integrated at the user interface level. SunNet Manager is useful for networks with more than ten nodes or any network with at least one gateway.

SunNet Manager is an open, heterogeneous platform based on the OSI management framework. It offers extensibility to third parties through well-known open protocols, such as ONC (RPC/XDR), SNMP, and DECnet NICE (with SunLink® DNI).

Resources managed by SunNet Manager include communication protocol layers and interfaces; network devices; system and OS resources; and application, database, and network services. You can easily add your own resources, known as managed objects, to this list.

Tools provide event reporting and performance monitoring, with control through specific applications and terminal-emulation windows integrated at the user interface. SunNet Manager also allows users to customize their needs by providing open data formats and APIs. Users can integrate with existing SunOS management programs as well as the X Window System, 3270 or VT100 emulation or Telnet Windows.

Network Licensing

Network licensing is a mechanism that provides for the use and management of licensed software products in a network. With network licensing, software vendors need not *lock* a package to a specific machine to ensure compliance with a license agreement. Instead, network licensing provides a pool of *floating* licenses, also called *concurrent usage* licenses, that can be used anywhere in the network. For example, in a 20-user network with five license units for a software package, up to five users can work with the package concurrently.

When a user invokes, for example, a SPARCompiler or SPARCworks tool, a license, if one is available, is automatically checked out. Somewhere in the network a *license daemon* runs. The workstation or server running the daemon is called the *license server*.

When you run a licensed program, it looks in a *license data file* for the name of the license server and contacts the daemon on that server. Users or administrators install the license data files so that they are accessible to both the program and the license server.

The license data files also contain unique codes supplied by the vendor. Vendors can send licenses to users or administrators by e-mail, FAX, or phone.

Connectivity Solutions

PC Networks and Solaris

Connectivity solutions that integrate PC networks with Solaris 2.x are offered by SunSelect™.

PC-NFS

With PC-NFS®, users can access files on NFS servers as if they were in their local MS-DOS file system. PC-NFS is an NFS client implementation that runs on MS-DOS and in Microsoft® Windows.

NetWare SunLink

NetWare SunLink® lets a Solaris 2.x system function as the equivalent of a NetWare server. This permits integration of the popular PC local area networks into larger UNIX-based enterprise networks.

NetWare SunLink lets MS-DOS PC users use NetWare services to access files in Solaris file systems. In Solaris, NFS extends the reach of the NetWare services to files throughout the enterprise network.

NetWare SunLink also makes printers throughout the UNIX network available to the PC users.

Platform Networks and Solaris

SunLink SNA products are available for connecting with IBM SNA networks, included is IBM connectivity through 3270 emulation.

SunLink DNI product lets Solaris interoperate with DECnet.

Standards and OSI

SunLink OSI is a family of products that allows Solaris 2.x to interoperate with networks and services conforming to the ISO Open System Interconnection standards. SunLink MHS provides a gateway to the ISO X.400 mail handling protocol.

Migration Strategies

The transport independence of TLI and TI-RPC provides a ready migration path to OSI transports. For maximum transport independent developers using TLI and low-level TI-RPC should follow the guidelines in the *SunOS 5.2 Network Interfaces Programmer's Guide*, Chapter 6 on “Guidelines to Protocol Independence,” also use the new mechanisms described in Chapter 5 on “Transport Selection and Name-to-Address Mapping” for transport independence.

Present and Future Solaris Directions

The ToolTalk Service

The ToolTalk® service is a high-level, network-transparent interprocess communication service that is presently a part of Solaris 2.x. It is discussed in the chapter on “Application Interoperability” in this book. You should consider using ToolTalk to implement some of your distributed applications.

TookTalk is especially suitable for frameworks of integrated applications, where it is important to be able to add new applications and where applications should be able to interoperate with a minimum of dependency on each other. ToolTalk is also desirable for those moving toward object-oriented distributed applications.

Distributed Object Technology

Under Project DOE (“Distributed Objects Everywhere”), SunSoft is developing object technology in accordance with the Object Management Group (OMG) specifications.

The object-oriented Interface Definition Language (IDL) will offer a powerful model for transparent distributed computing. A Distributed Object Management Facility (DOMF) will provide communications between objects using optimal methods of transport.

Developers aiming for DOE environments can use ToolTalk now, for its object orientation. ToolTalk will continue to be supported under DOE, as will TI-RPC. For more information, see the chapter on “Project DOE: Future Solaris Technologies” in this book.

System Administration Framework

Solaris 2.x will advance system and network administration in two ways. First, it provides powerful new system management tools that are easy to use. Secondly, through advances in automation, it makes key components of the system self-administering.

Solaris will provide an object-oriented framework for distributed system administration. This framework will support the development of advanced graphical user interfaces for administering network objects. These user interfaces, reflecting the object-orientation of the framework, will be natural, consistent, and intuitive.

Physical Networks and Data Links

Solaris will add support for new, high speed wide area networking services. These include Integrated Services Digital Network (ISDN), and new packet switching techniques such as Asynchronous Transfer Mode (ATM). Solaris will also support wireless networks and nomadic devices.

Solaris Distributed Computing and Federated Services

SunSoft has announced a Federated Services strategy that will provide technology that enables ONC+ to coexist with other distributed computing technologies, from NetWare to DCE to OSI and tomorrow’s emerging standards. The Federated Services technology consists of a series of interfaces that enable easy integration of different types of filing, naming, and security services. Networking technology providers can utilize these interfaces for the full integration of their networking services alongside ONC+ without conflict.

Federated Naming

Solaris 2.x will support a diversity of naming services by forming a federation. Software in a Solaris system will have at least a minimum level of access to all naming services in the federation. The members of the federation, while providing their own distinctive sets of services, will export basic lookup services through a standard protocol.

Federated Filing

The federated filing strategy is to make the kernel virtual file system interface (VFS/VNODE) available for the addition of other distributed file systems. The interface already supports a variety of file system types. Examples in Solaris 2.x include the distributed file systems NFS and RFS, as well as the UNIX File System (UFS), the MS-DOS file system (PCFS) and a number of special purpose file systems.

Federated Security

Solaris will support dynamic selection of authentication technologies through a generic security interface. Since higher levels of security tend to inconvenience users more, a choice of levels will let administrators make a trade-off between user convenience and degree of security. Another reason for offering a choice of technologies is to replace older methods that may have been compromised with new methods.

Networking Sample Code

See the accompanying floppy disk for a sample of TI-RPC code.

Bibliography

SunOS 5.2 Network Interfaces Programmer's Guide, Part Number 801-2854-10, SunSoft, Inc., Mountain View, CA.

SunOS 5.2 Administering Security, Performance, and Accounting, Part Number 800-7362-10, SunSoft, Inc., Mountain View, CA.

SunOS 5.2 Administering NIS+ and DNS, Part Number 800-6155-10, SunSoft, Inc., Mountain View, CA.

Solaris 2.1 Application Developer's Guide (porting from RPC to TI-RPC), Part Number 800-6363-10, SunSoft, Inc., Mountain View, CA.

Design and Implementation of Transport-Independent RPC, Chuck McManis and Vipin Samar, Part Number 91028-001, Sun Microsystems, Inc., Mountain View, CA.

Network Information Service Plus (NIS+), Chuck McManis and Saqib Jang, Part Number 91027-001, Sun Microsystems, Inc., Mountain View, CA.

ONC/NFS: A Technology Guide to Distributed Computing, Part Number 92015-001, SunSoft, Inc., Mountain View, CA.

The Sun Network File System: Design, Implementation and Experience, Russel Sandberg, Part Number FE-199/0, Sun Microsystems, Inc., Mountain View, CA.

Secure Networking in the Sun Environment; by Bradley Taylor and David Goldberg, Sun Technical Report, 1987.

Analyzing Sun Networks; by Carl Malamud; VNR.

Managing NFS and NIS, Hal Stern, O'Reilly & Associates, Sebastopol, CA.

Power Programming with RPC, John Bloomer, O'Reilly & Associates, Sebastopol, CA.

All About Administering NIS+, Rick Ramsey, SunSoft Press/Prentice Hall, Englewood Cliffs, NJ.