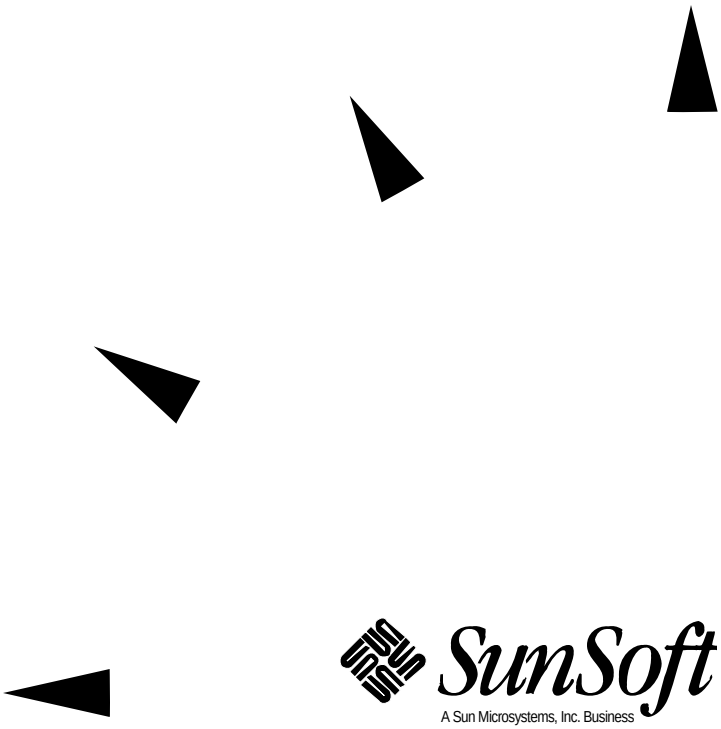
Network Interfaces Programmer's Guide



2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

Contents

Figures

Tables

Code Samples

Preface

The *Network Interfaces Programmer's Guide* describes the primary facilities for implementing distributed applications.

All utilities, their options, and library functions in this manual reflect the current Solaris™ system software developed by SunSoft Inc. If you are using a previous version of Solaris system software, some utilities and library functions may function differently.

Who Should Use This Book

The guide assists you in:

- Converting an existing single-computer application to a networked, distributed application
- Designing a distributed application
- Implementing a distributed application
- Maintaining a distributed application in the Solaris operating system

Use of this guide assumes basic competence in programming, a working familiarity with the C programming language, and a working familiarity with the UNIX® operating system. Previous experience in network programming is helpful, but is not required to use this manual.

How This Book Is Organized

Part One—Introduction

Chapter 1, “Introduction to Network Interfaces,” gives a high-level introduction to networking concepts and the topics covered in this book.

Part Two—Remote Procedure Call (RPC)

Chapter 2, “Introduction to Remote Procedure Call (RPC),” describes the basic model of RPC and its use in developing distributed applications.

Chapter 3, “rpcgen Programming Guide,” describes how the `rpcgen` tool generates client and server stubs. The RPC compiler generates standard TI-RPC code and distributes the specified procedure to the application.

Chapter 4, “RPC Programming Guide,” describes the use of RPC in the programming environment.

Part Three—Transport Layer

Chapter 5, “Transport Selection and Name-to-Address Mapping,” describes the network selection mechanisms used by applications in selecting a network transport and its configuration.

Chapter 6, “Transport -level Interface (TLI) Programming Guide,” describes the UNIX System Transport Interface.

Chapter 7, “Socket Interface,” describes the socket interface at the transport layer.

Part Four—Naming Services

Chapter 8, “NIS+ Programming Guide,” describes the NIS + programming interface.

Part Five—Appendixes

Appendix A, “XDR Protocol Specification,” describes the XDR protocol and language.

Appendix B, "XDR Technical Note," describes XDR and how it is used in data formatting and type conversion. These services may be utilized not only by distributed applications, but in any application in which common or uniform data representation is necessary.

Appendix C, "RPC Protocol and Language Specification," describes the protocol of RPC usage, both syntax and limitations.

Appendix D, "Writing a Port Monitor With the Service Access Facility (SAF)," describes the process of writing a port monitor application under the SAF and is included as a reference for applications development.

Appendix E, "The portmap Utility," describes the portmap utility and its function. This appendix is included in this document to aid migrating applications written to run on earlier releases of SunOS.

Appendix F, "Live RPC Code Examples," contains complete functional listings of some of the code included in the document as examples. These modules are furnished under the proviso stated at the beginning of the appendix.

Related Books

The following Solaris information products further explore a variety of related topics discussed in this book. You may want to reference the following on-line System AnswerBook® products:

- *Solaris 2.4 Reference Manual AnswerBook*
- *Solaris 2.4 Software Developer AnswerBook*
- *Solaris 2.4 System Administrator AnswerBook*

For network-related topics the following book may be helpful:

- *TCP/IP Network Administration Guide*

For RPC-related topics, the following books are recommended:

- *Multithreaded Programming Guide*
- *Security, Performance, and Accounting Administration*

For NIS+, the following books provide complementary topics:

- *Name Services Administration Guide*
- *Name Services Configuration Guide*
- *NIS+ Transition Guide*

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`system% su` `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
Code samples are included in boxes and may display the following:		
%	UNIX C shell prompt	`system%`
$	UNIX Bourne and Korn shell prompt	`system$`
#	Superuser prompt, all shells	`system#`

Part 1 — Introduction

Introduction to Network Interfaces 1

This manual describes the programmer's interface to network services in the SunOS 5.4 operating system. In this guide, the terms SunOS and Solaris are used interchangeably because the interfaces described in this manual are common to both. Solaris 2.4 is SunSoft's™ distributed computing operating environment. It is comprised of SunOS release 5.4 with ONC™, OpenWindows™, ToolTalk™, DeskSet™, and OPEN LOOK® as well as other utilities. This release of Solaris is fully compatible with System V, Release 4 (SVR4) and conforms to the third edition of the System V Interface Description (SVID). It supports all System V network services.

Applications that must adjust options or use specific addresses can still do so. But you can now write applications with relative ease to be portable over different protocol stacks.

Solaris Networking Features

This section summarizes the main networking themes in the current Solaris release.

Multithreaded (MT) Remote Procedure Call

This release provides multithreaded-safe client and server RPC interfaces. MT-safe server interfaces are new in the SunOS 5.4 release and are documented in this revision of the manual. To determine which interfaces are safe or unsafe, refer to routines from section 3N of the *man Pages(3): Library Routines*.

Transport-Independent Remote Procedure Call (TI-RPC)

Transport-independent RPC provides interfaces that let applications be free of, or more closely tied to the underlying transport. It is the developer's choice to use the most appropriate level.

Standardized Network Interfaces at the Transport and Link Layers

At the transport level, the AT&T transport provider interface (TPI) is required. At the link level, the UNIX international data link provider interface (DLPI) is required. Standardizing on these interfaces lets you interchange STREAMS drivers at the transport and link levels with no changes to the modules or drivers communicating with them. In particular, transport layer interface (TLI) and sockets can interface to any transport provider supporting TPI, and any device driver supporting DLPI can be linked beneath the Internet protocol (IP).

Functions and Network Selection

The sockets, TLI, and name-to-address translation functions work with the network selection facility to free user applications from the details of specific protocols and address formats.

Open Systems Interconnection Reference Model

The open systems interconnection (OSI) reference model is the basis of commercially available network service architectures. Other network protocols developed independently conform loosely to the model. The transport control protocol/interface program (TCP/IP) is an example. For more information on TCP/IP, see the *TCP/IP Network Administration Guide.*

The OSI reference model is a convenient framework for networking concepts. Basically, data are added to a network by a sender. The data are transmitted along a communication connection and are delivered to a receiver. To do this, a variety of networking hardware and software must work together.

Industry standards have been or are being defined for each layer of the reference model. Two standards are defined for each layer: one specifies the interface to the services provided by the layer, and the other specifies the protocol observed by the services in the layer. Users of a service interface standard should be able to ignore the protocol and any other implementation details of the layer.

The OSI reference model divides networking functions into seven layers, as shown in Figure 1-1.

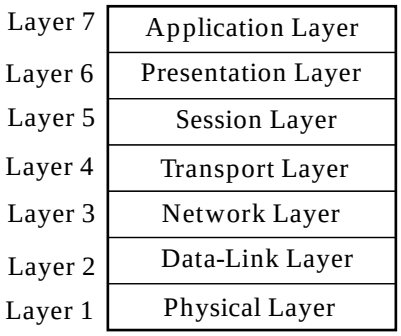


Figure 1-1 OSI Reference Model

Each protocol layer performs services for the layer above it. The ISO definition of the protocol layers provides designers some freedom of implementation. For example, some applications skip the presentation and session layers to interface directly with the transport layer.

Layer 1: Physical Layer

The hardware layer of the model. It specifies the physical connections between hosts and networks, and the procedures used to transfer packets between machines.

Layer 2: Data-Link Layer

Manages the delivery of data across the physical network. It describes how the internet protocol (IP) should use existing data link protocols, such as Ethernet / 802.

Layer 3: Network Layer

This layer is responsible for machine-to machine communications. It determines the path a transmission must take, based upon the receiving machine's IP address. Besides message routing, it also translates from logical to physical addresses.

Layer 4: Transport Layer

Controls the flow of data on the network and assures that received and transmitted data are identical. TLI, TCP/IP, or the user datagram protocol (UDP) may be used to enable communications between application programs running on separate machines.

Layer 5: Session Layer

Manages reliable sessions between cooperating applications. The interface at this layer enables remote communication using function call semantics.

Layer 6: Presentation Layer

Performs the translation between the data representation local to the computer and the processor-independent format that is sent across the network.

Layer 7: Application Layer

At this top layer are the user-level programs and network services. Some examples are `telnet`, `ftp`, `tftp`, and the domain name service (DNS).

Transport Interface Overview

This section gives a brief overview of the transport layer interfaces.

Description

The transport layer (layer 4) is the lowest layer of the model that provides applications and higher layers with end-to-end service. This layer hides the topology and characteristics of the underlying network from users. The transport layer also defines a set of services common to many contemporary protocol suites including the ISO protocols, TCP/IP, Xerox® network systems (XNS)™, and systems network architecture (SNA).

Industry Standard

The transport layer interface (TLI) is modeled on the Transport Service Definition (ISO 8072). The TLI can be used to access any transport support on your system. It is implemented as a user library using the STREAMS I/O mechanism.

Sockets

The socket interface is implemented with a set of user-level library routines and a STREAMS module. Applications that were implemented on 4.1.x or earlier versions of SunOS, or on operating systems from other vendors, can run on the current Solaris release because BCP mode (4.1.x library) is the default. To use the Solaris 2.x socket library, you must recompile the application.

MT Safe and MT Hot

TLI and sockets were made MT safe in SunOS 5.3. Some of the name service interfaces are safe, such as NIS+ and name-to-address translation. SunOS 5.4 provides MT-hot RPC server interfaces. The RPC client interfaces were MT-safe in SunOS 5.3. See Chapter 3, “rpcgen Programming Guide” and Chapter 4, “RPC Programming Guide” for details. To determine which interfaces are safe or unsafe, refer to section 3N of the *man Pages(3): Library Routines.*

NIS+ Overview

This section is a brief introduction of the NIS+ name service. Chapter 8, “NIS+ Programming Guide covers the NIS+ API.

NIS+ is the network information service in Solaris. It is an information retrieval system for well-known UNIX databases, such as the password tables, host tables, and mail aliases maps. It also supports Solaris databases such as the automount maps and the credentials tables. NIS+ is an enterprise-wide information service. The enterprise is partitioned into organizational units that are arranged into a tree and assigned hierarchical *domain* names.

The types of enterprise objects that NIS+ understands are principals, directories, tables, entries, and groups. There is no concept of a user or host context, per se. Information about an entity such as a user appears in various different tables, such as the credentials table, the password table, the automount maps, and the mail aliases map. This information is retrieved using NIS+ indexed names. For example, the password entry is obtained by using the name `[name=mjones]passwd.org_dir.sales.wiz.com.`, while the credentials for the same user are obtained using the name `[name=mjones.sales.wiz.com.]cred.org_dir.sales.wiz.com`.

1

Part 2 — Remote Procedure Call (RPC)

Introduction to Remote Procedure Call (RPC) 2

This manual addresses the C interface to RPC, and the way RPC is used to communicate between processes on same or different hosts. Other language bindings are not available in Solaris.

This chapter contains capsule overviews of the key components and leading characteristics of RPC. See “RPC Programming Terms” on page 449 for the definition of the terms used in this chapter.

RPC Server Is Multithread Safe

In SunOS 5.3 client-side interfaces were made MT safe. SunOS 5.4 now provides MT-safe server-side interfaces that can be used freely in a multithreaded application. For more information on these interfaces, refer to “MT Server Issues” on page 118.”

2

What Is RPC

RPC is a high-level network applications model. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

RPC specifically supports network applications. It runs on available networking mechanisms such as TCP/IP. Generic facilities, such as `rpcbind`, associate network services with universal network addresses where they may be accessed. In RPC programming, the term network is frequently used as a synonym for transport or transport type.

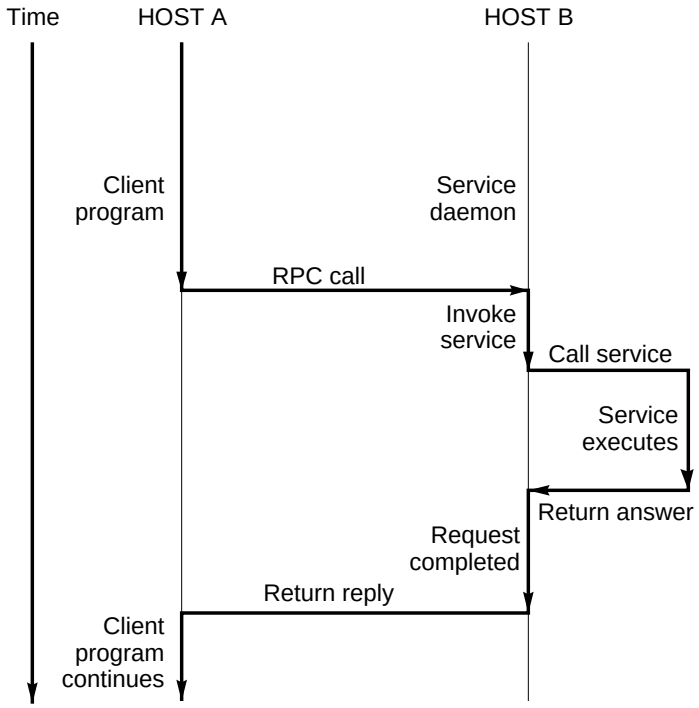


Figure 2-1 How a Remote Procedure Call Works

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. Figure 2-1 shows the flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out**.** When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues.

RPC programs follow the client/server model of distributed programming. RPC programs provide network services to callers without requiring any awareness of the underlying network. For example, a program can call `rusers`(), a routine in the `librpcsvc` library that returns the number of users on a remote host. The caller is not explicitly aware of using the network. The call to `rusers`() is as simple as a call to a standard system service call like `malloc`().

RPC Levels

You can use the RPC services at a number of levels, from the simplified level to the lower level. The levels are described in detail in “Introduction to RPC Interface Levels” on page 60.” Understanding the lower levels of RPC is helpful but not necessary when you use the `rpcgen` tool to generate your RPC applications. For use of `rpcgen`, see Chapter 3, “rpcgen Programming Guide.”

External Data Representation (XDR)

For RPC to function on a variety of system architectures requires a standard data representation. RPC uses external data representation (XDR). XDR is a machine-independent data description and encoding protocol. Using XDR, RPC can handle arbitrary data structures, regardless of different hosts’ byte orders or structure layout conventions. For a detailed discussion of XDR, see Appendix A, “XDR Protocol Specification,” and Appendix B, “XDR Technical Note.”

2

RPC Versions and Numbers

Each RPC procedure is uniquely identified by a program number, version number, and procedure number.

The program number identifies a group of related remote procedures, each of which has a unique procedure number. Each program also has a version number, so when a change is made to a remote service (such as adding a new procedure), a new program number does not have to be assigned.

Changes in a program, such as adding a new procedure, changing the arguments or return value of a procedure, or changing the side effects of the procedure *require* that the version number be changed.

Appendix C, "RPC Protocol and Language Specification," tells you how to assign a program number to an RPC program.

Network Selection

You can write programs to run on a specific transport or transport type, or to operate on a system- or user-chosen transport. It uses two mechanisms, the `/etc/netconfig` database and the environmental variable `NETPATH`. `/etc/netconfig` lists the transports available to the host and identifies them by type. `NETPATH` is optional and allows a user to specify a transport or selection of transports from the list in `/etc/netconfig`.

The major features of `/etc/netconfig` are:

- The first field of each entry is the network identifier of the transport.
- Each entry contains a flag or set of flags (the third field) that identifies the transport type — the `v` flag, for example, identifies a transport that can be selected ("visible").
- The second field identifies the transport type. Connectionless transports are identified by `tpi_clts`. Connection oriented transports are identified by `tpi_cots`. Connection oriented transports with orderly release are identified by `tpi_cots_ord`.
- The last field names one or more run-time linkable modules that contain the name-to-address translation routines associated with the transport.

- The loopback transports are used to register services with `rpcbind`. They are local transports, available only to local clients and servers, and are more secure than other transports.

The `/etc/netconfig` file contains several lines, each of which corresponds to an available transport. Code Example 2-1 shows some possible entries.

Code Example 2-1 Sample `/etc/netconfig` File

```
# The Network Configuration File.
#
# Each entry is of the form:
#
# network_id semantics: flags protofamily protocol device_name
# nametoaddr_libs
#
tcp tpi_cots_ord v inet tcp /dev/tcp tcpip.so
#
udp tpi_clts v inet udp /dev/udp tcpip.so
#
icmp tpi_raw - inet icmp /dev/icmp tcpip.so
#
rawip tpi_raw - inet - /dev/rawip tcpip.so
#
ticlts tpi_clts v loopback - /dev/ticlts straddr.so
#
ticots tpi_cots v loopback - /dev/ticots straddr.so
#
ticotsord tpi_cots_ord v loopback - /dev/ticotsord straddr.so
#
#   end of netconfig file
```

For the details of `/etc/netconfig` and the application interface to it, see the `getnetconfig(3N)` and `netconfig`(4) manpages and the *TCP/IP Network Administration Guide.*

`NETPATH` is an environment variable. Its format is a simple ordered list of network identifiers separated by colons (:). An example is: `udp:tcp`.

By setting `NETPATH`, the user specifies the order in which the application tries the available transports. If `NETPATH` is not set, the system defaults to all visible transports specified in `/etc/netconfig`, in the order they appear in that file.

An application can ignore a user's NETPATH and set its own transport-selection order. The ability to make a predetermined choice of transport is a convenience to the developer but is not mandatory.

RPC divides selectable transports into the following types:

NULL	Same as selecting netpath.
visible	Uses the transports chosen with the visible flag ('v') set in their /etc/netconfig entries.
circuit_v	Same as visible, but restricted to connection-oriented transports. Transports are selected in the order listed in /etc/netconfig.
datagram_v	Same as visible, but restricted to connectionless transports.
circuit_n	Uses the connection-oriented transports chosen in the order defined in NETPATH.
datagram_n	Uses the connectionless transports chosen in the order defined in NETPATH.
udp	Specifies Internet user datagram protocol (UDP).
tcp	Specifies Internet transport control protocol (TCP).

A transport-aware application starts by calling setnetconfig(3N), getnetconfig(3N), and endnetconfig(3N) to search /etc/netconfig for the right type of transport. The data are stored in local netconfig data structures for later use.

These mechanisms allow a fine degree of control over network selection: a user can specify a preferred transport, and if it can, an application uses it. If the specified transport is inappropriate, the application automatically tries others with the right characteristics.

Transport Selection

RPC services are supported on both circuit-oriented and datagram transports. The selection of the transport depends on the requirements of the application.

A datagram transport is the transport of choice if the application has all of the following characteristics:

- Calls to the procedures do not change the state of the procedure or of associated data.
- The size of both the arguments and results is smaller than the transport packet size.
- The server is required to handle hundreds of clients. A datagram server does not keep any state data on clients, so it can potentially handle many clients. A circuit-oriented server keeps state data on each open client connection, so the number of clients is limited by the host resources.

A circuit-oriented transport is the transport of choice if the application has any of the following characteristics:

- The application can tolerate or amortize the higher cost of connection setup compared to datagram transports.
- Calls to the procedures can change the state of the procedure or of associated data.
- The size of either the arguments or the results exceed the maximum size of a datagram packet.

Name-to-Address Translation

Each transport has an associated set of routines that translate between universal network addresses (string representations of transport addresses) and the local address representation. These universal addresses are passed around within the RPC system (for example, between `rpcbind` and a client). A run-time linkable library that contains the name-to-address translation routines is associated with each transport. Table 2-1 shows the main translation routines.

For more details on these routines, see the `netdir(3N)` manpage and Chapter 5, “Transport Selection and Name-to-Address Mapping.” Note that the `netconfig` structure in each case provides the context for name-to-address translations.

Table 2-1 Name-to-Address Translation Routines

`netdir_getbyname`	Translates from host/service pairs (e.g. `server1`, `rpcbind`) and a `netconfig` structure to a set of `netbuf` addresses. `netbuf`s are Transport Level Interface (TLI) structures that contain transport-specific addresses at run-time.
`netdir_getbyaddr`	Translates from `netbuf` addresses and a `netconfig` structure to host/service pairs.
`uaddr2taddr`	Translates from universal addresses and a `netconfig` structure to `netbuf` addresses.
`taddr2uaddr`	Translates from `netbuf` addresses and a `netconfig` structure to universal addresses.

Address Lookup Using `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`. `rpcbind` replaces the `portmap` utility.

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)` and `rpcbind(3N)`manpages.

`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

`rpcbind` maps RPC services to their addresses, so its address must be known. The name-to-address translation routines must reserve a known address for each type of transport used. 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`(1M) and to programs using library routines named in the `rpcbind(3N)` manpage. 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. For details, see Appendix C, "RPC Protocol and Language Specification."

2

The `rpcinfo` *Utility*

`rpcinfo` is a utility that reports current RPC information registered with `rpcbind`. `rpcinfo` (with either `rpcbind` or the `portmap` utility) reports the universal addresses and the transports for all registered RPC services on a specified host. It 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(1M)` manpage.

rpcgen Programming Guide 3

This chapter introduces the `rpcgen` tool and provides a tutorial with code examples and usage of the available compile-time flags. See “RPC Programming Terms” on page 449 for the definition of the terms used in this chapter.

What is `rpcgen`

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

The default output of `rpcgen` is:

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

`rpcgen` can optionally generate:

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

`rpcgen` significantly reduces the development time that would otherwise be spent developing low-level routines. Handwritten routines link easily with the `rpcgen` output. (For a discussion of RPC programming without `rpcgen`, see Chapter 4, "RPC Programming Guide.")

SunOS 5.x Features

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

Template Generation

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

C-style Mode

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

Multithread-Safe Code

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

Multithread Auto Mode

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

Library Selection

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

ANSI C -compliant Code

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

An `rpcgen` *Tutorial*

`rpcgen` provides programmers a simple and direct way to write distributed applications. Server procedures may be written in any language that observes procedure-calling conventions. They are linked with the server stub produced by `rpcgen` to form an executable server program. Client procedures are written and linked in the same way.

This section presents some basic `rpcgen` programming examples. Refer also to the `rpcgen(1)`manpage.

Converting Local Procedures to Remote Procedures

Assume that an application runs on a single computer and you want to convert it to run in a “distributed” manner on a network. This example shows the stepwise conversion of this program that writes a message to the system console. Code Example 3-1 shows the original program.

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

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

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

```
{
    char *message;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <message>\n", argv[0]);
        exit(1);
    }
    message = argv[1];
    if (!printmessage(message)) {
        fprintf(stderr,"%s: couldn't print your message\n",argv[0]);
    exit(1);
    }
    printf("Message Delivered!\n");
    exit(0);
}

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

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

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

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

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

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

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

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

Remote procedures are always declared as part of remote programs. The code above declares an entire remote program that contains the single procedure `PRINTMESSAGE`. In this example, the `PRINTMESSAGE` procedure is declared to be procedure `1`, in version `1` of the remote program `MESSAGEPROG`, with the program number `0x20000001`. (See Appendix C, "RPC Protocol and Language Specification" for guidance on choosing program numbers.) Version numbers are incremented when functionality is changed in the remote program. Existing procedures can be changed or new ones can be added. More than one version of a remote program can be defined and a version can have more than one procedure defined.

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

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

There are just two more programs to write:

- The remote procedure itself
- The main client program that calls it

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

Code Example 3-2 RPC Version of `printmsg.c`

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

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

int *
printmessage_1(msg, req)
    char **msg;
    struct svc_req *req;         /* details of call */
{
    static int result;           /* must be static! */
    FILE *f;

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

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

1. It takes a pointer to the character array instead of the pointer itself. This is true of all remote procedures when the `-N` option is not used: They always take pointers to their arguments rather than the arguments themselves. Without the `-N` option, remote procedures are always called with a single argument. If more than one argument is required the arguments must be passed in a `struct`.

2. It is called with two arguments. The second argument contains information on the context of an invocation: the program, version, and procedure numbers, raw and canonical credentials, and an SVCXPRT structure pointer (the SVCXPRT structure contains transport information). This information is made available in case the invoked procedure requires it to perform the request.

3. It returns a pointer to an integer instead of the integer itself. This is also true of remote procedures when the `-N` option is not used: They return pointers to the result. The result should be declared `static` *unless* the `-M` (multithread) or `-A` (Auto mode) options are used. Ordinarily, if the result is

declared local to the remote procedure, references to it by the server stub are invalid after the remote procedure returns. In the case of `-M` and `-A` options, a pointer to the result is passed as a third argument to the procedure, so the result is not declared in the procedure.

4. An `_1` is appended to its name. In general, all remote procedures calls generated by `rpcgen` are named as follows: the procedure name in the program definition (here `PRINTMESSAGE`) is converted to all lowercase letters, an underbar (_) is appended to it, and the version number (here `1`) is appended. This naming scheme allows multiple versions of the same procedure.

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

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

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

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

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

    server = argv[1];
    message = argv[2];
    /*
     * Create client "handle" used for calling MESSAGEPROG on the
     * server designated on the command line.
     */
    clnt = clnt_create(server, MESSAGEPROG, PRINTMESSAGEVERS,
                       "visible");
```

```
    if (clnt == (CLIENT *)NULL) {
        /*
         * Couldn't establish connection with server.
         * Print error message and die.
         */
        clnt_pcreateerror(server);
        exit(1);
    }

    /*
     * Call the remote procedure "printmessage" on the server
     */
    result = printmessage_1(&message, clnt);
    if (result == (int *)NULL) {
        /*
         * An error occurred while calling the server.
         * Print error message and die.
         */
        clnt_perror(clnt, server);
        exit(1);
    }
    /* Okay, we successfully called the remote procedure. */
    if (*result == 0) {
        /*
         * Server was unable to print our message.
         * Print error message and die.
         */
        fprintf(stderr,
        "%s: could not print your message\n",argv[0]);
        exit(1);
    }

    /* The message got printed on the server's console */
    printf("Message delivered to %s\n", server);
    clnt_destroy( clnt );
    exit(0);
}
```

Note the following about Code Example 3-3:

1. First, a client handle is created by the RPC library routine `clnt_create()`. This client handle is passed to the stub routine that calls the remote procedure. (The client handle can be created in other ways as well. See

Chapter 4, “RPC Programming Guide“for details.) If no more calls are to be made using the client handle, destroy it with a call to `clnt_destroy()` to conserve system resources.

2. The last parameter to `clnt_create()` is `visible`, which specifies that any transport noted as visible in `/etc/netconfig` can be used. For further information on this, see the `/etc/netconfig` file and its description in “/etc/netconfig File” on page 153."

3. The remote procedure `printmessage_1()` is called exactly the same way as it is declared in `msg_proc.c`, except for the inserted client handle as the second argument. It also returns a pointer to the result instead of the result.

4. The remote procedure call can fail in two ways. The RPC mechanism can fail or there can be an error in the execution of the remote procedure. In the former case, the remote procedure (`printmessage_1()` returns a `NULL`. In the latter case, the error reporting is application dependent. Here, the error is returned through `*result`.

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

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

First, `rpcgen` was used to generate the header files (`msg.h`), client stub (`msg_clnt.c`), and server stub (`msg_svc.c`). Then, two programs are compiled: the client program `rprintmsg` and the server program `msg_server`. The C object files must be linked with the library `libnsl`, which contains all of the networking functions, including those for RPC and XDR.

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

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

1. It created a header file called `msg.h` that contained `#define` statements for `MESSAGEPROG`, `MESSAGEVERS`, and `PRINTMESSAGE` for use in the other modules. This file must be included by both the client and server modules.

2. It created the client stub routines in the `msg_clnt.c` file. Here there is only one, the `printmessage_1()` routine, that was called from the `rprintmsg` client program. If the name of an `rpcgen` input file is `FOO.x`, the client stub’s output file is called `FOO_clnt.c`.

3. It created the server program in `msg_svc.c` that calls `printmessage_1()` from `msg_proc.c`. The rule for naming the server output file is similar to that of the client: for an input file called `FOO.x`, the output server file is named `FOO_svc.c`.

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

```
remote$ msg_server
```

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

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

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

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

Passing Complex Data Structures

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

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

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

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

```
 */

const MAXNAMELEN = 255;          /* max length of directory entry */
typedef string nametype<MAXNAMELEN>;          /* directory entry */
typedef struct namenode *namelist;         /* link in the listing */

/* A node in the directory listing */
struct namenode {
    nametype name;                    /* name of directory entry */
    namelist next;                              /* next entry */
};

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

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

/* The directory program definition */
program DIRPROG {
    version DIRVERS {
        readdir_res
        READDIR(nametype) = 1;
    } = 1;
} = 0x20000076;
```

You can redefine types (like `readdir_res` in the example above) using the `struct`, `union`, and `enum` RPC language keywords. These keywords are not used in later declarations of variables of those types. For example, if you define a union, `foo`, you declare using only `foo`, and not `union foo`.

`rpcgen` compiles RPC unions into C structures. Do not declare C unions using the `union` keyword.

Running `rpcgen` on `dir.x` generates four output files: (1) the header file, (2) the client stub, (3) the server skeleton, and (4) the XDR routines in the file `dir_xdr.c`. This last file contains the XDR routines to convert declared data types from the host platform representation into XDR format, and vice versa.

For each RPCL data type used in the `.x` file, `rpcgen` assumes that `libnsl` contains a routine whose name is the name of the data type, prepended by the XDR routine header `xdr_` (for example, `xdr_int`). If a data type is defined in the `.x` file, `rpcgen` generates the required `xdr_` routine. If there is no data type definition in the.x source file (for example, `msg.x`), then no `_xdr.c` file is generated.

You can write a `.x` source file that uses a data type not supported by `libnsl`, and deliberately omit defining the type (in the `.x` file). In doing so, you must provide the `xdr_` routine. This is a way to provide your own customized `xdr_` routines. See Chapter 4, “RPC Programming Guide,“for more details on passing arbitrary data types. The server-side of the `READDIR` procedure is shown in Code Example 3-5.

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

```
/*
 * dir_proc.c: remote readdir implementation
 */
#include <dirent.h>
#include "dir.h"                /* Created by rpcgen */

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

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

{
    DIR *dirp;
    struct dirent *d;
    namelist nl;
    namelist *nlp;
    static readdir_res res; /* must be static! */

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

```
    if (dirp == (DIR *)NULL) {
        res.errno = errno;
        return (&res);
    }
    /* Free previous result */
    xdr_free(xdr_readdir_res, &res);
    /*
     * Collect directory entries. Memory allocated here is freed by
     * xdr_free the next time readdir_1 is called
     */
    nlp = &res.readdir_res_u.list;
    while (d = readdir(dirp)) {
        nl = *nlp = (namenode *) malloc(sizeof(namenode));
        if (nl == (namenode *) NULL) {
            res.errno = EAGAIN;
            closedir(dirp);
            return(&res);
        }
        nl->name = strdup(d->d_name);
        nlp = &nl->next;
    }
    *nlp = (namelist)NULL;
    /* Return the result */
    res.errno = 0;
    closedir(dirp);
    return (&res);
}
```

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

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

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

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

extern int errno;

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

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

    if (argc != 3) {
        fprintf(stderr, "usage: %s host directory\n",argv[0]);
        exit(1);
    }
    server = argv[1];
    dir = argv[2];
    /*
     * Create client "handle" used for calling MESSAGEPROG
     * on the server designated on the command line.
     */
    cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");
    if (clnt == (CLIENT *)NULL) {
        clnt_pcreateerror(server);
        exit(1);
    }
    result = readdir_1(&dir, clnt);
    if (result == (readdir_res *)NULL) {
        clnt_perror(clnt, server);
        exit(1);
    }
    /* Okay, we successfully called the remote procedure. */
    if (result->errno != 0) {
        /* Remote system error. Print error message and die. */
        errno = result->errno;
        perror(dir);
        exit(1);
    }
    /* Successfully got a directory listing. Print it. */
    for (nl = result->readdir_res_u.list; nl != NULL;
        nl = nl->next) {
        printf("%s\n", nl->name);
    }
    xdr_free(xdr_readdir_res, result);
    clnt_destroy(cl);
    exit(0);
}
```

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

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

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

```
local$ rls remote /usr/share/lib
ascii
eqnchar
greek
kbd
marg8
tabclr
tabs
tabs4
local$
```

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

Preprocessing Directives

`rpcgen` supports C and other preprocessing features. C preprocessing is performed on `rpcgen` input files before they are compiled. All standard C preprocessing directives are allowed in the `.x` source files. Depending on the type of output file being generated, five symbols are defined by `rpcgen`.

rpcgen provides an additional preprocessing feature: any line that begins with a percent sign (%) is passed directly to the output file, with no action on the line's content. Caution is required because rpcgen does not always place the lines where you intend. Check the output source file and, if needed, edit it.

Table 3-1 rpcgen Preprocessing Directives

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

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

Code Example 3-7 Time Protocol rpcgen Source

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

#ifdef RPC_SVC
%int *
%timeget_1()
%{
%   static int thetime;
%
%   thetime = time(0);
%   return (&thetime);
%}
#endif
```

cpp *Directive*

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

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

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

Compile-Time Flags

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

Table 3-2 rpcgen Compile-time Flags

Option	Flag	Comments
Templates	-a, -Sc, -Ss, -Sm	See Table 3-3 on page 38
C-style	-N	Also called Newstyle mode
ANSI C	-C	Often used with the -N option
MT-Safe code	-M	For use in multithreaded environments
MT Auto mode	-A	-A also turns on -M option
TS-RPC library	-b	TI-RPC library is default
xdr_inline count	-i	Uses 5 packed elements as default, but other number may be specified

3

Client and Server Templates

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

Table 3-3 `rpcgen` Template Selection Flags

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

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

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

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

The result is stored in the file `add_server_template.c`. A C-style mode, client template for the same `add.x` source is generated with the command line:

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

The result is stored in the file `add_client_template.c`. A make file template for the same add.x source is generated with the command line:

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

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

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

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

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

C-style Mode

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

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

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

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

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

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

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

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

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

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

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

Code Example 3-11 Default Mode Client

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

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

Code Example 3-12 C-style Mode Server

```
#include "add.h"

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

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

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

Code Example 3-13 Default Mode Server Stub

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

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

MT-Safe Code

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

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

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

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

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

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

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

Code Example 3-15 MT-Safe Client Stub

```
#include "msg.h"

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

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

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

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

    clnt_destroy(clnt);
}

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

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

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

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

```
int *
printmessage_1(argp, clnt)
    char **argp;
    CLIENT *clnt;
{
    static int clnt_res;
    memset((char *)&clnt_res, 0, sizeof (clnt_res));
    if (clnt_call(clnt, PRINTMESSAGE,
        (xdrproc_t) xdr_wrapstring, (caddr_t) argp,
```

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

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

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

Code Example 3-17 MT-Safe Server Stub

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

#include "add.h"

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

```
};

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

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

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

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

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

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

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

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

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

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

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

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

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

MT Auto Mode

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

Further discussion on multithreaded RPC begins on page 112; see also “MT Auto Mode” on page 121.

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

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

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

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

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

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

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

TI-RPC or TS-RPC Library Selection

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

ANSI C-compliant Code

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

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

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

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

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

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

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

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

xdr_inline Count

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

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

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

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

prevents any in-line code from being generated.

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

rpcgen Programming Techniques

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

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

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

Network Types/Transport Selection

`rpcgen` takes optional arguments that allow a programmer to specify desired network types or specific network identifiers. (For details of network selection, see Chapter 7, “Socket Interface.“)

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

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

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

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

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

Command Line Define Statements

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

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

Server Response to Broadcast Calls

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

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

Code Example 3-23 NFS Server Response to Broadcast Calls

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

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

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

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

Port Monitor Support

Port monitors such as `inetd` and `listen` can monitor network addresses for specified RPC services. When a request arrives for a particular service, the port monitor spawns a server process. After the call has been serviced, the server can exit. This technique conserves system resources. The `main` server function generated by `rpcgen` allows invocation by `inetd`. See “Using inetd” on page 107“for details.

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

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

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

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

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

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

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

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

Time-out Changes

After sending a request to the server, a client program waits for a default period (25 seconds) to receive a reply. This time-out may be changed using the `clnt_control()` routine. See “Lower Levels of RPC” on page 71,” for additional uses of the `clnt_control` routine. See also the `rpc`(3N) manpage. When considering time-out periods, be sure to allow the minimum amount of time required for “round-trip” communications over the network.
Code Example 3-24 illustrates the use of `clnt_control()`.

Code Example 3-24 `clnt_control` Routine

```
struct timeval tv;

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

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

Client Authentication

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

The following example illustrates one of the least secure authentication methods in common use. See “Authentication” on page 98, for information on the more secure DES authentication technique.

Code Example 3-25 `AUTH_SYS` Authentication Program

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

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

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

Code Example 3-26 `printmsg_1` for Superuser

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

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

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

Dispatch Tables

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

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

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

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

where:

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

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

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

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

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

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

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

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

And use the following define when writing the server:

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

Debugging Applications

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

Link the procedures from previous example by:

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

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

You can also use the Raw RPC mode to test the XDR routines. See “Testing Programs Using Low-level Raw RPC” on page 87” for details.

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

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

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

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

3

RPC Programming Guide 4

This chapter discusses writing network applications using remote procedure calls, and the RPC mechanisms usually hidden by `rpcgen`. See “RPC Programming Terms” on page 449 for the definition of the terms used in this chapter.

RPC Is Multithread Safe

The interfaces described in this chapter are multithread safe, except where noted (such as raw mode). This means that applications that contain RPC function calls can be used freely in a multithreaded application. For a complete specification of the routines in the RPC library, see the `rpc(3N)` and related manpages.

The RPC server interfaces were made safe with the Solaris 2.4/SunOS 5.4 release. The two multithreaded programming modes available for developing RPC servers are MT Auto Mode and MT User Mode.

4

Introduction to RPC Interface Levels

RPC has multiple levels of application interface to its services. These levels provide different degrees of control balanced with different amounts of interface code to implement. In order of increasing control and complexity, the levels are the *simplified level*, *top level*, *intermediate level*, *expert level*, and *bottom level*. This section gives a summary of the routines available at each level.

Simplified Interface Routines

The simplified interfaces are used to make remote procedure calls to routines on other machines, and specify only the *type* of transport to use. The routines at this level are used for most applications. There are three basic RPC routines, shown in Table 4-1.

Table 4-1 RPC Routines—Simplified Level

Routine	Function
`rpc_reg()`	Registers a procedure as an RPC program on all transports of the specified type.
`rpc_call()`	Remote calls the specified procedure on the specified remote host.
`rpc_broadcast()`	Broadcasts a call message across all transports of the specified type.

Use of these routines and code samples can be found in the section, “Simplified Interface” on page 63.

Top Level Routines

At the top level, the interface is still simple, but the program has to create a client handle before making a call or create a server handle before receiving calls. Like the routines in the simplified interface, these routines require a `nettype` argument that specifies the type or types of transport.

The top level has three routines, shown in Table 4-2.

Table 4-2 RPC Routines—Top Level

Routine	Description
`clnt_create()`	Generic client creation. The program tells `clnt_create()` where the server is located and the type of transport to use.
`svc_create()`	Creates server handles for all transports of the specified type. The program tells `svc_create()` which dispatch function to use.
`clnt_call()`	Client calls a procedure to send a request to the server.

If you want the application to run on all transports, use this interface. Use of these routines and code samples can be found in the section, “Top Level Interface” on page 71.

Intermediate Level Routines

The intermediate level interface of RPC lets you control details. Programs written at these lower levels are more complicated but run more efficiently. The intermediate level enables you to specify the transport to use. It contains the routines shown in Table 4-3.

Table 4-3 RPC Routines—Intermediate Level

Routine	Description
`clnt_tp_create()`	Creates a client handle for the specified transport.
`svc_tp_create()`	Creates a server handle for the specified transport.
`clnt_call()`	Client calls a procedure to send a request to the server.

Use of these routines and code samples can be found in the section, “Intermediate Level Interface” on page 75.

Expert Level Routines

The expert level contains a larger set of routines with which to specify transport-related parameters. It includes routines shown in Table 4-4.

Table 4-4 RPC Routines—Expert Level

Routine	Description
`clnt_tli_create()`	Creates a client handle for the specified transport.
`svc_tli_create()`	Creates a server handle for the specified transport.
`rpcb_set()`	Calls `rpcbind` to set a map between an RPC service and a network address.
`rpcb_unset()`	Deletes a mapping set by `rpcb_set`.
`rpcb_getaddr()`	Calls rpcbind to get the transport addresses of specified RPC services.
`svc_reg()`	Associates the specified program and version number pair with the specified dispatch routine.
`svc_unreg()`	Deletes an association set by `svc_reg`.
`clnt_call()`	Client calls a procedure to send a request to the server.

Use of these routines and code samples can be found in the section, “Expert Level Interface” on page 78.

Bottom Level Routines

The bottom level contains routines used for full control of transport options. It consists of the routines inTable 4-5.

Table 4-5 RPC Routines—Bottom Level

Routine	Description
`clnt_dg_create()`	Creates an RPC client handle for the specified remote program, using a connectionless transport.
`svc_dg_create()`	Creates an RPC server handle, using a connectionless transport.

Table 4-5 RPC Routines—Bottom Level

Routine	Description
`clnt_vc_create()`	Creates an RPC client handle for the specified remote program, using a connection-oriented transport.
`svc_vc_create()`	Creates an RPC server handle, using a connection-oriented transport.
`clnt_call()`	Client calls a procedure to send a request to the server.

Use of these routines and code samples can be found in the section, “Bottom Level Interface” on page 83.

Simplified Interface

The simplified interface is the highest level of RPC. It is the easiest level to use because it does not require the use of any other RPC routines. It also limits control of the underlying communications mechanisms. Program development at this level can be rapid, and is directly supported by the `rpcgen` compiler. For most applications, `rpcgen` and its facilities are sufficient.

Some RPC services are not available as C functions, but they are available as RPC programs. “RPC Library-Based Network Services” on page 63 describes these easy-to-use services, and how to create new services that are equally simple to use.

Data types passed to and received from remote procedures can be any of a set of predefined types, or can be programmer-defined types. “Passing Arbitrary Data Types” on page 67 explains how to declare and use these types.

RPC Library-Based Network Services

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

Code Example 4-1 `rusers` Program, Example One

```
#include <stdio.h>

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

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

    if (argc != 2) {
        fprintf(stderr, "usage: %s hostname\n", argv[0]);
        exit(1);
    }
    if ((num = rusers(argv[1])) < 0) {
        fprintf(stderr, "error: rusers\n");
        exit(1);
    }
    fprintf(stderr, "%d users on %s\n", num, argv[1] );
    exit(0);
}
```

Routines such as `rusers()` are in the RPC services library `librpcsvc`. Compile the program in Code Example 4-1 with:

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

The simplified interface library routines provide direct access to the RPC facilities for programs that do not require fine levels of control. The example in Code Example 4-1, changed to use the simplified interface, looks like Code Example 4-2.

Code Example 4-2 `rusers` Program, Example Two

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

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

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

    if (argc != 2) {
        fprintf(stderr, "usage: rusers hostname\n");
```

```
        exit(1);
    }
    if( clnt_stat = rpc_call(argv[1], RUSERSPROG, RUSERSVERS,
      RUSERSPROC_NUM, xdr_void, (char *)0, xdr_u_long,
      (char *)&nusers, "visible") != RPC_SUCCESS) {
        clnt_perrno(clnt_stat);
        exit(1);
    }
    fprintf(stderr, "%d users on %s\n", nusers, argv[1] );
    exit(0);
}
```

`rpc_call()` *Routine*

`rpc_call()` has nine parameters:

- The first is the name of the remote server machine
- The next three parameters are the program, version, and procedure numbers; together, they identify the remote procedure to be called
- The fifth and sixth parameters are the XDR translator to encode it and an argument to be passed to the remote procedure
- The seventh and eighth parameters are the XDR translator to decode the result returned by the remote procedure and a pointer to where the result is to be stored
- Ninth is the `nettype` specifier

Multiple arguments and results are handled by collecting them in structures. If `rpc_call()` completes successfully, it returns `RPC_SUCCESS`. The return codes (of type `enum clnt_stat`) are in `<rpc/clnt.h>`.

Since data types may be represented differently on different machines, `rpc_call()` needs both the type of, and a pointer to, the RPC argument (similarly for the result). For `RUSERSPROC_NUM`, the return value is an `unsigned long`, so the first return parameter of `rpc_call()` is `xdr_u_long()` (which is for an unsigned `long`) and the second is `&nusers` (which points to unsigned `long` storage). Because `RUSERSPROC_NUM` has no argument, the XDR encoding function of `rpc_call()` is `xdr_void()` and its argument is `NULL`.

If `rpc_call()` receives no answer within a certain time period, it returns an error code. In the example, all "visible" transports listed in `/etc/netconfig` are tried. Adjusting the number of retries requires use of the lower levels of the RPC library. Code Example 4-3 shows a remote server procedure for Code Example 4-2.

Code Example 4-3 `rusers` Remote Server Procedure

```
void *
rusers()
{
    static unsigned long nusers;
    /*
     * Code here to compute the number of users
     * and place result in variable nusers.
     */
    return((void *)&nusers);
}
```

`rpc_reg()` *Routine*

Server procedures and functions are registered with `rpcbind` to make them accessible to clients. A server registers all the RPC calls it handles, then blocks to wait for requests.

Use `rpc_reg()` to register a C program as a specific program, version, and procedure. `rpc_reg()` has seven calling arguments:

- The first three arguments are the program, version, and procedure numbers of the remote procedure to be registered (`unsigned long`).
- The fourth argument points to a string containing the name of the function that implements the remote procedure.
- The fifth argument points to the XDR translator that processes the calling argument (or argument structure) of the program.
- The sixth argument points to the XDR translator that processes the return value (or structure) of the program.
- The last parameter specifies the desired transport type.

The procedure is registered for each transport of the specified type. If the type parameter is `(char *)NULL`, the procedure is registered for all transports specified in `NETPATH`.

After registering the local procedure, the server program's main procedure calls `svc_run()`, the RPC library's remote procedure dispatcher. This function invokes service procedures in response to RPC call messages. (Because the `svc_run()` routine is used at all levels of programming, it does not "belong" any particular level of RPC programming.) The dispatcher in `rpc_reg()` takes care of decoding remote procedure arguments and encoding results, using the XDR filters specified when the remote procedure was registered.

You can sometimes implement faster or more compact code than can `rpcgen`. `rpcgen` handles the generic code-generation cases. The following program is an example of a hand-coded registration routine. It registers a single procedure and enters `svc_run()` to service requests.

Code Example 4-4 "Hand-Coding" Registration Server

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
void *rusers();

main()
{
    if(rpc_reg(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM, rusers,
        xdr_void, xdr_u_long, "visible") == -1) {
        fprintf(stderr, "Couldn't Register\n");
        exit(1);
    }
    svc_run();     /* Never returns */
    fprintf(stderr, "Error: svc_run returned!\n");
    exit(1);
}
```

`rpc_reg()` can be called as many times as is needed to register different programs, versions, and procedures.

Passing Arbitrary Data Types

RPC handles arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, by always converting them to a standard transfer format called external data representation (XDR) before sending them over the transport. The conversion from a machine representation to XDR is called serializing, and the reverse process is called deserializing.

The translator arguments of `rpc_call()` and `rpc_reg()` can specify an XDR primitive procedure, like `xdr_u_long()`, or a programmer-supplied routine that processes a complete argument structure. Argument processing routines must take only two arguments: a pointer to the result and a pointer to the XDR handle.

Table 4-6 XDR Primitive Type Routines

XDR Primitive Routines			
`xdr_int()`	`xdr_netobj()`	`xdr_u_long()`	`xdr_enum()`
`xdr_long()`	`xdr_float()`	`xdr_u_short()`	`xdr_bool()`
`xdr_short()`	`xdr_double()`	`xdr_u_short()`	`xdr_wrapstring()`
`xdr_char()`	`xdr_quadruple`	`xdr_u_char()`	`xdr_void()`

The nonprimitive `xdr_string`, which takes more than two parameters, is called from `xdr_wrapstring()`.

For an example of a programmer-supplied routine, the structure:

```
struct simple {
    int a;
    short b;
} simple;
```

contains the calling arguments of a procedure. The XDR routine `xdr_simple`() translates the argument structure as shown in Code Example 4-5.

Code Example 4-5 `xdr_simple` Routine

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

bool_t
xdr_simple(xdrsp, simplep)
    XDR *xdrsp;
    struct simple *simplep;
{
    if (!xdr_int(xdrsp, &simplep->a))
        return (FALSE);
    if (!xdr_short(xdrsp, &simplep->b))
        return (FALSE);
    return (TRUE);
}
```

An equivalent routine can be generated automatically by `rpcgen`.

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

Table 4-7 XDR Building Block Routines

Prefabricated Routines		
`xdr_array()`	`xdr_bytes()`	`xdr_reference()`
`xdr_vector()`	`xdr_union()`	`xdr_pointer()`
`xdr_string()`	`xdr_opaque()`	

For example, to send a variable-sized array of integers, it is packaged in a structure containing the array and its length:

```
struct varintarr {
    int *data;
    int arrlnth;
} arr;
```

Translate the array with `xdr_varintarr()`, as shown in Code Example 4-6.

Code Example 4-6 `xdr_varintarr` Syntax Use

```
bool_t
xdr_varintarr(xdrsp, arrp)
    XDR *xdrsp;
    struct varintarr *arrp;
{
    return(xdr_array(xdrsp, (caddr_t)&arrp->data,
        (u_int *)&arrp->arrlnth, MAXLEN, sizeof(int), xdr_int));
}
```

The arguments of `xdr_array()` are the XDR handle, a pointer to the array, a pointer to the size of the array, the maximum array size, the size of each array element, and a pointer to the XDR routine to translate each array element. If the size of the array is known in advance, use `xdr_vector()`, as shown in Code Example 4-7.

Code Example 4-7 `xdr_vector` Syntax Use

```
int intarr[SIZE];

bool_t
```

```
xdr_intarr(xdrsp, intarr)
    XDR *xdrsp;
    int intarr[];
{
    return (xdr_vector(xdrsp, intarr, SIZE, sizeof(int), xdr_int));
}
```

XDR converts quantities to 4-byte multiples when serializing. For arrays of characters, each character occupies 32 bits. `xdr_bytes()` packs characters. It has four parameters similar to the first four parameters of `xdr_array()`.

Null-terminated strings are translated by `xdr_string()`. It is like `xdr_bytes()` with no length parameter. On serializing it gets the string length from `strlen()`, and on deserializing it creates a null-terminated string.

Code Example 4-8 calls the built-in functions `xdr_string()` and `xdr_reference()`, which translates pointers to pass a string, and `struct simple` from the previous examples.

Code Example 4-8 `xdr_reference` Syntax Use

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

bool_t
xdr_finalexample(xdrsp, finalp)
    XDR *xdrsp;
    struct finalexample *finalp;
{
    if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
        return (FALSE);
    if (!xdr_reference( xdrsp, &finalp->simplep,
         sizeof(struct simple), xdr_simple))
        return (FALSE);
    return (TRUE);

}
```

Note that `xdr_simple()` could have been called here instead of `xdr_reference()`.

Lower Levels of RPC

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

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

The routines shown in Table 4-8 cannot be used through the simplified interface because they require a transport handle. For example, there is no way to allocate and free memory while serializing or deserializing with XDR routines at the simplified interface.

Table 4-8 XDR Routines Requiring a Transport Handle

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

Top Level Interface

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

Client

Assume the header file in Code Example 4-9.

Code Example 4-9 `time_prot.h` Header File

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

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

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

Code Example 4-10 shows the client side of a trivial date service using top-level service routines. The transport type is specified as an invocation argument of the program.

Code Example 4-10 Client for Trivial Date Service

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

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

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

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

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

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

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

Server

Code Example 4-11 shows a top-level implementation of a server for the trivial date service.

Code Example 4-11 Server for Trivial Date Service

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

static void time_prog();

main(argc,argv)
```

```
    int argc;
    char *argv[];
{
    int transpnum;
    char *nettype;

    if (argc > 2) {

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

/*
 * The server dispatch function
 */
static void
time_prog(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    struct timev rslt;
    time_t thetime;

    switch(rqstp->rq_proc) {
        case NULLPROC:
            svc_sendreply(transp, xdr_void, NULL);
            return;
        case TIME_GET:
            break;
        default:
            svcerr_noproc(transp);
            return;
    }
    thetime = time((time_t *) 0);
```

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

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

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

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

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

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

Intermediate Level Interface

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

Client

Code Example 4-12 shows the client side of the time service from “Top Level Routines” on page 60, written at the intermediate level of RPC. In this example, the user must name the transport over which the call is made on the command line.

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

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

```
#include "time_prot.h"

#define TOTAL (30)

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

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

In this example, the `netconfig` structure is obtained by a call to `getnetconfigent(netid)`. (See the `getnetconfig(3N)` manpage and Chapter 5, “Transport Selection and Name-to-Address Mapping“for more details.) At this level, the program explicitly selects the network.

Server

Code Example 4-13 shows the corresponding server. The command line that starts the service must specify the transport over which the service is provided.

Code Example 4-13 Server for Time Service, Intermediate Level

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

static void time_prog();

main(argc, argv)
    int argc;
    char *argv[];
{
    SVCXPRT *transp;
    struct netconfig *nconf;


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

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

```
}
```

Expert Level Interface

At the expert level, network selection is done the same as at the intermediate level. The only difference is in the increased level of control that the application has over the details of the `CLIENT` and `SVCXPRT` handles. These examples illustrate this control, which is exercised using the `clnt_tli_create()` and `svc_tli_create()` routines. For more information on TLI, see Chapter 6, “Transport -level Interface (TLI) Programming Guide."

Client

Code Example 4-14 shows a version of `clntudp_create()` (the client creation routine for UDP transport) using `clnt_tli_create()`. The example shows how to do network selection based on the family of the transport you choose. `clnt_tli_create()` is used to create a client handle and to:

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

Code Example 4-14 Client for RPC Lower Level

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

```
int madefd = FALSE;             /* Is fd opened here */
int fd = *sockp;                /* TLI fd */
struct t_bind *tbind;           /* bind address */
struct netconfig *nconf;        /* netconfig structure */
void *handlep;

if ((handlep = setnetconfig() ) == (void *) NULL) {
    /* Error starting network configuration */
    rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
    return((CLIENT *) NULL);
}
/*
 * Try all the transports until it gets one that is
 * connectionless, family is INET, and preferred name is UDP
 */
while (nconf = getnetconfig( handlep)) {
    if ((nconf->nc_semantics == NC_TPI_CLTS) &&
         (strcmp( nconf->nc_protofmly, NC_INET ) == 0) &&
         (strcmp( nconf->nc_proto, NC_UDP ) == 0))
     break;
}
if (nconf == (struct netconfig *) NULL)
    rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
    goto err;
}
if (fd == RPC_ANYFD) {
    fd = t_open(nconf->nc_device, O_RDWR, &tinfo);
    if (fd == -1) {
        rpc_createerr.cf_stat = RPC_SYSTEMERROR;
        goto err;
    }
}
if (raddr->sin_port == 0) { /* remote addr unknown */
    u_short sport;
    /*
     * rpcb_getport() is a user-provided routine that calls
     * rpcb_getaddr and translates the netbuf address to port
     * number in host byte order.
     */
    sport = rpcb_getport(raddr, prog, vers, nconf);
    if (sport == 0) {
        rpc_createerr.cf_stat = RPC_PROGUNAVAIL;
        goto err;
    }
    raddr->sin_port = htons(sport);
}
```

```
    /* Transform sockaddr_in to netbuf */
    tbind = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR);
    if (tbind == (struct t_bind *) NULL)
        rpc_createerr.cf_stat = RPC_SYSTEMERROR;
        goto err;
    }
    if (t_bind->addr.maxlen < sizeof( struct sockaddr_in))
        goto err;
    (void) memcpy( tbind->addr.buf, (char *)raddr,
                   sizeof(struct sockaddr_in));
    tbind->addr.len = sizeof(struct sockaddr_in);
    /* Bind fd */
    if (t_bind( fd, NULL, NULL) == -1) {
        rpc_createerr.ct_stat = RPC_TLIERROR;
        goto err;
    }
    cl = clnt_tli_create(fd, nconf, &(tbind->addr), prog, vers,
                         tinfo.tsdu, tinfo.tsdu);
    /* Close the netconfig file */
    (void) endnetconfig( handlep);
    (void) t_free((char *) tbind, T_BIND);
    if (cl) {
        *sockp = fd;
        if (madefd == TRUE) {
            /* fd should be closed while destroying the handle */
            (void)clnt_control(cl,CLSET_FD_CLOSE, (char *)NULL);
        }
        /* Set the retry time */
        (void) clnt_control( l, CLSET_RETRY_TIMEOUT,
                             (char *) &wait);
        return(cl);
    }
err:
    if (madefd == TRUE)
        (void) t_close(fd);
    (void) endnetconfig(handlep);
    return((CLIENT *) NULL);
}
```

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

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

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

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

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

Server

Code Example 4-15 shows the server side of Code Example 4-14. It is called `svcudp_create()`. The server side uses `svc_tli_create()`.

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

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

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

Code Example 4-15 Server for RPC Lower Level

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

SVCXPRT *
svcudp_create(fd)
    register int fd;
{
    struct netconfig *nconf;
    SVCXPRT *svc;
    int madefd = FALSE;
    int port;
```

```
    void *handlep;
    struct  t_info tinfo;

    /* If no transports available */
    if ((handlep = setnetconfig() ) == (void *) NULL) {
        nc_perror("server");
        return((SVCXPRT *) NULL);
    }
    /*
     * Try all the transports until it gets one which is
     * connectionless, family is INET and, name is UDP
     */
    while (nconf = getnetconfig( handlep)) {
        if ((nconf->nc_semantics == NC_TPI_CLTS) &&
            (strcmp( nconf->nc_protofmly, NC_INET) == 0 )&&
            (strcmp( nconf->nc_proto, NC_UDP) == 0 ))
            break;
    }
    if (nconf == (struct netconfig *) NULL) {
        endnetconfig(handlep);
        return((SVCXPRT *) NULL);
    }
    if (fd == RPC_ANYFD) {
        fd = t_open(nconf->nc_device, O_RDWR, &tinfo);
        if (fd == -1) {
            (void) endnetconfig();
            return((SVCXPRT *) NULL);
        }
        madefd = TRUE;
    } else
        t_getinfo(fd, &tinfo);
    svc = svc_tli_create(fd, nconf, (struct t_bind *) NULL,
                         tinfo.tsdu, tinfo.tsdu);
    (void) endnetconfig(handlep);
    if (svc == (SVCXPRT *) NULL) {
        if (madefd)
            (void) t_close(fd);
        return((SVCXPRT *)NULL);
    }
    return (svc);
}
```

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

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

Bottom Level Interface

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

Bottom-level routines create internal data structures, buffer management, RPC headers, and so on. Callers of these routines, like the expert level routine clnt_tli_create(), must initialize the cl_netid and cl_tp fields in the client handle. For a created handle, cl_netid is the network identifier (for example udp) of the transport and cl_tp is the device name of that transport (for example /dev/udp). The routines clnt_dg_create() and clnt_vc_create() set the clnt_ops and cl_private fields.

Client

Code Example 4-16 shows calls to clnt_vc_create() and clnt_dg_create().

Code Example 4-16 Client for Bottom Level

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

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

Server

Code Example 4-17 is an example of creating a bottom-level server.

Code Example 4-17 Server for Bottom Level

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

Server Caching

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

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

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

Low-Level Data Structures

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

First is the client RPC handle, defined in <rpc/clnt.h>. Low-level implementations must provide and initialize one handle per connection, as shown in Code Example 4-18.

Code Example 4-18 RPC Client Handle Structure

```
typedef struct {
    AUTH *cl_auth;                      /* authenticator */
    struct clnt_ops {
        enum clnt_stat(*cl_call)();   /* call remote procedure */
        void       (*cl_abort)();      /* abort a call */
        void       (*cl_geterr)();    /* get specific error code */
        bool_t     (*cl_freeres)();    /* frees results */
        void       (*cl_destroy)();    /* destroy this structure */
        bool_t     (*cl_control)();    /* the ioctl() of rpc */
    } *cl_ops;
    caddrt_t   cl_private;             /* private stuff */
    char       *cl_netid;              /* network token *l
    char       *cl_tp;                 /* device name */
} CLIENT;
```

The first field of the client-side handle is an authentication structure, defined in <rpc/auth.h>. By default, it is set to AUTH_NONE. A client program must initialize cl_auth appropriately, as shown in Code Example 4-19.

Code Example 4-19 Client Authentication Handle

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

In the AUTH structure, `ah_cred` contains the caller's credentials, and `ah_verf` contains the data to verify the credentials. See "Authentication" on page 98 for details.

Code Example 4-20 shows the server transport handle.

Code Example 4-20 Server Transport Handle

```
typedef struct {
    int         xp_fd;
#define xp_sock    xp_fd
    u_short xp_port; /* associated port number. Obsoleted */
    struct xp_ops {
        bool_t     (*xp_recv)();     /* receive incoming requests */
       enum xprt_stat (*xp_stat)();  /* get transport status */
        bool_t     (*xp_getargs)();    /* get arguments */
        bool_t     (*xp_reply)();      /* send reply */
        bool_t     (*xp_freeargs)();   /* free mem alloc for args */
        void       (*xp_destroy)();    /* destroy this struct */
    } *xp_ops;
    int     xp_addrlen;         /* length of remote addr. Obsoleted */
    char    *xp_tp;             /* transport provider device name */
    char    *xp_netid;            /* network token */
    struct netbuf  xp_ltaddr;   /* local transport address */
    struct netbuf  xp_rtaddr;   /* remote transport address */
    char           xp_raddr[16]; /* remote address. Now obsoleted */
    struct opaque_auth xp_verf; /* raw response verifier */
    caddr_t        xp_p1;        /* private: for use by svc ops */
    caddr_t        xp_p2;       /* private: for use by svc ops */
    caddr_t        xp_p3;       /* private: for use by svc lib */
} SVCXPRT;
```

Table 4-9 shows the fields for the server transport handle.

Table 4-9 RPC Server Transport Handle Fields

`xp_fd`	The file descriptor associated with the handle. Two or more server handles can share the same file descriptor.
`xp_netid`	The network identifier (e.g. `udp`) of the transport on which the handle is created and `xp_tp` is the device name associated with that transport.

Table 4-9 RPC Server Transport Handle Fields

`xp_ltaddr`	The server's own bind address.
`xp_rtaddr`	The address of the remote caller (and so may change from call to call).

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

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

For connection-oriented endpoints, the fields in Table 4-10 are not valid until a connection has been requested and accepted for the server:

Table 4-10 RPC Connection-Oriented Endpoints

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

Testing Programs Using Low-level Raw RPC

There are two pseudo-RPC interface routines that bypass all the network software. The routines shown in Code Example 4-21, `clnt_raw_create()` and `svc_raw_create()`, do not use any real transport.

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

Code Example 4-21 is compiled and linked using the following Makefile:

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

Code Example 4-21 Simple Program Using Raw RPC

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

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

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

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

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

    if (argc == 2)
        num = atoi(argv[1]);
        svc = svc_raw_create();
    if (svc == (SVCXPRT *) NULL) {
        fprintf(stderr, "Could not create server handle\n");
        exit(1);
    }
    flag = svc_reg( svc, prognum, versnum, server,
            (struct netconfig *) NULL );
    if (flag == 0) {
        fprintf(stderr, "Error: svc_reg failed.\n");
        exit(1);
    }
    cl = clnt_raw_create( prognum, versnum );
    if (cl == (CLIENT *) NULL) {
        clnt_pcreateerror("Error: clnt_raw_create");
        exit(1);
    }
```

```
    if (clnt_call(cl, INCR, xdr_int, (caddr_t) &num, xdr_int,
          (caddr_t) &ans, TIMEOUT)
      != RPC_SUCCESS) {
        clnt_perror(cl, "Error: client_call with raw");
        exit(1);
    }
    printf("Client: number returned %d\n", ans);
    exit(0);
}

static void
server(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    int num;

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

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

```
    }
    fprintf(stderr, "Leaving server procedure.\n");
}
```

Note the following points in Code Example 4-21:

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

Advanced RPC Programming Techniques

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

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

`poll()` on the Server Side

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

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

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

Code Example 4-22 shows `svc_run()`. `svc_pollset` is an array of `pollfd` structures that is derived, through a call to `__rpc_select_to_poll()`, from `svc_fdset`. The array can change every time *any* RPC library routine is called, because descriptors are constantly being opened and closed. `svc_getreq_poll()` is called when `poll()` determines that an RPC request has arrived on some RPC file descriptors.

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

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

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

```
int __rpc_dtbsize()
```

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

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

Code Example 4-22 `svc_run()` and `poll()`

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

    for (;;) {
```

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

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

Broadcast RPC

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

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

Code Example 4-23 shows how `rpc_broadcast()` is used and describes its arguments.

Code Example 4-23 RPC Broadcast

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

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

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

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

The function in Code Example 4-24 collects replies to the broadcast. Normal operation is to collect either the first reply or all replies. `bcast_proc` displays the IP address of the server that has responded. Since the function returns `FALSE` it will continue to collect responses, and the RPC client code will continue to resend the broadcast until it times out.

Code Example 4-24 Collect Broadcast Replies

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

    naddr = taddr2naddr(nconf, &taddr->addr);
    if (naddr == (char *) NULL) {
        fprintf(stderr,"Responded: unknown\n");
    } else {
        fprintf(stderr,"Responded: %s\n", naddr);
        free(naddr);
    }
    return(FALSE);
}
```

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

Batching

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

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

- The server does not respond to any intermediate message.
- The pipeline of calls is transported on a reliable transport, such as TCP.

- The result's XDR routine in the calls must be `NULL`.
- The RPC call's time-out must be zero.

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

Code Example 4-25 shows the unbatched version of the client. It scans the character array, `buf`, for delimited strings and sends each string to the server.

Code Example 4-25 Unbatched Client

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

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

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

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

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

Code Example 4-26 shows the batched version of the client. It does not wait after each string is sent to the server. It waits only for an ending response from the server.

Code Example 4-26 Batched Client

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

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

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

Code Example 4-27 shows the dispatch portion of the batched server. Because the server sends no message, the clients are not notified of failures.

Code Example 4-27 Batched Server

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

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

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

```
    /* Now free string allocated while decoding arguments */
    svc_freeargs(transp, xdr_wrapstring, &s);
}
```

Batching Performance

To illustrate the benefits of batching, the examples in Code Example 4-25, Code Example 4-26, and Code Example 4-27 were completed to render the lines in a 25144-line file. The rendering service simply throws the lines away. The batched version of the application was four times as fast as the unbatched version.

Authentication

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

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

Sun RPC currently supports the authentication flavors shown in Table 4-11.

Table 4-11 Authentication Methods Supported By Sun RPC

AUTH_NONE	Default. No authentication performed
AUTH_SYS	An authentication flavor based on UNIX operating system, process permissions authentication

Table 4-11 Authentication Methods Supported By Sun RPC

AUTH_SHORT	An alternate flavor of AUTH_SYS used by some servers for efficiency. Client programs using AUTH_SYS authentication can receive AUTH_SHORT response verifiers from some servers. See Appendix C, "RPC Protocol and Language Specification for details
AUTH_DES	An authentication flavor based on DES encryption techniques
AUTH_KERB	Version 4 Kerberos authentication based on DES framework

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

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

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

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

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

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

```
/*
 * Authentication data
 */
struct opaque_auth {
    enum_t    oa_flavor;         /* style of credentials */
    caddr_t   oa_base;           /* address of more auth stuff */
    u_int    oa_length;         /* not to exceed MAX_AUTH_BYTES */
};
```

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

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

AUTH_NONE flavor. rq_clntcred can be cast only as a pointer to an authsys_parms, short_hand_verf, authkerb_cred, or authdes_cred structure.

AUTH_SYS *Authentication*

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

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

This causes each RPC call associated with clnt to carry with it the following credentials-authentication structure shown in Code Example 4-28.

Code Example 4-28 AUTH_SYS Credential Structure

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

rpc.broadcast defaults to AUTH_SYS authentication.

Code Example 4-29 shows a server, with procedure RUSERPROC_1, that returns the number of users on the network. As an example of authentication, it checks AUTH_SYS credentials and does not service requests from callers whose uid is 16.

Code Example 4-29 Authentication Server

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

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

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

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

Note the following:

- The authentication parameters associated with the `NULLPROC` (procedure number zero) are usually not checked.
- The server calls `svcerr_weakauth()` if the authentication parameter's flavor is too weak; there is no way to get the list of authentication flavors the server requires.

- The service protocol should return status for access denied; in Code Example 4-29, the protocol calls the service primitive `svcerr_systemerr()`, instead.

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

AUTH_DES *Authentication*

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

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

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

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

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

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

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

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

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

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

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

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

The server side is simpler than the client. Code Example 4-30 shows the server in Code Example 4-29 changed to use `AUTH_DES`.

Code Example 4-30 `AUTH_DES` Server

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

    /* NULLPROC should never be authenticated */
    if (rqstp->rq_proc == NULLPROC) {
```

```
    /* same as before */
}
/* now get the uid */
switch(rqstp->rq_cred.oa_flavor) {
    case AUTH_DES:
        des_cred = (struct authdes_cred *) rqstp->rq_clntcred;
        if (! netname2user( des_cred->adc_fullname.name, &uid,
                            &gid, &gidlen, gidlist)) {
            fprintf(stderr, "unknown user: %s\n",
                     des_cred->adc_fullname.name);
            svcerr_systemerr(transp);
            return;
        }
        break;
    default:
        svcerr_weakauth(transp);
        return;
}
/* The rest is the same as before */
```

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

AUTH_KERB *Authentication*

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

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

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

Time Synchronization

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

Well-Known Names

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

- `root`.*host* represents a privileged user on client *host*.
- *user*.`ignored` represents the user whose user name is *user*. The instance is ignored.

Encryption

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

```
xdr_type(object)
```

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

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

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

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

The credential is:

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

The verifier is:

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

The nickname exchange yields:

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

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

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

The verifier is:

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

Using Port Monitors

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

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

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

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

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

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

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

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

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

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

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

Using inetd

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

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

where:

Table 4-12 RPC `inetd` Services

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

For example:

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

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

Using the Listener

Use `pmadm` to add RPC services:

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

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

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

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

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

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

For more information, such as how to set up the listener process, see the `listen`(1M), `pmadm`(1M), `sacadm`(1M) and `sysadm`(1M) manpages and the *Network Expansion and Setup Guide.*

Multiple Server Versions

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

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

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

Code Example 4-31 Server Handle for Two Versions of Single Routine

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

```
    fprintf(stderr, "can't register RUSER service\n");
    exit(1);
}
```

Both versions can be performed by a single procedure.

Code Example 4-32 Procedure for Two Versions of Single Routine

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

Multiple Client Versions

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

If the version on a server does not match the version number in the client creation call, clnt_call fails with an RPCPROGVERSMISMATCH error. You can get the version numbers supported by a server and then create a client handle with the appropriate version number. Use either the routine in Code Example 4-33, or clnt_create_vers(). See the rpc(3N)manpage for more details.

Code Example 4-33 RPC Versions on Client Side

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

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

    status = clnt_call(clnt, RUSERSPROC_NUM, xdr_void,
                 (caddr_t) NULL, xdr_u_short, (caddr_t)&num_s, to);
    if (status == RPC_SUCCESS) {    /* Found latest version number */
        printf("num = %d\n", num_s);
        exit(0);
    }
    if (status != RPC_PROGVERSMISMATCH) {      /* Some other error */
        clnt_perror(clnt, "rusers");
        exit(1);
    }
    /* This version not supported */
```

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

Multithreaded RPC Programming

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

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

TI-RPC supports multithreaded RPC servers in SunOS 5.4/Solaris 2.4. The difference between a multithreaded server and a single-threaded server is that a multithreaded server uses threading technology to process incoming client requests concurrently. Multithreaded servers can have higher performance and availability compared with single-threaded servers.

The section “MT Server Issues” on page 118 is a good place to start reading about the new interfaces available in this release.

MT Client Issues

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

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

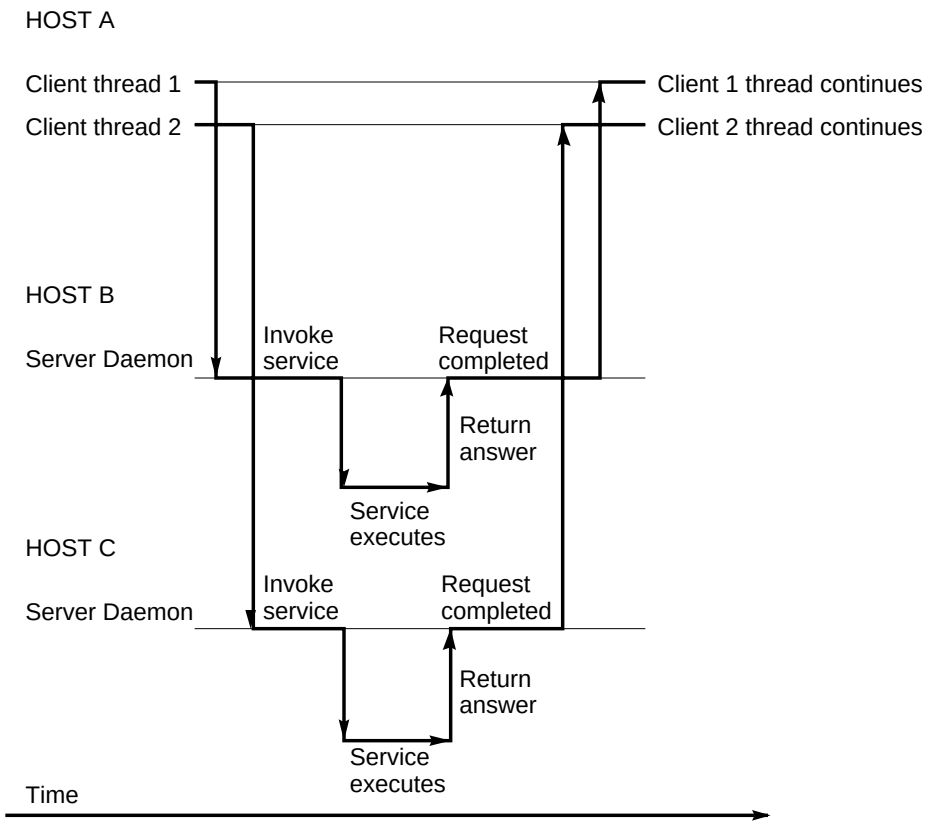


Figure 4-1 Two Client Threads Using Different Client Handles (Real time)

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

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

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

Code Example 4-34 Client for MT `rstat`

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

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

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

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

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

    mutex_lock(&tty);

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

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

```
    }

    exit(0);
}

bool_t rstatproc_stats();

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

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

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

        thr_exit(0);

    }

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

```
    thr_exit(0);

}

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

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

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

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

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

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

    hours_buf[0] = ’\0’;

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

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

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

/*
Client side implementation of MT rstat program
*/

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

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

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

    }
    return (TRUE);
}
```

MT Server Issues

Prior to SunOS 5.4, RPC servers were single threaded. That is, they process client requests sequentially, as the requests come in. For example, if two requests come in, and the first takes 30 seconds to process, and the second takes only 1 second to process, the client that made the second request will still have to wait for the first request to complete before it receives a response. This is not desirable, especially in a multiprocessor server environment, where each CPU could be processing a different request simultaneously; or in a situation where one request is waiting for I/O to complete, other requests could be processed by the server. SunOS 5.4 provides facilities in the RPC library for service developers to create servers that use multithreading to deliver better performance to end users. Two modes of server multithreading are supported in TI-RPC: the Automatic MT mode and the User MT mode.

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

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

New Calls for Server Multithreading

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

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

Making Service Procedures MT Safe

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

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

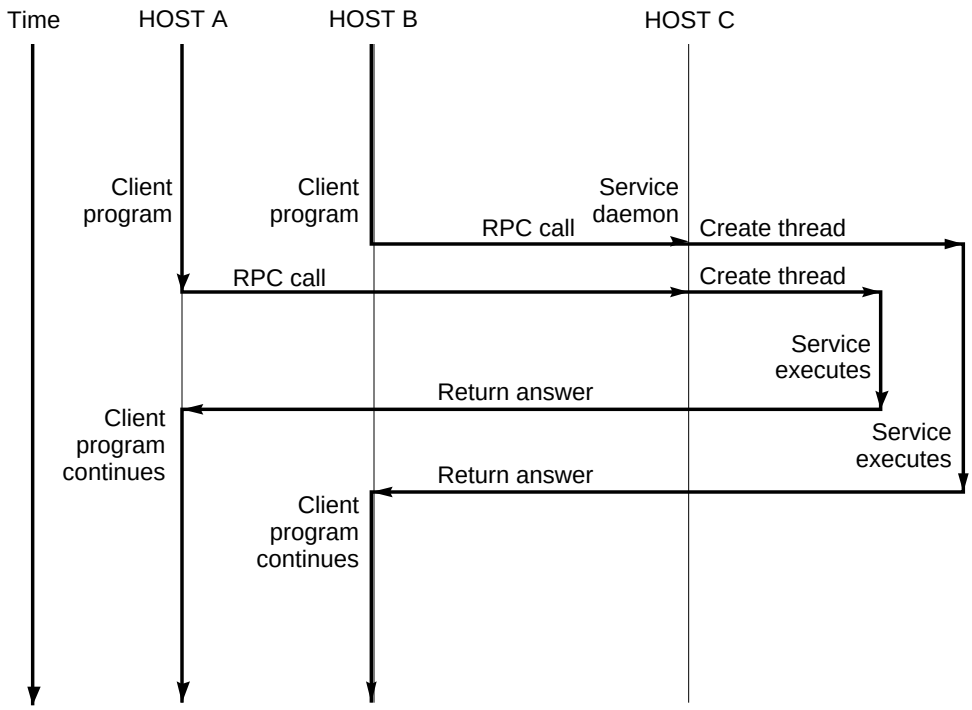


Figure 4-2 MT RPC Server Timing Diagram

Sharing the Service Transport Handle

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

MT Auto Mode

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

`rpc_control()` allows applications to set and modify global RPC attributes. At present, it supports only server-side operations. Table 4-13 shows the `rpc_control()` operations defined for Auto mode. See also the `rpc_control(3N)` manpage for additional information.

Table 4-13 `rpc_control()` Library Routines

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

Note – All of the get operations in Table 4-13, except `RPC_SVC_MTMODE_GET`, apply only to the Auto MT mode. If used in MT User mode or the single-threaded default mode, the results of the operations may be undefined.

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

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

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

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

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

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

Code Example 4-35 Server for MT Auto Mode

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

void time_prog();

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

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

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

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

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

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

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

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

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

    struct timev rslt;
    time_t thetime;

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

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

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

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

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

typedef struct timev timev;
bool_t xdr_timev();

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

MT User Mode

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

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

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

As in the Auto mode, the `rpc_control()` library call is used to turn on User mode. Note that the `rpc_control()` operations shown in Table 4-13 on page 121 (except for `RPC_SVC_MTMODE_GET`) apply only to MT Auto mode. If used in MT User mode or the single-threaded default mode, the results of the operations may be undefined.

Freeing Library Resources in User Mode

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

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

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

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

struct intpair {
    u_shorta;
    u_short b;
};

typedef struct intpair intpair;

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

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

extern bool_t xdr_intpair();

#define NTHREADS_CONST 500
```

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

Code Example 4-38 Client for MT User Mode

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

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

struct thread_info {
    thread_t client_id;
    int client_status;
};

struct thread_info save_thread[NTHREADS_CONST];

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

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

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

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

Code Example 4-39 Server for MT User Mode

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

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

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

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

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

    result = pair->a * pair->b;
    return (result);}

long add_one(arg)
    long arg;
{
    return (++arg);
}
```

Connection-Oriented Transports

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

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

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

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

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

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

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

Code Example 4-41 Remote Copy Client Routines

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

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

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

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

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

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

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

Code Example 4-42 Remote Copy Server Routines

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

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

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

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

Memory Allocation With XDR

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

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

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

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

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

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

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

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

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

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

The corresponding RPC call is:

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

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

To summarize:

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

4

Porting From TS-RPC to TI-RPC

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

This section is an informal guide to porting transport-specific RPC (TS-RPC) applications to TI-RPC. Table 4-14 shows the differences between selected routines and their counterparts. For more information on porting issues, see Socket-to-TLI Equivalents on page 216, and Moving Socket Applications to SunOS 5.x on page 259.

Porting an Application

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

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

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

Benefits of Porting

Some of the benefits of porting are:

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

Porting Issues

`libnsl` *Library*

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

Old Interfaces

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

Name-to-Address Mapping

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

4

Differences Between TI-RPC and TS-RPC

The major differences between transport-independent RPC and transport-specific RPC are illustrated in Table 4-14. Also see section “Comparison Examples” on page 144 for code examples comparing TS-RPC with TI-RPC.

Table 4-14 Differences Between TI-RPC and TS-RPC

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

Function Compatibility Lists

The RPC library functions are listed in this section and grouped into functional areas. Each section includes lists of functions that are unchanged, have added functionality, and are new relative to previous releases. Functions marked with an asterisk are retained for ease of porting and may be not be supported in future releases of Solaris.

Creating Client Handles

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

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

Creating and Destroying Services

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

```
svc_destroy
svcfd_create
*svc_raw_create
*svc_tp_create
*svcudp_create
*svc_udp_bufcreate
svc_create
svc_dg_create
svc_fd_create
svc_raw_create
```

```
svc_tli_create
svc_tp_create
svc_vc_create
```

The new functions for the SunOS 5.4 release are:

```
svc_done
rpc_control
```

Registering and Unregistering Services

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

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

SunOS 4.x Compatibility Calls

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

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

Broadcasting

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

```
*clnt_broadcast
```

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

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

`rpc_broadcast`

Address Management Functions

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

The following functions work with `portmap`:

```
pmap_set
pmap_unset
pmap_getport
pmap_getmaps
pmap_rmtcall
```

The following functions work with `rpcbind`:

```
rpcb_set
rpcb_unset
rpcb_getaddr
rpcb_getmaps
rpcb_rmtcall
```

Authentication Functions

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

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

Other Functions

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

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

Comparison Examples

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

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

Code Example 4-43 Client Creation in TS-RPC

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

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

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

Code Example 4-44 Client Creation in TI-RPC

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

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

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

Code Example 4-45 Broadcast in TS-RPC

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

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

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

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

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

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

Code Example 4-46 Broadcast in TI-RPC

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

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

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

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

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

4

Part 3 — Transport Level

Transport Selection and Name-to-Address Mapping 5

This chapter covers selecting transports and resolving network addresses. It describes interfaces that enable you to specify the available communication protocols for an application. The chapter also describes additional functions that provide direct mapping of names to network addresses.

Note – In this chapter the terms *network* and *transport* are used interchangeably to refer to the programmatic interface that conforms to the transport layer of the OSI Reference Model. In the OSI model, the network layer is one below the transport layer. The OSI model is briefly described in Chapter 1, “Introduction to Network Interfaces.” The term *network* is also used to refer to the physical collection of computers connected through some electronic medium.

Transport Selection Is Multithread Safe

The interface described in this chapter is multithread safe. This means that applications that contain transport selection function calls can be used freely in a multithreaded application.

5

Transport Selection

A distributed application must use a standard interface to the transport services if it is to be portable to different protocols. Transport selection services provide an interface that allows an application to select which protocols to use. This makes an application "protocol" and "medium independent."

Transport selection makes it easy for a client application to try each available transport until it establishes communication with a server. Transport selection lets server applications accept requests on multiple transports, and in doing so, communicate over a number of protocols. Transports may be tried in either the order specified by the local default sequence or in an order specified by the user.

Choosing from the available transports is the responsibility of the application. The transport selection mechanism makes that selection uniform and simple.

How Transport Selection Works

The transport selection component is built around:

- A network configuration database (the `/etc/netconfig` file), which contains an entry for each network on the system
- Optional use of the `NETPATH` environment variable

The `NETPATH` variable is set by the user; it contains an ordered list of transport identifiers. The transport identifiers match the `netconfig` network ID field and are links to records in the `netconfig` file. The `netconfig` file is described in "/etc/netconfig File" on page 153. The network selection interface is a set of access routines for the network-configuration database.

One set of library routines accesses only the `/etc/netconfig` entries identified by the `NETPATH` environment variable:

```
setnetpath()
getnetpath()
endnetpath()
```

They are described in "NETPATH Access to netconfig Data" on page 156 and in the `getnetpath(3N)` man page. These routines let the user influence the selection of transports used by the application.

To avoid user influence on transport selection, use the routines that access the `netconfig` database directly. These routines are described in “Accessing netconfig” on page 158 and in the `getnetconfig`(3N) manpage.

```
setnetconfig()
getnetconfig()
endnetconfig()
```

The following two routines manipulate `netconfig` entities and the data structures they represent.

```
getnetconfigent()
freenetconfigent()
```

`/etc/netconfig` *File*

The `netconfig` file describes all network transport protocols on a host. The entries in the `netconfig` file are explained briefly in Table 5-1 and in more detail in the `netconfig(4)` manpage.

Table 5-1 The `netconfig` File

Entries	Description
network ID	A local representation of a transport name (such as `tcp`). Do not assume that this field contains a well-known name (such as `tcp` or `udp`) or that two systems use the same name for the same transport.
semantics	The semantics of the particular transport protocol. Valid semantics are: `tpi_clts` – connectionless `tpi_cots` – connection oriented `tpi_cots_ord` – connection oriented with orderly release
flags	May take only the values, `v`, or hyphen (`-`). Only the visible flag (`-v`) is defined.
protocol family	The protocol family name of the transport provider (for example, `inet` or `loopback`).

Table 5-1 The `netconfig` File

Entries	Description
protocol name	The protocol name of the transport provider. For example, if *protocol family* is `inet`, then *protocol name* is `tcp`, `udp`, or `icmp`. Otherwise, the value of *protocol name* is a hyphen (`-`).
network device	The full path name of the device file to open when accessing the transport provider.
name-to-address translation libraries	Names of the shared objects. This field contains the comma-separated file names of the shared objects that contain name-to-address mapping routines. Shared objects are located through the path in the `LD_LIBRARY_PATH` variable. A "-" in this field indicates the absence of any translation libraries.

Code Example 5-1 shows a sample `netconfig` file. Use of the `netconfig` file has been changed for the `inet` transports, as described in the commented section in the sample file. This change is also described in “Name-to-Address Mapping” on page 160.

Code Example 5-1 Sample `netconfig` File

```
# The “Network Configuration” File.
#
# Each entry is of the form:
#
#<net <semantics>  <flags> <proto    <proto   <device>    <nametoaddr_libs>
# id>                       family>   name>
#
# The "-" in <nametoaddr_libs> for inet family transports indicates redirection
# to the name service switch policies for "hosts" and "services. The "-" may be
# replaced by nametoaddr libraries that comply with the SVR4 specs, in which
# case the name service switch will be be used for netdir_getbyname, netdir_
# get byaddr, gethostbyname, gethostbyaddr, getservbyname, and getservbyport.
# There are no nametoaddr_libs for the inet family in Solaris anymore.
#
udp    tpi_clts     v       inet      udp      /dev/udp      -
#
tcp    tpi_cots_ord v       inet      tcp      /dev/tcp      -
#
icmp   tpi_raw      -       inet      icmp     /dev/icmp     -
#
rawip  tpi_raw      -       inet      -        /dev/rawip    -
```

```
#
ticlts tpi_clts     v      loopback  -        /dev/ticlts    straddr.so
#
ticots tpi_cots     v      loopback  -        /dev/ticots    straddr.so
#
ticotsord tpi_cots_ord v   loopback  -        /dev/ticotsord straddr.so
#
```

Network selection library routines return pointers to `netconfig` entries. The `netconfig` structure is shown in Code Example 5-2.

Code Example 5-2 The `netconfig` Structure

```
struct netconfig {
    char          *nc_netid;    /* network identifier */
    unsigned long nc_semantics; /* semantics of protocol */
    unsigned long nc_flag;      /* flags for the protocol */
    unsigned long nc_protofmly; /* family name */
    unsigned long nc_proto;     /* proto specific */
    char          *nc_device;   /* device name for network id */
    unsigned long nc_nlookups;  /* # entries in nc_lookups */
    char          **nc_lookups; /* list of lookup libraries */
    unsigned long nc_unused[8];
};
```

Valid network IDs are defined by the system administrator, who must ensure that network IDs are locally unique. If they are not, some network selection routines can fail. For example, it is not possible to know which network `getnetconfigent(udp)` will use if there are two `netconfig` entries with the network ID `udp`.

The system administrator also sets the order of the entries in the `netconfig` database. The routines that find entries in `/etc/netconfig` return them in order, from the beginning of the file. The order of transports in the `netconfig` file is the default transport search sequence of the routines. Loopback entries should be at the end of the file.

The `netconfig` file and the `netconfig` structure are described in greater detail on the `netconfig(4)` manpage.

5

The NETPATH *Environment Variable*

An application usually uses the default transport search path set by the system administrator to locate an available transport. However, when a user wants to influence the choices made by an application, the application can modify the interface by using the environment variable NETPATH and the routines described in the section, "NETPATH Access to netconfig Data." These routines access only the transports specified in the NETPATH variable.

NETPATH is similar to the PATH variable. It is a colon-separated list of transport IDs. Each transport ID in the NETPATH variable corresponds to the network ID field of a record in the netconfig file. NETPATH is described on the environ(5) manpage.

The default transport set is different for the routines that access netconfig through the NETPATH environment variable (described in the next section) and the routines that access netconfig directly. The default transport set for routines that access netconfig via NETPATH consists of the visible transports in the netconfig file. For routines that access netconfig directly, the default transport set is the entire netconfig file. A transport is visible if the system administrator has included a v flag in the flags field of that transport's netconfig entry.

NETPATH *Access to* netconfig *Data*

Three routines access the network configuration database indirectly through the NETPATH environment variable. The variable specifies the transport or transports an application is to use and the order to try them. NETPATH components are read from left to right. The functions have the following interfaces:

```
#include <netconfig.h>

void *setnetpath(void);
struct netconfig *getnetpath(void *);
int endnetpath(void *);
```

A call to setnetpath()initializes the search of NETPATH. It returns a pointer to a database that contains the entries specified in a NETPATH variable. The pointer, called a handle, is used to traverse this database with getnetpath(). setnetpath() must be called before the first call to getnetpath().

When first called, `getnetpath()` returns a pointer to the `netconfig` file entry that corresponds to the first component of the `NETPATH` variable. On each subsequent call, `getnetpath()` returns a pointer to the `netconfig` entry that corresponds to the next component of the `NETPATH` variable. `getnetpath()` returns `NULL` if there are no more components in `NETPATH`. A call to `getnetpath()` without an initial call to `setnetpath()` causes an error. `getnetpath()` requires the pointer returned by `setnetpath()` as an argument.

`getnetpath()` silently ignores invalid `NETPATH` components. A `NETPATH` component is invalid if there is no corresponding entry in the `netconfig` database.

If the `NETPATH` variable is unset, `getnetpath()` behaves as if `NETPATH` were set to the sequence of default or visible transports in the `netconfig` database, in the order in which they are listed.

`endnetpath()` is called to release the database pointer to elements in the `NETPATH` variable when processing is complete. `endnetpath()` fails if `setnetpath()` was not called previously. Code Example 5-3 shows the `setnetpath()`, `getnetpath()`, and `endnetpath()` routines.

Code Example 5-3 `setnetpath()`, `getnetpath()`, and `endnetpath()`

```
#include <netconfig.h>

void *handlep;
struct netconfig *nconf;

if ((handlep = setnetpath()) == (void *)NULL) {
    nc_perror(argv[0]);
    exit(1);
}
while ((nconf = getnetpath(handlep)) != (struct netconfig *)NULL) {
    /*
     * nconf now describes a transport provider.
     */
}
endnetpath(handlep);
```

The `netconfig` structures obtained through `getnetpath()` become invalid after the execution of a `endnetpath()`. To preserve the data in the structure, use `getnetconfigent(nconf->nc_netid)` to copy them into a new data structure.

5

Accessing `netconfig`

Three functions access `/etc/netconfig` and locate `netconfig` entries. The routines `setnetconfig`, `getnetconfig`, and `endnetconfig` have the following interfaces:

```
#include <netconfig.h>

void *setnetconfig(void);
struct netconfig *getnetconfig(void *);
int endnetconfig(void *);
```

A call to `setnetconfig`() initializes the record pointer to the first index in the database. `setnetconfig`() must be used before the first use of `getnetconfig`(). `setnetconfig`() returns a unique handle (a pointer into the database) to be used by the `getnetconfig`() routine. Each call to `getnetconfig`() returns the pointer to the current record in the `netconfig` database and increments its pointer to the next record. It can be used to search the entire `netconfig` database. `getnetconfig`() returns a `NULL` at the end of file.

You must use `endnetconfig`() to release the database pointer when processing is complete. `endnetconfig`() must not be called before `setnetconfig`().

Code Example 5-4 `setnetconfig()`, `getnetconfig()`, and `endnetconfig()`

```
void *handlep;
struct netconfig *nconf;

if ((handlep = setnetconfig()) == (void *)NULL){
    nc_perror(argv[0]);
    exit(1);
}
/*
 * transport provider information is described in nconf.
 * process_transport is a user-supplied routine that
 * tries to connect to a server over transport nconf.
 */
while ((nconf = getnetconfig(handlep)) != (struct netconfig *)NULL){
    if (process_transport(nconf) == SUCCESS){
        break;
    }
}
endnetconfig(handlep);
```

The last two functions have the following interface:

```
#include <netconfig.h>
struct netconfig *getnetconfigent(char *);
int freenetconfigent(struct netconfig *);
```

`getnetconfigent`() returns a pointer to the `struct netconfig` structure corresponding to `netid`. It returns `NULL` if `netid` is invalid. `setnetconfig`() need not be called before `getnetconfigent`().

`freenetconfigent`() frees the structure returned by `getnetconfigent`(). Code Example 5-5 shows the `getnetconfigent()` and `freenetconfigent()` routines.

Code Example 5-5 `getnetconfigent()` and `freenetconfigent()`

```
/* assume udp is a netid on this host */
struct netconfig *nconf;

if ((nconf = getnetconfigent("udp")) == (struct netconfig *)NULL){
    nc_perror("no information about udp");
    exit(1);
}
process_transport(nconf);
freenetconfigent(nconf);
```

Loop Through All Visible `netconfig` *Entries*

The `setnetconfig()` call is used to step through all the transports marked *visible* in the `netconfig` database. The transport selection routine returns a `netconfig` pointer.

Use `getnetpath()` and `setnetpath()` to obtain or modify the network path variable. Code Example 5-6 shows the form and use, which are similar to the `getnetconfig` routines.

Code Example 5-6 Looping Through Visible Transports

```
void *handlep;
struct netconfig *nconf;

if ((handlep = setnetconfig() == (void *) NULL) {
    nc_perror("setnetconfig");
    exit(1);
}
```

```
while (nconf = getnetconfig(handlep))
    if (nconf->nc_flag & NC_VISIBLE)
        doit(nconf);
(void) endnetconfig(handlep);
```

Looping Through User-Defined `netconfig` *Entries*

Users can control the loop by setting the `NETPATH` environment variable to a colon-separated list of transport names. If `NETPATH` is set as follows:

```
NETPATH=tcp:udp
```

The loop first returns the `tcp` entry, and then the `udp` entry. If `NETPATH` is not defined, the loop returns all visible entries in the `netconfig` file in the order in which they are stored. The `NETPATH` environment variable lets users define the order in which client-side applications try to connect to a service. It also lets the server administrator limit transports on which a service can listen.

Name-to-Address Mapping

Name-to-address mapping lets an application obtain the address of a service on a specified host, independent of the transport used. Name-to-address mapping consists of the following functions:

```
netdir_getbyname
netdir_getbyaddr
netdir_free
taddr2uaddr
uaddr2taddr
netdir_options
```

The first argument of each routine points to a `netconfig` structure that describes a transport. The routine uses the array of directory lookup library paths in the `netconfig` structure to call each path until the translation succeeds.

The libraries are described in Table 5-2 on page 161. The routines described in the section, “Using the Name-to-Address Mapping Routines,” are in the `netdir(3N)` manpage.

Note – The following libraries no longer exist in Solaris 2.X: `tcpip.so`, `switch.so`, and `nis.so`. For more information on this change, see the `nsswitch.conf(4)`manpage and the NOTES section of the `gethostbyname(3N)` manpage.

Table 5-2 Name-to-Address Libraries

Library	Transport Family	Description
-	`inet`	For networks of the protocol family `inet`, its name-to-address mapping is provided by the name service switch based on the entries for *hosts* and *services* in the file `nsswitch.conf`. For networks of other families, the "-" indicates a non-functional name-to-address mapping.
`straddr.so`	loopback	Contains the name-to-address mapping routines of any protocol that accepts strings as addresses, such as the loopback transports.

`straddr.so` *Library*

Files for the library are created and maintained by the system administrator. The `straddr.so` files are `/etc/net/`*transport-name*`/hosts` and `/etc/net/`*transport-name*`/services`. *transport-name* is the local name of the transport that accepts string addresses (specified in the *network ID* field of the `/etc/netconfig` file). For example, the host file for `ticlts` would be `/etc/net/ticlts/hosts`, and the service file for `ticlts` would be `/etc/net/ticlts/services`.

Even though most string addresses do not distinguish between *host* and *service*, separating the string into a host part and a service part is consistent with other transports. The `/etc/net/`*transport-name*`/hosts` file contains a text string that is assumed to be the host address, followed by the host name. For example:

```
joyluckaddr        joyluck
carpediemaddr      carpediem
thehopaddr         thehop
pongoaddr          pongo
```

For loopback transports, it makes no sense to list other hosts because the service cannot go outside the containing host.

The /etc/net/*transport-name*/services file contains service names followed by strings identifying the service address. For example:

```
rpcbind         rpc
listen          serve
```

The routines create the full string address by concatenating the host address, a period (.), and the service address. For example, the address of the listen service on pongo is pongoaddr.serve.

When an application requests the address of a service on a particular host on a transport that uses this library, the host name must be in /etc/net/*transport*/hosts and the service name must be in /etc/net/*transport*/services. If either is missing, the name-to-address translation fails.

Using the Name-to-Address Mapping Routines

This section provides an overview of what routines are available to use. The routines return or convert the network names to their respective network addresses. Note that netdir_getbyname(), netdir_getbyaddr(), and taddr2uaddr() return pointers to data that must be freed by calls to netdir_free().

```
int netdir_getbyname(struct netconfig *nconf,
                     struct nd_hostserv *service,
                     struct nd_addrlist **addrs);
```

maps the host and service name specified in *service* to a set of addresses consistent with the transport identified in *nconf*. The nd_hostserv and nd_addrlist structures are defined in netdir(3N). A pointer to the addresses is returned in *addrs*.

To find all addresses of a host and service (on all available transports), call netdir_getbyname() with each netconfig structure returned by either getnetpath(3N) or getnetconfig(3N).

```
int netdir_getbyaddr(struct netconfig *nconf,
                     struct nd_hostservlist **service,
                     struct netbuf *netaddr);
```

maps addresses into host and service names. The function is called with an address in *netaddr* and returns a list of host name and service name pairs in *service*. The nd_hostservlist structure is defined in netdir(3N).

```
void netdir_free(void *ptr, int struct_type);
```

The `netdir_free()` routine frees structures allocated by the name-to-address translation routines. The parameters can take the values shown in Table 5-4.

Table 5-3 `netdir_free()` Routines

struct_type	***ptr***
`ND_HOSTSERV`	pointer to an `nd_hostserv` structure
`ND_HOSTSERVLIST`	pointer to an `nd_hostservlist` structure
`ND_ADDR`	pointer to a `netbuf` structure
`ND_ADDRLIST`	pointer to an `nd_addrlist` structure

```
char *taddr2uaddr(struct netconfig *nconf, struct netbuf *addr);
```

translates the address pointed to by *addr a*nd returns a transport independent character representation of the address (“universal address”). *nconf s*pecifies the transport for which the address is valid. The universal address can be freed by `free`().

```
struct netbuf *uaddr2taddr(struct netconfig *nconf, char *uaddr);
```

the “universal address” pointed to by *uaddr* is translated into a `netbuf` structure. *nconf* specifies the transport for which the address is valid.

```
int netdir_options(struct netconfig *nconf, int option, int fd,
                   char *point_to_args);
```

interfaces to transport-specific capabilities (such as the broadcast address and reserved port facilities of TCP and UDP). *nconf* specifies a transport. o*ption* specifies the transport-specific action to take. *fd* may or may not be used depending upon the value of *option*. The fourth argument points to operation-specific data.

Table 5-4 shows the values used for *option*:

Table 5-4 Values for `netdir_options`

option	**Description**
`ND_SET_BROADCAST`	Sets the transport for broadcast (if the transport supports broadcast)
`ND_SET_RESERVEDPORT`	Lets the application bind to a reserved port (if allowed by the transport)
`ND_CHECK_RESERVEDPORT`	Verifies that an address corresponds to a reserved port (if the transport supports reserved ports)
`ND_MERGEADDR`	Transforms a locally meaningful address into an address that client hosts can connect to

```
void netdir_perror(char *s);
```

displays the message stating why one of the name-to-address mapping routines failed on `stderr`.

```
char *netdir_sperror(void);
```

returns a string containing the error message stating why one of the name-to-address mapping routines failed.

Code Example 5-7 shows network selection and name-to-address mapping.

Code Example 5-7 Network Selection and Name-to-Address Mapping

```
#include <netconfig.h>
#include <netdir.h>
#include <sys/tiuser.h>

struct nd_hostserv nd_hostserv;   /* host and service information */
struct nd_addrlist *nd_addrlistp; /* addresses for the service */
struct netbuf *netbufp;           /* the address of the service */
struct netconfig *nconf;          /* transport information*/
int i;                            /* the number of addresses */
char *uaddr;                      /* service universal address */
void *handlep;                 /* a handle into network selection */
/*
 * Set the host structure to reference the "date"
 * service on host "gandalf"
 */
nd_hostserv.h_host = "gandalf";
```

```
nd_hostserv.h_serv = "date";
/*
 * Initialize the network selection mechanism.
 */
if ((handlep = setnetpath()) == (void *)NULL) {
    nc_perror(argv[0]);
    exit(1);
}
/*
 * Loop through the transport providers.
 */
while ((nconf = getnetpath(handlep)) != (struct netconfig *)NULL) {
    /*
     * Print out the information associated with the
     * transport provider described in the "netconfig"
     * structure.
     */
    printf("Transport provider name: %s\n", nconf->nc_netid);
    printf("Transport protocol family: %s\n", nconf->nc_protofmly);
    printf("The transport device file: %s\n", nconf->nc_device);
    printf("Transport provider semantics: ");
    switch (nconf->nc_semantics) {
        case NC_TPI_COTS:
            printf("virtual circuit\n");
            break;
        case NC_TPI_COTS_ORD:
            printf("virtual circuit with orderly release\n");
            break;

        case NC_TPI_CLTS:
            printf("datagram\n");
            break;
    }
    /*
     * Get the address for service "date" on the host
     * named "gandalf" over the transport provider
     * specified in the netconfig structure.
     */
    if (netdir_getbyname(nconf, &nd_hostserv, &nd_addrlistp)
            != ND_OK) {
        printf("Cannot determine address for service\n");
        netdir_perror(argv[0]);
        continue;
    }
    printf("<%d> address of date service on gandalf:\n",
            nd_addrlistp->n_cnt);
```

```
        /*
         * Print out all addresses for service "date" on
         * host "gandalf" on current transport provider.
         */
        netbufp = nd_addrlistp->n_addrs;
        for (i = 0; i < nd_addrlistp->n_cnt; i++, netbufp++) {
            uaddr = taddr2uaddr(nconf,netbufp);
            printf("%s\n",uaddr);
            free(uaddr);
        }
        netdir_free( nd_addrlistp, ND_ADDRLIST );
    }
    endnetconfig(handlep);
```

Portability From Previous Releases

The list that follows contains the names of the functions with functionality unchanged from earlier releases.

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

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

Other porting issues are described in “Porting From TS-RPC to TI-RPC” on page 138.”

Transport-level Interface (TLI) Programming Guide 6

The transport-level interface (TLI) is a set of functions that enable networked applications to be transport independent. TLI and the socket interface are the standard network interfaces in the current release of SunOS. TLI may be more appropriate for some applications than are the socket services.

TLI Is Multithread Safe

The interface described in this chapter is multithread safe. This means that applications that contain TLI function calls can be used freely in a multithreaded application.

6

Transport Interface Overview

A transport provider performs services, and the transport user requests the services. The transport user issues service requests to the transport provider. An example is a request to transfer data over a connection. See Figure 6-1 on page 168. The transport control protocol (TCP) is an example of a transport provider. A transport user may be a networking application or a session layer protocol.

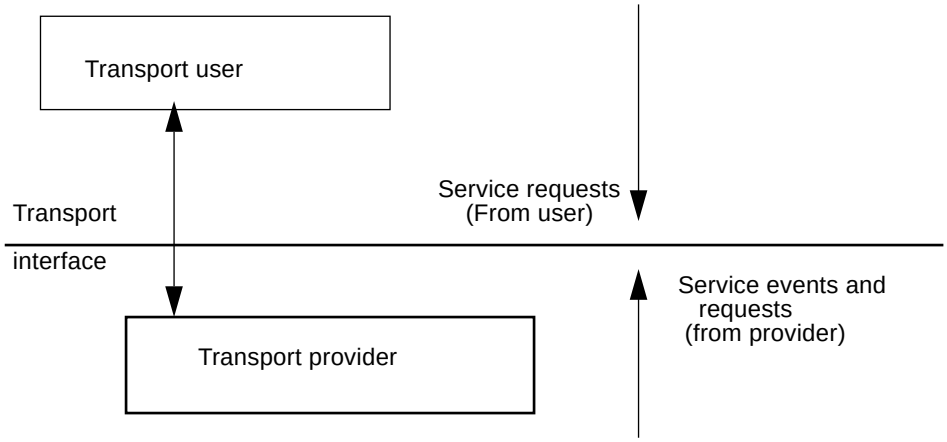


Figure 6-1 How TLI Works

TLI has two components:

- Library routines that perform the transport services. The network services library includes a set of functions that implement TLI for user processes. See the `intro`(3] manpage. Programs using TLI link with the `lnsl` library as follows:

  ```
  cc prog.c -lnsl
  ```

- State transition rules that define the sequence in which the transport routines may be invoked

For more information on state transition rules, see section, “State Transitions” on page 208. The state tables define the legal sequence of library calls based on the state and the handling of events. These events include user-generated library calls, as well as provider-generated event indications. TLI programmers should understand all state transitions before using the interface.

TLI provides two modes of service, connection mode and connectionless mode. The next two sections give an overview of these modes.

Connectionless Mode Overview

Connectionless mode is message oriented. Data are transferred in self-contained units with no relationship between the units. This service requires only an established association between the peer users that determines the characteristics of the data. All the information required to deliver a message (such as the destination address) is presented to the transport provider, with the data to be transmitted, in one service request. Each message is entirely self-contained. Use connectionless mode service for applications that:

- Have short-term request/response interactions
- Exhibit a high level of redundancy
- Are dynamically reconfigurable
- Do not require sequential delivery of data

Connectionless transports are usually called “unreliable,” since they do not necessarily maintain message sequence.

Connectionless Mode Routines

Connectionless-mode transport service has two phases: local management and data transfer. The local management phase defines the same local operations as for the connection mode service.

The data transfer phase lets a user transfer data units (usually called datagrams) to the specified peer user. Each data unit must be accompanied by the transport address of the destination user. `t_sndudata`() sends and `t_rcvudata`() receives messages. Table 6-1 summarizes all routines for connectionless mode data transfer.

6

Table 6-1 Routines for Connectionless-Mode Data Transfer

Command	***Description***
`t_sndudata`	Sends a message to another user of the transport
`t_rcvudata`	Receives a message sent by another user of the transport
`t_rcvuderr`	Retrieves error information associated with a previously sent message

Connection Mode Overview

Connection mode is circuit oriented. Data are transmitted in sequence over an established connection. The mode also provides an identification procedure that avoids address resolution and transmission in the data transfer phase. Use this service for applications that require data stream-oriented interactions. Connection mode transport service has four phases:

- Local management
- Connection establishment
- Data transfer
- Connection release

The local management phase defines local operations between a transport user and a transport provider. For example, a user must establish a channel of communication with the transport provider. Each channel between a transport user and transport provider is a unique endpoint of communication, and is called the transport endpoint. `t_open`() lets a user choose a particular transport provider to supply the connection mode services, and establishes the transport endpoint.

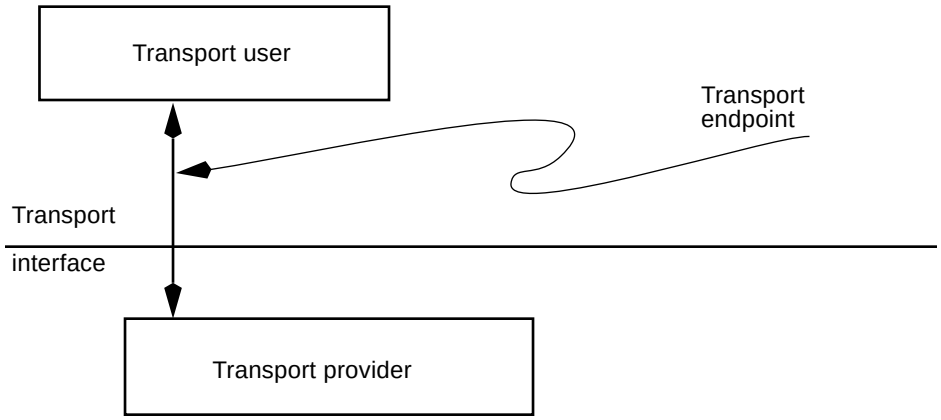


Figure 6-2 Transport Endpoint

Connection Mode Routines

Each user must establish an identity with the transport provider. A transport address is associated with each transport endpoint. One user process may manage several transport endpoints. In connection mode service, one user requests a connection to another user by specifying the other's address. The structure of a transport address is defined by the transport provider. An address may be as simple as an unstructured character string (for example, `file_server`), or as complex as an encoded bit pattern that specifies all information needed to route data through a network. Each transport provider defines its own mechanism for identifying users. Addresses may be assigned to the endpoint of a transport by `t_bind`().

In addition to `t_open` and `t_bind`, several routines support local operations. The following table summarizes all local management routines of TLI.

Table 6-2 Endpoint Establishment Routines of TLI

Command	Description
`t_alloc`	Allocates TLI data structures
`t_bind`	Binds a transport address to a transport endpoint
`t_close`	Closes a transport endpoint

Table 6-2 Endpoint Establishment Routines of TLI

Command	Description
`t_error`	Prints a TLI error message
`t_free`	Frees structures allocated using `t_alloc`
`t_getinfo`	Returns a set of parameters associated with a particular transport provider
`t_getstate`	Returns the state of a transport endpoint
`t_look`	Returns the current event on a transport endpoint
`t_open`	Establishes a transport endpoint connected to a chosen transport provider
`t_optmgmt`	Negotiates protocol-specific options with the transport provider
`t_sync`	Synchronizes a transport endpoint with the transport provider
`t_unbind`	Unbinds a transport address from a transport endpoint

Many TLI routines have equivalents in the socket service library. TLI routines with the same or similar function as the corresponding socket routine have the same name with `t_` prefixed.

The connection phase lets two users create a connection, or virtual circuit, between them, as shown in Figure 6-3.

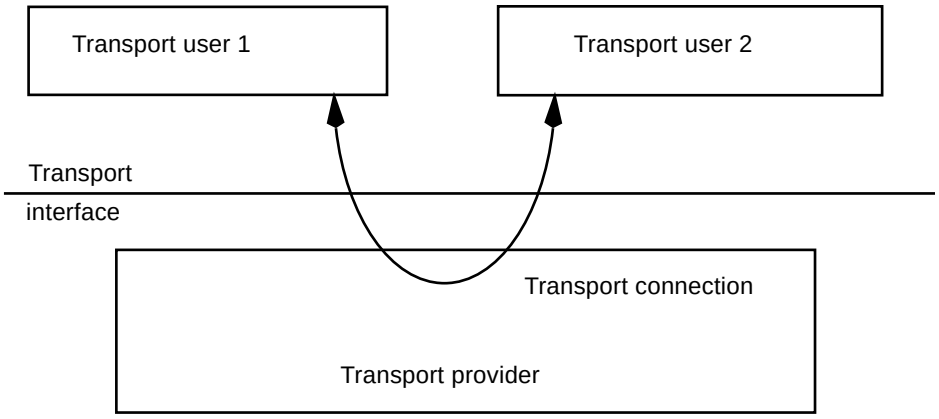


Figure 6-3 Transport Connection

For example, the connection phase occurs when a server advertises its service to a group of clients, and then blocks on `t_listen`() to wait for a request. A client tries to connect to the server at the advertised address by a call to `t_connect`(). The connection request causes `t_listen`() to return to the server, which can call `t_accept`() to complete the connection.

Table 6-3 summarizes all routines available for establishing a transport connection. Refer to Section 3N of the *man Pages(3): Library Routines* for the specifications on these routines.

Table 6-3 Routines for Establishing a Transport Connection

Command	Description
`t_accept`	Accepts a request for a transport connection
`t_connect`	Establishes a connection with the transport user at a specified destination
`t_listen`	Listens for connect request from another transport user
`t_rcvconnect`	Completes connection establishment if `t_connect` was called in asynchronous mode (see “Advanced Topics” on page 201)

The data transfer phase lets users transfer data in both directions via the connection. `t_snd`() sends and `t_rcv`() receives data through the connection. It is assumed that all data sent by one user is guaranteed to be delivered to the other user in the order in which it was sent. Table 6-4 summarizes the connection mode data transfer routines.

Table 6-4 Connection Mode Data Transfer Routines

Command	Description
`t_rcv`	Receives data that has arrived over a transport connection
`t_snd`	Sends data over an established transport connection

TLI has two types of connection release. The abortive release directs the transport provider to release the connection immediately. Any previously sent data that has not yet been transmitted to the other user may be discarded by the transport provider. `t_snddis`() initiates the abortive disconnect. `t_rcvdis`() cleans up after an abortive disconnect. All transport providers must support the abortive release procedure.

Transport providers may also support an orderly release that terminates communication without discarding data. `t_sndrel`() and `t_rcvrel`() perform this function. Table 6-5 summarizes the connection release routines. Refer to Section 3N of the *man Pages(3): Library Routines* for the specifications on these routines.

Table 6-5 Connection Release Routines

Command	Description
`t_rcvdis`	Returns a reason code for a disconnection and any remaining user data
`t_rcvrel`	Acknowledges receipt of an orderly release of a connection request
`t_snddis`	Aborts a connection or rejects a connect request
`t_sndrel`	Requests the orderly release of a connection

Connection Mode Service

The main concepts of connection mode service are illustrated through a client program and its server. The examples are presented in segments.

In the examples, the client establishes a connection to a server process. The server transfers a file to the client. The client receives the file contents and writes them to standard output.

Endpoint Initiation

Before a client and server can connect, each must first open a local connection to the transport provider (the transport endpoint) through `t_open`, and establish its identity (or address) through `t_bind`.

Many protocols perform a subset of the services defined in TLI. Each transport provider has characteristics that determine the services it provides and limit the services. Data defining the transport characteristics are returned by `t_open` in a `t_info` structure. Table 6-6 shows the fields in a `t_info` structure.

Table 6-6 `t_info` Structure

Field	Content
`addr`	Maximum size of a transport address
`options`	Maximum bytes of protocol-specific options that may be passed between the transport user and transport provider
`tsdu`	Maximum message size that may be transmitted in either connection mode or connectionless mode
`etsdu`	Maximum expedited data message size that may be sent over a transport connection
`connect`	Maximum number of bytes of user data that may be passed between users during connection establishment
`discon`	Maximum bytes of user data that may be passed between users during the abortive release of a connection
`servtype`	The type of service supported by the transport provider

The three service types defined by TLI are:

1. T_COTS
   The transport provider supports connection mode service but does not provide the optional orderly release facility.

2. T_COTS_ORD
   The transport provider supports connection mode service with the optional orderly release facility.
3. T_CLTS
   The transport provider supports connectionless mode service.

Only one such service can be associated with the transport provider identified by `t_open`.

`t_open` returns the default provider characteristics of a transport endpoint. Some characteristics may change after an endpoint has been opened. This happens with negotiated options (option negotiation is described later in this section). `t_getinfo` returns the current characteristics of a transport endpoint.

Once a user establishes an endpoint with the chosen transport provider, the client and server must establish their identities. `t_bind` does this by binding a transport address to the transport endpoint. For servers, this routine informs the transport provider that the endpoint is used to listen for incoming connect requests.

`t_optmgmt`() can be used during the local management phase. It lets a user negotiate the values of protocol options with the transport provider. Each transport protocol defines its own set of negotiable protocol options, such as quality-of-service parameters. Because the options are protocol-specific, only applications written for a specific protocol use this function.

Client

The local management requirements of the example client and server are used to discuss details of these facilities. Code Example 6-1 shows the definitions needed by the client program, followed by its necessary local management steps.

Code Example 6-1 Client Implementation of Open and Bind

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>
#define SRV_ADDR 1                          /* server's address */

main()
```

```
{
    int fd;
    int nbytes;
    int flags = 0;
    char buf[1024];
    struct t_call *sndcall;
    extern int t_errno;

    if ((fd = t_open("/dev/exmp", O_RDWR, (struct t_info *), NULL))
            == -1) {
        t_error("t_open failed");
        exit(1);
    }
    if (t_bind(fd, (struct t_bind *) NULL, (struct t_bind *) NULL)
              == -1) {
        t_error("t_bind failed");
        exit(2);
    }
```

The first argument of `t_open` is the path of a file system object that identifies the transport protocol. `/dev/exmp` is the example name of a `clone` device special file that identifies a generic, connection-based transport protocol. It must be created on the workstation for this purpose. The `clone` device finds an available minor device of the transport provider for the user. The second argument, `O_RDWR`, specifies to open for both reading and writing. The third argument points to a t_info structure in which to return the service characteristics of the transport. This data is useful to write protocol-independent software (see “Guidelines to Protocol Independence” on page 213). In this example a `NULL` pointer is passed. The transport provider should have the following characteristics:

- The transport address is an integer value that uniquely identifies each user.
- The transport provider supports the `T_COTS_ORD` service type, and the example uses orderly release.
- User data is not passed between users during connection establishment and abortive release.
- The transport provider does not support protocol-specific options.

If the user needs a service other than `T_COTS_ORD`, another transport provider can be opened. An example of the `T_CLTS` service invocation is shown in the section “Connectionless Mode Service” on page 192.

t_open returns the transport endpoint file handle that is used by all subsequent TLI function calls. The identifier is a file descriptor from opening the transport protocol file. See the open(2)manpage.

The client then calls t_bind to assign an address to the endpoint. The first argument of t_bind is the transport endpoint handle. The second argument points to a t_bind structure that describes the address to bind to the endpoint. The third argument points to a t_bind structure that describes the address that the provider bound.

The address of a client is rarely important, because no other process tries to access it. That is why the second and third arguments to t_bind are NULL. The NULL second argument directs the transport provider to choose an address for the user.

If t_open or t_bind fails, the program calls t_error() to display an appropriate error message via stderr. The global integer t_errno is assigned an error value. A set of error values is defined in <tiuser.h>. t_error is analogous to perror(). If the transport function error is a system error, t_errno is set to TSYSERR, and errno is set to the appropriate value.

Server

The server example must also establish a transport endpoint at which to listen for connection requests. Code Example 6-2 shows the definitions and local management steps.

Code Example 6-2 Server Implementation of Open and Bind

```
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>

#define DISCONNECT -1
#define SRV_ADDR 1                          /* server's address */
int conn_fd;/* connection established here */
extern int t_errno;

main()
{
    int listen_fd;              /* listening transport endpoint */
    struct t_bind *bind;
```

```
    struct t_call *call;

    if ((listen_fd = t_open/dev/exmp”, O_RDWR,
                              (struct t_info *) NULL)) == -1) {
        t_error(“t_open failed for listen_fd”);
        exit(1);
    }
    /*
     * Because it assumes the format of the provider’s address,
     * this program is transport-dependent
     */
    if ((bind = (struct t_bind *)t_alloc( listen_fd, T_BIND, T_ALL ))
                == (struct t_bind *) NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }
    bind->qlen = 1;
    bind->addr.len = sizeof(int);
    *(int *) bind->addr.buf = SRV_ADDR;
    if (t_bind (listen_fd, bind, bind) < 0 ) {
        t_error("t_bind failed for listen_fd");
        exit(3);
    }
    /* Was the correct address bound? */
    if (bind->addr.len != sizeof(int) ||
       *(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\n");
        exit(4);
    }
```

Like the client, the server first calls `t_open` to establish a transport endpoint with the desired transport provider. The endpoint, `listen_fd`, is used to listen for connect requests.

Next, the server binds its address to the endpoint. This address is used by each client to access the server. The second argument points to a `t_bind` structure that specifies the address to bind to the endpoint. The `t_bind` structure has the following format:

```
struct t_bind {
    struct netbuf addr;
    unsigned qlen;
}
```

where addr describes the address to be bound, and qlen specifies the maximum number of outstanding connect requests. All TLI structure and constant definitions are in <tiuser.h>.

The address is specified in the netbuf structure with the following format:

```
struct netbuf {
    unsigned int maxlen;
    unsigned int len;
    char *buf;
}
```

where maxlen specifies the maximum length of the buffer in bytes, len specifies the bytes of data in the buffer, and buf points to the buffer that contains the data.

In the t_bind structure, the data identifies a transport address. qlen specifies the maximum number of connect requests that can be queued. If the value of qlen is positive, the endpoint can be used to listen for connect requests. t_bind directs the transport provider to queue connect requests for the bound address immediately. The server must dequeue each connect request and accept or reject it. For a server that fully processes a single connect request and responds to it before receiving the next request, a value of 1 is appropriate for qlen. Servers that dequeue several connect requests before responding to any should specify a longer queue. The server in this example processes connect requests one at a time, so qlen is set to 1.

t_alloc() is called to allocate the t_bind structure. t_alloc has three arguments: a file descriptor of a transport endpoint; the identifier of the structure to allocate; and a flag that specifies which, if any, netbuf buffers to allocate. T_ALL specifies to allocate all netbuf buffers, and causes the addr buffer to be allocated in this example. Buffer size is determined automatically and stored in the maxlen field.

Each transport provider manages its address space differently. Some transport providers allow a single transport address to be bound to several transport endpoints, while others require a unique address per endpoint. TLI supports both. Based on its rules, a provider determines if it can bind the requested address. If not, it chooses another valid address from its address space and binds it to the transport endpoint. The server must check the bound address to ensure that it is the one previously advertised to clients.

If t_bind succeeds, the provider begins queueing connect requests, entering the next phase of communication.

Connection Establishment

TLI imposes different procedures in this phase for clients and servers. The client starts connection establishment by requesting a connection to a specified server using `t_connect`(). The server receives a client's request by calling `t_listen`(). The server must accept or reject the client's request. It calls `t_accept`() to establish the connection, or `t_snddis`() to reject the request. The client is notified of the result when `t_connect` returns.

TLI supports two facilities during connection establishment that may not be supported by all transport providers:

- Data transfer between the client and server when establishing the connection – The client may send data to the server when it requests a connection. This data is passed to the server by `t_listen`. The server can send data to the client when it accepts or rejects the connection. The connect characteristic returned by `t_open` determines how much data, if any, two users may transfer during connect establishment.
- The negotiation of protocol options – The client may specify preferred protocol options to the transport provider and/or the remote user. TLI supports both local and remote option negotiation. Option negotiation is a protocol-specific capacity.

These facilities produce protocol-dependent software (see "Guidelines to Protocol Independence" on page 213).

Client

The steps for the client to establish a connection are shown in Code Example 6-3.

Code Example 6-3 Client-to-Server Connection

```
/*
 * Because it assumes it knows the format of the provider's
 * address, this program is transport-dependent
 */
if ((sndcall = (struct t_call *) t_alloc(fd, T_CALL, T_ADDR))
      == (struct t_call *) NULL) {
    t_error("t_alloc failed");
    exit(3);
}
sndcall->addr.len = sizeof(int);
```

```
    *(int *) sndcall->addr.buf = SRV_ADDR;
    if (t_connect( fd, sndcall, (struct t_call *) NULL) == -1 ) {
        t_error("t_connect failed for fd");
        exit(4);
    }
```

The `t_connect` call connects to the server. The first argument of `t_connect` identifies the client's endpoint, and the second argument points to a `t_call` structure that identifies the destination server. This structure has the following format:

```
struct t_call {
    struct netbuf addr;
    struct netbuf opt;
    struct netbuf udata;
    int sequence;
}
```

`addr` identifies the address of the server, `opt` specifies protocol-specific options to the connection, and `udata` identifies user data that may be sent with the connect request to the server. The `sequence` field has no meaning for `t_connect`. In this example, only the server's address is passed.

`t_alloc` allocates the `t_call` structure dynamically. The third argument of `t_alloc` is `T_ADDR` to specify to allocate a `netbuf` buffer. The server's address is then copied to `buf`, and `len` is set accordingly.

The third argument of `t_connect` can be used to return information about the newly established connection, and can return any user data sent by the server in its response to the connect request. The third argument here is set to `NULL` by the client. The connection is established on successful return of `t_connect`. If the server rejects the connect request, `t_connect` sets `t_errno` to `TLOOK`.

Event Handling

The TLOOK error has special significance. TLOOK is set if a TLI routine is interrupted by an unexpected asynchronous transport event on the endpoint. TLOOK does not report an error with a TLI routine, but the normal processing of the routine is not done because of the pending event. The events defined by TLI are listed in Table 6-7.

Table 6-7 Asynchronous Endpoint Events

Name	Description
T_LISTEN	Connection request arrived at the transport endpoint
T_CONNECT	Confirmation of a previous connect request arrived (generated when a server accepts a connect request)
T_DATA	User data has arrived
T_EXDATA	Expedited user data arrived
T_DISCONNECT	Notice of an aborted connection or of a rejected connect request arrived
T_ORDREL	A request for orderly release of a connection arrived
T_UDERR	Notice of an error in a previous datagram arrived. (see “Connectionless Mode Service” on page 192)

The state table in “State Transitions” on page 208 shows which events can happen in each state. t_look() lets a user determine what event has occurred if a TLOOK error is returned. In the example, if a connect request is rejected, the client exits.

Server

When the client calls t_connect, a connect request is sent at the server’s transport endpoint. For each client, the server accepts the connect request and spawns a process to service the connection.

```
if ((call = (struct t_call *) t_alloc(listen_fd, T_CALL, T_ALL))
          == (struct t_call *) NULL) {
    t_error("t_alloc of t_call structure failed");
    exit(5);
}
while(1) {
    if (t_listen( listen_fd, call) == -1) {
```

```
        t_error("t_listen failed for listen_fd");
        exit(6);
    }
    if ((conn_fd = accept_call(listen_fd, call)) != DISCONNECT)
        run_server(listen_fd);
}
```

The server allocates a `t_call` structure, then does a closed loop. The loop blocks on `t_listen` for a connect request. When a request arrives, the server calls `accept_call` to accept the connect request. `accept_call` accepts the connection on an alternate transport endpoint (as discussed below) and returns the handle of that endpoint. (`conn_fd` is a global variable.) Because the connection is accepted on an alternate endpoint, the server can continue to listen on the original endpoint. If the call is accepted without error, `run_server` spawns a process to service the connection.

Note – TLI supports an asynchronous mode for these routines that prevents a process from blocking. See “Advanced Topics” on page 201.

When a connect request arrives, the server calls `accept_call` to accept the client’s request, as Code Example 6-4 shows.

Code Example 6-4 `accept_call` Function

```
accept_call(listen_fd, call)
int listen_fd;
struct t_call *call;
{
    int resfd;

    if ((resfd = t_open("/dev/exmp", O_RDWR, (struct t_info *)
NULL))
             == -1) {
        t_error("t_open for responding fd failed");
        exit(7);
    }
    if (t_bind(resfd,(struct t_bind *) NULL, (struct t_bin"t_bind
for responding fd failed");
        exit(8);
    }
    if (t_accept(listen_fd, resfd, call) == -1) {
        if (t_errno == TLOOK) {           /* must be a disconnect */
            if (t_rcvdis(listen_fd,(struct t_discon *) NULL) == -1) {
                t_error("t_rcvdis failed for listen_fd");
```

```
                exit(9);
            }
            if (t_close(resfd) == -1) {
                t_error("t_close failed for responding fd");
                exit(10);
            }
            /* go back up and listen for other calls */
            return(DISCONNECT);
        }
        t_error("t_accept failed");
        exit(11);
    }
    return(resfd);
}
```

`accept_call` has two arguments:

1. `listen_fd` is the file handle of the transport endpoint where the connect request arrived.
2. `call` points to a `t_call` structure that contains all information associated with the connect request.

The server first opens another transport endpoint by opening the clone device special file of the transport provider and binding an address. A `NULL` specifies not to return the address bound by the provider. The new transport endpoint, `resfd`, accepts the client's connect request.

The first two arguments of `t_accept` specify the listening transport endpoint and the endpoint where the connection is accepted respectively. Accepting a connection on the listening endpoint prevents other clients from accessing the server for the duration of the connection.

The third argument of `t_accept` points to the `t_call` structure containing the connect request. This structure should contain the address of the calling user and the sequence number returned by `t_listen`. The sequence number is significant if the server queues multiple connect requests. The"Advanced Topics" on page 201 shows an example of this. The `t_call` structure also identifies protocol options and user data to pass to the client. Because this transport provider does not support protocol options or the transfer of user data during connection, the `t_call` structure returned by `t_listen` is passed without change to `t_accept`.

The example is simplified. The server exits if either the `t_open` or `t_bind` call fails. `exit`(2) closes the transport endpoint of `listen_fd`, causing a disconnect request to be sent to the client. The client's `t_connect` call fails, setting `t_errno` to `TLOOK`.

`t_accept` can fail if an asynchronous event occurs on the listening endpoint before the connection is accepted, and `t_errno` will be set to `TLOOK`. Table 6-8 on page 207 shows that only a disconnect request can be sent in this state with only one queued connect request. This event can happen if the client undoes a previous connect request. If a disconnect request arrives, the server must respond by calling `t_rcvdis`. This routine argument is a pointer to a `t_discon` structure, which is used to retrieve the data of the disconnect request. In this example, the server passes a `NULL`.

After receiving a disconnect request, `accept_call` closes the responding transport endpoint and returns `DISCONNECT`, which informs the server that the connection was disconnected by the client. The server then listens for further connect requests.

The figure illustrates how the server establishes connections:

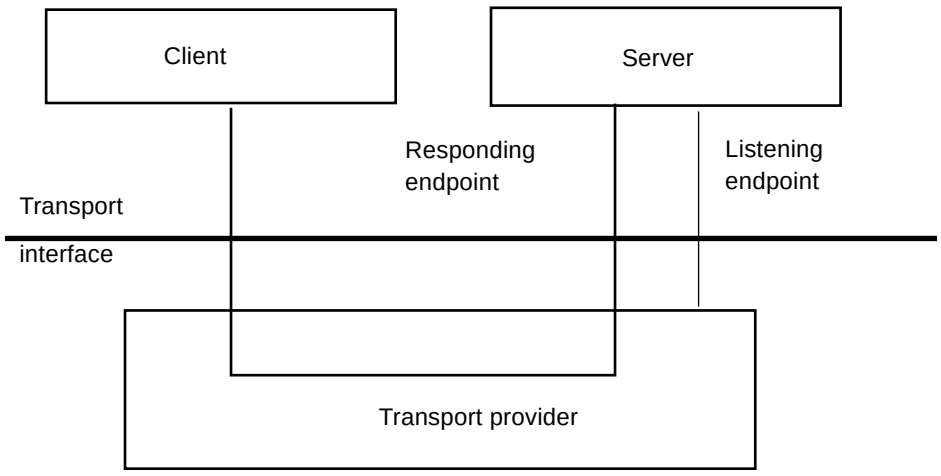


Figure 6-4 Listening and Responding Transport Endpoints

The transport connection is established on the new responding endpoint, and the listening endpoint is freed to retrieve further connect requests.

Data Transfer

Once the connection is established, both the client and the server can transfer data through the connection using `t_snd` and `t_rcv`. TLI does not differentiate the client from the server from this point on. Either user may send data, receive data, or release the connection.

There are two classes of data on a transport connection:

1. Normal data
2. Expedited data

Expedited mode is for urgent data. The exact semantics of expedited data vary between transport providers. Not all transport protocols support expedited data (see `t_open`()).

All connection oriented mode protocols must transfer data in byte streams. “Byte stream” implies no message boundaries in data sent over a connection. Some transport protocols preserve message boundaries over a transport connection. This service is supported by TLI, but protocol-independent software must not rely on it.

The message boundaries are invoked by the `T_MORE` flag of `t_snd` and `t_rcv`. The messages, called transport service data units (TSDU), may be transferred between two transport users as distinct units. The maximum message size is defined by the underlying transport protocol. Get the message size through `t_open` or `t_getinfo`. You can send a message in multiple units.

Set the `T_MORE` flag on every `t_snd` call except the last to send a message in multiple units. The flag specifies that the data in the current and the next `t_snd` calls are a logical unit. Send the last message unit with `T_MORE` turned off to specify the end of the logical unit.

Similarly, a logical unit may be sent in multiple units. If `t_rcv` returns with the `T_MORE` flag set, the user must call `t_rcv` again to receive the rest of the message. The last unit in the message is identified by a call to `t_rcv` that does not set `T_MORE`.

The `T_MORE` flag implies nothing about how the data is packaged below TLI or how the data is delivered to the remote user. Each transport protocol, and each implementation of a protocol, may package and deliver the data differently.

For example, if a user sends a complete message in a single call to `t_snd`, there is no guarantee that the transport provider delivers the data in a single unit to the receiving user. Similarly, a message transmitted in two units may be delivered in a single unit to the remote transport user. The message boundaries are preserved only by setting the value of `T_MORE` for `t_snd` and testing it after `t_rcv`. This guarantees that the receiver sees a message with the same contents and message boundaries as was sent.

Client

The example server transfers a log file to the client over the transport connection. The client receives the data and writes it to its standard output file. A byte stream interface is used by the client and server, with no message boundaries. The client receives data by the following:

```
while ((nbytes = t_rcv(fd, buf, nbytes, &flags))!= -1){
    if (fwrite(buf, 1, nbytes, stdout) == -1) {
        fprintf(stderr, "fwrite failed\n");
        exit(5);
    }
}
```

The client repeatedly calls `t_rcv` to receive incoming data. `t_rcv` blocks until data arrives. `t_rcv` writes up to `nbytes` of the data available into `buf` and returns the number of bytes buffered. The client writes the data to standard output and continues. The data transfer loop ends when `t_rcv` fails. `t_rcv` fails when an orderly release or disconnect request arrives. If `fwrite`() fails for any reason, the client exits, which closes the transport endpoint. If the transport endpoint is closed (either by `exit` or `t_close`) during data transfer, the connection is aborted and the remote user receives a disconnect request.

Server

The server manages its data transfer by spawning a child process to send the data to the client. The parent process continues the loop to listen for more connect requests. `run_server` is called by the server to spawn this child process, as shown in Code Example 6-5.

Code Example 6-5 Spawning Child Process to Loopback and Listen

```
connrelease()
{
    /* conn_fd is global because needed here */
```

```
    if (t_look(conn_fd) == T_DISCONNECT) {
        fprintf(stderr, “connection aborted\n”);
        exit(12);
    }
    /* else orderly release request - normal exit */
    exit(0);
}
run_server(listen_fd)
int listen_fd;
{
    int nbytes;
    FILE *logfp;                    /* file pointer to log file */
    char buf[1024];

    switch(fork()) {
    case -1:
        perror("fork failed");
        exit(20);
    default:                                            /* parent */
        /* close conn_fd and then go up and listen again*/
        if (t_close(conn_fd) == -1) {
            t_error("t_close failed for conn_fd");
            exit(21);
        }
        return;
    case 0:                                              /* child */
        /* close listen_fd and do service */
        if (t_close(listen_fd) == -1) {
            t_error("t_close failed for listen_fd");
            exit(22);
        }
        if ((logfp = fopen("logfile", "r")) == (FILE *) NULL) {
            perror("cannot open logfile");
            exit(23);
        }
        signal(SIGPOLL, connrelease);
        if (ioctl(conn_fd, I_SETSIG, S_INPUT) == -1) {
            perror("ioctl I_SETSIG failed");
            exit(24);
        }
        if (t_look(conn_fd) != 0){           /*disconnect there?*/
            fprintf(stderr, "t_look: unexpected event\n");
            exit(25);
        }
        while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
```

```
if (t_snd(conn_fd, buf, nbytes, 0) == -1) {
    t_error("t_snd failed");
    exit(26);
}
```

After the fork, the parent process returns to the main listening loop. The child process manages the newly established transport connection. If the `fork` fails, `exit` closes both transport endpoints, sending a disconnect request to the client, and the client's `t_connect` call fails.

The server process reads 1024 bytes of the log file at a time and sends the data to the client using `t_snd`. `buf` points to the start of the data buffer, and `nbytes` specifies the number of bytes to transmit. The fourth argument can be zero or one of the two optional flags below:

- `T_EXPEDITED` specifies that the data is expedited.
- `T_MORE` specifies that the next block will continue the message in this block.

Neither flag is set by the server in this example.

If the user floods the transport provider with data, `t_snd` blocks until enough data is removed from the transport.

`t_snd` does not look for a disconnect request (showing that the connection was broken). If the connection is aborted, the server should be notified since data may be lost. One solution is to call `t_look` to check for incoming events before each `t_snd` call or after a `t_snd` failure. The example has a cleaner solution. The `I_SETSIG ioctl` lets a user request a signal when a specified event occurs. See the `streamio(7)` manpage. `S_INPUT` causes a signal to the user when any input arrives at the endpoint `conn_fd`. If a disconnect request arrives, the signal-catching routine (`connrelease`) prints an error message and exits.

If the server alternates `t_snd` and `t_rcv` calls, it can use `t_rcv` to recognize an incoming disconnect request.

Connection Release

At any time during data transfer, either user can release the transport connection and end the conversation. There are two forms of connection release. The first way, abortive release, breaks the connection immediately and discards any data that has not been delivered to the destination user.

Either user can call `t_snddis` to perform an abortive release. The transport provider can abort a connection if a problem occurs below TLI. `t_snddis` lets a user send data to the remote user when aborting a connection. The abortive release is supported by all transport providers, the ability to send data when aborting a connection is not.

When the remote user is notified of the aborted connection, call `t_rcvdis` to receive the disconnect request. The call returns a code that identifies why the connection was aborted, and returns any data that may have accompanied the disconnect request (if the abort was initiated by the remote user). The reason code is specific to the underlying transport protocol, and should not be interpreted by protocol-independent software.

The second way, orderly release, ends a connection so that no data is lost. All transport providers must support the abortive release procedure, but orderly release is an option not supported by all connection-oriented protocols.

Server

This example assumes that the transport provider supports orderly release. When all the data has been sent by the server, the connection is released as follows:

```
if (t_sndrel(conn_fd) == -1) {
    t_error("t_sndrel failed");
    exit(27);
}
pause(); /* until orderly release request arrives */
```

Orderly release requires two steps by each user. The server can call `t_sndrel`. This routine sends a disconnect request. When the client receives the request, it can continue sending data back to the server. When all data have been sent, the client calls `t_sndrel` to send a disconnect request back. The connection is released only after both users have received a disconnect request.

In this example, data is transferred only from the server to the client. So there is no provision to receive data from the client after the server initiates release. The server calls `pause`() after initiating the release.

The client responds with its orderly release request, which generates a signal caught by `connrelease`. (In Code Example 6-5 on page 188, the server issued an `I_SETSIG` `ioctl` call to generate a signal on any incoming event.) The only TLI event possible in this state is a disconnect request or an orderly

release request, so `connrelease` `exit`s normally when the orderly release request arrives. `exit` from `connrelease` closes the transport endpoint and frees the bound address. To close a transport endpoint without exiting, call `t_close`.

Client

The client releases the connection similarly to the server. The client processes incoming data until `t_rcv` fails. When the server releases the connection (using either `t_snddis` or `t_sndrel`), `t_rcv` fails and sets `t_errno` to `TLOOK`. The client then processes the connection release as follows:

```
if ((t_errno == TLOOK) && (t_look(fd) == T_ORDREL)) {
    if (t_rcvrel(fd) == -1) {
        t_error("t_rcvrel failed");
        exit(6);
    }
    if (t_sndrel(fd) == -1) {
        t_error("t_sndrel failed");
        exit(7);
    }
    exit(0);
}
```

Each event on the client's transport endpoint is checked for an orderly release request. When one is received, the client calls `t_rcvrel` to process the request and `t_sndrel` to send the response release request. The client then exits, closing its transport endpoint.

If a transport provider does not support the orderly release, use abortive release with `t_snddis` and `t_rcvdis`. Each user must take steps to prevent data loss. For example, use a special byte pattern in the data stream to indicate the end of a conversation.

Connectionless Mode Service

Connectionless mode service is appropriate for short-term request/response interactions, such as transaction-processing applications. Data are transferred in self-contained units with no logical relationship required among multiple units.

A transaction server is used as an example. The server waits for incoming transaction queries, and processes and responds to each query.

Endpoint Initiation

Transport users must initiate TLI endpoints before transferring data. Choose the appropriate connectionless service provider using `t_open` and establish its identity using `t_bind`.

Use `t_optmgmt` to negotiate protocol options. Like connection mode service, each transport provider specifies the options, if any, it supports. Option negotiation is a protocol-specific activity.

Code Example 6-6 shows the definitions and initiation sequence of the transaction server.

Code Example 6-6 Transaction Server

```
#include <stdio.h>
#include <fcntl.h>
#include <tiuser.h>
#define SRV_ADDR 2             /* server's well known address */

main()
{
    int fd;
    int flags;
    struct t_bind *bind;
    struct t_unitdata *ud;
    struct t_uderr *uderr;
    extern int t_errno;

    if ((fd = t_open("/dev/exmp", O_RDWR, (struct t_info *) NULL))
            == -1) {
        t_error("unable to open /dev/exmp");
        exit(1);
    }
    if ((bind = (struct t_bind *)t_alloc(fd, T_BIND, T_ADDR))
             == (struct t_bind *) NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }
    bind->addr.len = sizeof(int);
    *(int *)bind->addr.buf = SRV_ADDR;
    bind->qlen = 0;
    if (t_bind(fd, bind, bind) == -1) {
        t_error("t_bind failed");
        exit(3);
```

```
}
/*
 * is the bound address correct?
 */
if (bind -> addr.len != sizeof(int) ||
        *(int *)bind->addr.buf != SRV_ADDR) {
    fprintf(stderr, "t_bind bound wrong address\n");
    exit(4);
}
```

The server establishes a transport endpoint with the desired transport provider using `t_open`. Each provider has an associated service type, so the user may choose a particular service by opening the appropriate transport provider file. This connectionless mode server ignores the characteristics of the provider returned by `t_open` by setting the third argument to `NULL`. The transaction server assumes the transport provider has the following characteristics:

- The transport address is an integer value that uniquely identifies each user.
- The transport provider supports the `T_CLTS` service type (connectionless transport service, or datagram).
- The transport provider does not support any protocol-specific options.

The connectionless server binds a transport address to the endpoint so that potential clients can access the server. A `t_bind` structure is allocated using `t_alloc` and the `buf` and `len` fields of the address are set accordingly.

A major difference between a connection mode server and a connectionless mode server is that the `qlen` field of the `t_bind` structure is 0 for connectionless mode service. There are no connection requests to queue.

TLI defines an inherent client-server relationship between two users while establishing a transport connection in the connection mode service. No such relationship exists in connectionless mode service. This example, not TLI, defines one user as a server and the other as a client.

Because the address of the server is known by all potential clients, the server checks the bound address returned by `t_bind` to ensure it is correct.

Data Transfer

Once a user has bound an address to the transport endpoint, datagrams may be sent or received over the endpoint. Each outgoing message carries the address of the destination user. TLI also lets you specify protocol options to the transfer of the data unit (for example, transit delay). Each transport provider defines the set of options, if any, on a datagram. When the datagram is passed to the destination user, the associated protocol options may be passed too.

Code Example 6-7 illustrates the data transfer phase of the connectionless mode server.

Code Example 6-7 Data Transfer Routine

```
if ((ud = (struct t_unitdata *) t_alloc(fd, T_UNITDATA, T_ALL))
       == (struct t_unitdata *) NULL) {
    t_error("t_alloc of t_unitdata struct failed");
    exit(5);
}
if ((uderr = (struct t_uderr *) t_alloc(fd, T_UDERROR, T_ALL))
          == (struct t_uderr *) NULL) {
    t_error("t_alloc of t_uderr struct failed");
    exit(6);
}
while(1) {
    if (t_rcvudata(fd, ud, &flags) == -1) {
        if (t_errno == TLOOK) {
            /* Error on previously sent datagram */
            if(t_rcvuderr(fd, uderr) == -1) {
                exit(7);
            }
            fprintf(stderr, "bad datagram, error=%d\n",
                 uderr->error);
            continue;
        }
        t_error("t_rcvudata failed");
        exit(8);
    }
    /*
     * Query() processes the request and places the response in
     * ud->udata.buf, setting ud->udata.len
     */
    query(ud);
    if (t_sndudata(fd, ud) == -1) {
        t_error("t_sndudata failed");
```

```
            exit(9);
        }
    }
}

/* ARGS USED */
void
query(ud)
struct t_unitdate *ud;
{
    /* Merely a stub for simplicity */
}
```

To buffer datagrams, the server first allocates a `t_unitdata` structure, which has the following format:

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

`addr` holds the source address of incoming datagrams and the destination address of outgoing datagrams. `opt` holds any protocol options on the datagram. `udata` holds the data. The `addr`, `opt`, and `udata` fields must all be allocated with buffers large enough to hold any possible incoming values. The `T_ALL` argument of `t_alloc` ensures this and sets the `maxlen` field of each `netbuf` structure accordingly. The provider does not support protocol options in this example, so `maxlen` is set to 0 in the `opt netbuf` structure. The server also allocates a `t_uderr` structure for datagram errors.

The transaction server loops forever, receiving queries, processing the queries, and responding to the clients. It first calls `t_rcvudata` to receive the next query. `t_rcvudata` blocks until a datagram arrives, and returns it. The second argument of `t_rcvudata` identifies the `t_unitdata` structure in which to buffer the datagram. The third argument, `flags`, points to an integer variable and may be set to `T_MORE` on return from `t_rcvudata` to indicate that the user's `udata` buffer is too small to store the full datagram. If this happens, the next call to `t_rcvudata` gets the rest of the datagram. Because `t_alloc` allocates a `udata` buffer large enough to store the maximum size datagram, this transaction server does not have to check `flags`. This is true only of `t_rcvudata` and not of any other receive primitives.

When a datagram is received, the transaction server calls its `query` routine to process the request. This routine stores a response in the structure pointed to by `ud`, and sets `ud->udata.len` to the number of bytes in the response. The source address returned by `t_rcvudata` in `ud->addr` is the destination address for `t_sndudata`.

When the response is ready, `t_sndudata` is called to send the response to the client. TLI prevents a user from flooding the transport provider with datagrams by the same flow control mechanism described for the connection mode service. In such cases, `t_sndudata` blocks until the flow is relieved, and then returns.

Datagram Errors

If the transport provider cannot process a datagram sent by `t_sndudata`, it returns a unit data error event, `T_UDERR`, to the user. This event includes the destination address and options of the datagram, and a protocol-specific error value that identifies the error. Datagram errors are protocol specific.

Note – A unit data error event does not always indicate success or failure in delivering the datagram to the specified destination. Remember, connectionless service does not guarantee reliable delivery of data.

The transaction server is notified of an error when it tries to receive another datagram. In this case, `t_rcvudata` will fail, setting `t_errno` to `TLOOK`. If `TLOOK` is set, the only possible event is `T_UDERR`, so the server calls `t_rcvuderr` to retrieve the event. The second argument of `t_rcvuderr` is the `t_uderr` structure that was allocated earlier. This structure is filled in by `t_rcvuderr` and has the following format:

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

where `addr` and `opt` identify the destination address and protocol options specified in the bad datagram, and `error` is a protocol-specific error code. The transaction server prints the error code and then continues.

6

A Read/Write Interface

A user may want to establish a transport connection and use `exec`() on an existing program such as `cat`() to process the data as it arrives over the connection. Existing programs use `read`() and `write`(). TLI does not directly support a read / write interface to a transport provider, but one is available. The interface lets you issue `read` and `write` calls over a transport connection in the data transfer phase. This section describes the read / write interface to the connection mode service of TLI. This interface is not available with the connectionless mode service.

The read / write interface is presented using the client example of "Connection Mode Service" on page 174 with modifications. The clients are identical until the data transfer phase. Then the client uses the read / write interface and `cat`() to process incoming data. `cat` is run without change over the transport connection. Only the differences between this client and that of the client in Code Example 6-1 on page 176 are shown in Code Example 6-8.

Code Example 6-8 Read / Write Interface

```
#include <stropts.h>
    .
    .   /*
         Same local management and connection establishment steps.
         */
    .
    if (ioctl(fd, I_PUSH, "tirdwr") == -1) {
        perror("I_PUSH of tirdwr failed");
        exit(5);
    }
    close(0);
    dup(fd);
    execl("/usr/bin/cat", "/usr/bin/cat", (char *) 0);
    perror("exec of /usr/bin/cat failed");
    exit(6);
}
```

The client invokes the read / write interface by pushing `tirdwr`() onto the stream associated with the transport endpoint. See `I_PUSH` in the `streamio(7)` manpage. `tirdwr` converts TLI above the transport provider into a pure read / write interface. With the module in place, the client calls `close`() and `dup`() to establish the transport endpoint as its standard input file, and uses `/usr/bin/cat` to process the input.

By pushing `tirdwr` onto the transport provider, TLI is changed. The semantics of `read` and `write` must be used, and message boundaries are not preserved. `tirdwr` can be popped from the transport provider to restore TLI semantics (see `I_POP` in the `streamio(7)` manpage.

Caution – The `tirdwr` module may only be pushed onto a stream when the transport endpoint is in the data transfer phase. Once the module is pushed, the user may not call any TLI routines. If a TLI routine is invoked, `tirdwr` generates a fatal protocol error, `EPROTO`, on the stream, rendering it unusable. If you then pop the `tirdwr` module off the stream, the transport connection is aborted. See `I_POP` in the `streamio(7)` manpage.

`write`

Send data over the transport connection with `write`. `tirdwr` passes data through to the transport provider. If you send a zero-length data packet, which the mechanism allows, `tirdwr` discards the message. If the transport connection is aborted (for example, because the remote user aborts the connection using `t_snddis`), a hang-up condition is generated on the stream, further `write` calls fail, and `errno` is set to `ENXIO`. You can still retrieve any available data after a hang-up.

`read`

Receive data that arrives at the transport connection with `read`. `tirdwr` passes data from the transport provider. Any other event or request passed to the user from the provider is processed by `tirdwr` as follows:

- `read` cannot identify expedited data to the user. If an expedited data request is received, `tirdwr` generates a fatal protocol error, `EPROTO`, on the stream. The error causes further system calls to fail. Do not use `read` to receive expedited data.
- `tirdwr` discards an abortive disconnect request and generates a hang-up condition on the stream. Subsequent `read` calls will retrieve any remaining data, then return zero for all further calls (indicating end of file).
- `tirdwr` discards an orderly release request and delivers a zero-length message to the user. As described in `read`()], this notifies the user of end of file by returning 0.

- If any other TLI request is received, `tirdwr` generates a fatal protocol error, `EPROTO`, on the stream. This causes further system calls to fail. If a user pushes `tirdwr` onto a stream after the connection has been established, no request is generated.

`close`

With `tirdwr` on a stream, you can send and receive data over a transport connection for the duration of the connection. Either user can terminate the connection by closing the file descriptor associated with the transport endpoint or by popping the `tirdwr` module off the stream. In either case, `tirdwr` does the following:

- If an orderly release request was previously received by `tirdwr`, it is passed to the transport provider to complete the orderly release of the connection. The remote user who initiated the orderly release procedure receives the expected request when data transfer completes.
- If a disconnect request was previously received by `tirdwr`, no special action is taken.
- If neither an orderly release nor a disconnect request was previously received by `tirdwr`, a disconnect request is passed to the transport provider to abort the connection.
- If an error previously occurred on the stream and a disconnect request has not been received by `tirdwr`, a disconnect request is passed to the transport provider.

A process cannot initiate an orderly release after `tirdwr` is pushed onto a stream. `tirdwr` handles an orderly release if it is initiated by the user on the other side of a transport connection. If the client in this section is communicating with the server program in “Connection Mode Service” on page 174, the server will terminate the transfer of data with an orderly release request. The server then waits for the corresponding request from the client. At that point, the client exits and the transport endpoint is closed. When the file descriptor is closed, `tirdwr` initiates the orderly release request from the client’s side of the connection. This generates the request that the server is blocked on.

Some protocols, like TCP, require this orderly release to ensure that the data is delivered intact.

Advanced Topics

This section presents additional TLI concepts:

- An optional nonblocking (asynchronous) mode for some library calls
- How to set and get TCP and UDP options under TLI
- A program example of a server supporting multiple outstanding connect requests and operating in an event-driven manner

Asynchronous Execution Mode

Many TLI library routines block to wait for an incoming event. However, some time-critical applications should not block for any reason. An application may do local processing while waiting for some asynchronous TLI event.

Asynchronous processing of TLI events is available to applications through the combination of asynchronous features and the non-blocking mode of TLI library routines. Use of the `poll`() system call and the `I_SETSIG ioctl` command to process events asynchronously is described in the section, “poll() on the Server Side” on page 90.

Each TLI routine that blocks for an event can be run in a special non-blocking mode. For example, `t_listen` normally blocks for a connect request. A server can periodically poll a transport endpoint for queued connect requests by calling `t_listen` in the non-blocking (or asynchronous) mode. The asynchronous mode is enabled by setting `O_NDELAY` or `O_NONBLOCK` in the file descriptor. These modes can be set as a flag through `t_open`, or by calling `fcntl`() before calling the TLI routine. `fcntl` enables or disables this mode at any time. All program examples in this chapter use the default synchronous processing mode.

`O_NDELAY` or `O_NONBLOCK` affect each TLI routine differently. You will need to determine the exact semantics of `O_NDELAY` or `O_NONBLOCK` for a particular routine.

Advanced Programming Example

The following example demonstrates two important concepts. The first is a server’s ability to manage multiple outstanding connect requests. The second is event-driven use of TLI and the system call interface.

The server example in Code Example 6-2 on page 178 supports only one outstanding connect request, but TLI lets a server manage multiple outstanding connect requests. One reason to receive several simultaneous connect requests is to prioritize the clients. A server can receive several connect requests, and accept them in an order based on the priority of each client. A second reason for handling several outstanding connect requests is the limits of single-threaded processing. Depending on the transport provider, while a server processes one connect request, other clients find it busy. If multiple connect requests are processed simultaneously, the server will be found busy only if more than the maximum number of clients try to call the server simultaneously.

The server example is event-driven: the process polls a transport endpoint for incoming TLI events, and takes the appropriate actions for the event received. The example demonstrates the ability to poll multiple transport endpoints for incoming events.

The definitions and endpoint establishment functions of Code Example 6-8 are similar to those of the server example in Code Example 6-2 on page 178.

Code Example 6-9 Endpoint Establishment (Convertible to Multiple Connections)

```
#include <tiuser.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <stropts.h>
#include <signal.h>

#define NUM_FDS 1
#define MAX_CONN_IND 4
#define SRV_ADDR 1                /* server's well known address */

int conn_fd;                        /* server connection here */
extern int t_errno;
/* holds connect requests */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];

main()
{
    struct pollfd pollfds[NUM_FDS];
    struct t_bind *bind;
    int i;
```

```
    /*
     * Only opening and binding one transport endpoint, but more can
     * be supported
     */
    if ((pollfds[0].fd = t_open("/dev/tivc", O_RDWR,
              (struct t_info *) NULL)) == -1) {
        t_error("t_open failed");
        exit(1);
    }
    if ((bind = (struct t_bind *) t_alloc(pollfds[0].fd, T_BIND,
                 T_ALL)) == (struct t_bind *) NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }
    bind->qlen = MAX_CONN_IND;
    bind->addr.len = sizeof(int);
    *(int *) bind->addr.buf = SRV_ADDR;
    if (bind->addr.len != sizeof(int) ||
         t_bind(pollfds[0].fd, bind, bind) == -1) {
        t_error("t_bind failed");
        exit(3);
    }
    /* Was the correct address bound? */
    if (bind->addr.len != sizeof(int) ||
          *(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\n");
        exit(4);
    }
}
```

The file descriptor returned by `t_open` is stored in a `pollfd` structure that controls polling the transport endpoints for incoming data. See the `poll(2)` manpage. Only one transport endpoint is established in this example. However, the remainder of the example is written to manage multiple transport endpoints. Several endpoints could be supported with minor changes to Code Example 6-9.

This server sets `qlen` to a value greater than 1 for `t_bind`. This specifies that the server queues multiple outstanding connect requests. The server accepts the current connect request before accepting additional connect requests. This example can queue up to `MAX_CONN_IND` connect requests. The transport provider may negotiate the value of `qlen` smaller if it cannot support `MAX_CONN_IND` outstanding connect requests.

Once the server has bound its address and is ready to process connect requests, it behaves as shown in Code Example 6-10.

Code Example 6-10 Processing Connection Requests

```
pollfds[0].events = POLLIN;

while (TRUE) {
    if (poll(pollfds, NUM_FDS, -1) == -1) {
        perror("poll failed");
        exit(5);
    }
    for (i = 0; i < NUM_FDS; i++) {
        switch (pollfds[i].revents) {
            default:
                perror("poll returned error event");
                exit(6);
            case 0:
                continue;
            case POLLIN:
                do_event(i, pollfds[i].fd);
                service_conn_ind(i, pollfds[i].fd);
        }
    }
}
```

The `events` field of the `pollfd` structure is set to `POLLIN`, which notifies the server of any incoming TLI events. The server then enters an infinite loop in which it `poll`s the transport endpoint(s) for events, and processes events as they occur.

The `poll` call blocks indefinitely for an incoming event. On return, each entry (one per transport endpoint) is checked for a new event. If `revents` is 0, no event has occurred on the endpoint and the server continues to the next endpoint. If `revents` is `POLLIN`, there is an event on the endpoint. `do_event` is called to process the event. Any other value in `revents` indicates an error on the endpoint, and the server exits. With multiple endpoints, it is better for the server to `close` this descriptor and continue.

For each iteration of the loop, `service_conn_ind` is called to process any outstanding connect requests. If another connect request is pending, `service_conn_ind` saves the new connect request and responds to it later.

`do_event`() in Code Example 6-11 is called to process an incoming event.

Code Example 6-11 Event Processing Routine

```
do_event( slot, fd)
int slot;
int fd;
{
    struct t_discon *discon;
    int i;

    switch (t_look(fd)) {
    default:
        fprintf(stderr, "t_look: unexpected event\n");
        exit(7);
    case T_ERROR:
        fprintf(stderr, "t_look returned T_ERROR event\n");
        exit(8);
    case -1:
        t_error("t_look failed");
        exit(9);
    case 0:
        /* since POLLIN returned, this should not happen */
        fprintf(stderr,"t_look returned no event\n");
        exit(10);
    case T_LISTEN:
        /* find free element in calls array */
        for (i = 0; i < MAX_CONN_IND; i++) {
            if (calls[slot][i] == (struct t_call *) NULL)
                break;
        }
        if ((calls[slot][i] = (struct t_call *) t_alloc( fd, T_CALL,
                    T_ALL)) == (struct t_call *) NULL) {
            t_error("t_alloc of t_call structure failed");
            exit(11);
        }
        if (t_listen(fd, calls[slot][i] ) == -1) {
            t_error("t_listen failed");
            exit(12);
        }
        break;
    case T_DISCONNECT:
        discon = (struct t_discon *) t_alloc(fd, T_DIS, T_ALL);
        if(t_rcvdis( fd, discon) == -1) {
            t_error("t_rcvdis failed");
            exit(13);
        }
        /* find call ind in array and delete it */
```

```
        for (i = 0; i < MAX_CONN_IND; i++) {
            if (discon->sequence == calls[slot][i]->sequence) {
                t_free(calls[slot][i], T_CALL);
                calls[slot][i] = (struct t_call *) NULL;
            }
        }
        t_free(discon, T_DIS);
        break;
    }
}
```

The arguments are a number (`slot`) and a file descriptor (`fd`). `slot` is the index into the global array `calls`. `calls` has an entry for each transport endpoint. Each entry is an array of `t_call` structures that hold incoming connect requests for the endpoint.

`do_event` calls `t_look` to identify the TLI event on the endpoint specified by `fd`. If the event is a connect request (`T_LISTEN` event) or disconnect request (`T_DISCONNECT` event), the event is processed. Otherwise, the server prints an error message and exits.

For connect requests, `do_event` scans the array of outstanding connect requests for the first free entry. A `t_call` structure is allocated for the entry, and the connect request is received by `t_listen`. The array is large enough to hold the maximum number of outstanding connect requests. The processing of the connect request is deferred.

A disconnect request must correspond to an earlier connect request. `do_event` allocates a `t_discon` structure to receive the request. This structure has the following fields:

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

`udata` contains any user data sent with the disconnect request. `reason` contains a protocol-specific disconnect reason code. `sequence` identifies the connect request that matches the disconnect request.

`t_rcvdis` is called to receive the disconnect request. The array of connect requests is scanned for one that contains the sequence number that matches the `sequence` number in the disconnect request. When the connect request is found, its structure is freed and the entry is set to `NULL`.

When an event is found on a transport endpoint, `service_conn_ind` is called to process all queued connect requests on the endpoint as Code Example 6-12 shows.

Code Example 6-12 Process All Connect Requests

```
service_conn_ind(slot, fd)
{
    int i;

    for (i = 0; i < MAX_CONN_IND; i++) {
        if (calls[slot][i] == (struct t_call *) NULL)
            continue;
        if((conn_fd = t_open( “/dev/tivc”, O_RDWR,
                    (struct t_info *) NULL)) == -1) {
            t_error("open failed");
            exit(14);
        }
        if (t_bind(conn_fd, (struct t_bind *) NULL,
                 (struct t_bind *) NULL) == -1) {
            t_error("t_bind failed");
            exit(15);
        }
        if (t_accept(fd, conn_fd, calls[slot][i]) == -1) {
            if (t_errno == TLOOK) {
                t_close(conn_fd);
                return;
            }
            t_error("t_accept failed");
            exit(16);
        }
        t_free(calls[slot][i], T_CALL);
        calls[slot][i] = (struct t_call *) NULL;
        run_server(fd);
    }
}
```

For each transport endpoint, the array of outstanding connect requests is scanned. For each request, the server opens a responding transport endpoint, binds an address to the endpoint, and accepts the connection on the endpoint. If another event (connect request or disconnect request) arrives before the current request is accepted, `t_accept` fails and sets `t_errno` to `TLOOK`. (You cannot accept an outstanding connect request if any pending connect request events or disconnect request events exist on the transport endpoint.)

If this error occurs, the responding transport endpoint is closed and `service_conn_ind` will return immediately (saving the current connect request for later processing). This causes the server's main processing loop to be entered, and the new event will be discovered by the next call to `poll`. In this way, multiple connect requests may be queued by the user.

Eventually, all events will be processed, and `service_conn_ind` will be able to accept each connect request in turn. Once the connection has been established, the `run_server` routine used by the server in the Code Example 6-3 on page 181 is called to manage the data transfer.

State Transitions

These tables describe all state transitions associated with TLI. First, however, the states and events are described.

TLI States

Table 6-8 defines the states used in TLI state transitions, along with the service types.

Table 6-8 TLI State Transitions and Service Types

State	Description	Service Type
`T_UNIT`	Uninitialized – initial and final state of interface	`T_COTS, COTS_ORD, T_CLTS`
`T_UNBND`	Initialized but not bound	`T_COTS, T_COTS_ORD, T_CLTS`
`T_IDLE`	No connection established	`T_COTS, T_COTS_ORD, T_CLTS`
`T_OUTCON`	Outgoing connection pending for client	`T_COTS, T_COTS_ORD`
`T_INCON`	Incoming connection pending for server	`COTS, T_COTS_ORD`

Table 6-8 TLI State Transitions and Service Types

State	Description	Service Type
T_DATAXFER	Data transfer	T_COTS, T_COTS_ORD
T_OUTREL	Outgoing orderly release (waiting for orderly release request	T_COTS_ORD
T_INREL	Incoming orderly release (waiting to send orderly release request)	T_COTS_ORD

Outgoing Events

The outgoing events described in Table 6-9 correspond to the status returned from the specified transport routines, where these routines send a request or response to the transport provider. In the table, some events (such as `accept`() are distinguished by the context in which they occur. The context is based on the values of the following variables:

- `ocnt` – count of outstanding connect requests
- `fd` – file descriptor of the current transport endpoint
- `resfd` – file descriptor of the transport endpoint where a connection is accepted

Table 6-9 Outgoing Events

Event	Description	Service Type
opened	Successful return of t_open	T_COTS, T_COTS_ORD, T_CLTS
bind	Successful return of t_bind	T_COTS, T_COTS_ORD, T_CLTS
optmgmt	Successful return of t_optmgmt	T_COTS, T_COTS_ORD,T_CLTS
unbind	Successful return of t_unbind	T_COTS, T_COTS_ORD,T_CLTS
closed	Successful return of t_close	T_COTS, T_COTS_ORD,T_CLT
connect1	Successful return of t_connect in synchronous mode	T_COTS, T_COTS_ORD
connect2	TNODATA error on t_connect in asynchronous mode, or TLOOK error due to a disconnect request arriving on the transport endpoint	T_COTS, T_COTS_ORD

Table 6-9 Outgoing Events

Event	Description	Service Type
accept1	Successful return of t_accept with ocnt == 1, fd == resfd	T_COTS, T_COTS_ORD
accept2	Successful return of t_accept with ocnt== 1,fd!= resfd	T_COTS, T_COTS_ORD
accept3	Successful return of t_accept with ocnt > 1	T_COTS, T_COTS_ORD
snd	Successful return of t_snd	T_COTS, T_COTS_ORD
snddis1	Successful return of t_snddis with ocnt <= 1	T_COTS, T_COTS_ORD
snddis2	Successful return of t_snddis with ocnt > 1	T_COTS, T_COTS_ORD
sndrel	Successful return of t_sndrel	T_COTS_ORD
sndudata	Successful return of t_sndudata	T_CLTS

Incoming Events

The incoming events correspond to the successful return of the specified routines. These routines return data or event information from the transport provider. The only incoming event not associated directly with the return of a routine is pass_conn, which occurs when a connection is transferred to another endpoint. The event occurs on the endpoint that is being passed the connection, although no TLI routine is called on the endpoint.

In Table 6-10, the rcvdis events are distinguished by the value of ocnt, the count of outstanding connect requests on the endpoint.

Table 6-10 Incoming Events

Event	Description	Service Type
listen	Successful return of t_listen	T_COTS, T_COTS_ORD
rcvconnect	Successful return of t_rcvconnect	T_COTS,T_COTS_ORD
rcv	Successful return of t_rcv	T_COTS,T_COTS_ORD

Table 6-10 Incoming Events

Event	Description	Service Type
rcvdis1	Successful return of rcvdis1t_rcvdis, onct <= 0	T_COTS, T_COTS_ORD
rcvdis2	Successful return of rcvdis2 t_rcvdis, ocnt == 1	T_COTS, T_COTS_ORD
rcvdis3	Successful return of t_rcvdis with ocnt > 1	T_COTS, T_COTS_ORD
rcvrel	Successful return of t_rcvrel	T_COTS_ORD
rcvudata	Successful return of t_rcvudata	T_CLTS
rcvuderr	Successful return of t_rcvuderr	T_CLTS
pass_conn	Receive a passed connection	T_COTS, T_COTS_ORD

Transport User Actions

Some state transitions (below) have a list of actions the transport user must take. Each action is represented by a digit from the list below:

1. Set the count of outstanding connect requests to zero
2. Increment the count of outstanding connect requests
3. Decrement the count of outstanding connect requests
4. Pass a connection to another transport endpoint as indicated in t_accept

State Tables

The tables describe the TLI state transitions. Each box contains the next state, given the current state (column) and the current event (row). An empty box is an invalid state/event combination. Each box may also have an action list. Actions must be done in the order specified in the box.

The following should be understood when studying the state tables:

- t_close can be called from any state to close an endpoint. A transport address bound to an endpoint gets unbound. If the transport connection is active, the connection is aborted.

- If a transport user issues a function out of sequence, the function fails and `t_errno` is set to `TOUTSTATE`. The state does not change.
- The error codes `TLOOK` or `TNODATA` after `t_connect` can result in state changes described in "Event Handling" on page 183. The state tables assume correct use of TLI.
- Any other transport error does not change the state unless the manual page for the function says otherwise.
- The support functions `t_getinfo`, `t_getstate`, `t_alloc`, `t_free`, `t_sync`, `t_look`, and `t_error` are excluded from the state tables because they do not affect the state.

Table 6-11, Table 6-12, and Table 6-13 show endpoint establishment, data transfer in connectionless mode, and connection establishment / connection release / data transfer in connection mode.

Table 6-11 Connection Establishment State

Event/State	T_UNIT	T_UNBND	T_IDLE
opened	T_UNBND		
bind		T_IDLE[1]	
optmgmt			T_IDLE
unbind			T_UNBND
closed		T_UNINIT	

Table 6-12 Connection Mode State

Event/State	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER	T_OUTREL	T_INREL
connect1	T_DATAXFER					
connect2	T_OUTCON					
rcvconnect		T_DATAXFER				
listen	T_INCON [2]		T_INCON [2]			
accept1			T_DATAXFER [3]			

Table 6-12 Connection Mode State

Event/State	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER	T_OUTREL	T_INREL
accept2			T_IDLE [3] [4]			
accept3			T_INCON [3] [4]			
snd				T_DATAXFER		T_INREL
rcv				T_DATAXFER	T_OUTREL	
snddis1		T_IDLE	T_IDLE [3]	T_IDLE	T_IDLE	T_IDLE
snddis2			T_INCON [3]			
rcvdis1		T_IDLE		T_IDLE	T_IDLE	T_IDLE
rcvdis2			T_IDLE [3]			
rcvdis3			T_INCON [3]			
sndrel				T_OUTREL		T_IDLE
rcvrel				T_INREL	T_IDLE	
pass_conn	T_DATAXFER					

Table 6-13 Connectionless Mode State

Event/State	T_IDLE
snudata	T_IDLE
rcvdata	T_IDLE
rcvuderr	T_IDLE

Guidelines to Protocol Independence

TLI's set of services, common to many transport protocols, offers protocol independence to applications. Not all transport protocols support all TLI services. If software must run in a variety of protocol environments, use only the common services. The following is a list of services that may not be common to all transport protocols.

1. In connection mode service, a transport service data unit (TSDU) may not be supported by all transport providers. Make no assumptions about the preserving logical data boundaries across a connection.
2. Protocol and implementation specific service limits are returned by the `t_open` and `t_getinfo` routines. Use these limits to allocate buffers to store protocol-specific transport addresses and options.
3. Do not send user data with connect requests or disconnect requests, such as `t_connect` and `t_snddis()`. Not all transport protocols work this way.
4. The buffers in the `t_call` structure used for `t_listen` must be large enough to hold any data sent by the client during connection establishment. Use the `T_ALL` argument to `t_alloc` to set maximum buffer sizes to store the address, options, and user data for the current transport provider.
5. Do not test or change options of any TLI routine. These options are specific to the underlying transport protocol. Do not pass options with `t_connect` or `t_sndudata`. In such cases, the transport provider will use default values. Also, a server should use the options returned by `t_listen` to accept a connection.
6. Do not specify a protocol address on `t_bind`. Let the transport provider assign an appropriate address to the transport endpoint. A server should retrieve its address for `t_bind` in such a way that it does not require knowledge of the transport provider's name space.
7. Do not make assumptions about formats of transport addresses. Transport addresses should not be constants in a program. Chapter 5, "Transport Selection and Name-to-Address Mapping, "contains detailed information.
8. The reason codes associated with `t_rcvdis` are protocol-dependent. Do not interpret this information if protocol-independence is important.
9. The `t_rcvuderr` error codes are protocol dependent. Do not interpret this information if protocol independence is a concern.
10. Do not code the names of devices into programs. The device node identifies a particular transport provider and is not protocol independent.

11. Do not use the optional orderly release facility of the connection mode service (provided by `t_sndrel` and `t_rcvrel`) in programs targeted for multiple protocol environments. This facility is not supported by all connection-based transport protocols. Its use can prevent programs from successfully communicating with open systems.

TLI Versus Socket Interfaces

TLI and sockets are different ways of doing the same tasks. Mostly, they provide mechanisms and services that are functionally similar. They do not provide one-to-one compatibility of routines or low-level services. Observe the similarities and differences between the TLI and socket-based interfaces before you decide to port an application.

The following issues are related to transport independence, and may have some bearing on RPC applications:

- *Privileged ports* – Privileged ports are an artifact of the Berkeley Software Distribution (BSD) implementation of the TCP/IP Internet Protocols (TCP and UDP on top of IP). They are not portable. The notion of privileged ports is not supported in the transport-independent environment.
- *Opaque addresses* – 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 that assumes it can discern the host address of a network service must be changed.
- *Broadcast* – There is no transport independent form of broadcast.

6

Socket-to-TLI Equivalents

Table 6-14 shows approximate equivalents between TLI functions and socket functions. The comment field describes the differences. Where there is no comment, either the functions are the same or there is no equivalent function in one or the other interface.

Table 6-14 TLI and Socket Equivalent Functions

TLI function	Socket function	Comments
t_open()	socket()	
–	socketpair()	
t_bind()	bind()	t_bind() sets the queue depth for passive sockets, but bind() doesn't. For sockets, the queue length is specified in the call to listen().
t_optmgmt()	getsockopt()	
	setsockopt()	t_optmgmt() manages only transport options. getsockopt() and setsockopt() can manage options at the transport layer, but also at the socket layer and at the arbitrary protocol layer.
t_unbind()		–
t_close()	close()	
t_getinfo()	getsockopt()	t_getinfo() returns information about the transport. getsockopt() can return information about the transport and the socket.
t_getstate()	-	
t_sync()	-	
t_alloc()	-	
t_free()	-	
t_look()	-	getsockopt with the SO_ERROR option returns the same kind of error information as t_look().
t_error()	perror()	

Table 6-14 TLI and Socket Equivalent Functions

TLI function	Socket function	Comments
t_connect()	connect()	A connect() can be done without first binding the local endpoint. The endpoint must be bound before calling t_connect(). A connect() can be done on a connectionless endpoint to set the default destination address for datagrams. Data may be sent on aconnect().
t_rcvconnect()	-	
t_listen()	listen()	t_listen() waits for connection indications. listen() merely sets the queue depth.
t_accept()	accept()	
t_snd()	send()	
	sendto()	
	sendmsg()	sendto() and sendmsg() operate in connection mode as well as datagram mode.
t_rcv()	recv(),	
	recvfrom(),	
	recvmsg()	recvfrom() and recvmsg() operate in connection mode as well as datagram mode.
t_snddis()	-	
t_rcvdis()	-	
t_sndrel()	shutdown()	
t_rcvrel()	-	
t_sndudata()	sendto()	
	recvmsg()	
t_rcvuderr()	-	
read(), write()	read(), write()	In TLI you must push the tirdwr module before calling read() or write(); in sockets, just callread() or write().

6

Socket Interface 7

The socket interface was introduced in 1981 in the Berkeley 4.2 Software Distribution. It is still an integral part of SunOS releases. This chapter presents some of the alternatives and illustrates them with sample programs. The programs demonstrate both the datagram socket and stream socket communications models.

Sockets are Multithread Safe

The interface described in this chapter is multithread safe. This means that applications that contain socket function calls can be used freely in a multithreaded application.

SunOS Binary Compatibility

There are two major changes from SunOS 4.x that hold true for SunOS 5.x/Solaris 2.x releases. The binary compatibility package allows SunOS 4.x–based dynamically linked socket applications to run in SunOS 5.x.

1. You must explicitly specify the socket library (`-lsocket`) when you compile.
2. All existing socket-based applications should be recompiled with the socket library to run under SunOS 5.x. The differences in the two socket implementations are outlined in "Moving Socket Applications to SunOS 5.x" on page 259.

What Are Sockets

Sockets are a basic component of interprocess (intersystem) communication. The socket interface provides access to transport protocols. A socket is an endpoint of communication to which a name can be bound. A socket has a *type* and one or more associated processes. Sockets exist within communications domains. A socket domain is an abstraction that provides an addressing structure (address family) and a set of protocols. Sockets exchange data only with sockets in the same domain.

UNIX domain sockets are named with UNIX paths. For example, a socket may be named `/dev/foo`. UNIX domain sockets communicate only between processes on a single host.

Internet domain communication uses the TCP/IP suite.

Socket Libraries

The socket interface routines are in a library that must be linked with the application. The libraries `libsocket.so` and `libsocket.a` are contained in `/usr/lib` with the rest of the system service libraries.

Socket Types

Socket types define the communication properties visible to a user. Processes communicate only between sockets of the same type.

There are several types of sockets:

1. A stream socket provides bidirectional, reliable, sequenced, and unduplicated flow of data with no record boundaries. Stream communication is similar to a telephone conversation. Under Internet (AF_INET), the socket type is SOCK_STREAM which uses TCP.

2. A datagram socket supports bidirectional flow of messages. A process on a datagram socket may receive messages in a different order from the sending sequence and may receive duplicate messages. Record boundaries in the data are preserved. Datagram communication is similar to mailing a letter. Under Internet (`AF_INET`), the socket type is `SOCK_DGRAM`, which uses User Datagram Protocol (UDP).

3. A raw socket provides access to the underlying communication protocols. These sockets are normally datagram oriented, although their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not for most applications. They are provided to support developing new communication protocols or for access to more esoteric facilities of an existing protocol. For more information on raw sockets, see "Advanced Topics" on page 246.

Socket Tutorial

This section covers the basic methodologies of using sockets.

Socket Creation

The `socket()` call creates a socket,

```
s = socket(domain, type, protocol);
```

in the specified domain and of the specified type. If the protocol is unspecified (a value of `0`), the system selects a protocol that supports the requested socket type. The socket handle (a descriptor) is returned.

The domain is specified by one of the constants defined in `<sys/socket.h>`. For the UNIX domain the constant is `AF_UNIX`. For the Internet domain it is `AF_INET`. Constants named AF_*suite* specify the address format to use in interpreting names.

Socket types are defined in `<sys/socket.h>`. `SOCK_STREAM, SOCK_DGRAM`, or `SOCK_RAW` is supported by `AF_INET` and `AF_UNIX`. The following creates a stream socket in the Internet domain:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call results in a stream socket with the TCP protocol providing the underlying communication. A datagram socket for intramachine use is created by:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

Use the default protocol (used when the `protocol` argument is `0`) in most situations. You can specify a protocol other than the default, as described in "Advanced Topics" on page 246.

Binding Local Names

A socket is created with no name. A remote process has no way to refer to a socket until an address is bound to it. Communicating processes are connected through addresses. In the Internet domain, a connection is composed of local and remote addresses, and local and remote ports. In the UNIX domain, a connection is composed of (usually) one or two path names. In most domains, connections must be unique.

In the Internet domain, there may never be duplicate ordered sets, such as: `<protocol, local address, local port, foreign address, foreign port>`. UNIX domain sockets need not always be bound to a name, but when bound there may never be duplicate ordered sets such as: `<local pathname, foreign pathname>`. The path names may not refer to existing files.

The `bind`() call allows a process to specify half of an association, such as: `<local address, local port> (or <local pathname>)` while `connect`() and `accept`() complete a socket's association. The `bind`() system call is used as follows:

`bind (`*s, name, namelen*`);`

`s` is the socket handle. The bound name is a byte string that is interpreted by the supporting protocol(s). Internet domain names contain an Internet address and port number. UNIX domain names contain a path name and a family.

Code Example 7-1 binds the name `/tmp/foo` to a UNIX domain socket.

Code Example 7-1 Bind Name to Socket

```
#include <sys/un.h>
 ...
struct sockaddr_un addr;
 ...
strcpy(addr.sun_path, "/tmp/foo");
```

```
addr.sun_family = AF_UNIX;
bind (s, (struct sockaddr *) &addr,
     strlen(addr.sun_path) + sizeof (addr.sun_family));
```

Note that in determining the size of an AF_UNIX socket address, null bytes are not counted, which is why `strlen()` is used.

The file name referred to in `addr.sun_path` is created as a socket in the system file name space. The caller must have write permission in the directory where `addr.sun_path` is created. The file should be deleted by the caller when it is no longer needed. AF_UNIX sockets can be deleted with `unlink`().

Binding an Internet address is more complicated. The call is similar:

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
 ...
bind (s, (struct sockaddr *) &sin, sizeof sin);
```

The content of the address `sin` is described in “Address Binding” on page 252, where Internet address bindings are discussed.

Connection Establishment

Connection establishment is usually asymmetric, with one process acting as the client and the other as the server. The server binds a socket to a well-known address associated with the service and blocks on its socket for a connect request. An unrelated process can then connect to the server. The client requests services from the server by initiating a connection to the server’s socket. On the client side, the `connect()` call initiates a connection. In the UNIX domain, this might appear as:

```
struct sockaddr_un server;
server.sun.family = AF_UNIX;
 ...
connect(s, (struct sockaddr *)&server,
        strlen(server.sun_path) + sizeof (server.sun_family));
```

while in the Internet domain it might be:

```
struct sockaddr_in server;
 ...
connect(s, (struct sockaddr *)&server, sizeof server);
```

If the client's socket is unbound at the time of the connect call, the system automatically selects and binds a name to the socket. See "Signals and Process Group ID" on page 250. This is the usual way that local addresses are bound to a socket on the client side.

In the examples that follow, only AF_INET sockets are described.

To receive a client's connection, a server must perform two steps after binding its socket. The first is to indicate how many connections can be queued. The second step is to accept a connection:

```
struct sockaddr_in from;
 ...
listen(s, 5);                /* Allow queue of 5 connections */
fromlen = sizeof(from);
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```

s is the socket bound to the address to which the connection request is sent. The second parameter of listen() specifies the maximum number of outstanding connections that may be queued. from is a structure that is filled with the address of the client. A NULL pointer may be passed. fromlen is the length of the structure. (In the UNIX domain, from is declared a struct sockaddr_un.)

accept() normally blocks. accept() returns a new socket descriptor that is connected to the requesting client. The value of from is changed to the actual size of the address.

There is no way for a server to indicate that it will accept connections only from specific addresses. The server can check the from-address returned by accept() and close a connection with an unacceptable client. A server can accept connections on more than one socket, or avoid blocking on the accept call. These techniques are presented in "Advanced Topics" on page 246.

Connection Errors

An error is returned if the connection is unsuccessful (however, an address bound by the system remains). Otherwise, the socket is associated with the server and data transfer may begin.

Table 7-1 lists some of the more common errors returned when a connection attempt fails.

Table 7-1 Socket Connection Errors

Socket Errors	Error Description
ENOBUFS	Lack of memory available to support the call.
EPROTONOSUPPORT	Request for an unknown protocol.
EPROTOTYPE	Request for an unsupported type of socket.
ETIMEDOUT	No connection established in specified time. This happens when the destination host is down or when problems in the network result in lost transmissions.
ECONNREFUSED	The host refused service. This happens when a server process is not present at the requested address.
ENETDOWN or EHOSTDOWN	These errors are caused by status information delivered by the underlying communication interface.
ENETUNREACH or EHOSTUNREACH	These operational errors can occur either because there is no route to the network or host, or because of status information returned by intermediate gateways or switching nodes. The status returned is not always sufficient to distinguish between a network that is down and a host that is down.

Data Transfer

This section describes the functions to send and receive data. You can send or receive a message with the normal `read()` and `write()` system calls:

```
write(s, buf, sizeof buf);
read(s, buf, sizeof buf);
```

Or the calls `send`() and `recv`() can be used:

```
send(s, buf, sizeof buf, flags);
recv(s, buf, sizeof buf, flags);
```

`send`() and `recv`() are very similar to `read`() and `write`(), but the `flags` argument is important. The flags, defined in `<sys/socket.h>`, can be specified as a nonzero value if one or more of the following is required:

MSG_OOB	send and receive out-of-band data
MSG_PEEK	look at data without reading
MSG_DONTROUTE	send data without routing packets

Out-of-band data is specific to stream sockets. When MSG_PEEK is specified with a recv( ) call, any data present is returned to the user but treated as still unread. The next read() or recv() call on the socket returns the same data. The option to send data without routing applied to the outgoing packets is currently used only by the routing table management process and is unlikely to be interesting to most users.

Closing Sockets

A SOCK_STREAM socket can be discarded by a close() system call. If data is queued to a socket that promises reliable delivery after a close, the protocol continues to try to transfer the data. If the data is still undelivered after an arbitrary period, it is discarded.

shutdown() closes SOCK_STREAM sockets gracefully. Both processes can acknowledge that they are no longer sending. This call has the form:

```
shutdown(s, how);
```

where how is 0 disallows further receives, 1 disallows further sends, and 2 disallows both.

Connecting Stream Sockets

The next two code examples illustrate initiating and accepting an Internet domain stream connection.

To initiate a connection, the client program in Code Example 7-2 creates a stream socket and calls connect(), specifying the address of the socket to connect to. If the target socket exists and the request is accepted, the connection is complete and the program can send data. Data are delivered in sequence with no message boundaries. The connection is destroyed when either socket is closed.

Code Example 7-2 Internet Domain Stream Connection (Client)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <netdb.h>
#include <stdio.h>

#define DATA “Half a league, half a league . . ."

/*
 * This program creates a socket and initiates a connection with the
 * socket given in the command line. Some data are sent over the
 * connection and then the socket is closed, ending the connection.
 * The form of the command line is: streamwrite hostname portnumber
 */
main(argc, argv)
    int argc;
    char *argv[];
{
    int sock;
    struct sockaddr_in server;
    struct hostent *hp, *gethostbyname();
    char buf[1024];

    /* Create socket. */
    sock = socket( AF_INET, SOCK_STREAM, 0 );
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
    /* Connect socket using name specified by command line. */
    server.sin_family = AF_INET;
    hp = gethostbyname(argv[1] );
/*
 * gethostbyname returns a structure including the network address
 * of the specified host.
 */
    if (hp == (struct hostent *) 0) {
        fprintf(stderr, "%s: unknown host\n", argv[1]);
        exit(2);
    }
    memcpy((char *) &server.sin_addr, (char *) hp->h_addr,
        hp->h_length);
    server.sin_port = htons(atoi( argv[2]));
    if (connect(sock, (struct sockaddr *) &server, sizeof server)
        == -1) {
        perror("connecting stream socket");
        exit(1);
    }
    if (write( sock, DATA, sizeof DATA ) == -1)
```

```
        perror("writing on stream socket");
    close(sock);
    exit(0);
}
```

The program in Code Example 7-3 is a server. It creates a socket and binds a name to it, then displays the port number. The program calls `listen`() to mark the socket ready to accept connection requests and initialize a queue for the requests. The rest of the program is an infinite loop. Each pass of the loop accepts a new connection and removes it from the queue, creating a new socket. The server reads and displays the messages from the socket and closes it. The use of `INADDR_ANY` is explained in "Address Binding" on page 252.

Code Example 7-3 Accepting an Internet Stream Connection (Server)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1

/*
 * This program creates a socket and then begins an infinite loop.
 * Each time through the loop it accepts a connection and prints
 * data from it. When the connection breaks, or the client closes
 * the connection, the program accepts a new connection.
 */

main()
{
    int sock, length;
    struct sockaddr_in server;
    int msgsock;
    char buf[1024];
    int rval;

    /* Create socket. */
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
    /* Bind socket using wildcards.*/
```

```
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = 0;
    if (bind(sock, (struct sockaddr *) &server, sizeof server)
        == -1)
        perror("binding stream socket");
        exit(1);
    }
    /* Find out assigned port number and print it out. */
    length = sizeof server;
    if (getsockname(sock,(struct sockaddr *) &server,&length)
         == -1) {
        perror("getting socket name");
        exit(1);
    }
    printf("Socket port #%d\n", ntohs(server.sin_port));
    /* Start accepting connections. */
    listen(sock, 5);
    do {
        msgsock = accept(sock,(struct sockaddr *) 0,(int *) 0);
        if (msgsock == -1
            perror("accept");
        else do {
            memset(buf, 0, sizeof buf);
            if ((rval = read(msgsock,buf, 1024)) == -1)
                perror("reading stream message");
            if (rval == 0)
                printf("Ending connection\n");
            else
                printf("-->%s\n", buf);
        } while (rval != 0);
        close(msgsock);
    } while(TRUE);
    /*
     * Since this program has an infinite loop, the socket "sock" is
     * never explicitly closed. However, all sockets will be closed
     * automatically when a process is killed or terminates normally.
     */
     exit(0);
}
```

7

Datagram Sockets

A datagram socket provides a symmetric data exchange interface. There is no requirement for connection establishment. Each message carries the destination address.

Datagram sockets are created as described in “Socket Creation” on page 221. If a particular local address is needed, the `bind()` operation must precede the first data transmission. Otherwise, the system sets the local address and/or port when data is first sent. To send data, the `sendto()` call is used:

```
sendto(s, buf, buflen, flags, (struct sockaddr *) &to, tolen);
```

The `s`, `buf`, `buflen`, and `flags` parameters are the same as in connection-oriented sockets. The `to` and `tolen` values indicate the address of the intended recipient of the message. A locally detected error condition (such as an unreachable network) causes a return of `-1` and `errno` to be set to the error number.

To receive messages on a datagram socket, the `recvfrom()` call is used:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *) &from,
          &fromlen);
```

Before the call, `fromlen` is set to the size of the `from` buffer. On return it is set to the size of the address from which the datagram was received.

Datagram sockets can also use the `connect`() call to associate a socket with a specific destination address. It can then use the `send`() call. Any data sent on the socket without explicitly specifying a destination address is addressed to the connected peer, and only data received from that peer is delivered. Only one connected address is permitted for one socket at a time. A second `connect`() call changes the destination address. Connect requests on datagram sockets return immediately. The system simply records the peer’s address.

`accept()` and `listen()` are not used with datagram sockets.

While a datagram socket is connected, errors from previous `send()` calls may be returned asynchronously. These errors can be reported on subsequent operations on the socket, or an option of `getsockopt`, `SO_ERROR`, can be used to interrogate the error status. Code Example 7-4 shows how to read an Internet call, and Code Example 7-5 shows how to send an Internet call.

Code Example 7-4 Reading Internet Domain Datagrams

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

/*
 * The include file <netinet/in.h> defines sockaddr_in as:
 *  struct sockaddr_in {
 *      short      sin_family;
 *      u_short    sin_port;
 *      struct     in_addr sin_addr;
 *      char       sin_zero[8];
 * };
 * This program creates a datagram socket, binds a name to it, then
 * reads from the socket.
 */

main()
{
    int sock, length;
    struct sockaddr_in name;
    char buf[1024];

    /* Create socket from which to read. */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
    }
    /* Create name with wildcards. */
    name.sin_family = AF_INET;
    name.sin_addr.s_addr = INADDR_ANY;
    name.sin_port = 0;
    if (bind(sock,(struct sockaddr *)&name, sizeof name) == -1) {
        perror("binding datagram socket");
        exit(1);
    }
    /* Find assigned port value and print it out. */
    length = sizeof(name);
    if (getsockname(sock,(struct sockaddr *) &name, &length)
        == -1) {
        perror("getting socket name");
        exit(1);
    }
```

```
    printf("Socket port #%d\n", ntohs( name.sin_port));
    /* Read from the socket. */
    if ( read(sock, buf, 1024) == -1 )
        perror("receiving datagram packet");
    printf("-->%s\n", buf);
    close(sock);
    exit(0);
}
```

Code Example 7-5 Sending an Internet Domain Datagram

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "The sea is calm, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the
 * command line arguments. The form of the command line is:
 * dgramsend hostname portnumber
 */
main(argc, argv)
    int argc;
    char *argv[];
{
    int sock;
    struct sockaddr_in name;
    struct hostent *hp, *gethostbyname();

    /* Create socket on which to send. */
    sock = socket(AF_INET,SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
    }
    /*
     * Construct name, with no wildcards, of the socket to ‘‘send’’
     * to. gethostbyname returns a structure including the network
     * address of the specified host. The port number is taken from
     * the command line.
     */
    hp = gethostbyname(argv[1]);
    if (hp == (struct hostent *) 0) {
```

```
        fprintf(stderr, "%s: unknown host\n", argv[1]);
        exit(2);
    }
    memcpy((char *) &name.sin_addr, (char *) hp->h_addr,
            hp->h_length);
    name.sin_family = AF_INET;
    name.sin_port = htons( atoi( argv[2] ));
    /* Send message. */
    if (sendto(sock,DATA, sizeof DATA ,0,
        (struct sockaddr *) &name,sizeof name) == -1)
        perror("sending datagram message");
    close(sock);
    exit(0);
}
```

Input/Output Multiplexing

Requests can be multiplexed among multiple sockets or files. The `select()` call is used to do this:

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/select.h>
 ...
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

The first argument of `select()` is the number of file descriptors in the lists pointed to by the next three arguments.

The second, third, and fourth arguments of `select()` are pointers to three sets of file descriptors: a set of descriptors to read on, a set to write on, and a set on which exception conditions are accepted. Out-of-band data is the only exceptional condition. Any of these pointers can be a properly cast null. Each set is a structure containing an array of long integer bit masks. The size of the array is set by `FD_SETSIZE` (defined in `select.h`). The array is long enough to hold one bit for each `FD_SETSIZE` file descriptor.

The macros `FD_SET(`*fd*, &*mask*) and `FD_CLR(`*fd*, &*mask*) add and delete, respectively, the file descriptor `fd` in the set `mask`. The set should be zeroed before use, and the macro `FD_ZERO`(&*mask*) clears the set `mask`.

A time-out value may be specified. If the `timeout` pointer is `NULL`, `select`() blocks until a descriptor is selectable, or until a signal is received. If the fields in `timeout` are set to `0`, `select`() polls and returns immediately.

`select`() normally returns the number of file descriptors selected. `select`() returns a `0` if the time-out has expired. `select`() returns `-1` for an error or interrupt with the error number in `errno` and the file descriptor masks unchanged.

For a successful return, the three sets indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending.

Test the status of a file descriptor in a select mask with the `FD_ISSET`(*fd*, &*mask*) macro. It returns a nonzero value if `fd` is in the set `mask`, and `0` if it is not. Use `select`() followed by a `FD_ISSET`(*fd*, &*mask*) macro on the read set to check for queued connect requests on a socket.

Code Example 7-6 shows how to select on a "listening" socket for readability to determine when a new connection can be picked up with a call to `accept`(). The program accepts connection requests, reads data, and disconnects on a single socket.

Code Example 7-6 Check for Pending Connections With `select()`

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1

/*
 * This program uses select to check that someone is
 * trying to connect before calling accept.
 */

main()
{
    int sock, length;
    struct sockaddr_in server;
    int msgsock;
    char buf[1024];
    int rval;
```

```
    fd_set ready;
    struct timeval to;

    /* Open a socket and bind it as in previous examples. */

    /* Start accepting connections. */
    listen(sock, 5);
    do {
        FD_ZERO(&ready);
        FD_SET(sock, &ready);
        to.tv_sec = 5;
        to.tv_usec = 0;
        if (select(1, &ready, (fd_set *)0, (fd_set *)0, &to) == -1) {
            perror("select");
            continue;
        }
        if (FD_ISSET(sock, &ready)) {
            msgsock = accept(sock, (struct sockaddr *)0,
             (int *)0);
            if (msgsock == -1)
                perror("accept");
            else do {
                memset(buf, 0, sizeof buf);
                if ((rval = read(msgsock, buf, 1024)) == -1)
                    perror("reading stream message");
                else if (rval == 0)
                    printf("Ending connection\n");
                else
                    printf("-->%s\n", buf);
            } while (rval > 0);
            close(msgsock);
        } else
            printf("Do something else\n");
    } while (TRUE);
    exit(0);
}
```

In previous versions of the `select`() routine, its arguments were pointers to integers instead of pointers to `fd_set`s. This style of call still works if the number of file descriptors is smaller than the number of bits in an integer.

`select()` provides a synchronous multiplexing scheme. The `SIGIO` and `SIGURG` signals described in "Advanced Topics" on page 246 provide asynchronous notification of output completion, input availability, and exceptional conditions.

7

Standard Routines

You may need to locate and construct network addresses. This section describes the new routines that manipulate network addresses. Unless otherwise stated, functions presented in this section apply only to the Internet domain.

Locating a service on a remote host requires many levels of mapping before client and server communicate. A service has a name for human use. The service and host names must be translated to network addresses. Finally, the address is used to locate and route to the host. The specifics of the mappings may vary between network architectures. Preferably, a network will not require that hosts be named, thus protecting the identity of their physical locations. It is more flexible to discover the location of the host when it is addressed.

Standard routines map host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers, and the appropriate protocol to use in communicating with the server process. The file `<netdb.h>` must be included when using any of these routines.

Host Names

An Internet host name to address mapping is represented by the `hostent` structure:

```
struct hostent {
    char    *h_name;          /* official name of host */
    char    **h_aliases;      /* alias list */
    int     h_addrtype;       /* hostaddrtype(e.g.,AF_INET) */
    int     h_length;         /* length of address */
    char    **h_addr_list;    /* list of addrs, null terminated */
};
/*1st addr, net byte order*/
#define h_addr h_addr_list[0]
```

`gethostbyname`() maps an Internet host name to a `hostent` structure.
`gethostbyaddr`() maps an Internet host address to a `hostent` structure.
`inet_ntoa`() maps an Internet host address to a displayable string.

The routines return a `hostent` structure containing the name of the host, its aliases, the address type (address family), and a `NULL`-terminated list of variable length addresses. The list of addresses is required because a host can have many addresses. The `h_addr` definition is for backward compatibility, and is the first address in the list of addresses in the `hostent` structure.

Network Names

There are routines to map network names to numbers, and back. These routines return a `netent` structure:

```
/*
 * Assumes that a network number fits in 32 bits.
 */
struct netent {
    char    *n_name;     /* official name of net */
    char    **n_aliases; /* alias list */
    int     n_addrtype;  /* net address type */
    int     n_net;       /* net number, host byte order */
};
```

`getnetbyname()`, `getnetbyaddr()`, and `getnetent()` are the network counterparts to the host routines described above.

Protocol Names

The `protoent` structure defines the protocol-name mapping used with `getprotobyname`(), `getprotobynumber`(), and `getprotoent`():

```
struct protoent {
    char    *p_name;     /* official protocol name */
    char    **p_aliases; /* alias list */
    int     p_proto;     /* protocol number */
};
```

In the UNIX domain, no protocol database exists.

Service Names

An Internet domain service resides at a specific, well-known port and uses a particular protocol. A service name to port number mapping is described by the `servent` structure:

```
struct servent {
    char    *s_name;     /* official service name */
    char    **s_aliases; /* alias list */
    int     s_port;      /* port number, network byte order */
    char    *s_proto;    /* protocol to use */
};
```

`getservbyname()` maps service names and, optionally, a qualifying protocol to a `servent` structure. The call

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification of a telnet server using any protocol. The call

```
sp = getservbyname("telnet", “tcp”);
```

returns the telnet server that uses the TCP protocol. `getservbyport()` and `getservent()` are also provided. `getservbyport()` has an interface similar to that of `getservbyname()`; an optional protocol name may be specified to qualify lookups.

Other Routines

In addition to address-related database routines, there are several other routines that simplify manipulating names and addresses. Table 7-2 summarizes the routines for manipulating variable-length byte strings and byte-swapping network addresses and values.

Table 7-2 Run-Time Library Routines

Call	**Synopsis**
`memcmp`(*s1*, *s2*, *n*)	Compares byte-strings; `0` if same, not `0` otherwise
`memcpy`(*s1*, *s2*, *n*)	Copies n bytes from *s2* to *s1*
`memset`(*base*, *value*, *n*)	Sets *n* bytes to `value` starting at `base`
`htonl`(*val*)	32-bit quantity from host into network byte order
`htons`(*val*)	16-bit quantity from host into network byte order
`ntohl`(*val*)	32-bit quantity from network into host byte order
`ntohs`(*val*)	16-bit quantity from network into host byte order

The byte-swapping routines are provided because the operating system expects addresses to be supplied in network order. On some architectures the host byte ordering is different from network byte order, so, programs must sometimes byte-swap values. Routines that return network addresses do so in network order. There are byte-swapping problems only when interpreting network addresses. For example, the following code formats a TCP or UDP port:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On certain machines, where these routines are not needed, they are defined as null macros.

The Client/Server Model

The most common form of distributed application is the client/server model. In this scheme, client processes request services from a server process. In this section we consider some of the problems in developing client and server applications.

The client and server require a set of conventions to communicate. This is a protocol. The protocol may be symmetric or asymmetric. In a symmetric protocol either side can play the master or slave role. In an asymmetric protocol one side is the master and the other is the slave. A symmetric example is the Internet remote terminal-emulation protocol. An asymmetric example is the Internet file-transfer protocol.

A server process normally listens at a known address for service requests. When a request is received, the server is unblocked and does any legitimate actions the client requests.

An alternate scheme is a service server that can eliminate dormant server processes. An example is `inetd`, the Internet super-server. `inetd` listens at a variety of ports, determined at start up by reading a configuration file. When a connection is requested on an `inetd` serviced port, `inetd` spawns the appropriate server to serve the client. Clients are unaware that an intermediary has played any part in the connection. `inetd` is described in more detail in “Advanced Topics” on page 246.

7

Servers

Most servers are accessed at well-known Internet addresses or UNIX domain names. Code Example 7-7 illustrates the main loop of a remote-login server.

Code Example 7-7 Remote Login Server

```
main(argc, argv)
    int argc;
    char *argv[];
{
    int f;
    struct sockaddr_in from;
    struct sockaddr_in sin;
    struct servent *sp;

    sp = getservbyname("login", "tcp");
    if (sp == (struct servent *) NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service");
        exit(1);
    }
    ...
#ifndef DEBUG
    /* Disassociate server from controlling terminal. */
    ...
#endif
    sin.sin_port = sp->s_port;/* Restricted port */
    sin.sin_addr.s_addr = INADDR_ANY;
    ...
    f = socket(AF_INET, SOCK_STREAM, 0);
    ...
    if (bind( f, (struct sockaddr *) &sin, sizeof sin ) == -1) {
        ...
    }
    ...
    listen(f, 5);
    while (TRUE) {
        int g, len = sizeof from;
        g = accept(f, (struct sockaddr *) &from, &len);
        if (g == -1) {
            if (errno != EINTR)
                syslog(LOG_ERR, "rlogind: accept: %m"");
            continue;
        }
        if (fork() == 0) {
            close(f);
```

```
            doit(g, &from);
        }
        close(g);
    }
    exit(0);
}
```

First, the server gets its service definition, as Code Example 7-8 shows.

Code Example 7-8 Remote Login Server: Step 1

```
sp = getservbyname("login", "tcp");
if (sp == (struct servent *) NULL) {
    fprintf(stderr, "rlogind: tcp/login: unknown service\n");
    exit(1);
}
```

The result from `getservbyname()` is used later to define the Internet port at which the program listens for service requests. Some standard port numbers are in `/usr/include/netinet/in.h`.

In the non-`DEBUG` mode of operation, the server dissociates from the controlling terminal of its invoker, shown in Code Example 7-9.

Code Example 7-9 Dissociating from the Controlling Terminal

```
    (void) close(0);
    (void) close(1);
    (void) close(2);
    (void) open("/", O_RDONLY);
    (void) dup2(0, 1);
    (void) dup2(0, 2);
    setsid();
```

This prevents the server from receiving signals to the process group of the controlling terminal. Once a server has dissociated itself, it cannot send reports of errors to a terminal and must log errors with `syslog()`.

A server next creates a socket and listens for service requests. `bind()` insures that the server listens at the expected location. (The remote login server listens at a restricted port number, so it runs as super-user.)

Code Example 7-10 illustrates the main body of the loop.

Code Example 7-10 Remote Login Server: Main Body

```
while(TRUE) {
    int g, len = sizeof(from);
    if (g = accept(f, (struct sockaddr *) &from, &len) == -1) {
        if (errno != EINTR)
            syslog(LOG_ERR, "rlogind: accept: %m");
        continue;
    }
    if (fork() == 0) {                              /* Child */
        close(f);
        doit(g, &from);
    }
    close(g);                                       /* Parent */
}
```

`accept()` blocks until a client requests service. `accept`() returns a failure indication if it is interrupted by a signal such as `SIGCHLD`. The return value from `accept()` is checked and an error is logged with `syslog()` if an error has occurred.

The server then forks a child process and invokes the main body of the remote login protocol processing. The socket used by the parent to queue connection requests is closed in the child. The socket created by `accept()` is closed in the parent. The address of the client is passed to `doit()` for authenticating clients.

Clients

This section describes the steps taken by the client remote login process. As in the server, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == (struct servent *) NULL) {
    fprintf(stderr,"rlogin: tcp/login: unknown service");
    exit(1);
}
```

Next, the destination host is looked up with a `gethostbyname()` call:

```
hp = gethostbyname(argv[1]);
if (hp == (struct hostent *) NULL) {
    fprintf(stderr, "rlogin: %s: unknown host", argv[1]);
    exit(2);
}
```

Then, connect to the server at the requested host and start the remote login protocol. The address buffer is cleared and filled with the Internet address of the foreign host and the port number at which the login server listens:

```
memset((char *) &server, 0, sizeof server);
memcpy((char*) &server.sin_addr,hp->h_addr,hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created, and a connection initiated. `connect()` implicitly does a `bind()`, since `s` is unbound.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
    perror("rlogin: socket");
    exit(3);
}
 ...
if (connect(s, (struct sockaddr *) &server, sizeof server) < 0) {
    perror("rlogin: connect");
    exit(4);
}
```

Connectionless Servers

Some services use datagram sockets. The `rwho` service provides status information on hosts connected to a local area network. (Avoid running `in.rwho`; it causes heavy network traffic.) This service requires the ability to broadcast information to all hosts connected to a particular network. It is an example of datagram socket use.

A user on a host running the `rwho` server may get the current status of another host with `ruptime`. Typical output is illustrated in Code Example 7-11.

Code Example 7-11 Output of `ruptime` Program

```
itchy up 9:45, 5 users, load 1.15, 1.39, 1.31
scratchy up 2+12:04, 8 users, load 4.67, 5.13, 4.59
click up 10:10, 0 users, load 0.27, 0.15, 0.14
clack up 2+06:28, 9 users, load 1.04, 1.20, 1.65
ezekiel up 25+09:48, 0 users, load 1.49, 1.43, 1.41
dandy 5+00:05, 0 users, load 1.51, 1.54, 1.56
peninsula down 0:24
wood down 17:04
carpediem down 16:09
chances up 2+15:57, 3 users, load 1.52, 1.81, 1.86
```

7

Status information is periodically broadcast by the `rwho` server processes on each host. The server process also receives the status information and updates a database. This database is interpreted for the status of each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

Use of broadcast is fairly inefficient, since a lot of net traffic is generated. Unless the service is used widely and frequently, the expense of periodic broadcasts outweighs the simplicity.

A simplified version of the `rwho` server is shown in Code Example 7-12. It does two tasks: receives status information broadcast by other hosts on the network and supplies the status of its host. The first task is done in the main loop of the program: Packets received at the `rwho` port are checked to be sure they were sent by another `rwho` server process, and are stamped with the arrival time. They then update a file with the status of the host. When a host has not been heard from for an extended time, the database routines assume the host is down and logs it. This application is prone to error, as a server may be down while a host is up.

Code Example 7-12 `rwho` Server

```
main()
{
    ...
    sp = getservbyname("who", "udp");
    net = getnetbyname("localnet");
    sin.sin_addr = inet_makeaddr(net->n_net, INADDR_ANY);
    sin.sin_port = sp->s_port;
    ...
    s = socket(AF_INET, SOCK_DGRAM, 0);
    ...
    on = 1;
    if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof on)
              == -1) {
        syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
        exit(1);
    }
    bind(s, (struct sockaddr *) &sin, sizeof sin);
    ...
    signal(SIGALRM, onalrm);
    onalrm();
    while(1) {
        struct whod wd;
```

```
        int cc, whod, len = sizeof from;
        cc = recvfrom(s, (char *) &wd, sizeof(struct whod), 0,
             (struct sockaddr *) &from, &len);
        if (cc <= 0) {
            if (cc == -1 && errno != EINTR)
                syslog(LOG_ERR, “rwhod: recv: %m"");
            continue;
        }
        if (from.sin_port != sp->s_port) {
            syslog(LOG_ERR, "rwhod: %d: bad from port",
                   ntohs(from.sin_port));
            continue;
        }
        ...
        if (!verify( wd.wd_hostname)) {
            syslog(LOG_ERR, "rwhod: bad host name from %x"",
                   ntohl(from.sin_addr.s_addr));
            continue;
        }
        (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
        whod = open(path, O_WRONLY|O_CREAT|O_TRUNC, 0666);
        ...
        (void) time(&wd.wd_recvtime);
        (void) write(whod, (char *) &wd, cc);
        (void) close(whod);
    }
    exit(0);
}
```

The second server task is to supply the status of its host. This requires periodically acquiring system status information, packaging it in a message, and broadcasting it on the local network for other `rwho` servers to hear. This task is run by a timer and triggered with a signal. Locating the system status information is involved but uninteresting.

Status information is broadcast on the local network. For networks that do not support broadcast, another scheme must be used.

It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe problem. The system isolates host-specific data from applications by providing system calls that return the required data. (For example, `uname`() returns the host’s official name.) The `ioctl()` call lets you find the networks to which a host is directly connected. A local network broadcasting mechanism has been implemented at

the socket level. Combining these two features lets a process broadcast on any directly connected local network that supports broadcasting in a site-independent manner. This solves the problem of deciding how to propagate status with `rwho`, or more generally in broadcasting. Such status is broadcast to connected networks at the socket level, where the connected networks have been obtained through the appropriate `ioctl()` calls. “Broadcasting and Determining Network Configuration” on page 254 details the specifics of broadcasting.

Advanced Topics

For most programmers, the mechanisms already described are enough to build distributed applications. Others will need some of the features in this section.

Out-of-Band Data

The stream socket abstraction includes out-of-band data. Out-of-band data is a logically independent transmission channel between a pair of connected stream sockets. Out-of-band data is delivered independently of normal data. The out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message can contain at least one byte of data, and at least one message may be pending delivery at any time.

For communications protocols that support only in-band signaling (that is, urgent data is delivered in sequence with normal data), the message is extracted from the normal data stream and stored separately. This lets users choose between receiving the urgent data in order and receiving it out of sequence, without having to buffer the intervening data.

You can peek (with `MSG_PEEK`) at out-of-band data. If the socket has a process group, a `SIGURG` signal is generated when the protocol is notified of its existence. A process can set the process group or process id to be informed by `SIGURG` with the appropriate `fcntl`() call, as described in “Interrupt Driven Socket I/O” on page 250 for `SIGIO`. If multiple sockets have out-of-band data waiting delivery, a `select()` call for exceptional conditions can be used to determine the sockets with such data pending.

A logical mark is placed in the data stream at the point at which the out-of-band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal is received, all data up to the mark in the data stream is discarded.

To send an out-of-band message, the MSG_OOB flag is applied to send() or sendto(). To receive out-of-band data, specify MSG_OOB to recvfrom() or recv() (unless out-of-band data is taken in line, in which case the MSG_OOB flag is not needed). The SIOCATMARK ioctl tells whether the read pointer currently points at the mark in the data stream:

```
ioctl(s, SIOCATMARK, &yes);
```

If yes is 1 on return, the next read returns data after the mark. Otherwise, assuming out-of-band data has arrived, the next read provides data sent by the client before sending the out-of-band signal. The routine in the remote login process that flushes output on receipt of an interrupt or quit signal is shown in Code Example 7-13. This code reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

Code Example 7-13 Flushing Terminal I/O on Receipt of Out-of-Band Data

```
#include <sys/ioctl.h>
#include <sys/file.h>

...

oob()
{
    int out = FWRITE;
    char waste[BUFSIZ];
    int mark;

    /* flush local terminal output */
    ioctl(1, TIOCFLUSH, (char *) &out);
    while(1) {
        if (ioctl(rem, SIOCATMARK, &mark) == -1) {
            perror("ioctl");
            break;
        }
        if (mark)
            break;
        (void) read(rem, waste, sizeof waste);
    }
    if (recv(rem, &mark, 1, MSG_OOB) == -1) {
        perror("recv");
        ...
    }
    ...
}
```

A process can also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (for example, TCP, the protocol used to provide socket streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a `recv()` is done with the `MSG_OOB` flag. In that case, the call returns the error of `EWOULDBLOCK`. Also, there may be enough in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data before the urgent data can be delivered.

There is also a facility to retain the position of urgent in-line data in the socket stream. This is available as a socket-level option, `SO_OOBINLINE`. See the `setsockopt(3N)` manpage for usage. With this option, the position of urgent data (the mark) is retained, but the urgent data immediately follows the mark in the normal data stream returned without the `MSG_OOB` flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data are lost.

Nonblocking Sockets

Some applications require sockets that do not block. For example, requests that cannot complete immediately and would cause the process to be suspended (awaiting completion) are not executed. An error code would be returned. Once a socket is created and any connection to another socket is made, it may be made nonblocking by `fcntl()` as Code Example 7-14 shows.

Code Example 7-14 Set Nonblocking Socket

```
#include <fcntl.h>
#include <sys/file.h>
 ...
int fileflags;
int s;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
if (fileflags = fcntl(s, F_GETFL, 0) == -1)
    perror("fcntl F_GETFL");
    exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY) == -1)
    perror("fcntl F_SETFL, FNDELAY");
```

```
    exit(1);
}
 ...
```

When doing I/O on a nonblocking socket, check for the error EWOULDBLOCK (in <errno.h>), which occurs when an operation would normally block. accept(), connect(), send(), recv(), read(), and write() can all return EWOULDBLOCK. If an operation such as a send() cannot be done in its entirety, but partial writes work (such as when using a stream socket), the data that can be sent immediately are processed, and the return value is the amount actually sent.

Asynchronous Sockets

Asynchronous communication between processes is required in real-time applications. Asynchronous sockets must be SOCK_STREAM type. Make a socket asynchronous as shown in Code Example 7-15.

Code Example 7-15 Making a Socket Asynchronous

```
#include <fcntl.h>
#include <sys/file.h>
 ...
int fileflags;
int s;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
if (fileflags = fcntl(s, F_GETFL ) == -1)
    perror("fcntl F_GETFL");
    exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY | FASYNC) < 0)
    perror("fcntl F_SETFL, FNDELAY | FASYNC");
    exit(1);
}
 ...
```

After sockets are initialized, connected, and made asynchronous, communication is similar to reading and writing a file asynchronously. A send(), write(), recv(), or read() initiates a data transfer. A data transfer is completed by a signal-driven I/O routine, described in the next section.

Interrupt Driven Socket I/O

The SIGIO signal notifies a process when a socket (actually any file descriptor) has finished a data transfer. There are three steps in using SIGIO:

1. **Set up a SIGIO signal handler with the** signal() **or** sigvec() **calls.**
2. **Use** fcntl() **to set the process ID or process group ID to receive the signal to its own process ID or process group ID (the default process group of a socket is group 0).**
3. **Convert the socket to asynchronous as shown in "Asynchronous Sockets" on page 249.**

Sample code to allow a given process to receive information on pending requests as they occur for a socket is shown in Code Example 7-16. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

Code Example 7-16 Asynchronous Notification of I/O Requests

```
#include <fcntl.h>
#include <sys/file.h>
 ...
signal(SIGIO, io_handler);
/* Set the process receiving SIGIO/SIGURG signals to us. */
if (fcntl(s, F_SETOWN, getpid()) < 0) {
    perror("fcntl F_SETOWN");
    exit(1);
}
```

Signals and Process Group ID

For SIGURG and SIGIO, each socket has a process number and a process group ID. These values are initialized to zero, but may be redefined at a later time with the F_SETOWN fcntl(), as in the previous example. A positive third argument to fcntl() sets the socket's process ID. A negative third argument to fcntl() sets the socket's process group ID. The only allowed recipient of SIGURG and SIGIO signals is the calling process.

A similar fcntl(), F_GETOWN, returns the process number of a socket.

Reception of SIGURG and SIGIO can also be enabled by using ioctl() to assign the socket to the user's process group:

```
/* oobdata is the out-of-band data handling routine */
sigset(SIGURG, oobdata);
int pid = -getpid();
if (ioctl(client, SIOCSPGRP, (char *) &pid) < 0) {
    perror("ioctl: SIOCSPGRP");
}
```

Another signal that is useful in server processes is SIGCHLD. This signal is delivered to a process when any child process changes state. Normally, servers use the signal to “reap” child processes that have exited without explicitly awaiting their termination or periodically polling for exit status. For example, the remote login server loop shown previously can be augmented as shown in Code Example 7-17.

Code Example 7-17 SIGCHLD Signal

```
int reaper();
 ...
sigset(SIGCHLD, reaper);
listen(f, 5);
while (1) {
    int g, len = sizeof from;
    g = accept(f, (struct sockaddr *) &from, &len);
    if (g < 0) {
        if (errno != EINTR)
            syslog(LOG_ERR, "rlogind: accept: %m");
        continue;
    }
    ...
}

#include <wait.h>

reaper()
{
    int options;
    int error;
    siginfo_t info;

    options = WNOHANG | WEXITED;
    bzero((char *) &info, sizeof(info));
    error = waitid(P_ALL, 0, &info, options);
}
```

If the parent server process fails to reap its children, zombie processes result.

Selecting Specific Protocols

If the third argument of the `socket`() call is `0`, `socket`() selects a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. When using "raw" sockets to communicate directly with lower-level protocols or hardware interfaces, it may be important for the protocol argument to set up de-multiplexing. For example, raw sockets in the Internet domain can be used to implement a new protocol on IP, and the socket will receive packets only for the protocol specified. To obtain a particular protocol, determine the protocol number as defined in the protocol domain. For the Internet domain, use one of the library routines discussed in "Standard Routines" on page 236, such as `getprotobyname()`:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto) ;
```

This results in a socket `s` using a stream-based connection, but with protocol type of `newtcp` instead of the default `tcp`.

Address Binding

In the Internet Protocol family, bindings are composed of local and foreign IP addresses, and of local and foreign port numbers. Port numbers are allocated in separate spaces, one for each system and one for each transport protocol (TCP or UDP). Through `bind`(), a process specifies the <*local IP address, local port number*> half of an association, while `connect`() and `accept`() complete a socket's association by specifying the <*foreign IP address, foreign port number*> part. Since the association is created in two steps, the association-uniqueness requirement might be violated, unless care is taken. It is unrealistic to expect user programs to always know proper values to use for the local address and local port, since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

The wildcard address simplifies local address binding in the Internet domain. When an address is specified as `INADDR_ANY` (a constant defined in `<netinet/in.h>`), the system interprets the address as any valid address.

Code Example 7-18 binds a specific port number to a socket, and leaves the local address unspecified.

Code Example 7-18 Bind Port Number to Socket

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
    s = socket(AF_INET, SOCK_STREAM, 0);
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    sin.sin_port = htons(MYPORT);
    bind(s, (struct sockaddr *) &sin, sizeof sin);
```

Each network interface on a host typically has a unique IP address. Sockets with wildcard local addresses may receive messages directed to the specified port number and sent to any of the possible addresses assigned to a host. For example, if a host has two interfaces with addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as in Code Example 7-18, the process can accept connection requests addressed to 128.32.0.4 or 10.0.0.78. To allow only hosts on a specific network to connect to it, a server binds the address of the interface on the appropriate network.

Similarly, a local port number can be left unspecified (specified as `0`), in which case the system selects a port number. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

```
sin.sin_addr.s_addr = inet_addr("127.0.0.1");
sin.sin_family = AF_INET;
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The system uses two criteria to select the local port number:

- The first is that Internet port numbers less than 1024(`IPPORT_RESERVED`) are reserved for privileged users (that is, the superuser). Nonprivileged users may use any Internet port number greater than 1024. The largest Internet port number is 65535.
- The second criterion is that the port number is not currently bound to some other socket.

The port number and IP address of the client is found through either `accept`() (the `from` result) or `getpeername`().

In certain cases, the algorithm used by the system to select port numbers is unsuitable for an application. This is because associations are created in a two-step process. For example, the Internet file transfer protocol specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket still existed. To override the default port selection algorithm, an option call must be performed before address binding:

```
 ...
int on = 1;
...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof on);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

With this call, local addresses can be bound that are already in use. This does not violate the uniqueness requirement, because the system still verifies at connect time that any other sockets with the same local address and port do not have the same foreign address and port. If the association already exists, the error `EADDRINUSE` is returned.

Broadcasting and Determining Network Configuration

Messages sent by datagram sockets can be broadcast to reach all of the hosts on an attached network. The network must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Broadcasting is usually used for either of two reasons: to find a resource on a local network without having its address, or functions like routing require that information be sent to all accessible neighbors.

To send a broadcast message, create an Internet datagram socket:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

and bind a port number to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The datagram can be broadcast on only one network by send()ing to the network's broadcast address. A datagram can also be broadcast on all attached networks by send()ing to the special address INADDR_BROADCAST, defined in <netinet/in.h>.

The system provides a mechanism to determine a number of pieces of information (including the IP address and broadcast address) about the network interfaces on the system. The SIOCGIFCONF ioctl call returns the interface configuration of a host in a single ifconf structure. This structure contains an array of ifreq structures, one for each address domain supported by each network interface to which the host is connected. Code Example 7-19 shows these structures defined in <net/if.h>.

Code Example 7-19 net/if.h Header File

```
struct ifreq {
#define IFNAMSIZ 16
char ifr_name[IFNAMSIZ]; /* if name, e.g., "en0" */
union {
    struct sockaddr ifru_addr;
    struct sockaddr ifru_dstaddr;
    char ifru_oname[IFNAMSIZ]; /* other if name */
    struct sockaddr ifru_broadaddr;
    short ifru_flags;
    int ifru_metric;
    char ifru_data[1]; /* interface dependent data */
    char ifru_enaddr[6];
} ifr_ifru;
#define ifr_addr ifr_ifru.ifru_addr
#define ifr_dstaddr ifr_ifru.ifru_dstaddr
#define ifr_oname ifr_ifru.ifru_oname
#define ifr_broadaddr ifr_ifru.ifru_broadaddr
#define ifr_flags ifr_ifru.ifru_flags
#define ifr_metric ifr_ifru.ifru_metric
#define ifr_data ifr_ifru.ifru_data
#define ifr_enaddr ifr_ifru.ifru_enaddr
};
```

The call that obtains the interface configuration is:

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof buf;
ifc.ifc_buf = buf;
```

```
if (ioctl(s, SIOCGIFCONF, (char *) &ifc ) < 0) {
    ...
}
```

After this call, `buf` contains an array of `ifreq` structures, one for each network to which the host is connected. These structures are ordered first by interface name and then by supported address families. `ifc.ifc_len` is set to the number of bytes used by the `ifreq` structures.

Each structure has a set of interface flags that tell whether the corresponding network is up or down, point to point or broadcast, and so on. The `SIOCGIFFLAGS ioctl` returns these flags for an interface specified by an `ifreq` structure shown in Code Example 7-20.

Code Example 7-20 Obtaining Interface Flags

```
struct ifreq *ifr;
ifr = ifc.ifc_req;
for (n = ifc.ifc_len/sizeof (struct ifreq); --n >= 0; ifr++) {
    /*
     * Be careful not to use an interface devoted to an address
     * domain other than those intended.
     */
    if (ifr->ifr_addr.sa_family != AF_INET)
        continue;
    if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
        ...
    }
    /* Skip boring cases */
    if ((ifr->ifr_flags & IFF_UP) == 0 ||
        (ifr->ifr_flags & IFF_LOOPBACK) ||
        (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTOPOINT)) == 0)
        continue;
}
```

Code Example 7-21 shows the broadcast of an interface can be obtained with the `SIOGGIFBRDADDR ioctl()`.

Code Example 7-21 Broadcast Address of an Interface

```
    if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
        ...
    }
    memcpy((char *) &dst, (char *) &ifr->ifr_broadaddr,
        sizeof ifr->ifr_broadaddr);
```

The SIOGGIFBRDADDR ioctl can also be used to get the destination address of a point-to-point interface.

After the interface broadcast address is obtained, transmit the broadcast datagram with sendto():

```
sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof dst);
```

Use one sendto() for each interface to which the host is connected that supports the broadcast or point-to-point addressing.

Socket Options

You can set and get several options on sockets through setsockopt() and getsockopt(). These options include changing the send or receive buffer space, to linger on close, and so on. The general forms of the calls are:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

Table 7-3 shows the arguments of the calls.

Table 7-3 setsockopt() and getsockopt() Arguments

Arguments	Description
s	Socket on which the option is to be applied
level	Specifies the protocol level, i.e. socket level, indicated by the symbolic constant SOL_SOCKET in <sys/socket.h>
optname	Symbolic constant defined in <sys/socket.h> that specifies the option
optval	Points to the value of the option
optlen	Points to the length of the value of the option

For getsockopt(), optlen is a value-result argument, initially set to the size of the storage area pointed to by optval and set on return to the length of storage used.

It is sometimes useful to determine the type (for example, stream or datagram) of an existing socket. Programs invoked by inetd may need to do this by using the SO_TYPE socket option and the getsockopt() call:

```
#include <sys/types.h>
#include <sys/socket.h>

int type, size;

size = sizeof (int);
if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0) {
    ...
}
```

After `getsockopt`(), `type` is set to the value of the socket type, as defined in `<sys/socket.h>`. For a datagram socket, `type` would be `SOCK_DGRAM`.

`inetd` *Daemon*

One of the daemons provided with the system is `inetd`. `inetd` is invoked at start-up 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, `fork`()s, `dup`()s the new socket to file descriptors `0` and `1` (`stdin` and `stdout`), closes other open file descriptors, and `exec`()s the appropriate server.

Servers that use `inetd` are simplified, since `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.

Moving Socket Applications to SunOS 5.x

Sockets and the socket implementation are mostly compatible with previous releases of SunOS. But, an application programmer must be aware of some differences that are listed in the tables provided in this section.

Table 7-4 Connection-Mode Primitives (SunOS 4.x / SunOS 5.x)

SunOS 4.x (BSD)	SunOS 5.x (Solaris)
`connect`()	
When `connect`() is called on an unbound socket, the protocol determines whether the endpoint is bound before the connection takes place.	When `connect`() is called on an unbound socket, that socket is always bound to an address selected by the protocol.

Table 7-5 Data Transfer Primitives (SunOS 4.x / SunOS 5.x)

SunOS 4.x (BSD)	SunOS 5.x (Solaris)
`write`()	
`write`() fails with `errno` set to `ENOTCONN` if it is used on an unconnected socket.	A call to `write`() appears to succeed, but the data are discarded. The socket error option `SO_ERROR` returns `ENOTCONN` if this happens.
`write`() can be used on type `SOCK_DGRAM` sockets (either `AF_UNIX` or `AF_INET` domains) to send zero-length data.	A call to `write`() returns -1, with `errno` set to `ERANGE`. `send`(), `sendto`(), or `sendmsg`() should be used to send zero-length data.
`read`()	
`read`() fails with `errno` set to `ENOTCONN` if `read`() is used on an unconnected socket.	`read`() returns zero bytes read if the socket is in blocking mode. If the socket is in non-blocking mode, it returns `-1` with `errno` set to `EAGAIN`.

Table 7-6 Information Primitives (SunOS 4.x / SunOs 5.x)

SunOS 4.x (BSD)	SunOS 5.x (Solaris)
`getsockname`()	
`getsockname`() works when a previously existing connection has been closed.	`getsockname`() returns `-1` and `errno` is set to `EPIPE` if a previously existing connection has been closed.
`ioctl`() and `fcntl`()	
The argument of the `SIOCSPGRP` / `FIOSETOWN` / `F_SETOWN` `ioctl`()s and the `F_SETOWN` `fcntl`() are a positive process ID or negative process group ID of the intended recipient list of subsequent `SIGURG` and `SIGIO` signals.	This is not the case in SunOS 5.4. The only acceptable argument of these system calls is the caller's process ID or a negative of the caller's process group ID. So, the only recipient of `SIGURG` and `SIGIO` is the calling process.

Table 7-7 Local Management (SunOS 4.x / SunOS 5.x)

SunOS 4.x (BSD)	SunOS 5.x (Solaris)
`bind`()	
`bind`() uses the credentials of the user at the time of the `bind`() call to determine if the requested address is allocated or not.	`socket`() causes the user's credentials to be found and used to validate addresses used later in `bind`().
`setsockopt`()	
`setsockopt`() can be used at any time during the life of a socket.	If a socket is unbound and `setsockopt`() is used, the operation succeeds in the `AF_INET` domain but fails in the `AF_UNIX` domain.

Table 7-7 Local Management (SunOS 4.x / SunOS 5.x)

SunOS 4.x (BSD)	SunOS 5.x (Solaris)
`shutdown`()	
If `shutdown`() is called with `how` set to zero, further tries to receive data returns zero bytes (`EOF`).	If a `shutdown`() call with `how` set to zero is followed by a `read`(2) call and the socket is in nonblocking mode, `read`() returns `-1` with `errno` set to `EAGAIN`. If one of the socket receive primitives is used, the correct result (`EOF`) is returned.
If `shutdown`() is called with the value of `2` for how, further tries to receive data return `EOF`. Tries to send data return `-1` with `errno` set to `EPIPE` and a `SIGPIPE` is issued.	The same result happens, but tries to send data using write(2) cause `errno` to be set to `EIO`. If a socket primitive is used, the correct `errno` is returned.

Table 7-8 Signals (SunOS 4.x / SunOS 5.x)

SunOS 4.x (BSD)	SunOS 5.x (Solaris)
SIGIO	
`SIGIO` is delivered every time new data are appended to the socket input queue.	`SIGIO` is delivered only when data are appended to a socket queue that was previously empty.
SIGURG	
A `SIGURG` is delivered every time new data are anticipated or actually arrive.	A `SUGURG` is delivered only when data are already pending.
`S_ISSOCK`()	
The `S_ISSOCK` macro takes the mode of a file as an argument. It returns `1` if the file is a socket and `0` otherwise.	The `S_ISSOCK` macro does not exist. Here, a socket is a file descriptor associated with a STREAMS character device that has the socket module pushed onto it.

Table 7-9 Miscellaneous Socket Issues (SunOS 4.x / SunOS 5.x)

SunOS 4.x (BSD)	**SunOS 5.x (Solaris)**
Invalid buffers	
If an invalid buffer is specified in a function, the function returns `-1` with `errno` set to `EFAULT`.	If an invalid buffer is specified in a function, the user's program probably dumps core.
Sockets in Directories	
If `ls -l` is executed in a directory containing a UNIX domain socket, an `s` is displayed on the left side of the mode field.	If `ls -l` is executed in a directory that contains a UNIX domain socket, a `p` is displayed on the left side of the mode field.
An `ls -F` causes an equal sign (`=`) to be displayed after any file name of a UNIX domain socket.	Nothing is displayed after the file name.

Part 4 — Naming Services

Chapter 8 NIS+ Programming Guide

NIS+ Programming Guide 8

This chapter presents the fundamental principles of the NIS+ Name Service Applications Programming Interface and a detailed sample program. The NIS+ API is for programmers who need to build applications for Solaris-based networks. It provides the essential features for supporting enterprise-wide applications.

NIS+ Overview

The NIS+ naming service addresses the requirements of client/server networks ranging in size from 10 clients supported by a few servers on a simple local area network to 10,000 multi-vendor clients supported by 20 to 100 specialized servers located in sites throughout the world and connected by several public networks.

Domains

NIS+ supports hierarchical domains, illustrated as a simple case in Figure 8-1.

8

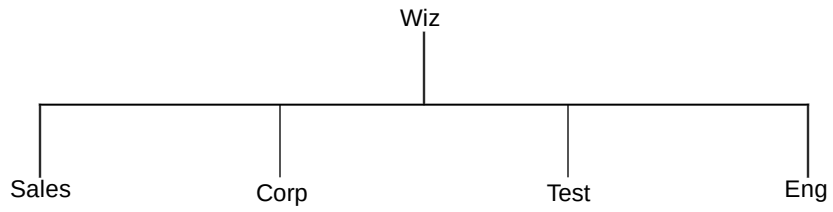


Figure 8-1 NIS+ Domain

A NIS+ domain is a set of data describing the workstations, users, and network services in a portion of an organization. NIS+ domains can be administered independently of each other. This allows NIS+ to be used in a range of networks, from small to very large.

Servers

Each domain is supported by a set of servers. The principal server is called the master server, and the backup servers are called replicas. Both master and replica servers run NIS+ server software. The master server stores the original tables, and the backup servers store copies.

NIS+ accepts incremental updates to the replicas. Changes are first made on the master server. Then they are automatically propagated to the replica servers and are soon available to the entire namespace.

Tables

NIS+ stores information in tables instead of maps or zone files. NIS+ provides 16 types of predefined, or system, tables, shown in Figure 8-2.

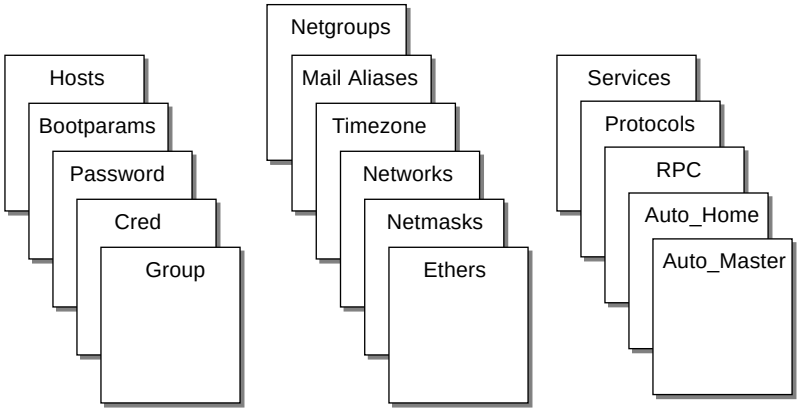


Figure 8-2 NIS+ Tables

Each table stores a different type of information. For instance, the Hosts table stores host name/Internet address pairs, and the Password table stores information about users of the network.

NIS+ tables have two major improvements over NIS maps. First, a NIS+ table can be accessed by any column, not just the first column (sometimes referred to as the “key”). This eliminates the need for duplicate maps, such as the `hosts.byname` and `hosts.byaddr` maps of NIS. Second, access to the information in NIS+ tables can be controlled at three levels of granularity: the table level, the entry level, and the column level.

NIS+ Security

The NIS+ security model provides both authorization and authentication mechanisms. First, every object in the namespace specifies the type of operation it accepts and from whom. This is authorization. Second, NIS+ attempts to authenticate every requestor accessing the namespace. Once it identifies the originator of the request, it determines whether the object has authorized that particular operation for that particular principal. Based on its authentication and the object’s authorization, NIS+ carries out or denies the access request.

8

Name Service Switch

NIS+ works in conjunction with a separate facility called the Name Service Switch. The Name Service Switch, sometimes referred to as “the Switch,” lets Solaris 2.x-based workstations obtain their information from more than one network information service; specifically, from local, or `/etc` files, from NIS maps, from DNS zone files, or from NIS+ tables. The Switch not only offers a choice of sources, but allows a workstation to specify different sources for different *types* of information. The name service is configured through the file `/etc/nsswitch.conf`().

NIS+ Administration Commands

NIS+ provides a full set of commands for administering a namespace. Table 8-1 summarizes them.

Table 8-1 NIS+ Namespace Administration Commands

Command	Description
`nischgrp`	Changes the group owner of a NIS+ object.
`nischmod`	Changes an object’s access rights.
`nischown`	Changes the owner of a NIS+ object.
`nisgrpadm`	Creates or destroys a NIS+ group, or displays a list of its members. Also adds members to a group, removes them, or tests them for membership in the group.
`niscat`	Displays the contents of NIS+ tables.
`nisgrep`	Searches for entries in a NIS+ table.
`nisls`	Lists the contents of a NIS+ directory.
`nismatch`	Searches for entries in a NIS+ table.
`nisaddent`	Adds information from `/etc` files or NIS maps into NIS+ tables.
`nistbladm`	Creates or deletes NIS+ tables, and adds, modifies or deletes entries in a NIS+ table.
`nisaddcred`	Creates credentials for NIS+ principals and stores them in the Cred table.
`nispasswd`	Changes password information stored in the NIS+ Passwd table.
`nisupdkeys`	Updates the public keys stored in a NIS+ object.

Table 8-1 NIS+ Namespace Administration Commands

Command	Description
`nisinit`	Initializes a NIS+ client or server.
`nismkdir`	Creates a NIS+ directory and specifies its master and replica servers.
`nisrmdir`	Removes NIS+ directories and replicas from the namespace.
`nissetup`	Creates `org_dir` and `groups_dir` directories and a complete set of (unpopulated) NIS+ tables for a NIS+ domain.
`rpc.nisd`	The NIS+ server process.
`nis_cachemgr`	Starts the NIS+ Cache Manager on a NIS+ client.
`nischttl`	Changes a NIS+ object's time-to-live value.
`nisdefaults`	Lists a NIS+ object's default values: domain name, group name, workstation name, NIS+ principal name, access rights, directory search path, and time-to-live.
`nisln`	Creates a symbolic link between two NIS+ objects.
`nisrm`	Removes NIS+ objects (except directories) from the namespace.
`nisshowcache`	Lists the contents of the NIS+ shared cache maintained by the NIS+ Cache Manager.

NIS+ API

The NIS+ application programming interface (API) is a group of functions that can be called by an application to access and modify NIS+ objects. The NIS+ API has 54 functions that fall into nine categories:

- Object Manipulation Functions (`nis_names`)
- Table Access Functions (`nis_tables`)
- Local Name Functions (`nis_local_names`)
- Group Manipulation Functions (`nis_groups`)
- Server Related Functions (`nis_server`)
- Database Access Functions (`nis_db`)
- Error Message Display Functions (`nis_error`)
- Transaction Log Functions (`nis_admin`)
- Miscellaneous Functions (`nis_subr`)

The functions in each category are summarized in Table 8-2.
The category names match the names by which they are grouped in the NIS+ manpages.

Table 8-2 NIS+ API Functions

Function	Description
nis_names	**Locate and Manipulate Objects**
nis_lookup()	Returns a copy of an NIS+ object. Can follow links. Though it cannot search for an entry object, if a link points to one, it can return an entry object.
nis_add()	Adds an NIS+ object to the namespace.
nis_remove()	Removes an NIS+ object in the namespace.
nis_modify()	Modifies an NIS+ object in the namespace.
nis_tables	**Search and Update Tables**
nis_list()	Searches a table in the NIS+ namespace and returns entry objects that match the search criteria. Can follow links and search paths from one table to another.
nis_add_entry()	Adds an entry object to an NIS+ table. Can be instructed to either fail or overwrite if the entry object already exists. Can return a copy of the resulting object if the operation was successful.
nis_freeresult()	Frees all memory associated with a nis_result structure.
nis_remove_entry()	Removes one or more entry objects from an NIS+ table. Can identify the object to be removed by using search criteria or by pointing to a cached copy of the object. If using search criteria, can remove all objects that match the search criteria; therefore, with the proper search criteria, can remove all entries in a table. Can return a copy of the resulting object if the operation was successful.
nis_modify_entry()	Modifies one or more entry objects in an NIS+ table. Can identify the object to be modified by using search criteria or by pointing to a cached copy of the object.
nis_first_entry()	Returns a copy of the first entry object in an NIS+ table.
nis_next_entry()	Returns a copy of the “next” entry object in an NIS+ table. Because a table can be updated and entries removed or modified between calls to this function, the order of entries returned may not match the actual order of entries in the table.
nis_local_names	**Get Default Names for the Current Process**

Table 8-2 NIS+ API Functions

Function	Description
`nis_local_directory()`	Returns the name of the workstation's NIS+ domain.
`nis_local_host()`	Returns the fully-qualified name of the workstation. A fully qualified name has the form <host-name>.<domain-name>.
`nis_local_group()`	Returns the name of the current NIS+ group, which is specified by the environment variable NIS_GROUP.
`nis_local_principal()`	Returns the name of the NIS+ principal whose UID is associated with the calling process.
`nis_getnames()`	Returns a list of possible expansions to a particular name.
`nis_freenames()`	Frees the memory containing the list generated by `nis_getnames`.
`nis_groups`	**Group Manipulation and Authorization**
`nis_ismember()`	Test whether a principal is a member of a group.
`nis_addmember()`	Adds a member to a group. The member can be a principal, a group, or a domain.
`nis_removemember()`	Deletes a member from a group.
`nis_creategroup()`	Create a group object.
`nis_destroygroup()`	Delete a group object.
`nis_verifygroup()`	Tests whether a group object exists.
`nis_print_group_ entry()`	Lists the principals that are members of a group object.
`nis_server`	**Various services for NIS+ applications.**
`nis_mkdir()`	Creates the databases to support service for a named directory on a specified host.
`nis_rmdir()`	Removes the directory from a host.
`nis_servstate()`	Sets and reads state variables of NIS+ servers and flushes internal cashes.
`nis_stats()`	Retrieves statistics about a server's performance.
`nis_getservlist()`	Returns a list of servers that support a particular domain.
`nis_freeservlist()`	Frees the list of servers returned by `nis_getservlist`.

Table 8-2 NIS+ API Functions

Function	Description
`nis_freetags()`	Frees the memory associated with the results of `nis_servstate` and `nis_stats`.
`nis_db`	**Interface Between the NIS+ Server and the Database. Not To Be Used By a NIS+ Client.**
`db_first_entry()`	Returns a copy of the first entry of the specified table.
`db_next_entry()`	Returns a copy of the entry succeeding the specified entry.
`db_reset_next_entry()`	Terminates a first/next entry sequence.
`db_list_entries()`	Returns copies of entries that meet specified attributes.
`db_remove_entry()`	Removes all entries that meet specified attributes.
`db_add_entry()`	Replaces an entry in a table identified by specified attributes with a copy of the specified object, or adds the object to the table.
`db_checkpoint()`	Reorganizes the contents of a table to make access to the table more efficient.
`db_standby()`	Advises the database manager to release resources.
`nis_error`	**Functions that supply descriptive strings equivalent to NIS+ status values**
`nis_sperrno()`	Returns a pointer to the appropriate string constant.
`nis_perror()`	Displays the appropriate string constant on standard output.
`nis_lerror()`	Sends the appropriate string constant to syslog
`nis_sperror()`	Returns a pointer to a statically allocated string to be used or to be copied with `strdup`().
`nis_admin`	**Transaction logging functions used by servers**
`nis_ping()`	Used by the master server of a directory to time stamp it. This forces replicas of the directory to be updated.
`nis_checkpoint()`	Forces logged data to be stored in the table on disk.
`nis_subr`	**Functions To Help Operate on NIS+ Names and Objects.**
`nis_leaf_of()`	Returns the first label in an NIS+ name. The returned name does not have a trailing dot.

Table 8-2 NIS+ API Functions

Function	Description
`nis_name_of()`	Removes all domain-related labels and returns only the unique object portion of the name. The name passed to the function must be either in the local domain or in one of its child domains, or the function returns NULL.
`nis_domain_of()`	Returns the name of the domain in which an object resides. The returned name ends in a dot.
`nis_dir_cmp()`	Compares any two NIS+ names. The comparison ignores case and states whether the names are the same, descendants of each other, or not related.
`nis_clone_object()`	Creates an exact duplicate of an NIS+ object.
`nis_destroy_object()`	Destroys an object created by `nis_clone_object`.
`nis_print_object()`	Prints the contents of an NIS+ object structure to `stdout`.

NIS+ Sample Program

This program performs the following tasks:

- Determines the local principal and local domain
- Looks up the local directory object
- Creates a directory called `foo` under the local domain
- Creates the `groups_dir` and `org_dir` directories under domain foo
- Creates a group object `admins.foo`
- Adds the local principal to the admins group
- Creates a table under `org_dir.foo`
- Adds two entries to the `org_dir.foo` table
- Retrieves and displays the new membership list of the `admins` group
- Lists the namespace under the `foo` domain using callbacks
- Lists the contents of the table created using callbacks
- Cleans up all the objects that were created by removing the following:
  - the local principal from the admins group
  - the `admins` group
  - the entries in the table followed by the table
  - the `groups_dir` and `org_dir` directory objects
  - the `foo` directory object

The example program is not a typical application. In a normal situation the directories and tables would be created or removed through the command line interface, and applications would manipulate NIS+ entry objects.

Unsupported Macros

The sample program uses unsupported macros that are defined in the file <rpcsvc/nis.h>. These are not public APIs and can change or disappear in the future. They are used for illustration purposes only and if you choose to use them, you do so at your own risk. The macros used are:

- NIS_RES_OBJECT
- ENTRY_VAL
- DEFAULT_RIGHTS

Functions Used in the Example

The use of the following NIS+ C API functions is illustrated through this example:

```
nis_add()              nis_add_entry()        nis_addmember()
nis_creategroup()      nis_destroygroup()     nis_domain_of()
nis_freeresult()       nis_leaf_of ()         nis_list()
nis_local_directory()  nis_local_principal()  nis_lookup()
nis_mkdir()            nis_perror()           nis_remove()
nis_remove_entry()     nis_removemember()
```

Program Compilation

The program shown in Code Example 8-1assumes that the NIS+ principal running this application has permission to create directory objects in the local domain. The program is compiled:

```
yourhost% cc -o example.c example -lnsl
```

It is invoked:

```
yourhost% example [dir]
```

where dir is the NIS+ directory in which the program creates all the NIS+ objects. Specifying no directory argument causes the objects to be created in the parent directory of the local domain. Note that for the call to nis_lookup(), a space and the name of the local domain are appended to the string that

names the directory. The argument is the name of the NIS+ directory in which to create the NIS+ objects. The principal running this program should have create permission in the directory.

Code Example 8-1 NIS+ Program Main `example.c`

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <rpcsvc/nis.h>

#define    MAX_MESG_SIZE 512
#define    BUFFER_SIZE 64
#define    TABLE_TYPE "test_data"

main(argc, argv)
    int    argc;
    char   *argv[];
{
    char   *saved_grp, *saved_name, *saved_owner;
    char   dir_name[NIS_MAXNAMELEN];
    char   local_domain[NIS_MAXNAMELEN];
    char   local_princip [NIS_MAXNAMELEN];
    char   org_dir_name [NIS_MAXNAMELEN];
    char   grp_name [NIS_MAXNAMELEN];
    char   grp_dir_name [NIS_MAXNAMELEN];
    char   table_name [NIS_MAXNAMELEN];
    nis_object     *dirobj, entdata;
    nis_result     *pres;
    u_int  saved_num_servers;
    int    err;

    if (argc == 2)
        sprintf (local_domain, "%s.", argv[1]);
    else
        strcpy (local_domain, "");

    strcat (local_domain, (char *) nis_local_directory());
    strcpy (local_princip, (char *) nis_local_principal());
    /*
     * Lookup the directory object for the local domain for two reasons:
     * 1.To get a template of a nis_object.
     * 2.To reuse some of the information contained in the directory
     * object returned. We could have declared a static nis_object, but
     * since we need to change very little, it is easier to make the
     * changes and not initialize the nis_object structure.
```

```
 */

pres = nis_lookup (local_domain, 0);
if (pres->status != NIS_SUCCESS) {
    nis_perror (pres->status, "unable to lookup local directory");
    exit (1);
}

/*
 * re-use most of the fields in the parent directory object - save
 * pointers to the fields that are being changed so that we can
 * free the original object and avoid dangling pointer references.
 */
dirobj = NIS_RES_OBJECT (pres);
saved_name = dirobj->DI_data.do_name;
saved_owner = dirobj->zo_owner;
saved_grp = dirobj->zo_group;

/*
 * set the new name, group, owner and new access rights for the
 * foo domain.
 */
sprintf (dir_name, "%s.%s", "foo", local_domain);
sprintf (grp_name, "%s.%s", "admins", dir_name);
dirobj->DI_data.do_name = dir_name;
dirobj->zo_group = grp_name;
dirobj->zo_owner = local_princip;

/*
 * Access rights in NIS+ are stored in a u_long with the highest
 * order bytes reserved for the "nobody" category, the next eight
 * bytes reserved for the owner, followed by group and world. In
 * this example we are giving access to the directory based on the
 * "----rmcdrmcd----" access right pattern.
 */
dirobj->zo_access = ((NIS_READ_ACC + NIS_MODIFY_ACC
                  + NIS_CREATE_ACC + NIS_DESTROY_ACC) << 16)
                  | ((NIS_READ_ACC + NIS_MODIFY_ACC
                  + NIS_CREATE_ACC + NIS_DESTROY_ACC) << 8);

/*
 * Save the number of servers the parent directory object had so
 * that we can restore this value before calling nis_freeresult()
 * later and avoid memory leaks.
 */
saved_num_servers = dirobj->DI_data.do_servers.do_servers_len;
```

```
/* We want only one server to serve this directory */
dirobj->DI_data.do_servers.do_servers_len = 1;

dir_create (dir_name, dirobj);

/* create the groups_dir and org_dir directories under foo. */
sprintf (grp_dir_name, "groups_dir.%s", dir_name);
dirobj->DI_data.do_name = grp_dir_name;
dir_create (grp_dir_name, dirobj);

sprintf (org_dir_name, "org_dir.%s", dir_name);
dirobj->DI_data.do_name = org_dir_name;
dir_create (org_dir_name, dirobj);

grp_create (grp_name);

printf ("\nAdding principal %s to group %s ... \n",
           local_princip, grp_name);
err = nis_addmember (local_princip, grp_name);
if (err != NIS_SUCCESS) {
    nis_perror (err,
        "unable to add local principal to group.");
    exit (1);
}

sprintf (table_name, "test_table.org_dir.%s", dir_name);
tbl_create (dirobj, table_name);

/*
 * Now create NIS+ entry objects in the table that was just created
 */
stuff_table (table_name);

/* Display what we stuffed */
list_objs(dir_name, table_name, grp_name);

/* Clean out everything we created. */
cleanup (local_princip, grp_name, table_name, dir_name, dirobj);

/*
 * Restore the saved pointers from the original pres structure
 * so that we can free up the associated memory and have no
 * memory leaks.
 */
dirobj->DI_data.do_name = saved_name;
```

```
    dirobj->zo_group = saved_grp;
    dirobj->zo_owner = saved_owner;
    dirobj->DI_data.do_servers.do_servers_len = saved_num_servers;
    (void) nis_freeresult (pres);
}
```

Code Example 8-2 shows the routine is called by `main()` to create directory objects.

Code Example 8-2 NIS+ Routine to Create Directory Objects

```
void
dir_create (dir_name, dirobj)
    nis_name   dir_name;
    nis_object *dirobj;
{
    nis_result *cres;
    nis_error  err;

    printf ("\n Adding Directory %s to namespace ... \n", dir_name);
    cres = nis_add (dir_name, dirobj);
    if (cres->status != NIS_SUCCESS) {
        nis_perror (cres->status, "unable to add directory foo.");
        exit (1);
    }

    (void) nis_freeresult (cres);

    /*
     * NOTE: you need to do a nis_mkdir to create the table to store the
     * contents of the directory you are creating.
     */
    err = nis_mkdir (dir_name,
                     dirobj->DI_data.do_servers.do_servers_val);
    if (err != NIS_SUCCESS) {
        (void) nis_remove (dir_name, 0);
        nis_perror (err,
                "unable to create table for  directory object foo.");
        exit (1);
    }
}
```

This routine is called by `main()` to create the group object. Since `nis_creategroup`() works only on group objects, the “groups_dir” literal is not needed in the group name.

Code Example 8-3 NIS+ Routine to Create Group Objects

```
void
grp_create (grp_name)
    nis_name   grp_name;
{
    nis_error  err;

    printf ("\n Adding %s group to namespace ... \n", grp_name);
    err = nis_creategroup (grp_name, 0);
    if (err != NIS_SUCCESS) {
        nis_perror (err, "unable to create group.");
        exit (1);
    }
}
```

The routine shown in Code Example 8-4 is called by `main()` to create a table object laid out as shown in Table 8-3.

Table 8-3 NIS+ Table Objects

	Column1	**Column2**
Name:	id	name
Attributes:	Searchable, Text	Searchable, Text
Access Rights	----rmcdr---r---	----rmcdr---r---

The TA_SEARCHABLE constant indicates to the service that the column is searchable. Only TEXT (the default) columns are searchable. TA_CASE indicates to the service that the column value is to be treated in a case-insensitive manner during searches.

Code Example 8-4 NIS+ Routine to Create Table Objects

```
#define    TABLE_MAXCOLS 2
#define    TABLE_COLSEP ':'
#define    TABLE_PATH 0

void
tbl_create (dirobj, table_name)
    nis_object *dirobj;   /* need to use some of the fields */
    nis_name   table_name;
{
    nis_result     *cres;
    static nis_object  tblobj;
    static table_col   tbl_cols[TABLE_MAXCOLS] = {
```

```
        {"Id", TA_SEARCHABLE | TA_CASE, DEFAULT_RIGHTS},
        {"Name", TA_SEARCHABLE | TA_CASE, DEFAULT_RIGHTS}
    };

    tblobj.zo_owner = dirobj->zo_owner;
    tblobj.zo_group = dirobj->zo_group;
    tblobj.zo_access = DEFAULT_RIGHTS;   /* macro defined in nis.h  */
    tblobj.zo_data.zo_type = TABLE_OBJ;  /* enumerated type in nis.h
*/
    tblobj.TA_data.ta_type = TABLE_TYPE;
    tblobj.TA_data.ta_maxcol = TABLE_MAXCOLS;
    tblobj.TA_data.ta_sep = TABLE_COLSEP;
    tblobj.TA_data.ta_path = TABLE_PATH;
    tblobj.TA_data.ta_cols.ta_cols_len =
        tblobj.TA_data.ta_maxcol;       /* ALWAYS ! */
    tblobj.TA_data.ta_cols.ta_cols_val = tbl_cols;

/*
 * Use a fully qualified table name i.e. the "org_dir" literal should
 * be embedded in the table name. This is necessary because nis_add
 * operates on all types of NIS+ objects and needs the full path name
 * if a table is created.
 */
    printf ("\n Creating table %s ... \n", table_name);
    cres = nis_add (table_name, &tblobj);
    if (cres->status != NIS_SUCCESS) {
        nis_perror (cres->status, "unable to add table.");
        exit (1);
    }
    (void) nis_freeresult (cres);
}
```

The routine shown in Code Example 8-5 is called by main() to add entry objects to the table object. Two entries are added to the table object. Note that the column width in both entries is set to include the NULL character for a string terminator.

Code Example 8-5 NIS+ Routine to Add Objects to Table

```
#define    MAXENTRIES 2
void
stuff_table(table_name)
    nis_name table_name;
{
    int        i;
    nis_object entdata;
```

```
    nis_result *cres;
    static entry_col ent_col_data[MAXENTRIES][TABLE_MAXCOLS] = {
           {0, 2, "1", 0, 5, "John"},
           {0, 2, "2", 0, 5, "Mary"}
    };

    printf ("\n Adding entries to table ... \n");

    /*
     * Look up the table object first since the entries being added
     * should have the same owner, group owner and access rights as
     * the table they go in.
     */
    cres = nis_lookup (table_name, 0);
    if (cres->status != NIS_SUCCESS) {
        nis_perror (cres->status, "Unable to lookup table");
        exit(1);
    }
    entdata.zo_owner = NIS_RES_OBJECT (cres)->zo_owner;
    entdata.zo_group = NIS_RES_OBJECT (cres)->zo_group;
    entdata.zo_access = NIS_RES_OBJECT (cres)->zo_access;

    /* Free cres, so that it can be reused. */
    (void) nis_freeresult (cres);

    entdata.zo_data.zo_type = ENTRY_OBJ; /* enumerated type in nis.h
*/
    entdata.EN_data.en_type = TABLE_TYPE;
    entdata.EN_data.en_cols.en_cols_len = TABLE_MAXCOLS;
    for (i = 0; i < MAXENTRIES; ++i) {
        entdata.EN_data.en_cols.en_cols_val = &ent_col_data[i][0];
        cres = nis_add_entry (table_name, &entdata, 0);
        if (cres->status != NIS_SUCCESS) {
            nis_perror (cres->status, "unable to add entry.");
            exit (1);
        }
        (void) nis_freeresult (cres);
    }
}
```

The routine shown in Code Example 8-6 is the print function for the `nis_list()` call. When `list_objs()` calls `nis_list()`, a pointer to `print_info()` is one of the calling arguments. Each time the service calls this

function, it prints the contents of the entry object. The return value indicates to the library to call with the next entry from the table.

Code Example 8-6 NIS+ Routine for `nis_list` Call

```
int
print_info (name, entry, cbdata)
    nis_name   name;      /* Unused */
    nis_object *entry;    /* The NIS+ entry object */
    void       *cbdata;   /* Unused */
{
    static u_int   firsttime = 1;
    entry_col      *tmp;  /* only to make source more readable */
    u_int          i, terminal;

    if (firsttime) {
        printf ("\tId.\t\t\tName\n");
        printf ("\t---\t\t\t----\n");
        firsttime = 0;
    }
    for (i = 0; i < entry->EN_data.en_cols.en_cols_len; ++i) {
        tmp = &entry->EN_data.en_cols.en_cols_val[i];
        terminal = tmp->ec_value.ec_value_len;
        tmp->ec_value.ec_value_val[terminal] = '\0';
    }

    /*
     * ENTRY_VAL is a macro that returns the value of a specific
     * column value of a specified entry.
     */
    printf("\t%s\t\t\t%s\n", ENTRY_VAL (entry, 0),
                             ENTRY_VAL (entry, 1));
    return (0); /* always ask for more */
}
```

The routine shown in Code Example 8-7 is called by `main()` to list the contents of the group, table and directory objects. The routine demonstrates the use of callbacks also. It retrieves and displays the membership of the group. The group membership list is not stored as the contents of the object. So, it is queried through the `nis_lookup()` instead of the `nis_list`() call. You must use the “groups_dir” form of the group name since `nis_lookup`() works on all types of NIS+ objects.

Code Example 8-7 NIS+ Routine to List Objects

```
void
list_objs(dir_name, table_name, grp_name)
         nis_name   dir_name, table_name, grp_name;
{
    group_obj  *tmp; /* only to make source more readable */
    u_int      i;
    char       grp_obj_name [NIS_MAXNAMELEN];
    nis_result *cres;
    char       index_name [BUFFER_SIZE];

    sprintf (grp_obj_name, "%s.groups_dir.%s",
               nis_leaf_of (grp_name), nis_domain_of (grp_name));
    printf ("\nGroup %s membership is: \n", grp_name);

    cres = nis_lookup(grp_obj_name, 0);
    if (cres->status != NIS_SUCCESS) {
        nis_perror (cres->status, "Unable to lookup group object.");
        exit(1);
    }
    tmp = &(NIS_RES_OBJECT(cres)->GR_data);
    for (i = 0; i < tmp->gr_members.gr_members_len; ++i)
        printf ("\t %s\n", tmp->gr_members.gr_members_val[i]);
    (void) nis_freeresult (cres);

    /*
     * Display the contents of the foo domain without using callbacks.
     */
    printf ("\nContents of Directory %s are: \n", dir_name);
    cres = nis_list (dir_name, 0, 0, 0);
    if (cres->status != NIS_SUCCESS) {
        nis_perror (cres->status,
                        "Unable to list Contents of Directory foo.");
        exit(1);
    }
    for (i = 0; i < NIS_RES_NUMOBJ(cres); ++i)
        printf("\t%s\n", NIS_RES_OBJECT(cres)[i].zo_name);
    (void) nis_freeresult (cres);

    /*
     * List the contents of the table we created using the callback form
     * of nis_list().
     */
    printf ("\n Contents of Table %s are: \n", table_name);
    cres = nis_list (table_name, 0, print_info, 0);
```

```
    if(cres->status != NIS_CBRESULTS && cres->status !=
NIS_NOTFOUND){
        nis_perror (cres->status,
            "Listing entries using callback failed");
        exit(1);
    }
    (void) nis_freeresult (cres);

    /*
     * List only one entry from the table we created. We will
     * use indexed names to do this retrieval.
     */

    printf("\n Entry corresponding to id 1 is:\n");
    /*
     * The name of the column is usually extracted from the table
     * object, which would have to be retrieved first.
     */
    sprintf(index_name, "[Id=1],%s", table_name);
    cres = nis_list (index_name, 0, print_info, 0);
    if(cres->status != NIS_CBRESULTS && cres->status !=
NIS_NOTFOUND){
        nis_perror (cres->status,
        "Listing entry using indexed names and callback failed");
        exit(1);
    }
    (void) nis_freeresult (cres);
}
```

The routine in Code Example 8-8 is called by `cleanup()` to remove a directory object from the namespace. It also informs the servers serving the directory about this deletion. Notice that the memory containing result structure, pointed to by `cres`, must be freed after the result has been tested.

Code Example 8-8 NIS+ Routine to Remove Directory Objects

```
void
dir_remove(dir_name, srv_list, numservers)
    nis_name   dir_name;
    nis_server *srv_list;
    u_int      numservers;
{
    nis_result *cres;
    nis_error  err;
    u_int      i;
```

```
    printf ("\nRemoving %s directory object from namespace ... \n",
                                dir_name);
    cres = nis_remove (dir_name, 0);
    if (cres->status != NIS_SUCCESS) {
        nis_perror (cres->status, "unable to remove directory");
        exit (1);
    }
    (void) nis_freeresult (cres);

    for (i = 0; i < numservers; ++i) {
        err = nis_rmdir (dir_name, &srv_list[i]);
        if (err != NIS_SUCCESS) {
            nis_perror (err,
            "unable to remove server from directory");
            exit (1);
        }
    }
}
```

This routine, Code Example 8-9, is called by `main()` to delete all the objects that were created in this example. Note the use of the `REM_MULTIPLE` flag in the call to `nis_remove_entry()`. All entries must be deleted from a table before the table itself can be deleted.

Code Example 8-9 NIS+ Routine to Remove All Objects

```
void
cleanup(local_princip, grp_name, table_name, dir_name, dirobj)
    nis_name   local_princip, grp_name, table_name, dir_name;
    nis_object *dirobj;
{
    char   grp_dir_name [NIS_MAXNAMELEN];
    char   org_dir_name [NIS_MAXNAMELEN];
    nis_error  err;
    nis_result *cres;

    sprintf(grp_dir_name, "%s.%s", "groups_dir", dir_name);
    sprintf(org_dir_name, "%s.%s", "org_dir", dir_name);

    printf("\n\n\nStarting to Clean up ... \n");
    printf("\n\nRemoving principal %s from group %s \n",
                      local_princip, grp_name);
    err = nis_removemember (local_princip, grp_name);
    if (err != NIS_SUCCESS) {
        nis_perror (err,
        "unable to delete local principal from group.");
```

```
    exit (1);
}

/*
 * Delete the admins group. We do not use the "groups_dir" form
 * of the group name since this API is applicable to groups only.
 * It automatically embeds the groups_dir literal in the name of
 * the group.
 */
printf("\nRemoving %s group from namespace ... \n", grp_name);
err = nis_destroygroup (grp_name);
if (err != NIS_SUCCESS) {
    nis_perror (err, "unable to delete group.");
    exit (1);
}

printf("\n Deleting all entries from table %s ... \n", table_name);
cres = nis_remove_entry(table_name, 0, REM_MULTIPLE);
switch (cres->status) {
    case NIS_SUCCESS:
    case NIS_NOTFOUND:
        break;
    default:
        nis_perror(cres->status, "Could not delete entries from
                                  table");
        exit(1);
}
(void) nis_freeresult (cres);

printf("\n Deleting table %s itself ... \n", table_name);
cres = nis_remove(table_name, 0);
if (cres->status != NIS_SUCCESS) {
    nis_perror(cres->status, "Could not delete table.");
exit(1);
}
(void) nis_freeresult (cres);

/* delete the groups_dir, org_dir and foo  directory objects. */
dir_remove (grp_dir_name,
                  dirobj->DI_data.do_servers.do_servers_val,
                  dirobj->DI_data.do_servers.do_servers_len);
dir_remove (org_dir_name,
                  dirobj->DI_data.do_servers.do_servers_val,
                  dirobj->DI_data.do_servers.do_servers_len);
dir_remove (dir_name, dirobj-
```

```
>DI_data.do_servers.do_servers_val,
                        dirobj->DI_data.do_servers.do_servers_len);
}
```

Running the program displays on the screen, as shown in Figure 8-3.

```
myhost% domainname
sun.com
myhost% ./sample
Adding Directory foo.sun.com. to namespace ...
Adding Directory groups_dir.foo.sun.com. to namespace ...
Adding Directory org_dir.foo.sun.com. to namespace ...
Adding admins.foo.sun.com. group to namespace ...
Adding principal myhost.sun.com. to group admins.foo.sun.com. ...
Creating table test_table.org_dir.foo.sun.com. ...
Adding entries to table ...
Group admins.foo.sun.com. membership is:
        myhost.sun.com.
Contents of Directory foo.sun.com. are:
        groups_dir
        org_dir

Contents of Table test_table.org_dir.foo.sun.com. are:
        Id.                     Name
        ---                     ----
        1                       John
        2                       Mary

Entry corresponding to id 1 is:
        1                       John

Starting to Clean up ...

Removing principal myhost.sun.com. from group admins.foo.sun.com.
Removing admins.foo.sun.com. group from namespace ...
Deleting all entries from table test_table.org_dir.foo.sun.com. ...
Deleting table test_table.org_dir.foo.sun.com. itself ...
Removing groups_dir.foo.sun.com. directory object from namespace ...
Removing org_dir.foo.sun.com. directory object from namespace ...
Removing foo.sun.com. directory object from namespace ...
myhost%
```

Figure 8-3 NIS+ Program Execution

As a debugging aid, the same operations are performed by the following command sequence. The first command:

```
% niscat -o ‘domainname‘
```

displays the name of the master server. Substitute the master server name where the variable *master* appears below.

```
% nismkdir -m master foo.‘domainname‘.

# Create the org_dir.foo subdirectory with the specified master
% nismkdir -m master org_dir.foo.‘domainname‘.
# Create the groups_dir.foo subdirectory with the specified master
% nismkdir -m master groups_dir.foo.‘domainname‘.
# Create the “admins” group
% nisgrpadm -c admins.foo.‘domainname‘.

# Add yourself as a member of this group
% nisgrpadm -a admins.foo.‘domainname‘. ‘nisdefaults -p‘

# Create a test_table with two columns : Id and Name
% nistbladm -c test_data id=SI Name=SI \
test_table.org_dir.foo.‘domainname‘

# Add one entry to that table.
% nistbladm -a id=1 Name=John test_table.org_dir.foo.‘domainname‘.
# Add another entry to that table.
% nistbladm -a id=2 Name=Mary test_table.org_dir.foo.‘domainname‘.

# List the members of the group admins
% nisgrpadm -l admins.foo.‘domainname‘.
# List the contents of the foo directory
% nisls foo.‘domainname‘.
# List the contents of the test_table along with its header
% niscat -h test_table.org_dir.foo.‘domainname‘.

# Get the entry from the test_table where id = 1
% nismatch id=1 test_table.org_dir.foo.‘domainname‘.

# Delete all we created.
# First, delete yourself from the admins group
% nisgrpadm -r admins.foo.‘domainname‘. ‘nisdefaults -p‘
# Delete the admins group
% nisgrpadm -d admins.foo.‘domainname‘.
# Delete all the entries from the test_table
% nistbladm -r “[],test_table.org_dir.foo.‘domainname‘.”
# Delete the test_table itself.
% nistbladm -d test_table.org_dir.foo.‘domainname‘.
# Delete all three directories that we created
```

```
% nisrmdir groups_dir.foo.‘domainname‘.
% nisrmdir org_dir.foo.‘domainname‘.
% nisrmdir foo.‘domainname‘.
```

8

Part 5 — Appendixes

XDR Protocol Specification A

This appendix contains the XDR Protocol Language Specification.

XDR Protocol Introduction

External data representation (XDR) is a standard for the description and encoding of data. The XDR protocol is useful for transferring data between different computer architectures and has been used to communicate data between such diverse machines as the Sun® Workstation®, VAX®, IBM® PC, and Cray. XDR fits into the ISO reference model's presentation layer (layer 6) and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference between the two is that XDR uses implicit typing, while X.409 uses explicit typing.

XDR uses a language to describe data formats and only can be used to describe data; it is not a programming language. This language makes it possible to describe intricate data formats in a concise manner. The XDR language is similar to the C language. Protocols such as RPC and the NFS use XDR to describe the format of their data.

The XDR standard assumes that bytes (or octets) are portable and that a byte is defined to be 8 bits of data.

A

Graphic Box Notation

This appendix uses graphic box notation for illustration and comparison. In most illustrations, each box depicts a byte. The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered `0` through `n-1`. The bytes are read or written to some byte stream such that byte `m` always precedes byte `m+1`. The `n` bytes are followed by enough (0 to 3) residual zero bytes, `r`, to make the total byte count a multiple of four. Ellipses (`...`) between boxes show zero or more additional bytes where required. For example:

A Block

```
 +--------+--------+...+--------+--------+...+--------+
 | byte 0 | byte 1 |...|byte n-1|   0    |...|   0    |
 +--------+--------+...+--------+--------+...+--------+
 |<----------n bytes----------->|<------r bytes------>|
 |<----------n+r (where (n+r) mod 4 = 0)>------------>|
```

Basic Block Size

Choosing the XDR block size requires a trade off. Choosing a small size such as two makes the encoded data small, but causes alignment problems for machines that are not aligned on these boundaries. A large size such as eight means the data will be aligned on virtually every machine, but causes the encoded data to grow too large. Four was chosen as a compromise. Four is big enough to support most architectures efficiently, except for rare machines such as the eight-byte aligned Cray.

This is not to say that the computers cannot utilize standard `XDR`, just that they do so at a greater overhead per data item than 4-byte (32-bit) architectures. Four is also small enough to keep the encoded data restricted to a reasonable size.

The same data should encode into an equivalent result on all machines, so that encoded data can be compared or checksummed. So, variable length data must be padded with trailing zeros.

XDR Data Type Declarations

Each of the sections that follow:

- Describe a data type defined in the XDR standard
- Show how that data type is declared in the language
- Include a graphic illustration of the encoding

For each data type in the language we show a general paradigm declaration. Note that angle brackets (< and >) denote variable length sequences of data and square brackets ([and]) denote fixed-length sequences of data. `n`, `m` and `r` denote integers. For the full language specification, refer to “The XDR Language Specification” on page 307.

For some data types, specific examples are included. A more extensive example is given in the section, “XDR Data Description” on page 310.

Signed Integer

Description

An XDR signed integer is a 32-bit datum that encodes an integer in the range [-2147483648,2147483647]. The integer is represented in two’s complement notation; the most and least significant bytes are 0 and 3, respectively.

Declaration

Integers are declared:

```
int identifier;
```

Encoding

```
Integer
 (MSB)                           (LSB)
 +--------+--------+--------+--------+
 | byte 0 | byte 1 | byte 2 | byte 3 |
 +--------+--------+--------+--------+
 <---------------32 bits------------->
```

Unsigned Integer

Description

An XDR unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range [0, 4294967295]. The integer is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively.

Declaration

An unsigned integer is declared as follows:

```
unsigned int identifier;
```

Encoding

```
Unsigned Integer
 (MSB)                           (LSB)
 +--------+--------+--------+--------+
 | byte 0 | byte 1 | byte 2 | byte 3 |
 +--------+--------+--------+--------+
 <---------------32 bits------------->
```

Enumerations

Description

Enumerations have the same representation as signed integers and are handy for describing subsets of the integers.

Declaration

Enumerated data is declared as follows:

`enum {`*`name-identifier`* = *constant*, ... } *identifier*;

For example, an enumerated type could represent the three colors red, yellow, and blue as follows:

```
enum {RED = 2, YELLOW = 3, BLUE = 5} colors;
```

It is an error to assign to an `enum` an integer that has not been assigned in the `enum` declaration.

Encoding

See “Signed Integer,”shown previously.

Booleans

Description

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Booleans are integers of value `0` or `1`.

Declaration

Booleans are declared as follows:

`bool` *`identifier`*;

This is equivalent to:

`enum {FALSE = 0, TRUE = 1}` *`identifier`*;

Encoding

See “Signed Integer,” shown in a previous section.

Hyper Integer and Unsigned Hyper Integer

Description

The standard defines 64-bit (8-byte) numbers called `hyper int` and `unsigned hyper int` whose representations are the obvious extensions of `integer` and `unsigned integer`, defined above. They are represented in two’s complement notation; the most and least significant bytes are 0 and 7, respectively.

Declaration

Hyper integers are declared as follows:

```
hyper int identifier;
unsigned hyper int identifier;
```

Encoding

```
Hyper Integer
(MSB)                                                 (LSB)
+------+------+------+------+------+------+------+------+
|byte 0|byte 1|byte2 |byte 3|byte 4|byte 5|byte 6|byte 7|
+------+------+------+------+------+------+------+------+
<-------------------------64 bits----------------------->
```

Floating Point

Description

The standard defines the floating-point data type `float` (32-bits or 4-bytes). The encoding used is the IEEE standard for normalized single-precision floating-point numbers [1]. The following three fields describe the single-precision floating-point number:

S: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E: The exponent of the number, base 2. There are eight bits in this field. The exponent is biased by 127.

F: The fractional part of the number's mantissa, base 2. There are 23 bits are in this field.

Therefore, the floating-point number is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

Declaration

Single-precision floating-point data is declared as follows:

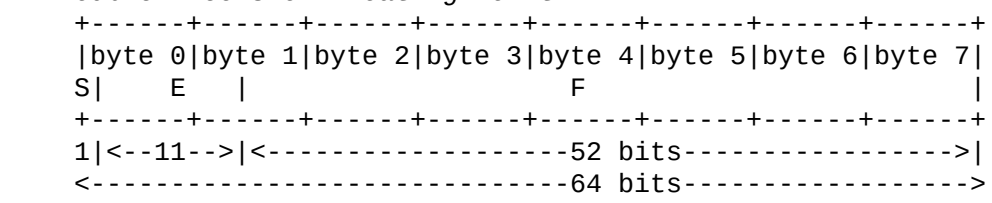
```
float identifier;
```

Declaration

`double` *identifier*;

Encoding

Double-Precision Floating Point



Just as the most and least significant bytes of an integer are 0 and 3, the most and least significant bits of a double-precision floating- point number are 0 and 63. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 12, respectively.

These offsets refer to the logical positions of the bits, *not* to their physical locations (which vary from medium to medium).

The IEEE specifications should be consulted about the encoding for signed zero, signed infinity (overflow), and de-normalized numbers (underflow) [1]. According to IEEE specifications, the `NaN` (not a number) is system dependent and should not be used externally.

Quadruple-Precision Floating Point

Description

The standard defines the encoding for the quadruple-precision floating-point data type `quadruple` (128 bits or 16 bytes). The encoding used is the IEEE standard for normalized quadruple-precision floating-point numbers [1]. The standard encodes the following three fields, which describe the quadruple-precision floating-point number:

S: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E: The exponent of the number, base 2. There are 15 bits in this field. The exponent is biased by 16383.

F: The fractional part of the number's mantissa, base 2. There are 111 bits in this field.

Therefore, the floating-point number is described by:

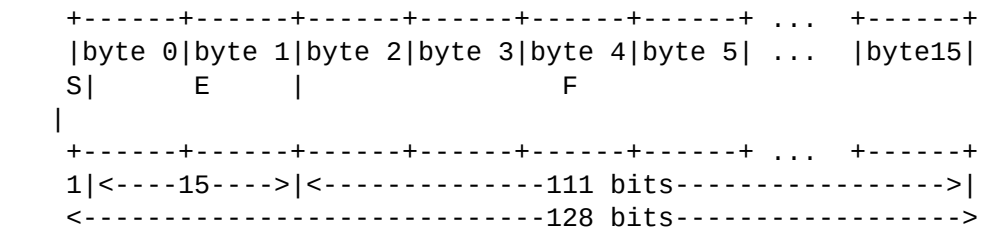
```
(-1)**S * 2**(E-Bias) * 1.F
```

Declaration

quadruple *identifier*;

Encoding

Quadruple-Precision Floating Point



Just as the most and least significant bytes of an integer are 0 and 3, the most and least significant bits of a quadruple-precision floating- point number are 0 and 127. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 16, respectively. These offsets refer to the logical positions of the bits, *not* to their physical locations (which vary from medium to medium).

The IEEE specifications should be consulted about the encoding for signed zero, signed infinity (overflow), and de-normalized numbers (underflow) [1]. According to IEEE specifications, the NaN (not a number) is system dependent and should not be used externally.

Fixed-Length Opaque Data

Description

At times, fixed-length uninterpreted data needs to be passed among machines. This data is called opaque.

Declaration

Opaque data is declared as follows:

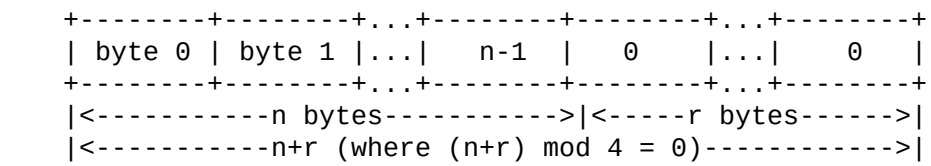
```
opaque identifier[n];
```

where the constant *n* is the (static) number of bytes necessary to contain the opaque data.The *n* bytes are followed by enough (0 to 3) residual zero bytes, *r*, to make the total byte count of the opaque object a multiple of four.

Encoding

The *n* bytes are followed by enough (0 to 3) residual zero bytes, `r`, to make the total byte count of the opaque object a multiple of four.

Fixed-Length Opaque



Variable-Length Opaque Data

Description

The standard also provides for variable-length (counted) opaque data, defined as a sequence of `n` (numbered `0` through `n-1`) arbitrary bytes to be the number `n` encoded as an unsigned integer (as described subsequently), and followed by the `n` bytes of the sequence.

Byte `b` of the sequence always precedes byte `b+1` of the sequence, and byte `0` of the sequence always follows the sequence's length (count). The `n` bytes are followed by enough (0 to 3) residual zero bytes, `r`, to make the total byte count a multiple of four.

Declaration

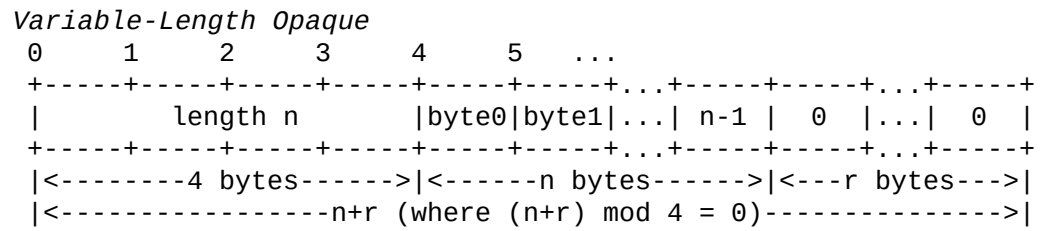
Variable-length opaque data is declared in the following way:

```
opaque identifier<m>;
 or
opaque identifier<>;
```

The constant *m* denotes an upper bound of the number of bytes that the sequence may contain. If *m* is not specified, as in the second declaration, it is assumed to be `(2**32) - 1`, the maximum length. For example, a filing protocol may state that the maximum data transfer size is `8192` bytes, as follows:

```
opaque filedata<8192>;
```

Encoding



It is an error to encode a length greater than the maximum described in the specification.

Counted Byte Strings

Description

The standard defines a string of `n` (numbered `0` through `n-1`) ASCII bytes to be the number `n` encoded as an unsigned integer (as described previously), and followed by the `n` bytes of the string. Byte `b` of the string always precedes byte `b+1` of the string, and byte `0` of the string always follows the string's length. The `n` bytes are followed by enough (0 to 3) residual zero bytes, `r`, to make the total byte count a multiple of four.

Declaration

Counted byte strings are declared as follows:

```
string object<m>;
```

or

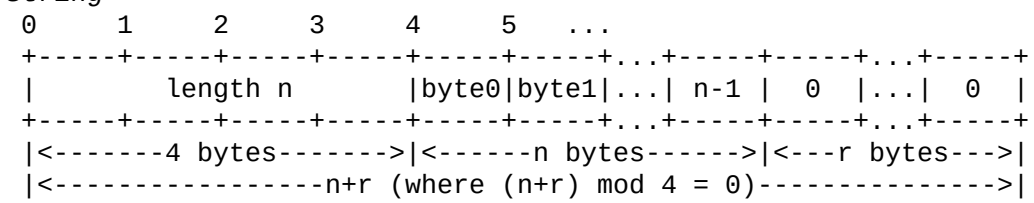
```
string object<>;
```

The constant *m* denotes an upper bound of the number of bytes that a string may contain. If *m* is not specified, as in the second declaration, it is assumed to be `(2**32) - 1`, the maximum length. The constant *m* would normally be found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than `255` bytes, as follows:

```
string filename<255>;
```

Encoding

String



It is an error to encode a length greater than the maximum described in the specification.

Fixed-Length Array

Fixed-length arrays of elements numbered `0` through `n-1` are encoded by individually encoding the elements of the array in their natural order, `0` through `n-1`. Each element's size is a multiple of four bytes. Though all elements are of the same type, the elements may have different sizes. For example, in a fixed-length array of strings, all elements are of type `string`, yet each element will vary in its length.

Declaration

Declarations for fixed-length arrays of homogenous elements are in the following form:

type-name identifier`[`*n*`];`

Encoding

```
Fixed-Length Array
 +---+---+---+---+---+---+---+---+...+---+---+---+---+
 |   element 0   |   element 1   |...|   element n-1 |
 +---+---+---+---+---+---+---+---+...+---+---+---+---+
 |<--------------------n elements------------------->|
```

Variable-Length Array

Description

Counted arrays allow variable-length arrays to be encoded as homogeneous elements: the element count n (an unsigned integer) is followed by each array element, starting with element `0` and progressing through element `n-1`.

Declaration

The declaration for variable-length arrays follows this form:

type-name identifier<m>;
or
type-name identifier<>;

The constant *m* specifies the maximum acceptable element count of an array. If *m* is not specified, it is assumed to be `(2**32) - 1`.

Encoding

```
Counted Array
 0  1  2  3
 +--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
 |     n     | element 0 | element 1 |...|element n-1|
 +--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
 |<-4 bytes->|<--------------n elements------------->|
```

It is an error to encode a length greater than the maximum described in the specification.

Structure

Description

The components of the structure are encoded in the order of their declaration in the structure. Each component's size is a multiple of four bytes, though the components may be different sizes.

Declaration

Structures are declared as follows:

```
struct {
    component-declaration-A;
    component-declaration-B;
    ...
 } identifier;
```

Encoding

```
Structure
 +-------------+-------------+...
 | component A | component B |...
 +-------------+-------------+...
```

Discriminated Union

Description

A discriminated union is a type composed of a discriminant followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of discriminant is either `int`, `unsigned int`, or an enumerated type, such as `bool`. The component types are called *arms* of the union, and are preceded by the value of the discriminant that implies their encoding.

Declaration

Discriminated unions are declared as follows:

```
union switch (discriminant-declaration) {
    case discriminant-value-A:
        arm-declaration-A;
```

```
        case discriminant-value-B:
                arm-declaration-B;
        ...
        default:
                default-declaration;
} identifier;
```

Each `case` keyword is followed by a legal value of the discriminant. The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. The size of the implied arm is always a multiple of four bytes.

The discriminated union is encoded as its discriminant followed by the encoding of the implied arm.

Encoding

```
Discriminated Union
 0   1   2   3
 +---+---+---+---+---+---+---+---+
 | discriminant  |  implied arm  |
 +---+---+---+---+---+---+---+---+
 |<---4 bytes--->|
```

Void

Description

An XDR `void` is a 0-byte quantity. Voids are useful for describing operations that take no data as input or no data as output. They are also useful in unions, where some arms may contain data and others do not.

Declaration

The declaration is simply as follows:

```
void;
```

Voids are illustrated as follows:

```
   ++
   ||
   ++
 --><-- 0 bytes
```

Constant

Description

`const` is used to define a symbolic name for a constant; it does not declare any data. The symbolic constant may be used anywhere a regular constant may be used.

The following example defines a symbolic constant `DOZEN`, equal to `12`.

```
const DOZEN = 12;
```

Declaration

The declaration of a constant follows this form:

```
const name-identifier = n;
```

Typedef

`typedef` does not declare any data either, but serves to define new identifiers for declaring data. The syntax is:

```
typedef declaration;
```

The new type name is actually the variable name in the declaration part of the `typedef`. The following example defines a new type called `eggbox` using an existing type called `egg` and the symbolic constant `DOZEN`:

```
typedef egg eggbox[DOZEN];
```

Variables declared using the new type name have the same type as the new type name would have in the `typedef`, if it were considered a variable. For example, the following two declarations are equivalent in declaring the variable `fresheggs`:

```
eggbox fresheggs;
egg fresheggs[DOZEN];
```

When a `typedef` involves a `struct`, `enum`, or `union` definition, there is another (preferred) syntax that may be used to define the same type. In general, a `typedef` of the following form:

```
typedef <<struct, union, or enum definition>> identifier;
```

may be converted to the alternative form by removing the `typedef` part and placing the identifier after the `struct`, `enum`, or `union` keyword, instead of at the end. For example, here are the two ways to define the type `bool`:

```
typedef enum {/* using typedef */
    FALSE = 0,
    TRUE = 1
} bool;
enum bool {/* preferred alternative */
    FALSE = 0,
    TRUE = 1
};
```

This syntax is preferred because one does not have to go to the end of a declaration to learn the name of the new type.

Optional-Data

Optional-data is one kind of union that occurs so frequently that it is given a special syntax of its own for declaring it. It is declared as follows:

type-name ***identifier*;

This is equivalent to the following union:

```
union switch (bool opted) {
    case TRUE:
    type-name element;
    case FALSE:
    void;
} identifier;
```

It is also equivalent to the following variable-length array declaration, since the Boolean `opted` can be interpreted as the length of the array:

type-name identifier<1>;

Optional-data is useful for describing recursive data-structures, such as linked lists and trees.

The XDR Language Specification

Notational Conventions

This specification uses a modified Backus-Naur Form notation for describing the XDR language. Here is a brief description of the notation:

1. The characters `|`, *(,),* *[,],* and `*` are special.
2. Terminal symbols are strings of any characters embedded in quotes (").
3. Nonterminal symbols are strings of nonspecial *italic* characters.
4. Alternative items are separated by a vertical bar (`|`).
5. Optional items are enclosed in brackets.
6. Items are grouped together by enclosing them in parentheses.
7. A * following an item means 0 or more occurrences of the item.

For example, consider the following pattern:

```
"a " "very" (", " " very")* [" cold " "and"] " rainy " ("day" |
"night")
```

An infinite number of strings match this pattern. A few of them are:

```
a very rainy day
a very, very rainy day
a very cold and rainy day
a very, very, very cold and rainy night
```

Lexical Notes

1. Comments begin with `/*` and end with `*/`.
2. White space serves to separate items and is otherwise ignored.
3. An identifier is a letter followed by an optional sequence of letters, digits, or underbars (_). The case of identifiers is not ignored.
4. A constant is a sequence of one or more decimal digits, optionally preceded by a minus-sign (-).

A

Code Example A-1 XDR Specification

```
Syntax Information

declaration:
    type-specifier identifier
    | type-specifier identifier "[" value "]"
    | type-specifier identifier "<" [ value ] ">"
    | "opaque" identifier "[" value "]"
    | "opaque" identifier "<" [ value ] ">"
    | "string" identifier "<" [ value ] ">"
    | type-specifier "*" identifier
    | "void"

value:
    constant
    | identifier

type-specifier:
     [ "unsigned" ] "int"
    | [ "unsigned" ] "hyper"
    | "float"
    | "double"
    | "quadruple"
    | "bool"
    | enum-type-spec
    | struct-type-spec
    | union-type-spec
    | identifier

enum-type-spec:
    "enum" enum-body

enum-body:
    "{"
    ( identifier "=" value )
    ( "," identifier "=" value )*
    "}"

struct-type-spec:
    "struct" struct-body

struct-body:
    "{"
```

```
    ( declaration ";" )
    ( declaration ";" )*
    "}"

 union-type-spec:
    "union" union-body

union-body:
    "switch" "(" declaration ")" "{"
    ( "case" value ":" declaration ";" )
    ( "case" value ":" declaration ";" )*
    [ "default" ":" declaration ";" ]
    "}"

constant-def:
    "const" identifier "=" constant ";"

type-def:
    "typedef" declaration ";"
    | "enum" identifier enum-body ";"
    | "struct" identifier struct-body ";"
    | "union" identifier union-body ";"

definition:
    type-def
    | constant-def

specification:
    definition *
```

Syntax Notes

1. The following are keywords and cannot be used as identifiers:

Table A-1 XDR Keywords

bool	**const**	**enum**	**int**	**string**	**typedef**	void
case	**default**	**float**	**opaque**	**struct**	union	
char	**double**	**hyper**	**quadruple**	**switch**	unsigned	

2. Only unsigned constants may be used as size specifications for arrays. If an identifier is used, it must have been declared previously as an unsigned constant in a `const` definition.

3. Constant and type identifiers within the scope of a specification are in the same name space and must be declared uniquely within this scope.

4. Similarly, variable names must be unique within the scope of `struct` and `union` declarations. Nested `struct` and `union` declarations create new scopes.

5. The discriminant of a `union` must be of a type that evaluates to an integer. That is, `int`, `unsigned int`, `bool`, an `enum` type, or any `typedef` that evaluates to one of these. Also, the `case` values must be legal discriminant values. Finally, a `case` value may not be specified more than once within the scope of a `union` declaration.

XDR Data Description

Here is a short XDR data description of a file data structure, which might be used to transfer files from one machine to another.

Code Example A-2 XDR File Data Structure

```
const MAXUSERNAME = 32;    /* max length of a user name */
const MAXFILELEN = 65535;  /* max length of a file */
const MAXNAMELEN = 255;    /* max length of a file name */

/* Types of files: */
enum filekind {
    TEXT = 0, /* ascii data */
    DATA = 1, /* raw data */
    EXEC = 2  /* executable */
 };

/* File information, per kind of file: */
union filetype switch (filekind kind) {
    case TEXT:
        void;                           /* no extra information */
    case DATA:
        string creator<MAXNAMELEN>;     /* data creator */
    case EXEC:
        string interpretor<MAXNAMELEN>; /*proginterptr*/
};
```

```
/* A complete file: */
struct file {
    string filename<MAXNAMELEN>;        /* name of file */
    filetype type;                      /* info about file */
    string owner<MAXUSERNAME>;          /* owner of file */
    opaque data<MAXFILELEN>;            /* file data */
};
```

Suppose now that there is a user named `john` who wants to store his LISP program `sillyprog` that contains just the data "quit." His file would be encoded as follows:

Table 8-4 XDR Data Description Example

Offset	Hex Bytes	ASCII	Description
0	00 00 00 09	....	Length of filename = 9
4	73 69 6c 6c	sill	Filename characters
8	79 70 72 6f	ypro	... and more characters ...
12	67 00 00 00	g...	.. and 3 zero-bytes of fill
16	00 00 00 02	....	Filekind is EXEC = 2
20	00 00 00 04	....	Length of interpretor = 4
24	6c 69 73 70	lisp	Interpretor characters
28	00 00 00 04	....	Length of owner = 4
32	6a 6f 68 6e	john	Owner characters
36	00 00 00 06	....	Length of file data = 6
40	28 71 75 69	(qu	File data bytes ...
44	74 29 00 00	t)..	... and 2 zero-bytes of fill

[1]
"IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August 1985.

A

RPC Language Reference

The RPC language is an extension of the XDR language. The sole extension is the addition of the `program` and `version` types.

For a description of the RPC extensions to the XDR language, see Appendix C, "RPC Protocol and Language Specification."

The RPC language is similar to C. This section describes the syntax of the RPC language, showing a few examples along the way. It also shows how RPC and XDR type definitions get compiled into C type definitions in the output header file.

An RPC language file consists of a series of definitions.

```
definition-list:
    definition;
    definition; definition-list
```

It recognizes six types of definitions.

```
definition:
        enum-definition
        const-definition
        typedef-definition
        struct-definition
        union-definition
        program-definition
```

Definitions are not the same as declarations. No space is allocated by a definition – only the type definition of a single or series of data elements. This means that variables still must be declared.

Enumerations

RPC/XDR enumerations have similar syntax as C enumerations.

```
enum-definition:
    "enum" enum-ident "{"
        enum-value-list
    "}"
enum-value-list:
    enum-value
    enum-value "," enum-value-list
```

```
enum-value:
    enum-value-ident
    enum-value-ident "=" value
```

Here is an example of an XDR `enum` and the C `enum` to which it gets compiled.

```
enum colortype {                enum colortype {
    RED = 0,                        RED = 0,
    GREEN = 1,        -->           GREEN = 1,
    BLUE = 2                        BLUE = 2,
};                              };
                                typedef enum colortype colortype;
```

Constants

XDR symbolic constants may be used wherever an integer constant is used. For example, in array size specifications:

```
    const-definition:
        const const-ident = integer
```

The following example defines a constant, `DOZEN` as equal to `12`:

```
const DOZEN = 12; --> #define DOZEN 12
```

Type Definitions

XDR `typedef`s have the same syntax as C `typedef`s.

```
typedef-definition:
        typedef declaration
```

This example defines an `fname_type` used for declaring file name strings that have a maximum length of 255 characters.

```
typedef string fname_type<255>; --> typedef char *fname_type;
```

Declarations

In XDR, there are four kinds of declarations. These declarations must be a part of a `struct` or a `typedef`; they cannot stand alone:

```
declaration:
```

```
simple-declaration
fixed-array-declaration
variable-array-declaration
pointer-declaration
```

Simple Declarations

Simple declarations are just like simple C declarations:

```
simple-declaration:
        type-ident variable-ident
```

Example:

```
    colortype color; --> colortype color;
```

Fixed-Length Array Declarations

Fixed-length array declarations are just like C array declarations:

```
fixed-array-declaration:
        type-ident variable-ident [value]
```

Example:

```
colortype palette[8]; --> colortype palette[8];
```

Many programmers confuse variable declarations with type declarations. It is important to note that `rpcgen` does not support variable declarations. This example is a program that will not compile:

```
int data[10];
program P {
    version V {
        int PROC(data) = 1;
    } = 1;
} = 0x200000;
```

The example above will not compile because of the variable declaration:

```
int data[10]
```

Instead, use:

```
typedef int data[10];
```

or

```
struct data {int dummy [10]};
```

Variable-Length Array Declarations

Variable-length array declarations have no explicit syntax in C. The XDR language does have a syntax, using angle brackets:

```
variable-array-declaration:
        type-ident variable-ident <value>
        type-ident variable-ident < >
```

The maximum size is specified between the angle brackets. The size may be omitted, indicating that the array may be of any size:

```
int heights<12>; /* at most 12 items */
int widths<>; /* any number of items */
```

Because variable-length arrays have no explicit syntax in C, these declarations are compiled into `struct` declarations. For example, the `heights` declaration compiled into the following `struct`:

```
struct {
    u_int heights_len;              /* # of items in array */
    int *heights_val;               /* pointer to array */
} heights;
```

The number of items in the array is stored in the `_len` component and the pointer to the array is stored in the `_val` component. The first part of each component name is the same as the name of the declared XDR variable (`heights`).

Pointer Declarations

Pointer declarations are made in XDR exactly as they are in C. Address pointers are not really sent over the network; instead, XDR pointers are useful for sending recursive data types such as lists and trees. The type is called “optional-data,” not “pointer,” in XDR language:

```
pointer-declaration:
    type-ident *variable-ident
```

Example:

```
listitem *next; --> listitem *next;
```

A

Structures

An RPC/XDR `struct` is declared almost exactly like its C counterpart. It looks like the following:

```
struct-definition:
    struct struct-ident "{"
        declaration-list
    "}"
declaration-list:
    declaration ";"
    declaration ";" declaration-list
```

The following XDR structure is an example of a two-dimensional coordinate and the C structure that it compiles into:

```
struct coord {                  struct coord {
    int x;            -->           int x;
    int y;                          int y;
};                                  };
                                    typedef struct coord coord;
```

The output is identical to the input, except for the added `typedef` at the end of the output. This enables one to use `coord` instead of `struct coord` when declaring items.

Unions

XDR unions are discriminated unions, and do not look like C unions – they are more similar to Pascal variant records:

```
union-definition:
    "union" union-ident "switch" "("simple declaration")" "{"
        case-list
    "}"
case-list:
    "case" value ":" declaration ";"
    "case" value ":" declaration ";" case-list
    "default" ":" declaration ";"
```

The following is an example of a type returned as the result of a “read data” operation: If there is no error, return a block of data; otherwise, don’t return anything.

```
union read_result switch (int errno) {
    case 0:
        opaque data[1024];
    default:
        void;
    };
```

It compiles into the following:

```
struct read_result {
    int errno;
    union {
        char data[1024];
    } read_result_u;
};
typedef struct read_result read_result;
```

Notice that the union component of the output struct has the same name as the type name, except for the trailing _u.

Programs

RPC programs are declared using the following syntax:

```
program-definition:
    "program" program-ident "{"
        version-list
    "}" "=" value;

version-list:
    version ";"
    version ";" version-list

version:
    "version" version-ident "{"
        procedure-list
    "}" "=" value;

procedure-list:
    procedure ";"
    procedure ";" procedure-list

procedure:
    type-ident procedure-ident "(" type-ident ")" "=" value;
```

When the `-N` option is specified, `rpcgen` also recognizes the following syntax:

```
procedure:
    type-ident procedure-ident "(" type-ident-list ")" "=" value;
```

```
type-ident-list:
    type-ident
    type-ident "," type-ident-list
```

For example:

```
/*
 * time.x: Get or set the time. Time is represented as seconds
 * since 0:00, January 1, 1970.
 */
program TIMEPROG {
    version TIMEVERS {
        unsigned int TIMEGET(void) = 1;
        void TIMESET(unsigned) = 2;
    } = 1;
} = 0x20000044;
```

Note that the `void` argument type means that no argument is passed.

This file compiles into these `#define` statements in the output header file:

```
#define TIMEPROG 0x20000044
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

Special Cases

There are several exceptions to the RPC language rules.

C-style Mode

In the new features section we talked about the features of the C-style mode of `rpcgen`. These features have implications with regard to the passing of `void` arguments. No arguments need be passed if their value is `void`.

Booleans

C has no built-in boolean type. However, the RPC library uses a boolean type called `bool_t` that is either `TRUE` or `FALSE`. Parameters declared as type `bool` in XDR language are compiled into `bool_t` in the output header file.

Example:

```
bool married; --> bool_t married;
```

Strings

The C language has no built-in string type, but instead uses the null-terminated `char *` convention. In C, strings are usually treated as null-terminated single-dimensional arrays.

In XDR language, strings are declared using the `string` keyword, and compiled into type `char *` in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the `NULL` character). The maximum size may be omitted, indicating a string of arbitrary length.

Examples:

```
string name<32>;   --> char *name;
string longname<>; --> char *longname;
```

Note – `NULL` strings cannot be passed; however, a zero-length string (that is, just the terminator or `NULL` byte) can be passed.

Opaque Data

Opaque data is used in XDR to describe untyped data, that is, sequences of arbitrary bytes. It may be declared either as a fixed length or variable length array. Examples:

```
opaque diskblock[512]; --> char diskblock[512];

opaque filedata<1024>; --> struct {
                               u_int filedata_len;
                               char *filedata_val;
                           } filedata;
```

Voids

In a void declaration, the variable is not named. The declaration is just `void` and nothing else. Void declarations can only occur in two places: union definitions and program definitions (as the argument or result of a remote procedure, for example no arguments are passed.)

A

XDR Technical Note B

This appendix is a technical note on SunSoft's implementation of the external data representation (XDR) standard, a set of library routines that enable C programmers to describe arbitrary data structures in a machine-independent fashion.

What is XDR

XDR is the backbone of SunSoft's Remote Procedure Call package, in the sense that data for RPCs are transmitted using this standard. XDR library routines should be used to transmit data accessed (read or written) by more than one type of machine.

XDR works across different languages, operating systems, and machine architectures. Most users (particularly RPC users) only need the information in the sections on Number Filters, Floating Point Filters, and Enumeration Filters. Programmers wanting to implement RPC and XDR on new machines will be interested in this technical note and the protocol specification.

`rpcgen` can be used to write XDR routines even in cases where no RPC calls are being made.

C programs that use XDR routines must include the file `<rpc/xdr.h>`, which contains all the necessary interfaces to the XDR system. Since the library `libnsl.a` contains all the XDR routines, compile as follows:

```
example% cc program.c
```

B

In many environments several criteria must be observed to accomplish porting. It is not always easy to see the ramifications of a small programmatic change, but they can often have far reaching implications. Consider the examples of a program to read / write a line of text, shown in Code Example B-1 and Code Example B-2.

Code Example B-1 Writer Example (initial)

```
#include <stdio.h>

main()         /* writer.c */
{
    long i;

for (i = 0; i < 8; i++) {
        if (fwrite((char *) &i, sizeof(i), 1, stdout) != 1) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
    }
    exit(0);
}
```

Code Example B-2 Reader Example (initial)

```
#include <stdio.h>

main()       /* reader.c */
{
    long i, j;

    for (j = 0; j < 8; j++) {
        if (fread((char *) &i, sizeof(i), 1, stdin) != 1) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
        printf("%ld ", i);
    }
    printf("\n");
    exit(0);
}
```

The two programs appear to be portable, because (a) they pass `lint` checking, and (b) they exhibit the same behavior when executed on two different hardware architectures, a SPARC and a VAX.

Piping the output of the `writer` program to the `reader` program gives identical results on SPARC or VAX.

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%
vax% writer | reader
0 1 2 3 4 5 6 7
vax%
```

With the advent of local area networks and 4.2BSD came the concept of "network pipes" — a process produces data on one machine, and a second process consumes data on another machine. A network pipe can be constructed with `writer` and `reader`. Here are the results if the first produces data on a SPARC, and the second consumes data on a VAX.

```
sun% writer | rsh vax reader
0 16777216 33554432 50331648 67108864 83886080 100663296
117440512
sun%
```

Identical results can be obtained by executing `writer` on the VAX and `reader` on the SPARC. These results occur because the byte ordering of long integers differs between the VAX and the SPARC, even though word size is the same. Note that 16777216 is 2^{24} — when four bytes are reversed, the 1 is placed in the 24th bit.

Whenever data is shared by two or more machine types, there is a need for portable data. Programs can be made data-portable by replacing the read() and write() calls with calls to an XDR library routine, xdr_long(), a filter that knows the standard representation of a long integer in its external form. The revised versions of writer are shown in Code Example B-3.

Code Example B-3 Writer Example (XDR modified)

```
#include <stdio.h>
#include <rpc/rpc.h>/* xdr is a sub-library of rpc */

main()       /* writer.c */
{
    XDR xdrs;
    long i;

xdrstdio_create(&xdrs, stdout, XDR_ENCODE);
    for (i = 0; i < 8; i++) {
        if (!xdr_long(&xdrs, &i)) {
```

```
            fprintf(stderr, "failed!\n");
            exit(1);
        }
    }
    exit(0);
}
```

Code Example B-4 shows the revised versions of reader.

Code Example B-4 Reader Example (XDR modified)

```
#include <stdio.h>
#include <rpc/rpc.h>/* xdr is a sub-library of rpc */

main()                /* reader.c */
{
    XDR xdrs;
    long i, j;

xdrstdio_create(&xdrs, stdin, XDR_DECODE);
    for (j = 0; j < 8; j++) {
        if (!xdr_long(&xdrs, &i)) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
        printf("%ld ", i);
    }
    printf("\n");
    exit(0);
}
```

The new programs were executed on a SPARC, on a VAX, and from a SPARC to a VAX; the results are shown below.

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%
vax% writer | reader
0 1 2 3 4 5 6 7
vax%
sun% writer | rsh vax reader
0 1 2 3 4 5 6 7
sun%
```

Note – Integers are just the tip of the portable-data iceberg. Arbitrary data structures present portability problems, particularly with respect to alignment and pointers. Alignment on word boundaries may cause the size of a structure to vary from machine to machine. And pointers, which are very convenient to use, have no meaning outside the machine where they are defined.

A Canonical Standard

XDR's approach to standardizing data representations is canonical. That is, XDR defines a single byte order, a single floating-point representation (IEEE), and so on. Any program running on any machine can use XDR to create portable data by translating its local representation to the XDR standard representations. Similarly, any program running on any machine can read portable data by translating the XDR standard representations to its local equivalents. The single standard completely decouples programs that create or send portable data from those that use or receive portable data. The advent of a new machine or a new language has no effect upon the community of existing portable-data creators and users. A new machine joins this community by being "taught" how to convert the standard representations and its local representations; the local representations of other machines are irrelevant. Conversely, to existing programs running on other machines, the local representations of the new machine are also irrelevant; such programs can immediately read portable data produced by the new machine because such data conforms to the canonical standards that they already understand.

There are strong precedents for XDR's canonical approach. For example, TCP/IP, UDP/IP, XNS, Ethernet, and, indeed, all protocols below layer five of the ISO model, are canonical protocols. The advantage of any canonical approach is simplicity; in the case of XDR, a single set of conversion routines is written once and is never touched again. The canonical approach has a disadvantage, but it is unimportant in real-world data transfer applications. Suppose two Little-Endian machines are transferring integers according to the XDR standard. The sending machine converts the integers from Little-Endian byte order to XDR (Big-Endian) byte order; the receiving machine performs the reverse conversion. Because both machines observe the same byte order, their conversions are unnecessary. The point, however, is not necessity, but cost as compared to the alternative.

B

The time spent converting to and from a canonical representation is insignificant, especially in distributed applications. Most of the time required to prepare a data structure for transfer is not spent in conversion but in traversing the elements of the data structure. To transmit a tree, for example, each leaf must be visited and each element in a leaf record must be copied to a buffer and aligned there; storage for the leaf may have to be de-allocated as well. Similarly, to receive a tree, storage must be allocated for each leaf, data must be moved from the buffer to the leaf and properly aligned, and pointers must be constructed to link the leaves together. Every machine pays the cost of traversing and copying data structures whether or not conversion is required. In distributed applications, communications overhead—the time required to move the data down through the sender's protocol layers, across the network and up through the receiver's protocol layers—dwarfs conversion overhead.

The XDR Library

The XDR library not only solves data portability problems, it also allows you to write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it can make sense to use the library even when the data is not shared among machines on a network.

The XDR library has filter routines for strings (null-terminated arrays of bytes), structures, unions, and arrays, to name a few. Using more primitive routines, you can write your own specific XDR routines to describe arbitrary data structures, including elements of arrays, arms of unions, or objects pointed at from other structures. The structures themselves may contain arrays of arbitrary elements, or pointers to other structures.

Look closely at the two programs. There is a family of XDR stream creation routines in which each member treats the stream of bits differently. In the example, data is manipulated using standard I/O routines, so you use `xdrstdio_create()`. The parameters to XDR stream creation routines vary according to their function. In the example, `xdrstdio_create()` takes a pointer to an XDR structure that it initializes, a pointer to a `FILE` that the input or output is performed on, and the operation. The operation may be `XDR_ENCODE` for serializing in the writer program, or `XDR_DECODE` for deserializing in the reader program.

Note – RPC users never need to create XDR streams; the RPC system itself creates these streams, which are then passed to the users.

The xdr_long() primitive is characteristic of most XDR library primitives and all client XDR routines. First, the routine returns FALSE (0) if it fails, and TRUE (1) if it succeeds. Second, for each data type, xxx, there is an associated XDR routine of the form:

```
xdr_xxx(xdrs, xp)
    XDR *xdrs;
    xxx *xp;
{
}
```

In this case, xxx is long, and the corresponding XDR routine is a primitive, xdr_long(). The client could also define an arbitrary structure xxx in which case the client would also supply the routine xdr_xxx(), describing each field by calling XDR routines of the appropriate type. In all cases the first parameter, xdrs can be treated as an opaque handle, and passed to the primitive routines.

XDR routines are direction independent; that is, the same routines are called to serialize or deserialize data. This feature is critical to software engineering of portable data. The idea is to call the same routine for either operation — this almost guarantees that serialized data can also be deserialized. One routine is used by both producer and consumer of networked data. This is implemented by always passing the address of an object rather than the object itself — only in the case of deserialization is the object modified. This feature is not shown in our trivial example, but its value becomes obvious when nontrivial data structures are passed among machines. If needed, the user can obtain the direction of the XDR operation. For details, see the section, “Operation Directions” on page 344.

A slightly more complicated example follows. Assume that a person’s gross assets and liabilities are to be exchanged among processes. Also assume that these values are important enough to warrant their own data type:

```
struct gnumbers {
    long g_assets;
    long g_liabilities;
};
```

The corresponding XDR routine describing this structure is:

```
bool_t                      /* TRUE is success, FALSE is failure */
xdr_gnumbers(xdrs, gp)
    XDR *xdrs;
    struct gnumbers *gp;
```

```
{
    if (xdr_long(xdrs, &gp->g_assets) &&
           xdr_long(xdrs, &gp->g_liabilities))
        return(TRUE);
    return(FALSE);
}
```

Note that the parameter `xdrs` is never inspected or modified; it is only passed on to the subcomponent routines. It is imperative to inspect the return value of each XDR routine call, and to give up immediately and return `FALSE` if the subroutine fails.

This example also shows that the type `bool_t` is declared as an integer whose only values are `TRUE` (1) and `FALSE` (0). This document uses the following definitions:

```
#define bool_t int
#define TRUE 1
#define FALSE 0
```

Keeping these conventions in mind, `xdr_gnumbers()` can be rewritten as follows:

```
xdr_gnumbers(xdrs, gp)
    XDR *xdrs;
    struct gnumbers *gp;
{
    return(xdr_long(xdrs, &gp->g_assets) &&
             xdr_long(xdrs, &gp->g_liabilities));
}
```

This document uses both coding styles.

XDR Library Primitives

This section gives a synopsis of each XDR primitive. It starts with memory allocation and the basic data types, then moves on to constructed data types. Finally, XDR utilities are discussed. The interface to these primitives and utilities is defined in the include file `<rpc/xdr.h>`, automatically included by `<rpc/rpc.h>`.

Memory Requirements for XDR Routines

When using XDR routines, there is sometimes a need to pre-allocate memory (or to determine memory requirements). In these instances where the developer needs to control the allocation and de-allocation of memory for XDR conversion routines to use there is a routine, `xdr_sizeof`, that is used to return the number of bytes needed to encode and decode data using one of the XDR filter functions (`func`). `xdr_sizeof`'s output does not include RPC headers or record markers and they must be added in to get a complete accounting of the memory required. `xdr_sizeof` returns a zero on error.

`xdr_sizeof(xdrproc_t` *`func`*`, void *`*`data`*`)`

`xdr_sizeof` is specifically useful the allocation of memory in applications that use XDR outside of the RPC environment; to select between transport protocols; and at the lower levels of RPC to perform client and server creation functions.

Code Example B-5 and Code Example B-6 illustrate two uses of `xdr_sizeof`.

Code Example B-5 `xdr_sizeof` Example #1

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

/*
 *  This function takes as input a CLIENT handle, an XDR function and
 *  a pointer to data to be XDR'd. It returns TRUE if the amount of
 *  data to be XDR'd may be sent using the transport associated with
 *  the CLIENT handle, and false otherwise.
 */
bool_t
cansend(cl, xdrfunc, xdrdata)
    CLIENT *cl;
    xdrproc_t xdrfunc;
    void   *xdrdata;
{
    int fd;
    struct t_info tinfo;

    if (clnt_control(cl, CLGET_FD, &fd) == -1) {
        /* handle clnt_control() error */
        return (FALSE);
    }
```

```
    if (t_getinfo(fd, &tinfo) == -1) {
        /* handle t_getinfo() error */
        return (FALSE);
    } else {
        if (tinfo.servtype == T_CLTS) {
            /*
             * This is a connectionless transport. Use xdr_sizeof()
             * to compute the size of this request to see whether it
             * is too large for this transport.
             */
            switch(tinfo.tsdu) {
                case 0:                     /* no concept of TSDUs */
                case -2:                        /* can't send normal
data */
                    return (FALSE);
                    break;
                case -1:                  /* no limit on TSDU size */
                    return (TRUE);
                    break;
                default:
                    if (tinfo.tsdu < xdr_sizeof(xdrfunc, xdrdata))
                        return (FALSE);
                    else
                        return (TRUE);
            }
        } else
            return (TRUE);
    }
}
```

Code Example B-6 is the second `xdr_sizeof` example.

Code Example B-6 `xdr_sizeof` Example #2

```
#include <sys/statvfs.h>
#include <sys/sysmacros.h>

/*
 *  This function takes as input a file name, an XDR function, and a
 *  pointer to data to be XDR'd. It returns TRUE if the filesystem
 *  on which the file resides has room for the additional amount of
 *  data to be XDR'd. Note that since the information statvfs(2)
 *  returns about the filesystem is in blocks you must convert the
 *  value returned by xdr_sizeof() from bytes to disk blocks.
 */
bool_t
```

```
canwrite(file, xdrfunc, xdrdata)
    char     *file;
    xdrproc_t xdrfunc;
    void     *xdrdata;
{
    struct statvfs s;

    if (statvfs(file, &s) == -1) {
        /* handle statvfs() error */
        return (FALSE);
    }

    if (s.f_bavail >= btod(xdr_sizeof(xdrfunc, xdrdata)))
        return (TRUE);
    else
        return (FALSE);
}
```

Number Filters

The XDR library provides primitives to translate between numbers and their corresponding external representations. Primitives cover the set of numbers in the types:

```
[signed, unsigned] * [short, int, long]
```

Specifically, the eight primitives are:

```
bool_t xdr_char(xdrs, op)
    XDR *xdrs;
    char *cp;

bool_t xdr_u_char(xdrs, ucp)
    XDR *xdrs;
    unsigned char *ucp;

bool_t xdr_int(xdrs, ip)
    XDR *xdrs;
    int *ip;

bool_t xdr_u_int(xdrs, up)
    XDR *xdrs;
    unsigned *up;

bool_t xdr_long(xdrs, lip)
    XDR *xdrs;
    long *lip;
```

```
bool_t xdr_u_long(xdrs, lup)
    XDR *xdrs;
    u_long *lup;
bool_t xdr_short(xdrs, sip)
    XDR *xdrs;
    short *sip;
bool_t xdr_u_short(xdrs, sup)
    XDR *xdrs;
    u_short *sup;
```

The first parameter, `xdrs`, is an XDR stream handle. The second parameter is the address of the number that provides data to the stream or receives data from it. All routines return TRUE if they complete successfully, and FALSE otherwise.

Floating Point Filters

The XDR library also provides primitive routines for C floating point types:

```
bool_t xdr_float(xdrs, fp)
    XDR *xdrs;
    float *fp;
bool_t xdr_double(xdrs, dp)
    XDR *xdrs;
    double *dp;
```

The first parameter, `xdrs` is an XDR stream handle. The second parameter is the address of the floating point number that provides data to the stream or receives data from it. Both routines return TRUE if they complete successfully, and FALSE otherwise.

Note – Since the numbers are represented in IEEE floating point, routines may fail when decoding a valid IEEE representation into a machine-specific representation, or vice versa.

Enumeration Filters

The XDR library provides a primitive for generic enumerations. The primitive assumes that a C `enum` has the same representation inside the machine as a C integer. The Boolean type is an important instance of the `enum`. The external representation of a Boolean is always TRUE (1) or FALSE (0).

```
#define bool_t int
#define FALSE  0
#define TRUE   1

#define enum_t int

bool_t xdr_enum(xdrs, ep)
    XDR *xdrs;
    enum_t *ep;

bool_t xdr_bool(xdrs, bp)
    XDR *xdrs;
    bool_t *bp;
```

The second parameters `ep` and `bp` are addresses of the associated type that provides data to, or receives data from, the stream `xdrs`.

No-Data Routine

Occasionally, an XDR routine must be supplied to the RPC system, even when no data is passed or required. The library provides such a routine:

```
bool_t xdr_void(); /* always returns TRUE */
```

Constructed Data Type Filters

Constructed or compound data type primitives require more parameters and perform more complicated functions than the primitives discussed previously. This section includes primitives for strings, arrays, unions, and pointers to structures.

Constructed data type primitives may use memory management. In many cases, memory is allocated when deserializing data with `XDR_DECODE`. Therefore, the `XDR` package must provide means to de-allocate memory. This is done by an `XDR` operation, `XDR_FREE`. To review, the three `XDR` directional operations are `XDR_ENCODE`, `XDR_DECODE`, and `XDR_FREE`.

Strings

In the C language, a string is defined as a sequence of bytes terminated by a null byte, which is not considered when calculating string length. However, when a string is passed or manipulated, a pointer to it is employed. Therefore,

the XDR library defines a string to be a `char *`, and not a sequence of characters. The external representation of a string is drastically different from its internal representation.

Externally strings are represented as sequences of ASCII characters, while internally they are represented with character pointers. Conversion between the two representations is accomplished with the routine `xdr_string()`:

```
bool_t xdr_string(xdrs, sp, maxlength)
    XDR *xdrs;
    char **sp;
    u_int maxlength;
```

The first parameter `xdrs` is the XDR stream handle. The second parameter `sp` is a pointer to a string (type `char **`). The third parameter `maxlength` specifies the maximum number of bytes allowed during encoding or decoding. Its value is usually specified by a protocol. For example, a protocol specification may say that a file name may be no longer than 255 characters. The routine returns `FALSE` if the number of characters exceeds `maxlength`, and `TRUE` if it doesn't.

The behavior of `xdr_string()` is similar to the behavior of other routines discussed in this section. The direction `XDR_ENCODE` is easiest to understand. The parameter `sp` points to a string of a certain length; if the string does not exceed `maxlength`, the bytes are serialized.

The effect of deserializing a string is subtle. First the length of the incoming string is determined; it must not exceed `maxlength`. Next `sp` is dereferenced; if the value is `NULL`, a string of the appropriate length is allocated and `*sp` is set to this string. If the original value of `*sp` is nonnull, the XDR package assumes that a target area has been allocated, which can hold strings no longer than `maxlength`. In either case, the string is decoded into the target area. The routine then appends a null character to the string.

In the `XDR_FREE` operation the string is obtained by dereferencing `sp`. If the string is not `NULL`, it is freed and `*sp` is set to `NULL`. In this operation `xdr_string()` ignores the `maxlength` parameter.

Note that you can use XDR on an empty string ("") but not on a NULL string.

Byte Arrays

Often variable-length arrays of bytes are preferable to strings. Byte arrays differ from strings in the following three ways: (1) the length of the array (the byte count) is explicitly located in an unsigned integer, (2) the byte sequence is not terminated by a null character, and (3) the external representation of the bytes is the same as their internal representation. The primitive `xdr_bytes()` converts between the internal and external representations of byte arrays:

```
bool_t xdr_bytes(xdrs, bpp, lp, maxlength)
    XDR *xdrs;
    char **bpp;
    u_int *lp;
    u_int maxlength;
```

The usage of the first, second, and fourth parameters is identical to the first, second and third parameters of `xdr_string()` respectively. The length of the byte area is obtained by dereferencing `lp` when serializing; `*lp` is set to the byte length when deserializing.

Arrays

The XDR library package provides a primitive for handling arrays of arbitrary elements. The `xdr_bytes()` routine treats a subset of generic arrays, in which the size of array elements is known to be 1, and the external description of each element is built-in. The generic array primitive, `xdr_array()` requires parameters identical to those of `xdr_bytes()` plus two more: the size of array elements, and an XDR routine to handle each of the elements. This routine is called to encode or decode each element of the array.

```
bool_t
xdr_array(xdrs, ap, lp, maxlength, elementsize, xdr_element)
    XDR *xdrs;
    char **ap;
    u_int *lp;
    u_int maxlength;
    u_int elementsize;
    bool_t (*xdr_element)();
```

The parameter `ap` is the address of the pointer to the array. If `*ap` is `NULL` when the array is being deserialized, XDR allocates an array of the appropriate size and sets `*ap` to that array. The element count of the array is obtained from `*lp` when the array is serialized; `*lp` is set to the array length when the array is deserialized. The parameter `maxlength` is the maximum number of

elements that the array is allowed to have; `elementsiz` is the byte size of each element of the array (the C function `sizeof`() can be used to obtain this value). The `xdr_element()` routine is called to serialize, deserialize, or free each element of the array.

Before defining more constructed data types, it is appropriate to present three examples.

Array Example 1

A user on a networked machine can be identified by (a) the machine name, such as `krypton`; (b) the user's UID: see the `geteuid` man page; and (c) the group numbers to which the user belongs: see the `getgroups` man page. A structure with this information and its associated XDR routine could be coded as in Code Example B-7.

Code Example B-7 Array Example #1

```
struct netuser {
    char  *nu_machinename;
    int   nu_uid;
    u_int nu_glen;
    int   *nu_gids;
 };
#define NLEN 255       /* machine names < 256 chars */
#define NGRPS 20       /* user can't be in > 20 groups */

bool_t
xdr_netuser(xdrs, nup)
    XDR *xdrs;
    struct netuser *nup;
{
    return(xdr_string(xdrs, &nup->nu_machinename, NLEN) &&
            xdr_int(xdrs, &nup->nu_uid) &&
            xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen, NGRPS,
                      sizeof (int), xdr_int));
}
```

Array Example 2

A party of network users could be implemented as an array of `netuser` structure. The declaration and its associated XDR routines are as shown in Code Example B-8.

Code Example B-8 Array Example #2

```
struct party {
    u_int p_len;
    struct netuser *p_nusers;
};
#define PLEN 500 /* max number of users in a party*/

bool_t
xdr_party(xdrs, pp)
    XDR *xdrs;
    struct party *pp;
{
    return(xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
     sizeof (struct netuser), xdr_netuser));
}
```

Array Example 3

The well-known parameters to `main`, `argc` and `argv` can be combined into a structure. An array of these structures can make up a history of commands. The declarations and XDR routines might look like Code Example B-9.

Code Example B-9 Array Example #3

```
struct cmd {
    u_int c_argc;
    char **c_argv;
};
#define ALEN 1000           /* args cannot be > 1000 chars */
 #define NARGC 100          /* commands cannot have > 100 args */

struct history {
    u_int h_len;
    struct cmd *h_cmds;
};
#define NCMDS 75           /* history is no more than 75 commands */

bool_t
xdr_wrapstring(xdrs, sp)
    XDR *xdrs;
    char **sp;
{
    return(xdr_string(xdrs, sp, ALEN));
}

bool_t
```

```
xdr_cmd(xdrs, cp)
    XDR *xdrs;
    struct cmd *cp;
{
    return(xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
            sizeof (char *), xdr_wrapstring));
}

bool_t
xdr_history(xdrs, hp)
    XDR *xdrs;
    struct history *hp;
{
    return(xdr_array(xdrs, &hp->h_cmds, &hp->h_len, NCMDS,
            sizeof (struct cmd), xdr_cmd));
}
```

The most confusing part of this example is that the routine `xdr_wrapstring()` is needed to package the `xdr_string()` routine, because the implementation of `xdr_array()` passes only two parameters to the array element description routine; `xdr_wrapstring()` supplies the third parameter to `xdr_string()`.

By now the recursive nature of the XDR library should be obvious. Let's continue with more constructed data types.

Opaque Data

In some protocols, handles are passed from a server to client. The client passes the handle back to the server at some later time. Handles are never inspected by clients; they are obtained and submitted. That is to say, handles are opaque. The `xdr_opaque()` primitive is used for describing fixed sized, opaque bytes.

```
bool_t
xdr_opaque(xdrs, p, len)
    XDR *xdrs;
    char *p;
    u_int len;
```

The parameter `p` is the location of the bytes; `len` is the number of bytes in the opaque object. By definition, the actual data contained in the opaque object are not machine portable.

In SunOS/SVR4 there is another routine for manipulating opaque data. This routine, xdr_netobj sends counted opaque data, much like xdr_opaque. Code Example B-10 illustrates the syntax of `xdr_netobj`.

Code Example B-10 `xdr_netobj` Routine

```
struct netobj {
    u_int   n_len;
    char    *n_bytes;
};
typedef struct netobj netobj;

bool_t
xdr_netobj(xdrs, np)
    XDR *xdrs;
    struct netobj *np;
```

The `xdr_netobj` routine is a filter primitive that translates between variable length opaque data and its external representation. The parameter np is the address of the netobj structure containing both a length and a pointer to the opaque data. The length may be no more than MAX_NETOBJ_SZ bytes. This routine returns TRUE if it succeeds, FALSE otherwise.

Fixed-Length Arrays

The XDR library provides a primitive, `xdr_vector()`, for fixed-length arrays, shown in Code Example B-11.

Code Example B-11 `xdr_vector` Routine

```
#define NLEN 255/* machine names must be < 256 chars */
#define NGRPS 20/* user belongs to exactly 20 groups */

struct netuser {
    char *nu_machinename;
    int nu_uid;
    int nu_gids[NGRPS];
};

bool_t
xdr_netuser(xdrs, nup)
    XDR *xdrs;
    struct netuser *nup;
{
    int i;
```

```
    if (!xdr_string(xdrs, &nup->nu_machinename, NLEN))
        return(FALSE);
    if (!xdr_int(xdrs, &nup->nu_uid))
        return(FALSE);
    if (!xdr_vector(xdrs, nup->nu_gids, NGRPS, sizeof(int),
         xdr_int))
        return(FALSE);
    return(TRUE);
}
```

Discriminated Unions

The XDR library supports discriminated unions. A discriminated union is a C union and an `enum_t` value that selects an "arm" of the union.

```
struct xdr_discrim {
    enum_t value;
    bool_t (*proc)();
};

bool_t
xdr_union(xdrs, dscmp, unp, arms, defaultarm)
    XDR *xdrs;
    enum_t *dscmp;
    char *unp;
    struct xdr_discrim *arms;
    bool_t (*defaultarm)(); /* may equal NULL */
```

First the routine translates the discriminant of the union located at `*dscmp`. The discriminant is always an `enum_t`. Next the union located at `*unp` is translated. The parameter `arms` is a pointer to an array of `xdr_discrim` structures. Each structure contains an ordered pair of `[value,proc]`. If the union's discriminant is equal to the associated `value`, then the `proc` is called to translate the union. The end of the `xdr_discrim` structure array is denoted by a routine of value `NULL` (0). If the discriminant is not found in the `arms` array, then the `defaultarm` procedure is called if it is nonnull; otherwise the routine returns `FALSE`.

Discriminated Union Example

Suppose the type of a union may be integer, character pointer (a string), or a `gnumbers` structure. Also, assume the union and its current type are declared in a structure. The declaration is:

```
enum utype {INTEGER=1, STRING=2, GNUMBERS=3};
struct u_tag {
    enum utype utype;/* the union's discriminant */
    union {
        int ival;
        char *pval;
        struct gnumbers gn;
    } uval;
};
```

Code Example B-12 constructs and XDR procedure (de)serialize the discriminated union.

Code Example B-12 XDR Discriminated Union

```
struct xdr_discrim u_tag_arms[4] = {
    {INTEGER, xdr_int},
    {GNUMBERS, xdr_gnumbers}
    {STRING, xdr_wrapstring},
    {__dontcare__, NULL}
    /* always terminate arms with a NULL xdr_proc*/
}

bool_t
xdr_u_tag(xdrs, utp)
    XDR *xdrs;
    struct u_tag *utp;
{
    return(xdr_union(xdrs, &utp->utype, &utp->uval,
           u_tag_arms, NULL));
}
```

The routine `xdr_gnumbers()` was presented above in the XDR Library section. `xdr_wrapstring()` was presented in example C. The default `arm` parameter to `xdr_union()` (the last parameter) is `NULL` in this example. Therefore the value of the union's discriminant may legally take on only values listed in the `u_tag_arms` array. This example also demonstrates that the elements of the arm's array do not need to be sorted.

It is worth pointing out that the values of the discriminant may be sparse, though in this example they are not. It is always good practice to assign explicitly integer values to each element of the discriminant's type. This practice both documents the external representation of the discriminant and guarantees that different C compilers emit identical discriminant values.

Exercise

Implement `xdr_union()` using the other primitives in this section.

Pointers

In C it is often convenient to put pointers to another structure within a structure. The `xdr_reference()` primitive makes it easy to serialize, deserialize, and free these referenced structures.

```
bool_t
xdr_reference(xdrs, pp, size, proc)
    XDR *xdrs;
    char **pp;
    u_int ssize;
    bool_t (*proc)();
```

Parameter `pp` is the address of the pointer to the structure; parameter `ssize` is the size in bytes of the structure (use the C function `sizeof()` to obtain this value); and `proc` is the XDR routine that describes the structure. When decoding data, storage is allocated if `*pp` is `NULL`.

There is no need for a primitive `xdr_struct()` to describe structures within structures, because pointers are always sufficient.

Exercise

Implement `xdr_reference()` using `xdr_array()`.

Warning – `xdr_reference()` and `xdr_array()` are *NOT* interchangeable external representations of data.

Pointer Example

Suppose there is a structure containing a person's name and a pointer to a `gnumbers` structure containing the person's gross assets and liabilities. The construct is:

```
struct pgn {
    char *name;
    struct gnumbers *gnp;
};
```

The corresponding XDR routine for this structure is:

```
bool_t
xdr_pgn(xdrs, pp)
    XDR *xdrs;
    struct pgn *pp;
{
    return(xdr_string(xdrs, &pp->name, NLEN) &&
        xdr_reference(xdrs, &pp->gnp, sizeof(struct gnumbers),
                      xdr_gnumbers));
}
```

Pointer Semantics

In many applications, C programmers attach double meaning to the values of a pointer. Typically the value `NULL` (or zero) means data is not needed, yet some application-specific interpretation applies. In essence, the C programmer is encoding a discriminated union efficiently by overloading the interpretation of the value of a pointer. For instance, in example E a `NULL` pointer value for `gnp` could indicate that the person's assets and liabilities are unknown. That is, the pointer value encodes two things: whether or not the data is known; and if it is known, where it is located in memory. Linked lists are an extreme example of the use of application-specific pointer interpretation.

The primitive `xdr_reference()` cannot and does not attach any special meaning to a null-value pointer during serialization. That is, passing an address of a pointer whose value is `NULL` to `xdr_reference()` when serializing data will most likely cause a memory fault and, on the UNIX system, a core dump.

`xdr_pointer()` correctly handles `NULL` pointers.

Nonfilter Primitives

XDR streams can be manipulated with the primitives discussed in this section.

```
u_int xdr_getpos(xdrs)
    XDR *xdrs;

bool_t xdr_setpos(xdrs, pos)
    XDR *xdrs;
    u_int pos;

xdr_destroy(xdrs)
    XDR *xdrs;
```

The routine `xdr_getpos()` returns an unsigned integer that describes the current position in the data stream. Warning: In some XDR streams, the returned value of `xdr_getpos()` is meaningless; the routine returns a –1 in this case (though –1 should be a legitimate value).

The routine `xdr_setpos`() sets a stream position to `pos`. Warning: In some XDR streams, setting a position is impossible; in such cases, `xdr_setpos()` will return `FALSE`. This routine will also fail if the requested position is out-of-bounds. The definition of bounds varies from stream to stream.

The `xdr_destroy`() primitive destroys the XDR stream. Usage of the stream after calling this routine is undefined.

Operation Directions

At times you may want to optimize XDR routines by taking advantage of the direction of the operation — `XDR_ENCODE`, `XDR_DECODE` or `XDR_FREE`. The value `xdrs->x_op` always contains the direction of the XDR operation. An example in “Linked Lists” on page 349 demonstrates the usefulness of the `xdrs->x_op` field.

Stream Access

An XDR stream is obtained by calling the appropriate creation routine. These creation routines take arguments that are tailored to the specific properties of the stream. Streams currently exist for (de)serialization of data to or from standard I/O `FILE` streams, record streams, and UNIX files, and memory.

Standard I/O Streams

XDR streams can be interfaced to standard I/O using the xdrstdio_create() routine:

```
#include <stdio.h>
#include <rpc/rpc.h>/* xdr is part of rpc */

void
xdrstdio_create(xdrs, fp, xdr_op)
    XDR *xdrs;
    FILE *fp;
    enum xdr_op x_op;
```

The routine `xdrstdio_create()` initializes an XDR stream pointed to by `xdrs`. The XDR stream interfaces to the standard I/O library. Parameter `fp` is an open file, and `x_op` is an XDR direction.

Memory Streams

Memory streams allow the streaming of data into or out of a specified area of memory:

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

void
xdrmem_create(xdrs, addr, len, x_op)
    XDR *xdrs;
    char *addr;
    u_int len;
    enum xdr_op x_op;
```

The routine `xdrmem_create`() initializes an XDR stream in local memory. The memory is pointed to by parameter `addr`; parameter `len` is the length in bytes of the memory. The parameters `xdrs` and `x_op` are identical to the corresponding parameters of `xdrstdio_create()`. Currently, the datagram implementation of RPC uses `xdrmem_create()`. Complete call or result messages are built in memory before calling the `t_sndndata`() TLI routine.

Record (TCP/IP) Streams

A record stream is an XDR stream built on top of a record marking standard that is built on top of the UNIX file or 4.2 BSD connection interface.

```
#include <rpc/rpc.h>      /* xdr is part of rpc */

xdrrec_create(xdrs, sendsize, recvsize, iohandle, readproc,
              writeproc)
    XDR *xdrs;
    u_int sendsize, recvsize;
    char *iohandle;
    int (*readproc)(), (*writeproc)();
```

The routine `xdrrec_create()` provides an XDR stream interface that allows for a bidirectional, arbitrarily long sequence of records. The contents of the records are meant to be data in XDR form. The stream's primary use is for interfacing RPC to TCP connections. However, it can be used to stream data into or out of normal UNIX files.

The parameter `xdrs` is similar to the corresponding parameter described above. The stream does its own data buffering similar to that of standard I/O. The parameters `sendsize` and `recvsize` determine the size in bytes of the output and input buffers, respectively; if their values are zero (0), then predetermined defaults are used. When a buffer needs to be filled or flushed, the routine `readproc()` or `writeproc()` is called, respectively. The usage and behavior of these routines are similar to the UNIX system calls `read()` and `write()`. However, the first parameter to each of these routines is the opaque parameter `iohandle`. The other two parameters (and `nbytes`) and the results (byte count) are identical to the system routines. If `xxx` is `readproc()` or `writeproc()`, then it has the following form:

```
/* returns the actual number of bytes transferred. -1 is an error */
int
xxx(iohandle, buf, len)
    char *iohandle;
    char *buf;
    int nbytes;
```

The XDR stream provides means for delimiting records in the byte stream. The implementation details of delimiting records in a stream are discussed in "Advanced Topics" on page 349. The primitives that are specific to record streams are as follows:

```
bool_t
xdrrec_endofrecord(xdrs, flushnow)
    XDR *xdrs;
    bool_t flushnow;

bool_t
```

```
xdrrec_skiprecord(xdrs)
    XDR *xdrs;

bool_t
xdrrec_eof(xdrs)
    XDR *xdrs;
```

The routine `xdrrec_endofrecord()` causes the current outgoing data to be marked as a record. If the parameter `flushnow` is `TRUE`, then the stream's `writeproc` will be called; otherwise, `writeproc` will be called when the output buffer has been filled.

The routine `xdrrec_skiprecord()` causes an input stream's position to be moved past the current record boundary and onto the beginning of the next record in the stream.

If there is no more data in the stream's input buffer, then the routine `xdrrec_eof`() returns `TRUE`. That is not to say that there is no more data in the underlying file descriptor.

XDR Stream Implementation

This section provides the abstract data types needed to implement new instances of XDR streams.

The XDR Object

The structure in Code Example B-13 defines the interface to an XDR stream.

Code Example B-13 XDR Stream Interface Example

```
enum xdr_op {XDR_ENCODE=0, XDR_DECODE=1, XDR_FREE=2};

typedef struct {
    enum xdr_op x_op;
    struct xdr_ops {
        bool_t (*x_getlong)();       /* get long from stream */
        bool_t (*x_putlong)();       /* put long to stream */
        bool_t (*x_getbytes)();      /* get bytes from stream */
        bool_t (*x_putbytes)();      /* put bytes to stream */
        u_int (*x_getpostn)();       /* return stream offset */
        bool_t (*x_setpostn)();      /* reposition offset */
        caddr_t (*x_inline)();       /* ptr to buffered data */
        VOID (*x_destroy)();         /* free private area */
```

```
    } *x_ops;
    caddr_t x_public;               /* users' data */
    caddr_t x_private;              /* pointer to private data */
    caddr_t x_base;                /* private for position info */
    int     x_handy;                /* extra private word */
} XDR;
```

The `x_op` field is the current operation being performed on the stream. This field is important to the XDR primitives, but should not affect a stream's implementation. That is, a stream's implementation should not depend on this value. The fields `x_private`, `x_base`, and `x_handy` are private to the particular stream's implementation. The field `x_public` is for the XDR client and should never be used by the XDR stream implementations or the XDR primitives. `x_getpostn`(), `x_setpostn`(), and `x_destroy`() are macros for accessing operations. The operation `x_inline`() takes two parameters: an `XDR *`, and an unsigned integer, which is a byte count. The routine returns a pointer to a piece of the stream's internal buffer. The caller can then use the buffer segment for any purpose. From the stream's point of view, the bytes in the buffer segment have been consumed. The routine may return NULL if it cannot return a buffer segment of the requested size. (The `x_inline()` routine is used to squeeze cycles, and the resulting buffer is not data portable. Users are cautioned against using this feature.)

The operations `x_getbytes`() and `x_putbytes`() blindly get and put sequences of bytes from or to the underlying stream; they return `TRUE` if they are successful, and `FALSE` otherwise. The routines have identical parameters (replace `xxx`):

```
bool_t
xxxbytes(xdrs, buf, bytecount)
    XDR *xdrs;
    char *buf;
    u_int bytecount;
```

The operations `x_getlong()` and `x_putlong()` receive and put long numbers from and to the data stream. It is the responsibility of these routines to translate the numbers between the machine representation and the (standard) external representation. The UNIX primitives `htonl()` and `ntohl()` can be helpful in accomplishing this. The higher-level XDR implementation assumes that signed and unsigned long integers contain the same number of bits, and that nonnegative integers have the same bit representations as unsigned integers. The routines return *TRUE* if they succeed, and FALSE otherwise.

They have identical parameters:

```
bool_t
xxxlong(xdrs, lp)
    XDR *xdrs;
    long *lp;
```

Implementors of new XDR streams must make an XDR structure (with new operation routines) available to clients, using some kind of create routine.

Advanced Topics

This section describes techniques for passing data structures that are not covered in the preceding sections. Such structures include linked lists (of arbitrary lengths). Unlike the simpler examples covered in the earlier sections, the following examples are written using both the XDR C library routines and the XDR data description language. Appendix A, "XDR Protocol Specification," describes this language in detail.

Linked Lists

The "Pointer Example" on page 343 presented a C data structure and its associated XDR routines for an individual's gross assets and liabilities. Code Example B-14 uses a linked list to duplicate the pointer example.

Code Example B-14 Linked List

```
struct gnumbers {
    long g_assets;
    long g_liabilities;
 };

bool_t
xdr_gnumbers(xdrs, gp)
    XDR *xdrs;
    struct gnumbers *gp;
{
    return(xdr_long(xdrs, &(gp->g_assets) &&
             xdr_long(xdrs, &(gp->g_liabilities)));
}
```

Now assume that you want to implement a linked list of such information. A data structure could be constructed as follows:

```
struct gnumbers_node {
    struct gnumbers gn_numbers;
    struct gnumbers_node *gn_next;
};
typedef struct gnumbers_node *gnumbers_list;
```

The head of the linked list can be thought of as the data object; that is, the head is not merely a convenient shorthand for a structure. Similarly the `gn_next` field is used to indicate whether the object has terminated. Unfortunately, if the object continues, the `gn_next` field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

The XDR data description of this linked list is described by the recursive declaration of `gnumbers_list`:

```
struct gnumbers {
    int g_assets;
    int g_liabilities;
};
struct gnumbers_node {
    gnumbers gn_numbers;
    gnumbers_node *gn_next;
};
```

In this description, the Boolean indicates whether there is more data following it. If the Boolean is `FALSE`, it is the last data field of the structure. If it is `TRUE`, it is followed by a `gnumbers` structure and (recursively) by a `gnumbers_list`. Note that the C declaration has no Boolean explicitly declared in it (though the `gn_next` field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for writing the XDR routines for a `gnumbers_list` follow easily from the XDR description above. Note how the primitive `xdr_pointer()` is used to implement the XDR union above.

Code Example B-15 `xdr_pointer`

```
bool_t
xdr_gnumbers_node(xdrs, gn)
    XDR *xdrs;
    gnumbers_node *gn;
{
```

```
    return(xdr_gnumbers(xdrs, &gn->gn_numbers) &&
            xdr_gnumbers_list(xdrs, &gp->gn_next));
}

bool_t
xdr_gnumbers_list(xdrs, gnp)
    XDR *xdrs;
    gnumbers_list *gnp;
{
    return(xdr_pointer(xdrs, gnp, sizeof(struct gnumbers_node),
                        xdr_gnumbers_node));
 xdr_pointer}
```

The unfortunate side effect of using XDR on a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list. This is due to the recursion. Code Example B-16 collapses the above two mutually recursive routines into a single, nonrecursive one.

Code Example B-16 Nonrecursive Stack in XDR

```
bool_t
xdr_gnumbers_list(xdrs, gnp)
    XDR *xdrs;
    gnumbers_list *gnp;
{
    bool_t more_data;
    gnumbers_list *nextp;

    for(;;) {
        more_data = (*gnp != NULL);
        if (!xdr_bool(xdrs, &more_data))
            return(FALSE);
        if (! more_data)
            break;
        if (xdrs->x_op == XDR_FREE)
            nextp = &(*gnp)->gn_next;
        if (!xdr_reference(xdrs, gnp,
            sizeof(struct gnumbers_node), xdr_gnumbers))
        return(FALSE);
        gnp = (xdrs->x_op == XDR_FREE) ? nextp : &(*gnp)->gn_next;
    }
    *gnp = NULL;
    return(TRUE);
}
```

The first task is to find out whether there is more data, so that this Boolean information can be serialized. Notice that this statement is unnecessary in the XDR_DECODE case, since the value of more_data is not known until you deserialize it in the next statement.

The next statement implements XDR on the more_data field of the XDR union. Then if there is truly no more data, you set this last pointer to NULL to indicate the end of the list, and return TRUE because you are done. Note that setting the pointer to NULL is only important in the XDR_DECODE case, since it is already NULL in the XDR_ENCODE and XDR_FREE cases.

Next, if the direction is XDR_FREE, the value of nextp is set to indicate the location of the next pointer in the list. We do this now because you need to dereference gnp to find the location of the next item in the list, and after the next statement, the storage pointed to by gnp will be freed up and no be longer valid. We can't do this for all directions though, because in the XDR_DECODE direction the value of gnp won't be set until the next statement.

Next, you XDR the data in the node using the primitive xdr_reference(). xdr_reference() is like xdr_pointer() which you used before, but it does not send over the Boolean indicating whether there is more data. We use it instead of xdr_pointer() because you have already used XDR on this information yourself. Notice that the XDR routine passed is not the same type as an element in the list. The routine passed is xdr_gnumbers(), but each element in the list is actually of type gnumbers_node. You don't pass xdr_gnumbers_node() because it is recursive. Instead use xdr_gnumbers() which uses XDR on all of the nonrecursive part. Note that this trick works only if the gn_numbers field is the first item in each element, so that their addresses are identical when passed to xdr_reference().

Finally, you update *gnp* to point to the next item in the list. If the direction is XDR_FREE, you set it to the previously saved value; otherwise you can dereference *gnp* to get the proper value. Though harder to understand than the recursive version, this nonrecursive routine will run more efficiently since much of the procedure call overhead has been removed. Most lists are small though (in the hundreds of items or less) and the recursive version should be sufficient for them.

RPC Protocol and Language Specification

C

This appendix specifies a message protocol used in implementing the RPC package. The message protocol is specified with the XDR language. The companion appendix to this one is Appendix A, “XDR Protocol Specification." In addition, “RPC Programming Terms” on page 449 gives definitions for the RPC terms used in this appendix.

Protocol Overview

The RPC protocol provides for the following:

- Unique specification of a procedure to be called.
- Provisions for matching response messages to request messages.
- Provisions for authenticating the caller to service and vice-versa. In addition, the RPC package provides features that detect the following:
  - RPC protocol mismatches
  - Remote program protocol version mismatches
  - Protocol errors (such as incorrect specification of a procedure’s parameters)
  - Reasons why remote authentication failed

Consider a network file service composed of two programs. One program may deal with high-level applications such as file system access control and locking. The other may deal with low-level file I/O and have procedures like "read" and "write." A client machine of the network file service would call the procedures associated with the two programs of the service on behalf of some user on the client machine. In the client-server model a remote procedure call is used to call the service.

The RPC Model

The RPC model is similar to the local procedure call model. In the local case, the caller places arguments to a procedure in some well-specified location. The caller then transfers control to the procedure, and eventually regains control. At that point, the results of the procedure are extracted from a well-specified location, and the caller continues execution.

The RPC is similar, in that one thread of control logically winds through two processes. One is the caller's process, the other is a server's process. Conceptually, the caller process sends a call message to the server process and waits (blocks) for a reply message. The call message contains the procedure's parameters, among other things. The reply message contains the procedure's results, among other things. Once the reply message is received, the results of the procedure are extracted, and the caller's execution is resumed.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message.

Note that in this description only one of the two processes is active at any given time. However, this need not be the case. The RPC protocol makes no restrictions on the concurrency model implemented. For example, an implementation may choose to have RPC calls be asynchronous, so that the client may do useful work while waiting for the reply from the server. Another possibility is to have the server create a task to process an incoming request, so that the server can be free to receive other requests.

Transports and Semantics

The RPC protocol is independent of transport protocols. That is, RPC disregards how a message is passed from one process to another. The protocol deals only with specification and interpretation of messages.

RPC does not attempt to ensure transport reliability. Therefore, you must supply the application with information about the type of transport protocol used under RPC. If you tell the RPC service it is running on top of a reliable transport such as TCP, most of the work is already done for it. On the other hand, if RPC is running on top of an unreliable transport such as UDP, the service must devise its own retransmission and time-out policy. RPC does not provide this service.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, consider RPC running on top of an unreliable transport. If an application retransmits RPC messages after short time-outs, the only thing it can infer if it receives no reply is that the procedure was executed zero or more times. If it does receive a reply, it can infer that the procedure was executed at least once.

A server may choose to remember previously granted requests from a client and not regrant them to insure some degree of execute-at-most-once semantics. A server can do this by taking advantage of the transaction ID that is packaged with every RPC request. The main use of this transaction ID is by the RPC client for matching replies to requests. However, a client application may choose to reuse its previous transaction ID when retransmitting a request. The server application, checking this fact, may choose to remember this ID after granting a request and not regrant requests with the same ID. The server is not allowed to examine this ID in any other way except as a test for equality.

On the other hand, if using a reliable transport such as TCP, the application can infer from a reply message that the procedure was executed exactly once, but if it receives no reply message, it cannot assume the remote procedure was not executed. Note that even if a connection-oriented protocol like TCP is used, an application still needs time-outs and reconnection to handle server crashes.

Binding and Rendezvous Independence

The act of binding a client to a service is *not* part of the remote procedure call specification. This important and necessary function is left up to some higher-level software. (The software may use RPC itself; see “rpcbind Protocol” on page 386.)

Implementers should think of the RPC protocol as the jump-subroutine instruction (JSR) of a network; the loader (binder) makes JSR useful, and the loader itself uses JSR to accomplish its task. Likewise, the network makes RPC useful, enabling RPC to accomplish this task.

The RPC protocol provides the fields necessary for a client to identify itself to a service and vice-versa. Security and access control mechanisms can be built on top of the message authentication. Several different authentication protocols can be supported. A field in the RPC header specifies the protocol being used. More information on authentication protocols can be found in the section “Record-Marking Standard” on page 363.

Programs and Procedures

The RPC call message has three unsigned fields that uniquely identify the procedure to be called:

- remote program number
- remote program version number
- remote procedure number

Program numbers are administered by a central authority, as described in “Program Number Registration” on page 361.

The first implementation of a program will most likely have version number 1. Because most new protocols evolve into better, stable, and mature protocols, a version field of the call message identifies the version of the protocol the caller is using. Version numbers make “speaking” old and new protocols through the same server process possible.

The procedure number identifies the procedure to be called. These numbers are documented in the individual program’s protocol specification. For example, a file service’s protocol specification may state that its procedure number 5 is “read” and procedure number 12 is “write.”

Just as remote program protocols may change over several versions, the RPC message protocol itself may change. Therefore, the call message also has in it the RPC version number, which is always equal to 2 for the version of RPC described here.

The reply message to a request message has enough information to distinguish the following error conditions:

- The remote implementation of RPC does not “speak” protocol version 2. The lowest and highest supported RPC version numbers are returned.
- The remote program is not available on the remote system.
- The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
- The requested procedure number does not exist. (This is usually a caller-side protocol or programming error.)
- The parameters to the remote procedure appear to be garbage from the server’s point of view. (Again, this is usually caused by a disagreement about the protocol between client and service.)

Provisions for authentication of caller to service and vice versa are provided as a part of the RPC protocol. The call message has two authentication fields, the credentials and verifier. The reply message has one authentication field, the response verifier. The RPC protocol specification defines all three fields to be the following opaque type:

```
enum auth_flavor {
    AUTH_NONE = 0,
    AUTH_SYS = 1,
    AUTH_SHORT = 2,
    AUTH_DES = 3,
    AUTH_KERB = 4
    /* and more to be defined */
 };
 struct opaque_auth {
    enum       auth_flavor;        /* style of credentials */
    caddr_t    oa_base;          /* address of more auth stuff */
    u_int      oa_length;        /* not to exceed MAX_AUTH_BYTES */
 };
```

An `opaque_auth` structure is an `auth_flavor` enumeration followed by bytes that are opaque to the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields are specified by individual, independent authentication protocol specifications. (See “Record-Marking Standard” on page 363 for definitions of the various authentication protocols.)

If authentication parameters are rejected, the response message contains information stating why they are rejected.

Program Number Assignment

Program numbers are distributed in groups of `0x20000000`, as shown in Table C-1.

Table C-1 RPC Program Number Assignment

Program Numbers	Description
00000000 - 1fffffff	Defined by Sun
20000000 - 3fffffff	Defined by user
40000000 - 5fffffff	Transient (Reserved for customer-written applications)
60000000 - 7fffffff	Reserved
80000000 - 9fffffff	Reserved
a0000000 - bfffffff	Reserved
c0000000 - dfffffff	Reserved
e0000000 - ffffffff	Reserved

Sun Microsystems administers the first group of numbers, which should be identical for all customers. If a customer develops an application that might be of general interest, that application should be given an assigned number in the first range.

The second group of numbers is reserved for specific customer applications. This range is intended primarily for debugging new programs.

The third group is reserved for applications that generate program numbers dynamically.

The final groups are reserved for future use, and should not be used.

Program Number Registration

To register a protocol specification, send a request by email to `rpc@sun.com`, or write to:

RPC Administrator
Sun Microsystems
2550 Garcia Avenue
Mountain View, CA 94043

Please include a compilable `rpcgen ".x"` file describing your protocol. You will be given a unique program number in return.

The RPC program numbers and protocol specifications of standard RPC services can be found in the include files in `/usr/include/rpcsvc`. These services, however, constitute only a small subset of those that have been registered.

Other Uses of the RPC Protocol

The intended use of this protocol is for calling remote procedures. That is, each call message is matched with a response message. However, the protocol itself is a message-passing protocol with which other (non-RPC) protocols can be implemented. Some of the non-RPC protocols supported by the RPC package are batching and broadcasting.

Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server; batching typically uses reliable byte stream protocols (like TCP) for its transport. In batching, the client never waits for a reply from the server, and the server does not send replies to batch requests. A sequence of batch calls is usually finished by a non-batch RPC call to flush the pipeline (with positive acknowledgment). For more information, see “Batching” on page 94.

Broadcast RPC

In broadcast RPC, the client sends a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses connectionless, packet-based protocols (like UDP) as its transports. Servers that support broadcast protocols only respond when the request is successfully processed, and are silent in the face of errors. Broadcast RPC uses the `rpcbind` service to achieve its semantics. See “Broadcast RPC” on page 92 and “rpcbind Protocol” on page 386 for further information.

The RPC Message Protocol

This section defines the RPC message protocol in the XDR data description language. The message is defined in a top-down style, as shown in Code Example C-1.

Code Example C-1 RPC Message Protocol

```
enum msg_type {
    CALL = 0,
    REPLY = 1
 };

/*
 * A reply to a call message can take on two forms: The message was
 * either accepted or rejected.
 */
 enum reply_stat {
    MSG_ACCEPTED = 0,
    MSG_DENIED = 1
 };

/*
 * Given that a call message was accepted, the following is the
 * status of an attempt to call a remote procedure.
 */
enum accept_stat {
    SUCCESS = 0,       /* RPC executed successfully */
    PROG_UNAVAIL = 1,  /* remote service hasn't exported prog */
    PROG_MISMATCH = 2, /* remote service can't support versn # */
    PROC_UNAVAIL = 3,  /* program can't support proc */
    GARBAGE_ARGS = 4   /* procedure can't decode params */
};
```

```
/*
 * Reasons a call message was rejected:
 */
enum reject_stat {
    RPC_MISMATCH = 0,  /* RPC version number != 2 */
    AUTH_ERROR = 1     /* remote can't authenticate caller */
};
/*
 * Why authentication failed:
 */
enum auth_stat {
    AUTH_BADCRED = 1,       /* bad credentials */
    AUTH_REJECTEDCRED = 2,  /* clnt must do new session */
    AUTH_BADVERF = 3,       /* bad verifier */
    AUTH_REJECTEDVERF = 4,  /* verif expired or replayed */
    AUTH_TOOWEAK = 5        /* rejected for security */
};

/*
 * The RPC message:
 * All messages start with a transaction identifier, xid, followed
 * by a two-armed discriminated union. The union's discriminant is
 * a msg_type which switches to one of the two types of the message.
 * The xid of a REPLY message always matches that of the initiating
 * CALL message. NB: The xid field is only used for clients matching
 * reply messages with call messages or for servers detecting
 * retransmissions; the service side cannot treat this id as any type
 * of sequence number.
 */
struct rpc_msg {
    unsigned int xid;
    union switch (msg_type mtype) {
        case CALL:
            call_body cbody;
        case REPLY:
            reply_body rbody;
    } body;
};

/*
 * Body of an RPC request call:
 * In version 2 of the RPC protocol specification, rpcvers must be
 * equal to 2. The fields prog, vers, and proc specify the remote
 * program, its version number, and the procedure within the remote
 * program to be called. After these fields are two authentication
 * parameters: cred (authentication credentials) and verf
```

```
 * (authentication verifier). The two authentication parameters are
 * followed by the parameters to the remote procedure, which are
 * specified by the specific program protocol.
 */
struct call_body {
    unsigned int rpcvers; /* must be equal to two (2) */
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque_auth cred;
    opaque_auth verf;
    /* procedure specific parameters start here */
 };

/*
 * Body of a reply to an RPC request:
 * The call message was either accepted or rejected.
 */
union reply_body switch (reply_stat stat) {
    case MSG_ACCEPTED:
        accepted_reply areply;
    case MSG_DENIED:
        rejected_reply rreply;
} reply;

/*
 * Reply to an RPC request that was accepted by the server: there
 * could be an error even though the request was accepted. The first
 * field is an authentication verifier that the server generates in
 * order to validate itself to the caller. It is followed by a union
 * whose discriminant is an enum accept_stat. The SUCCESS arm of the
 * union is protocol specific. The PROG_UNAVAIL, PROC_UNAVAIL, and
 * GARBAGE_ARGP arms of the union are void. The PROG_MISMATCH arm
 * specifies the lowest and highest version numbers of the remote
 * program supported by the server.
 */
struct accepted_reply {
    opaque_auth verf;
    union switch (accept_stat stat) {
        case SUCCESS:
            opaque results[0];
            /* procedure-specific results start here */
        case PROG_MISMATCH:
            struct {
                unsigned int low;
                unsigned int high;
```

```
            } mismatch_info;
        default:
            /*
             * Void. Cases include PROG_UNAVAIL, PROC_UNAVAIL, and
             * GARBAGE_ARGS.
             */
            void;
    } reply_data;
};

/*
 * Reply to an RPC request that was rejected by the server:
 * The request can be rejected for two reasons: either the server is
 * not running a compatible version of the RPC protocol
 * (RPC_MISMATCH), or the server refuses to authenticate the caller
 * (AUTH_ERROR). In case of an RPC version mismatch, the server
 * returns the lowest and highest supported RPC version numbers. In
 * case of refused authentication, failure status is returned.
 */
union rejected_reply switch (reject_stat stat) {
    case RPC_MISMATCH:
        struct {
            unsigned int low;
            unsigned int high;
        } mismatch_info;
    case AUTH_ERROR:
        auth_stat stat;
};
```

Record-Marking Standard

When RPC messages are passed on top of a byte stream transport (like TCP), it is necessary, or at least desirable, to delimit one message from another to detect and possibly recover from user protocol errors. This is called record marking (RM). One RPC message fits into one RM record.

A record is composed of one or more record fragments. A record fragment is a four-byte header followed by `0` to `(2**31) - 1` bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is the network byte order.

The header encodes two values:

- A Boolean that specifies whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment).
- A 31-bit unsigned binary value that is the length in bytes of the fragment's data. The Boolean value is the highest-order bit of the header; the length is the 31 low-order bits. (This record specification is *not* in XDR standard form).

Authentication Protocols

Authentication parameters are opaque but open-ended to the rest of the RPC protocol. This section defines some flavors of authentication that have already been implemented. Other sites are free to invent new authentication types, with the same rules of flavor number assignment for program number assignment. SunSoft maintains and administers a range of authentication flavors. To have authentication numbers (like RPC program numbers) allocated (or registered to them), contact the Sun RPC Administrator, as described in "Program Number Registration" on page 361.

AUTH_NONE

Calls are often made where the caller does not authenticate itself and the server disregards who the caller is. In these cases, the `flavor` value (the "discriminant" of the `opaque_auth` "union") of the RPC message's credentials, verifier, and response verifier is `AUTH_NONE`. The body length is zero when `AUTH_NONE` authentication flavor is used.

AUTH_SYS

This is the same as the authentication flavor previously known as `AUTH_UNIX`. The caller of a remote procedure may wish to identify itself using traditional UNIX process permissions authentication. The `flavor` of the `opaque_auth` of such an RPC call message is `AUTH_SYS`. The bytes of the `body` encode the following structure:

```
struct auth_sysparms {
    unsigned int stamp;
    string machinename<255>;
    uid_t uid;
```

```
    gid_t gid;
    gid_t gids<10>;
};
```

- *stamp* is an arbitrary ID that the caller machine may generate.
- *machinename* is the name of the caller's machine.
- *uid* is the caller's effective user ID.
- *gid* is the caller's effective group ID.
- *gids* is a counted array of groups in which the caller is a member.

The `flavor` of the verifier accompanying the credentials should be AUTH_NONE.

The AUTH_SHORT Verifier

When using AUTH_SYS authentication, the `flavor` of the response verifier received in the reply message from the server may be AUTH_NONE or AUTH_SHORT.

If AUTH_SHORT, the bytes of the response verifier's string encode a `short_hand_verf` structure. This opaque structure may now be passed to the server instead of the original AUTH_SYS credentials.

The server keeps a cache that maps shorthand opaque structures (passed back by way of an AUTH_SHORT style response verifier) to the original credentials of the caller. The caller can save network bandwidth and server cpu cycles by using the new credentials.

The server may flush the shorthand opaque structure at any time. If this happens, the remote procedure call message will be rejected owing to an authentication error. The reason for the failure will be AUTH_REJECTEDCRED. At this point, the caller may wish to try the original AUTH_SYS style of credentials. See Figure C-1.

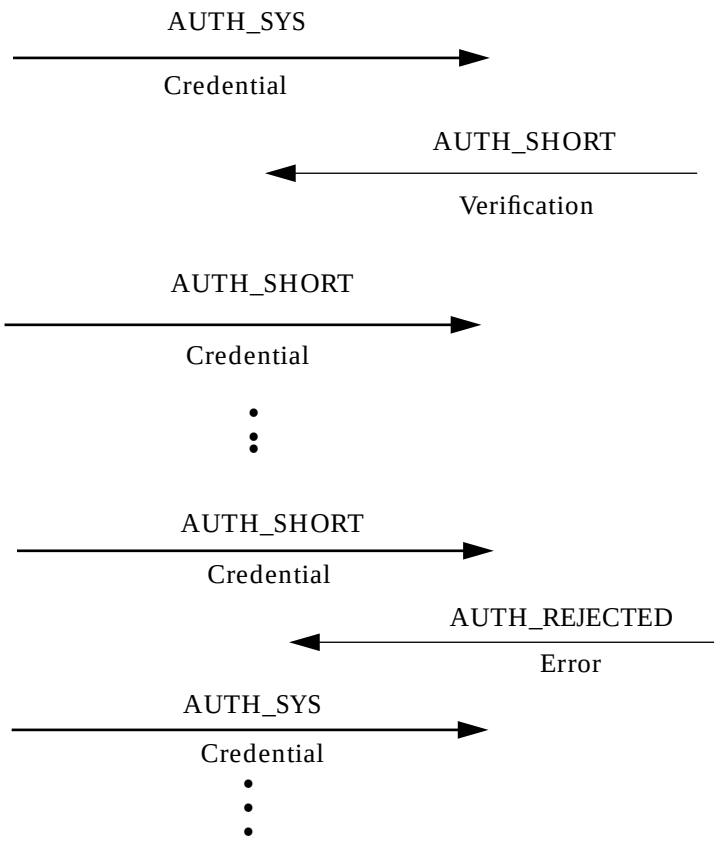


Figure C-1 Authentication Process Map

AUTH_DES Authentication

AUTH_SYS authentication has the following problems:

1. Caller identification cannot be guaranteed to be unique if machines with differing operating systems are on the same network.
2. There is no verifier, so credentials can easily be faked. AUTH_DES authentication attempts to fix these two problems.

The first problem is handled by addressing the caller by a simple string of characters instead of by an operating system specific integer. This string of characters is known as the netname or network name of the caller. The server should not interpret the caller's name in any way other than as the identity of the caller. Thus, netnames should be unique for every caller in the naming domain.

It is up to each operating system's implementation of AUTH_DES authentication to generate netnames for its users that ensure this uniqueness when they call remote servers. Operating systems already distinguish users local to their systems. It is usually a simple matter to extend this mechanism to the network. For example, a user with a user ID of 515 might be assigned the following netname: "UNIX.515@sun.com". This netname contains three items that serve to ensure it is unique. Going backward, there is only one naming domain called sun.com in the Internet. Within this domain, there is only one UNIX user with user ID 515. However, there may be another user on another operating system, for example VMS, within the same naming domain who, by coincidence, happens to have the same user ID. To ensure that these two users can be distinguished you add the operating system name. So one user is "UNIX.515@sun.com" and the other is "VMS.515@sun.com".

The first field is actually a naming method rather than an operating system name. It just happens that there is almost a one-to-one correspondence between naming methods and operating systems. If the world could agree on a naming standard, the first field could be a name from that standard, instead of an operating system name.

AUTH_DES Authentication Verifiers

Unlike AUTH_SYS authentication, AUTH_DES authentication does have a verifier so the server can validate the client's credential (and vice versa). The contents of this verifier are primarily an encrypted timestamp. The server can decrypt this timestamp, and if it is close to its current real time, then the client must have encrypted it correctly. The only way the client could encrypt it correctly is to know the conversation key of the RPC session. If the client knows the conversation key, it must be the real client.

The conversation key is a DES [5] key that the client generates and notifies the server of in its first RPC call. The conversation key is encrypted using a public key scheme in this first transaction. The particular public key scheme used in AUTH_DES authentication is Diffie-Hellman [3] with 192-bit keys. The details of this encryption method are described later.

The client and the server need the same notion of the current time for this to work. If network time synchronization cannot be guaranteed, then client can synchronize with the server before beginning the conversation. rpcbind provides a procedure, RPCBPROC_GETTIME, which may be used to obtain the current time.

A server can determine if a client timestamp is valid. For any transaction after the first, the server checks for two things:

- The timestamp is greater than the one previously seen from the same client.
- The timestamp has not expired. A timestamp is expired if the server's time is later than the sum of the client's timestamp plus what is known as the client's *window*. The window is a number the client passes (encrypted) to the server in its first transaction. The window can be thought of as a lifetime for the credential.

For the first transaction, the server checks that the timestamp has not expired. As an added check, the client sends an encrypted item in the first transaction known as the *window verifier* which must be equal to the window minus 1, or the server will reject the credential.

The client must check the verifier returned from the server to be sure it is legitimate. The server sends back to the client the encrypted timestamp it received from the client, minus one second. If the client gets anything other than this, it will reject it.

Nicknames and Clock Synchronization

After the first transaction, the server's AUTH_DES authentication subsystem returns in its verifier to the client an integer *nickname* that the client may use in its further transactions instead of passing its netname, encrypted DES key and window every time. The nickname is most likely an index into a table on the server that stores for each client its netname, decrypted DES key and window. It should however be treated an opaque data by the client.

Though originally synchronized, client and server clocks can get out of sync. If this happens, the client RPC subsystem most likely will receive an RPC_AUTHERROR at which point it should resynchronize.

A client may still get the RPC_AUTHERROR error even though it is synchronized with the server. The reason is that the server's nickname table is a limited size, and it may flush entries whenever it wants. The client should resend its original credential and the server will give it a new nickname. If a server crashes, the entire nickname table will be flushed, and all clients will have to resend their original credentials.

DES Authentication Protocol (in XDR language)

Code Example C-2 AUTH_DES Authentication Protocol

```
/*
 * There are two kinds of credentials: one in which the client uses
 * its full network name, and one in which it uses its "nickname"
 * (just an unsigned integer) given to it by the server. The client
 * must use its full name in its first transaction with the server,
 * in which the server will return to the client its nickname. The
 * client may use its nickname in all further transactions with the
 * server. There is no requirement to use the nickname, but it is
 * wise to use it for performance reasons.
 */
enum authdes_namekind {
    ADN_FULLNAME = 0,
    ADN_NICKNAME = 1
 };

/*
 * A 64-bit block of encrypted DES data
 */
 typedef opaque des_block[8];

/*
 * Maximum length of a network user's name
 */
const MAXNETNAMELEN = 255;

/*
 * A fullname contains the network name of the client, an encrypted
 * conversation key and the window. The window is actually a lifetime
 * for the credential. If the time indicated in the verifier
 * timestamp plus the window has passed, then the server should
```

```
 * expire the request and not grant it. To insure that requests are
 * not replayed, the server should insist that timestamps be greater
 * than the previous one seen, unless it is the first transaction.
 * In the first transaction, the server checks instead that the
 * window verifier is one less than the window.
 */
struct authdes_fullname {
    string name<MAXNETNAMELEN>; /* name of client */
    des_block key;              /* PK encrypted conversation key */
    unsigned int window;        /* encrypted window */
};                             /* NOTE: PK means "public key" */

/*
 * A credential is either a fullname or a nickname
 */
unionauthdes_credswitch(authdes_namekindadc_namekind){
    case ADN_FULLNAME:
        authdes_fullname adc_fullname;
    case ADN_NICKNAME:
        unsigned int adc_nickname;
};

/*
 * A timestamp encodes the time since midnight, January 1, 1970.
 */
struct timestamp {
    unsigned int seconds;      /* seconds */
    unsigned int useconds;     /* and microseconds */
};

/*
 * Verifier: client variety
 */
struct authdes_verf_clnt {
    timestamp adv_timestamp;   /* encrypted timestamp */
    unsigned int adv_winverf;  /* encrypted window verifier */
};

/*
 * Verifier: server variety
 * The server returns (encrypted) the same timestamp the client gave
 * it minus one second. It also tells the client its nickname to be
 * used in future transactions (unencrypted).
 */
struct authdes_verf_svr {
```

```
    timestamp adv_timeverf;    /* encrypted verifier */
    unsigned int adv_nickname; /* new nickname for clnt */
};
```

Diffie-Hellman Encryption

In this scheme, there are two constants, PROOT and HEXMODULUS. The particular values chosen for these for the DES authentication protocol are:

```
const PROOT = 3;
const HEXMODULUS =/* hex */
 "d4a0ba0250b6fd2ec626e7efd637df76c716e22d0944b88b";
```

The way this scheme works is best explained by an example. Suppose there are two people "A" and "B" who want to send encrypted messages to each other. A and B each generate a random secret key that they do not disclose to anyone. Let these keys be represented as SK(A) and SK(B). They also publish in a public directory their public keys. These keys are computed as follows:

```
PK(A) = (PROOT ** SK(A)) mod HEXMODULUS
PK(B) = (PROOT ** SK(B)) mod HEXMODULUS
```

The ** notation is used here to represent exponentiation.

Now, both A and B can arrive at the common key between them, represented here as CK(A,B), without disclosing their secret keys.

A computes:

```
CK(A, B) = (PK(B) ** SK(A)) mod HEXMODULUS
```

while B computes:

```
CK(A, B) = (PK(A) ** SK(B)) mod HEXMODULUS
```

These two can be shown to be equivalent: (PK(B)**SK(A)) mod HEXMODULUS = (PK(A)**SK(B)) mod HEXMODULUS Drop the mod HEXMODULUS parts and assume modulo arithmetic to simplify the process:

```
PK(B) ** SK(A) = PK(A) ** SK(B)
```

Then replace PK(B) by what B computed earlier and likewise for PK(A).

```
((PROOT ** SK(B)) ** SK(A) = (PROOT ** SK(A)) ** SK(B)
```

which leads to:

```
PROOT ** (SK(A) * SK(B)) = PROOT ** (SK(A) * SK(B))
```

This common key `CK(A,B)` is not used to encrypt the timestamps used in the protocol. It is used only to encrypt a conversation key that is then used to encrypt the timestamps. The reason for doing this is to use the common key as little as possible, for fear that it could be broken. Breaking the conversation key is a far less serious offense, because conversations are comparatively short-lived.

The conversation key is encrypted using 56-bit DES keys, yet the common key is 192 bits. To reduce the number of bits, 56 bits are selected from the common key as follows. The middle-most 8 bytes are selected from the common key, and then parity is added to the lower order bit of each byte, producing a 56-bit key with 8 bits of parity.

AUTH_KERB Authentication

To avoid compiling Kerberos code into the operating system kernel, the kernel used in the SunOS 5.x implementation of `AUTH_KERB` uses a proxy RPC daemon called `kerbd`. The daemon exports three procedures. Refer to the `kerbd(1M)` manpage for more details.

1. `KGETKCRED` is used by the server-side RPC to check the authenticator presented by the client.
2. `KSETKCRED` returns the encrypted ticket and DES session key, given a primary name, instance, and realm.
3. `KGETUCRED` is UNIX-specific. It returns the user's ID, the group ID, and groups list, assuming that the primary name is mapped to a user name known to the server.

The best way to describe how Kerberos works is to use an example based on a service currently implementing Kerberos: the network file system (NFS). The NFS service on server *s* is assumed to have the well-known principal name nfs.*s* A privileged user on client *c* is assumed to have the primary name `root` and an instance *c*. Note that (unlike `AUTH_DES`) when the user's ticket-granting ticket has expired, `kinit`() must be reinvoked. NFS service for Kerberos mounts will fail until a new ticket-granting ticket is obtained.

NFS Mount Example

This section follows an NFS mount request from start to finish using AUTH_KERB. Since mount requests are executed as root in SunOS, the user's identity is root.*c*.

Client *c* makes a MOUNTPROC_MOUNT request to the server *s* to obtain the file handle for the directory to be mounted. The client mount program makes an NFS mount system call, handing the client kernel the file handle, mount flavor, time synchronization address, and the server's well-known name, nfs.*s*. Next the client kernel contacts the server at the time synchronization host to obtain the client-server time bias.

The client kernel makes the following RPC calls: (1) KSETKCRED to the local kerbd to obtain the ticket and session key, (2) NFSPROC_GETATTR to the server's NFS service, using the full name credential and verifier. The server receives the calls and makes the KGETKCRED call to its local kerbd to check the client's ticket.

The server's kerbd and the Kerberos library decrypt the ticket and return, among other data, the principal name and DES session key. The server checks that the ticket is still valid, uses the session key to decrypt the DES-encrypted portions of the credential and verifier, and checks that the verifier is valid.

The possible Kerberos authentication errors returned at this time are:

- AUTH_BADCRED is returned if the verifier is invalid (the decrypted win in the credential and win +1 in the verifier do not match), or the timestamp is not within the window range
- AUTH_REJECTEDCRED is returned if a replay is detected
- AUTH_BADVERF is returned if the verifier is garbled

If no errors are received, the server caches the client's identity and allocates a nickname (small integer) to be returned in the NFS reply. The server then checks if the client is in the same realm as the server. If it is, the server calls KGETUCRED to its local kerbd to translate the principal's primary name into UNIX credentials. If it is not translatable, the user is marked anonymous. The server checks these credentials against the file system's export information. There are three cases to consider:

1. If the KGETUCRED call fails and anonymous requests are allowed, the UNIX credentials of the anonymous user are assigned.

2. If the KGETUCRED call fails and anonymous requests are not allowed, the NFS call fails with the AUTH_TOOWEAK.

3. If the KGETUCRED call succeeds, the credentials are assigned, and normal protection checking follows, including checking for root permission.

Next the server sends an NFS reply, including the nickname and server's verifier. The client receives the reply, decrypts and validates the verifier, and stores the nickname for future calls. The client makes a second NFS call to the server, and the calls to the server described earlier are repeated. The client kernel makes an NFSPROC_STATVFS call to the server's NFS service, using the nickname credential and verifier described previously. The server receives the call and validates the nickname. If it is out of range, the error AUTH_BADCRED is returned. The server uses the session key just obtained to decrypt the DES-encrypted portions of the verifier and validates the verifier.

The possible Kerberos authentication errors returned at this time are:
- AUTH_REJECTEDVERF is returned if the timestamp is invalid, a replay is detected, or if the timestamp is not within the window range
- AUTH_TIMEEXPIRE is returned if the service ticket is expired

If no errors are received, the server uses the nickname to retrieve the caller's UNIX credentials. Then it checks these credentials against the file system's export information, and sends an NFS reply that includes the nickname and the server's verifier. The client receives the reply, decrypts and validates the verifier, and stores the nickname for future calls. Last, the client's NFS mount system call returns, and the request is finished.

KERB Authentication Protocol (in XDR Language)

Code Example C-3 (AUTH_KERB) has many similarities to the one for AUTH_DES, shown in Code Example C-2 on page 369. Note the differences.

Code Example C-3 AUTH_KERB Authentication Protocol

```
#define AUTH_KERB 4

/*
 * There are two kinds of credentials: one in which the client sends
 * the (previously encrypted) Kerberos ticket, and one in which it
 * uses its "nickname" (just an unsigned integer) given to it by the
 * server. The client must use its full name in its first transaction
 * with the server, in which the server will return to the client
 * its nickname. The client may use its nickname in all further
```

```
 * transactions with the server (until the ticket expires). There is
 * no requirement to use the nickname, but it is wise to use it for
 * performance reasons.
 */
enum authkerb_namekind {
    AKN_FULLNAME = 0,
    AKN_NICKNAME = 1
 };

/*
 * A fullname contains the encrypted service ticket and the
 * window. The window is actually a lifetime
 * for the credential. If the time indicated in the verifier
 * timestamp plus the window has passed, then the server should
 * expire the request and not grant it. To insure that requests are
 * not replayed, the server should insist that timestamps be greater
 * than the previous one seen, unless it is the first transaction.
 * In the first transaction, the server checks instead that the
 * window verifier is one less than the window.
 */
struct authkerb_fullname {
    KTEXT_ST ticket;              /* Kerberos service ticket */
    unsigned long window;        /* encrypted window */
};
/*
 * A credential is either a fullname or a nickname
 */
union authkerb_credswitch(authkerb_namekind akc_namekind){
    case AKN_FULLNAME:
        authkerb_fullname akc_fullname;
    case AKN_NICKNAME:
        unsigned long akc_nickname;
};

/*
 * A timestamp encodes the time since midnight, January 1, 1970.
 */
struct timestamp {
    unsigned long seconds;      /* seconds */
    unsigned long useconds;     /* and microseconds */
};

/*
 * Verifier: client variety
 */
struct authkerb_verf_clnt {
```

```
    timestamp akv_timestamp;   /* encrypted timestamp */
    unsigned long akv_winverf;  /* encrypted window verifier */
};

/*
 * Verifier: server variety
 * The server returns (encrypted) the same timestamp the client gave
 * it minus one second. It also tells the client its nickname to be
 * used in future transactions (unencrypted).
 */
struct authkerb_verf_svr {
    timestamp akv_timeverf;    /* encrypted verifier */
    unsigned long akv_nickname; /* new nickname for clnt */
};
```

The RPC Language Specification

Just as there was a need to describe the XDR data types in a formal language, there is also need to describe the procedures that operate on these XDR data types in a formal language as well. The RPC Language, an extension to the XDR language, serves this purpose. The following example is used to describe the essence of the language.

An Example Service Described in the RPC Language

Code Example C-4 shows the specification of a simple `ping` program.

Code Example C-4 `ping` Service Using RPC Language

```
/*
 * Simple ping program
 */
program PING_PROG {
    version PING_VERS_PINGBACK {
        void
        PINGPROC_NULL(void) = 0;
        /*
         * ping the caller, return the round-trip time
         * in milliseconds. Return a minus one (-1) if
         * operation times-out
         */
        int
        PINGPROC_PINGBACK(void) = 1;
        /* void - above is an argument to the call */
```

```
    } = 2;
/*
 * Original version
 */
    version PING_VERS_ORIG {
        void
        PINGPROC_NULL(void) = 0;
    } = 1;
} = 200000;
const PING_VERS = 2; /* latest version */
```

The first version described is `PING_VERS_PINGBACK` with two procedures, `PINGPROC_NULL` and `PINGPROC_PINGBACK`.

`PINGPROC_NULL` takes no arguments and returns no results, but it is useful for such things as computing round-trip times from the client to the server and back again. By convention, procedure `0` of any RPC program should have the same semantics, and never require authentication.

The second procedure returns the amount of time (in microseconds) that the operation used.

The next version, `PING_VERS_ORIG`, is the original version of the protocol and it does not contain `PINGPROC_PINGBACK` procedure. It is useful for compatibility with old client programs, and as this program matures it may be dropped from the protocol entirely.

RPCL Syntax

The RPC language (RPCL) is similar to C. This section describes the syntax of the RPC language, showing a few examples along the way. It also shows how RPC and XDR type definitions get compiled into C type definitions in the output header file.

An RPC language file consists of a series of definitions.

```
definition-list:
    definition;
    definition; definition-list
```

It recognizes six types of definitions.

`definition:`

```
enum-definition
const-definition
typedef-definition
struct-definition
union-definition
program-definition
```

Definitions are not the same as declarations. No space is allocated by a definition – only the type definition of a single or series of data elements. This means that variables still must be declared.

The RPC language is identical to the XDR language, except for the added definitions described in Table C-2.

Table C-2 RPC Language Definitions

Term	Definition
program-definition	`program program-ident {version-list} = value`
version-list	`version`; version; version-list
version	version version-ident {procedure-list} = value
procedure-list	procedure; procedure; procedure-list
procedure	type-ident procedure-ident (type-ident) = value

- The following keywords are added and cannot be used as identifiers: program version.
- Neither version name nor a version number can occur more than once within the scope of a program definition.
- Neither a procedure name nor a procedure number can occur more than once within the scope of a version definition.
- Program identifiers are in the same name space as constant and type identifiers.
- Only unsigned constants can be assigned to programs, versions, and procedures.

Enumerations

RPC / XDR enumerations have the same syntax as C enumerations.

```
enum-definition:
    "enum" enum-ident "{"
        enum-value-list
    "}"
enum-value-list:
    enum-value
    enum-value "," enum-value-list
enum-value:
    enum-value-ident
    enum-value-ident "=" value
```

Here is an example of an XDR `enum` and the C `enum` to which it gets compiled.

```
enum colortype {                enum colortype {
    RED = 0,                        RED = 0,
    GREEN = 1,        -->           GREEN = 1,
    BLUE = 2                        BLUE = 2,
};                              };
                                typedef enum colortype colortype;
```

Constants

XDR symbolic constants may be used wherever an integer constant is used. For example, in array size specifications:

```
    const-definition:
        const const-ident = integer
```

The following example defines a constant, `DOZEN` as equal to `12`:

```
const DOZEN = 12; --> #define DOZEN 12
```

Type Definitions

XDR `typedef`s have the same syntax as C `typedef`s.

```
typedef-definition:
        typedef declaration
```

This example defines an `fname_type` used for declaring file name strings that have a maximum length of 255 characters.

```
typedef string fname_type<255>; --> typedef char *fname_type;
```

Declarations

In XDR, there are four kinds of declarations. These declarations must be a part of a `struct` or a `typedef`; they cannot stand alone:

```
declaration:
        simple-declaration
        fixed-array-declaration
        variable-array-declaration
        pointer-declaration
```

Simple Declarations

Simple declarations are just like simple C declarations:

```
simple-declaration:
        type-ident variable-ident
```

Example:

```
    colortype color; --> colortype color;
```

Fixed-Length Array Declarations

Fixed-length array declarations are just like C array declarations:

```
fixed-array-declaration:
        type-ident variable-ident [value]
```

Example:

```
colortype palette[8]; --> colortype palette[8];
```

Many programmers confuse variable declarations with type declarations. It is important to note that `rpcgen` does not support variable declarations. This example is a program that will not compile:

```
int data[10];
program P {
    version V {
        int PROC(data) = 1;
    } = 1;
} = 0x200000;
```

The example above will not compile because of the variable declaration:

```
int data[10]
```

Instead, use:

```
typedef int data[10];
```

or

```
struct data {int dummy [10]};
```

Variable-Length Array Declarations

Variable-length array declarations have no explicit syntax in C. The XDR language does have a syntax, using angle brackets:

```
variable-array-declaration:
        type-ident variable-ident <value>
        type-ident variable-ident < >
```

The maximum size is specified between the angle brackets. The size may be omitted, indicating that the array may be of any size:

```
int heights<12>; /* at most 12 items */
int widths<>; /* any number of items */
```

Because variable-length arrays have no explicit syntax in C, these declarations are compiled into `struct` declarations. For example, the `heights` declaration compiled into the following `struct`:

```
struct {
    u_int heights_len;              /* # of items in array */
    int *heights_val;               /* pointer to array */
} heights;
```

The number of items in the array is stored in the `_len` component and the pointer to the array is stored in the `_val` component. The first part of each component name is the same as the name of the declared XDR variable (`heights`).

Pointer Declarations

Pointer declarations are made in XDR exactly as they are in C. Address pointers are not really sent over the network; instead, XDR pointers are useful for sending recursive data types such as lists and trees. The type is called “optional-data,” not “pointer,” in XDR language:

```
pointer-declaration:
    type-ident *variable-ident
```

Example:

```
listitem *next; --> listitem *next;
```

Structures

An RPC/XDR `struct` is declared almost exactly like its C counterpart. It looks like the following:

```
struct-definition:
    struct struct-ident "{"
        declaration-list
    "}"
declaration-list:
    declaration ";"
    declaration ";" declaration-list
```

The following XDR structure is an example of a two-dimensional coordinate and the C structure that it compiles into:

```
struct coord {                  struct coord {
    int x;            -->           int x;
    int y;                          int y;
};                                  };
                                    typedef struct coord coord;
```

The output is identical to the input, except for the added `typedef` at the end of the output. This enables one to use `coord` instead of `struct coord` when declaring items.

Unions

XDR unions are discriminated unions, and do not look like C unions – they are more similar to Pascal variant records:

```
union-definition:
    "union" union-ident "switch" "("simple declaration")" "{"
        case-list
    "}"
case-list:
    "case" value ":" declaration ";"
    "case" value ":" declaration ";" case-list
    "default" ":" declaration ";"
```

The following is an example of a type returned as the result of a “read data” operation: If there is no error, return a block of data; otherwise, don’t return anything.

```
union read_result switch (int errno) {
    case 0:
        opaque data[1024];
    default:
        void;
    };
```

It compiles into the following:

```
struct read_result {
    int errno;
    union {
        char data[1024];
    } read_result_u;
};
typedef struct read_result read_result;
```

Notice that the union component of the output struct has the same name as the type name, except for the trailing _u.

Programs

RPC programs are declared using the following syntax:

```
program-definition:
    "program" program-ident "{"
        version-list
    "}" "=" value;

version-list:
    version ";"
    version ";" version-list

version:
    "version" version-ident "{"
        procedure-list
    "}" "=" value;

procedure-list:
    procedure ";"
    procedure ";" procedure-list

procedure:
    type-ident procedure-ident "(" type-ident ")" "=" value;
```

When the -N option is specified, rpcgen also recognizes the following syntax:

```
procedure:
    type-ident procedure-ident "(" type-ident-list ")" "=" value;
type-ident-list:
    type-ident
    type-ident "," type-ident-list
```

For example:

```
/*
 * time.x: Get or set the time. Time is represented as seconds
 * since 0:00, January 1, 1970.
 */
program TIMEPROG {
    version TIMEVERS {
        unsigned int TIMEGET(void) = 1;
        void TIMESET(unsigned) = 2;
    } = 1;
} = 0x20000044;
```

Note that the void argument type means that no argument is passed.

This file compiles into these #define statements in the output header file:

```
#define TIMEPROG 0x20000044
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

Special Cases

There are several exceptions to the RPC language rules.

C-style Mode

In the new features section we talked about the features of the C-style mode of rpcgen. These features have implications with regard to the passing of void arguments. No arguments need be passed if their value is void.

Booleans

C has no built-in boolean type. However, the RPC library uses a boolean type called `bool_t` that is either `TRUE` or `FALSE`. Parameters declared as type `bool` in XDR language are compiled into `bool_t` in the output header file.

Example:

```
bool married; --> bool_t married;
```

Strings

The C language has no built-in string type, but instead uses the null-terminated `char *` convention. In C, strings are usually treated as null-terminated single-dimensional arrays.

In XDR language, strings are declared using the `string` keyword, and compiled into type `char *` in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the `NULL` character). The maximum size may be omitted, indicating a string of arbitrary length.

Examples:

```
string name<32>;   --> char *name;
string longname<>; --> char *longname;
```

Note – `NULL` strings cannot be passed; however, a zero-length string (that is, just the terminator or `NULL` byte) can be passed.

Opaque Data

Opaque data is used in XDR to describe untyped data, that is, sequences of arbitrary bytes. It may be declared either as a fixed length or variable length array. Examples:

```
opaque diskblock[512]; --> char diskblock[512];

opaque filedata<1024>; --> struct {
                               u_int filedata_len;
                               char *filedata_val;
                           } filedata;
```

Voids

In a void declaration, the variable is not named. The declaration is just `void` and nothing else. Void declarations can only occur in two places: union definitions and program definitions (as the argument or result of a remote procedure, for example no arguments are passed.)

`rpcbind` *Protocol*

`rpcbind` maps RPC program and version numbers to universal addresses, thus making dynamic binding of remote programs possible.

`rpcbind` is bound to a well-known address of each supported transport, and other programs register their dynamically allocated transport addresses with it. `rpcbind` then makes those addresses publicly available. Universal addresses are string representations of the transport-dependent address. They are defined by the addressing authority of the given transport.

`rpcbind` also aids in broadcast RPC. RPC programs will have different addresses on different machines, so there is no way to broadcast directly to all these programs. `rpcbind`, however, has a well-known address. So, to broadcast to a given program, the client actually sends its message to the `rpcbind` process on the machine it chooses to reach. `rpcbind` picks up the broadcast and calls the local service specified by the client. When `rpcbind` gets a reply from the local service, it passes the reply on to the client.

Code Example C-5 `rpcbind` Protocol Specification (in RPC Language)

```
/*
 * rpcb_prot.x
 * RPCBIND protocol in rpc language
 */

/*
 * A mapping of (program, version, network ID) to universal address
 */
struct rpcb {
    unsigned long r_prog;           /* program number */
    unsigned long r_vers;           /* version number */
    string r_netid<>;               /* network id */
    string r_addr<>;                /* universal address */
    string r_owner<>;               /* owner of this service */ };
```

```
/* A list of mappings */
struct rpcblist {
    rpcb rpcb_map;
    struct rpcblist *rpcb_next;
};

/* Arguments of remote calls */
struct rpcb_rmtcallargs {
    unsigned long prog;             /* program number */
    unsigned long vers;             /* version number */
    unsigned long proc;             /* procedure number */
    opaque args<>;                  /* argument */
};

/* Results of the remote call */
struct rpcb_rmtcallres {
    string addr<>;                  /* remote universal address */
    opaque results<>;               /* result */
};

/*
 * rpcb_entry contains a merged address of a service on a particular
 * transport, plus associated netconfig information. A list of
 * rpcb_entrys is returned by RPCBPROC_GETADDRLIST. See netconfig.h
 * for values used in r_nc_* fields.
 */
struct rpcb_entry {
    string          r_maddr<>;      /* merged address of service */
    string          r_nc_netid<>;   /* netid field */
    unsigned long   r_nc_semantics; /* semantics of transport */
    string          r_nc_protofmly<>; /* protocol family */
    string          r_nc_proto<>;   /* protocol name */
};

/* A list of addresses supported by a service. */
struct rpcb_entry_list {
    rpcb_entry rpcb_entry_map;
    struct rpcb_entry_list *rpcb_entry_next;
};

typedef rpcb_entry_list *rpcb_entry_list_ptr;

/* rpcbind statistics */
const rpcb_highproc_2 = RPCBPROC_CALLIT;
const rpcb_highproc_3 = RPCBPROC_TADDR2UADDR;
const rpcb_highproc_4 = RPCBPROC_GETSTAT;
```

```
const RPCBSTAT_HIGHPROC = 13;  /* # of procs in rpcbind V4 plus one */
const RPCBVERS_STAT = 3;  /* provide only for rpcbind V2, V3 and V4 */
const RPCBVERS_4_STAT = 2;
const RPCBVERS_3_STAT = 1;
const RPCBVERS_2_STAT = 0;

/* Link list of all the stats about getport and getaddr */
struct rpcbs_addrlist {
    unsigned long prog;
    unsigned long vers;
    int success;
    int failure;
    string netid<>;
    struct rpcbs_addrlist *next;
};

/* Link list of all the stats about rmtcall */
struct rpcbs_rmtcalllist {
    unsigned long prog;
    unsigned long vers;
    unsigned long proc;
    int success;
    int failure;
    int indirect;   /* whether callit or indirect */
    string netid<>;
    struct rpcbs_rmtcalllist *next;
};

typedef int rpcbs_proc[RPCBSTAT_HIGHPROC];
typedef rpcbs_addrlist *rpcbs_addrlist_ptr;
typedef rpcbs_rmtcalllist *rpcbs_rmtcalllist_ptr;

struct rpcb_stat {
    rpcbs_proc              info;
    int                     setinfo;
    int                     unsetinfo;
    rpcbs_addrlist_ptr      addrinfo;
    rpcbs_rmtcalllist_ptr   rmtinfo;
};

/*
 * One rpcb_stat structure is returned for each version of rpcbind
 * being monitored.
 */
typedef rpcb_stat rpcb_stat_byvers[RPCBVERS_STAT];
```

```
/* rpcbind procedures */
program RPCBPROG {
    version RPCBVERS {
        void
        RPCBPROC_NULL(void) = 0;
        /*
         * Registers the tuple [r_prog, r_vers, r_addr, r_owner,
         * r_netid]. The rpcbind server accepts requests for this
         * procedure on only the loopback transport for security
         * reasons. Returns TRUE if successful, FALSE on failure.
         */
        bool
        RPCBPROC_SET(rpcb) = 1;

        /*
         * Unregisters the tuple [r_prog, r_vers, r_owner, r_netid].
         * If vers is zero, all versions are unregistered. The
rpcbind
         * server accepts requests for this procedure on only the
         * loopback transport for security reasons.  Returns TRUE if
         * successful, FALSE on failure.
         */
        bool
        RPCBPROC_UNSET(rpcb) = 2;

        /*
         * Returns the universal address where the triple [r_prog,
         * r_vers, r_netid] is registered.  If r_addr specified,
         * return a universal address merged on r_addr. Ignores
         * r_owner. Returns FALSE on failure.
         */
        string
        RPCBPROC_GETADDR(rpcb) = 3;

        /* Returns a list of all mappings. */
        rpcblist
        RPCBPROC_DUMP(void) = 4;

        /*
         * Calls the procedure on the remote machine.  If it is not
         * registered, this procedure IS quiet; that is, it DOES NOT
         * return error information.
         */
        rpcb_rmtcallres
        RPCBPROC_CALLIT(rpcb_rmtcallargs) = 5;
```

```
/*
 * Returns the time on the rpcbind server's system.
 */
unsigned int
RPCBPROC_GETTIME(void) = 6;

struct netbuf
RPCBPROC_UADDR2TADDR(string) = 7;

string
RPCBPROC_TADDR2UADDR(struct netbuf) = 8;
} = 3;
version RPCBVERS4 {
bool
RPCBPROC_SET(rpcb) = 1;

bool
RPCBPROC_UNSET(rpcb) = 2;

string
RPCBPROC_GETADDR(rpcb) = 3;

rpcblist_ptr
RPCBPROC_DUMP(void) = 4;

/*
 * NOTE: RPCBPROC_BCAST has the same functionality as CALLIT;
 * the new name is intended to indicate that this procedure
 * should be used for broadcast RPC, and RPCBPROC_INDIRECT
 * should be used for indirect calls.
 */
rpcb_rmtcallres
RPCBPROC_BCAST(rpcb_rmtcallargs) = RPCBPROC_CALLIT;

unsigned int
RPCBPROC_GETTIME(void) = 6;

struct netbuf
RPCBPROC_UADDR2TADDR(string) = 7;

string
RPCBPROC_TADDR2UADDR(struct netbuf) = 8;

/*
 * Same as RPCBPROC_GETADDR except that if the given version
 * number is not available, the address is not returned.
```

```
            */
           string
           RPCBPROC_GETVERSADDR(rpcb) = 9;

           /*
            * Calls the procedure on the remote machine.  If it is not
            * registered, this procedure IS NOT quiet; that is, it DOES
            * return error information.
            */
           rpcb_rmtcallres
           RPCBPROC_INDIRECT(rpcb_rmtcallargs) = 10;

           /*
            * Same as RPCBPROC_GETADDR except that it returns a list of
            * addresses registered for the combination (prog, vers).
            */
           rpcb_entry_list_ptr
           RPCBPROC_GETADDRLIST(rpcb) = 11;

           /*
            * Returns statistics about the rpcbind server's activity.
            */
           rpcb_stat_byvers
           RPCBPROC_GETSTAT(void) = 12;
       } = 4;
} = 100000;
```

rpcbind Operation

rpcbind is contacted by way of an assigned address specific to the transport being used. For TCP/IP and UDP/IP, for example, it is port number 111. Each transport has such an assigned well known address. The following is a description of each of the procedures supported by rpcbind.

RPCBPROC_NULL

This procedure does no work. By convention, procedure zero of any program takes no parameters and returns no results.

RPCBPROC_SET

When a program first becomes available on a machine, it registers itself with the `rpcbind` program running on the same machine. The program passes its program number *prog*; version number *vers*; network identifier *netid*; and the universal address *uaddr*; on which it awaits service requests.

The procedure returns a Boolean response with the value `TRUE` if the procedure successfully established the mapping and `FALSE` otherwise. The procedure refuses to establish a mapping if one already exists for the ordered set *(prog, vers, netid).*

Note that neither *netid* nor *uaddr* can be `NULL`, and that *netid* should be a valid network identifier on the machine making the call.

RPCBPROC_UNSET

When a program becomes unavailable, it should unregister itself with the `rpcbind` program on the same machine.

The parameters and results have meanings identical to those of `RPCBPROC_SET`. The mapping of the *(prog, vers, netid)* tuple with *uaddr* is deleted.

If *netid* is `NULL`, all mappings specified by the ordered set *(prog, vers, *)* and the corresponding universal addresses are deleted. Only the owner of the service or the super-user is allowed to unset a service.

RPCBPROC_GETADDR

Given a program number *prog*, version number *vers*, and network identifier *netid*, this procedure returns the universal address on which the program is awaiting call requests.

The *netid* field of the argument is ignored and the *netid* is inferred from the *netid* of the transport on which the request came in.

RPCBPROC_DUMP

This procedure lists all entries in `rpcbind`'s database.

The procedure takes no parameters and returns a list of program, version, netid, and universal addresses. Call this procedure using a stream rather than a datagram transport to avoid the return of a large amount of data.

RPCBPROC_CALLIT

This procedure allows a caller to call another remote procedure on the same machine without knowing the remote procedure's universal address. It is intended for supporting broadcasts to arbitrary remote programs via rpcbind's universal address.

The parameters *prog*, *vers*, *proc*, and the *args_ptr* are the program number, version number, procedure number, and parameters of the remote procedure.

Note – This procedure sends a response only if the procedure was successfully executed, and is silent (no response) otherwise.

The procedure returns the remote program's universal address, and the results of the remote procedure.

RPCBPROC_GETTIME

This procedure returns the local time on its own machine in seconds since the midnight on January 1, 1970.

RPCBPROC_UADDR2TADDR

This procedure converts universal addresses to transport (netbuf) addresses. RPCBPROC_UADDR2TADDR is equivalent to uaddr2taddr(). See the netdir(3N) manpage. Only processes that cannot link to the name-to-address library modules should use RPCBPROC_UADDR2TADDR.

RPCBPROC_TADDR2UADDR

This procedure converts transport (netbuf) addresses to universal addresses. RPCBPROC_TADDR2UADDR is equivalent to taddr2uaddr(). See the netdir(3N) manpage. Only processes that can not link to the name-to-address library modules should use RPCBPROC_TADDR2UADDR.

Version 4 rpcbind

Version 4 of the rpcbind protocol includes all of the above procedures, and adds several others.

RPCBPROC_BCAST

This procedure is identical to the version 3 RPCBPROC_CALLIT procedure. The new name indicates that the procedure should be used for broadcast RPCs only. RPCBPROC_INDIRECT, defined in the following text, should be used for indirect RPC calls.

RPCBPROC_GETVERSADDR

This procedure is similar to RPCBPROC_GETADDR. The difference is the r_vers field of the rpcb structure can be used to specify the version of interest. If that version is not registered, no address is returned.

RPCBPROC_INDIRECT

This procedure is similar to RPCBPROC_CALLIT. Instead of being silent about errors (such as the program not being registered on the system), this procedure returns an indication of the error. This procedure should not be used for broadcast RPC. It is intended to be used with indirect RPC calls only.

RPCBPROC_GETADDRLIST

This procedure returns a list of addresses for the given rpcb entry. The client may be able use the results to determine alternate transports that it can use to communicate with the server.

RPCBPROC_GETSTAT

This procedure returns statistics on the activity of the rpcbind server. The information lists the number and kind of requests the server has received.

Note – All procedures except RPCBPROC_SET and RPCBPROC_UNSET can be called by clients running on a machine other than a machine on which rpcbind is running. rpcbind accepts only RPCPROC_SET and RPCPROC_UNSET requests on the loopback transport.

Bibliography

For further information on the technologies and architectures discussed in this appendix, reference the following resources.

1. Birrel, Andrew D. & Nelson, Bruce Jay; "Implementing Remote Procedure Calls," XEROX CSL-83-7, October 1983.

2. Cheriton, D.; "VMTP: Versatile Message Transaction Protocol," Preliminary Version 0.3; Stanford University, January 1987.

3. Diffie and Hellman; "New Directions in Cryptography," *IEEE Transactions on Information Theory* IT-22, November 1976.

4. Harrenstien, K.; "Time Server," *RFC* 738; Information Sciences Institute, October 1977.

5. National Bureau of Standards; "Data Encryption Standard," *Federal Information Processing Standards Publication* 46, January 1977.

6. Postel, J.; "Transmission Control Protocol - DARPA Internet Program Protocol Specification," *RFC* 793; Information Sciences Institute, September 1981.

7. Postel, J.; "User Datagram Protocol," *RFC* 768; Information Sciences Institute, August 1980.

C

Writing a Port Monitor With the Service Access Facility (SAF)

D

This appendix gives a brief description of the functions a port monitor must perform to run under the service access facility (SAF) and service access controller (SAC).

What is the SAF

The SAF generalizes the procedures for service access so that login access on the local system and network access to local services are managed in similar ways. Under the SAF, systems may access services using a variety of port monitors, including `ttymon`, the listener, and port monitors written expressly for a user's application.

D

The manner in which a port monitor observes and manages access ports is specific to the port monitor and not to any component of the SAF. Users may therefore extend their systems by developing and installing their own port monitors. One of the important features of the SAF is that it can be extended in this way by users.

Relative to the SAF, a service is a process that is started. There are no restrictions on the functions a service may provide.

The SAF consists of a controlling process, the service access controller (SAC), and two administrative levels corresponding to two levels in the supporting directory structure. The top administrative level is concerned with port monitor administration, the lower level with service administration.

From an administrative point of view, the SAF consists of the following components:

- The SAC
- A per-system configuration script
- The SAC administrative file
- The SAC administrative command `sacadm`
- Port monitors
- Optional per-port monitor configuration scripts
- An administrative file (for each port monitor)
- The administrative command `pmadm`
- Optional per-service configuration scripts

What is the SAC

The SAC is the SAF's controlling process. The SAC is started by `init`() by means of an entry in `/etc/inittab`. Its function is to maintain the port monitors on the system in the state specified by the system administrator.

The administrative command `sacadm` is used to tell the SAC to change the state of a port monitor. `sacadm` can also be used to add or remove a port monitor from SAC supervision and to list information about port monitors known to the SAC.

The SAC's administrative file contains a unique tag for each port monitor known to the SAC and the path name of the command used to start each port monitor.

The SAC performs three main functions:

- Customizes its own environment
- Starts the appropriate port monitors
- Polls its port monitors and initiates recovery procedures when necessary

Basic Port Monitor Functions

A port monitor is a process that is responsible for monitoring a set of homogeneous, incoming ports on a machine. A port monitor's major purpose is to detect incoming service requests and to dispatch them appropriately.

A port is an externally seen access point on a system. A port may be an address on a network (TSAP or PSAP), a hardwired terminal line, an incoming phone line, etc. The definition of what constitutes a port is strictly a function of the port monitor itself.

A port monitor performs certain basic functions. Some of these are required to conform to the SAF; others may be specified by the requirements and design of the port monitor itself.

Port monitors have two main functions:

- Managing ports
- Monitoring ports for indications of activity

Port Management

The first function of a port monitor is to manage a port. The actual details of how a port is managed are defined by the person who defines the port monitor. A port monitor is not restricted to handling a single port; it may handle multiple ports simultaneously.

Note – Some examples of port management are setting the line speed on incoming phone connections, binding an appropriate network address, reinitializing the port when the service terminates, outputting a prompt, etc.

Activity Monitoring

The second function of a port monitor is to monitor the port or ports for which it is responsible for indications of activity. Two types of activity may be detected.

1. The first is an indication to the port monitor to take some port monitor-specific action. Pressing the break key to indicate that the line speed should be cycled is an example of a port monitor activity. Not all port monitors need to recognize and respond to the same indications. The indication used to attract the attention of the port monitor is defined by the person who defines the port monitor.
2. The second is an incoming service request. When a service request is received, a port monitor must be able to determine which service is being requested from the port on which the request is received. Note that the same service may be available on more than one port.

Other Port Monitor Functions

This section briefly describes other port monitor functions.

Restricting Access to the System

A port monitor must be able to restrict access to the system without disturbing services that are still running. In order to do this, a port monitor must maintain two internal states: enabled and disabled. The port monitor starts in the state indicated by the `ISTATE` environment variable provided by the `sac`. See “The SAC/Port Monitor Interface” on page 404.

Enabling or disabling a port monitor affects all ports for which the port monitor is responsible. If a port monitor is responsible for a single port, only that port will be affected. If a port monitor is responsible for multiple ports, the entire collection of ports will be affected.

Enabling or disabling a port monitor is a dynamic operation: It causes the port monitor to change its internal state. The effect does not persist across new invocations of the port monitor.

Enabling or disabling an individual port, however, is a static operation: It causes a change to an administrative file. The effect of this change will persist across new invocations of the port monitor.

Creating `utmp` *Entries*

Port monitors are responsible for creating `utmp` entries with the type field set to `USER_PROCESS` for services they start, if this action has been specified (that is, if `-fu` was specified in the `pmadm` line that added the service). These `utmp` entries may in turn be modified by the service. When the service terminates, the `utmp` entry must be set to `DEAD_PROCESS`.

Port Monitor Process IDs and Lock Files

When a port monitor starts, it writes its process id into a file named `_pid` in the current directory and places an advisory lock on the file.

Changing the Service Environment: Running `doconfig()`

Before invoking the service designated in the port monitor administrative file, `_pmtab`, a port monitor must arrange for the per-service configuration script to be run (if one exists by calling the library function `doconfig()`. Because the per-service configuration script may specify the execution of restricted commands, as well as for other security reasons, port monitors are invoked with root permissions. The details of how services are invoked are specified by the person who defines the port monitor.

Terminating a Port Monitor

A port monitor must terminate itself gracefully on receipt of the signal `SIGTERM`. The termination sequence is the following:

1. The port monitor enters the *stopping* state; no further service requests are accepted.
2. Any attempt to re-enable the port monitor will be ignored.
3. The port monitor yields control of all ports for which it is responsible. It must be possible for a new instantiation of the port monitor to start correctly while a previous instantiation is stopping.
4. The advisory lock on the process id file is released. Once this lock is released, the contents of the process id file are undefined and a new invocation of the port monitor may be started.

D

SAF Files

This section briefly covers the files used by the SAF.

The Port Monitor Administrative File

A port monitor's current directory contains an administrative file named `_pmtab`. `_pmtab` is maintained by the `pmadm` command in conjunction with a port monitor-specific administrative command.

Note – The port monitor administrative command for a `listen` port monitor is `nlsadmin`(); the port monitor administrative command for `ttymon` is `ttyadm`. Any port monitor written by a user must be provided with an administrative command specific to that port monitor to perform similar functions.

Per-Service Configuration Files

A port monitor's current directory also contains the per-service configuration scripts, if they exist. The names of the per-service configuration scripts correspond to the service tags in the `_pmtab` file.

Private Port Monitor Files

A port monitor may create private files in the directory `/var/saf/tag`, where *tag* is the name of the port monitor. Examples of private files are log files or temporary files.

The SAC/Port Monitor Interface

The `sac` creates two environment variables for each port monitor it starts:

1. `PMTAG`
2. `ISTATE PMTAG`
   This variable is set to a unique port monitor tag by the `sac`. The port monitor uses this tag to identify itself in response to `sac` messages. `ISTATE` is used to indicate to the port monitor what its initial internal state should

be. ISTATE is set to “enabled” or “disabled” to indicate that the port monitor is to start in the enabled or disabled state respectively. The sac performs a periodic sanity poll of the port monitors.

The sac communicates with port monitors through FIFOs. A port monitor should open _pmpipe, in the current directory, to receive messages from the sac and ../_sacpipe to send return messages to the sac.

Message Formats

This section describes the messages that may be sent from the sac to a port monitor (sac messages), and from a port monitor to the sac (port monitor messages). These messages are sent through FIFOs and are in the form of C structures. See Code Example D-2 on page 421.

sac Messages

The format of messages from the sac is defined by the structure sacmsg:

```
struct sacmsg
    int sc_size; /* size of optional data portion */
    char sc_type; /* type of message */
};
```

The sac may send four types of messages to port monitors. The type of message is indicated by setting the sc_type field of the sacmsg structure to one of the following:

SC_STATUS	status request
SC_ENABLE	enable message
SC_DISABLE	disable message
SC_READDB	message indicating that the port monitor’s _pmtab file should be read

sc_size indicates the size of the optional data part of the message. See “Message Classes” on page 405. For Solaris, sc_size should always be set to 0.

A port monitor must respond to every message sent by the sac.

Port Monitor Messages

The format of messages from a port monitor to the *sac* is defined by the structure *pmmsg*:

```
struct pmmsg {
    char pm_type;                /* type of message */
    unchar pm_state;            /* current state of port monitor */
    char pm_maxclass;            /* maximum message class this port
                                    monitor understands */
    char pm_tag[PMTAGSIZE + 1];  /* port monitor's tag */
    int pm_size;                /* size of optional data portion */
};
```

Port monitors may send two types of messages to the sac. The type of message is indicated by setting the pm_type field of the pmmsg structure to one of the following:

PM_STATUS	state information
PM_UNKNOWN	negative acknowledgment

For both types of messages, the pm_tag field is set to the port monitor's tag and the pm_state field is set to the port monitor's current state. Valid states are:

PM_STARTING	starting
PM_ENABLED	enabled
PM_DISABLED	disabled
PM_STOPPING	stopping

The current state reflects any changes caused by the last message from the sac.

The status message is the normal return message. The negative acknowledgment should be sent only when the message received is not understood.

pm_size indicates the size of the optional data part of the message. pm_maxclass is used to specify a message class. Both are discussed under "Message Classes.". In Solaris, always set pm_maxclass to 1 and sc_size to 0.

Port monitors may never initiate messages; they may only respond to messages that they receive.

Message Classes

The concept of message class has been included to accommodate possible SAF extensions. The messages described above are all class 1 messages. None of these messages contains a variable data portion; all pertinent information is contained in the message header.

If new messages are added to the protocol, they will be defined as new message classes (for example, class 2). The first message the `sac` sends to a port monitor will always be a class 1 message. Since all port monitors, by definition, understand class 1 messages, the first message the `sac` sends is guaranteed to be understood. In its response to the `sac`, the port monitor sets the `pm_maxclass` field to the maximum message class number for that port monitor. The `sac` will not send messages to a port monitor from a class with a larger number than the value of `pm_maxclass`. Requests that require messages of a higher class than the port monitor can understand will fail. For Solaris, always set `pm_maxclass` to `1`.

Note – For any given port monitor, messages of class `pm_maxclass` and messages of all classes with values lower than `pm_maxclass` are valid. Thus, if the `pm_maxclass` field is set to `3`, the port monitor understands messages of classes `1`, `2`, and `3`. Port monitors may not generate messages; they may only respond to messages. A port monitor's response must be of the same class as the originating message.

Since only the `sac` can generate messages, this protocol will function even if the port monitor is capable of dealing with messages of a higher class than the `sac` can generate.

`pm_size` (an element of the `pmmsg` structure) and `sc_size` (an element of the `sacmsg` structure) indicate the size of the optional data part of the message. The format of this part of the message is undefined. Its definition is inherent in the type of message. For Solaris, always set both `sc_size` and `pm_size` to `0`.

The Port Monitor Administrative Interface

This section discusses the administrative files available under the SAC.

D

The SAC Administrative File _sactab

The service access controller's administrative file contains information about all the port monitors for which the SAC is responsible. This file exists on the delivered system. Initially, it is empty except for a single comment line that contains the version number of the SAC. Port monitors are added to the system by making entries in the SAC's administrative file. These entries should be made using the administrative command `sacadm` with a `–a` option. `sacadm` is also used to remove entries from the SAC's administrative file.

Each entry in the SAC's administrative file contains the following information, shown in Table D-1.

Table D-1 Service Access Controller _sactab File

Fields	Description
PMTAG	A unique tag that identifies a particular port monitor. The system administrator is responsible for naming a port monitor. This tag is then used by the SAC to identify the port monitor for all administrative purposes. PMTAG may consist of up to 14 alphanumeric characters.
PMTYPE	The type of the port monitor. In addition to its unique tag, each port monitor has a type designator. The type designator identifies a group of port monitors that are different invocations of the same entity. `ttymon` and `listen` are examples of valid port monitor types. The type designator is used to facilitate the administration of groups of related port monitors. Without a type designator, the system administrator has no way of knowing which port monitor tags correspond to port monitors of the same type. PMTYPE may consist of up to 14 alphanumeric characters.
FLGS	The flags that are currently defined are: `d` When started, do not enable the port monitor. `x` Do not start the port monitor. If no flag is specified, the default action is taken. By default a port monitor is started and enabled.
RCNT	The number of times a port monitor may fail before being placed in a failed state. Once a port monitor enters the failed state, the SAC will not try to restart it. If a count is not specified when the entry is created, this field is set to `0`. A restart count of `0` indicates that the port monitor is not to be restarted when it fails.
COMMAND	A string representing the command that will start the port monitor. The first component of the string, the command itself, must be a full path name.

The Port Monitor Administrative File _pmtab

Each port monitor will have two directories for its exclusive use. The current directory will contain files defined by the SAF (`_pmtab`, `_pid`) and the per-service configuration scripts, if they exist. The directory `/var/saf/`*`pmtag`*, where *pmtag* is the tag of the port monitor, is available for the port monitor's private files.

Each port monitor has its own administrative file. The `pmadm` command should be used to add, remove, or modify service entries in this file. Each time a change is made using `pmadm`, the corresponding port monitor rereads its administrative file. Each entry in a port monitor's administrative file defines how the port monitor treats a specific port and what service is to be invoked on that port.

Some fields must be present for all types of port monitors. Each entry must include a service tag to identify the service uniquely and an identity to be assigned to the service when it is started (for example, root).

Note – The combination of a service tag and a port monitor tag uniquely define an instance of a service. The same service tag may be used to identify a service under a different port monitor. The record must also contain port monitor specific data (for example, for a `ttymon` port monitor, this will include the prompt string which is meaningful to `ttymon`). Each type of port monitor must provide a command that takes the necessary port monitor-specific data as arguments and outputs these data in a form suitable for storage in the file. The `ttyadm`() command does this for `ttymon` and `nlsadmin`() does it for `listen`. For a user-defined port monitor, a similar administrative command must also be supplied.

Each service entry in the port monitor administrative file must have the following format and contain the information listed below:

`svctag``:`*`flgs`*`:`*`id`*`:reserved:reserved:reserved:`*`pmspecific`*`#`
`comment`

`SVCTAG` is a unique tag that identifies a service. This tag is unique only for the port monitor through which the service is available. Other port monitors may offer the same or other services with the same tag. A service requires both a port monitor tag and a service tag to identify it uniquely.

SVCTAG may consist of up to 14 alphanumeric characters. The service entries are defined in Table D-2.

.

Table D-2 SVCTAG Service Entries

Service Entries	Description
FLGS	Flags with the following meanings may currently be included in this field: x Do not enable this port. By default the port is enabled. u Create a utmp entry for this service. By default no utmp entry is created for the service.
ID	The identity under which the service is to be started. The identity has the form of a login name as it appears in /etc/passwd.
PMSPECIFIC	Examples of port monitor information are addresses, the name of a process to execute, or the name of a STREAMS pipe to pass a connection through. This information will vary to meet the needs of each different type of port monitor.
COMMENT	A comment associated with the service entry.

Note – Port monitors may ignore the u flag if creating a utmp entry for the service is not appropriate to the manner in which the service is to be invoked. Some services may not start properly unless utmp entries have been created for them (for example, login).

Each port monitor administrative file must contain one special comment of the form:

```
# VERSION=value
```

where *value* is an integer that represents the port monitor's version number. The version number defines the format of the port monitor administrative file. This comment line is created automatically when a port monitor is added to the system. It appears on a line by itself, before the service entries.

The SAC Administrative Command sacadm

sacadm is the administrative command for the upper level of the SAF hierarchy, that is, for port monitor administration (see the sacadm(1M) manpage). Under the SAF, port monitors are administered by using the sacadm command to make changes in the SAC's administrative file. sacadm performs the following functions:

- Prints requested port monitor information from the SAC administrative file
- Adds or removes a port monitor
- Enables or disables a port monitor
- Starts or stops a port monitor
- Installs or replaces a per-system configuration script
- Installs or replaces a per-port monitor configuration script
- Asks the SAC to reread its administrative file

The Port Monitor Administrative Command `pmadm`

`pmadm` is the administrative command for the lower level of the SAF hierarchy, that is, for service administration (see the `pmadm(1M)` manpage). A port may have only one service associated with it although the same service may be available through more than one port. `pmadm` performs the following functions:

- Prints service status information from the port monitor's administrative file
- Adds or removes a service
- Enables or disables a service
- Installs or replaces a per-service configuration script

Note that in order to identify an instance of a service uniquely, the `pmadm` command must identify both the service (`-s`) and the port monitor or port monitors through which the service is available (`-p` or `-t`).

Monitor-Specific Administrative Command

In the previous section, two pieces of information included in the *`_pmtab`* file were described: the port monitor's version number and the port monitor part of the service entries in the port monitor's *`_pmtab`* file. When a new port monitor is added, the version number must be known so that the *`_pmtab`* file can be correctly initialized. When a new service is added, the port monitor part of the *`_pmtab`* entry must be formatted correctly.

Each port monitor must have an administrative command to perform these two tasks. The person who defines the port monitor must also define such an administrative command and its input options. When the command is invoked with these options, the information required for the port monitor part of the service entry must be correctly formatted for inclusion in the port monitor's `_pmtab` file and must be written to the standard output. To request the version

number the command must be invoked with a –V option; when it is invoked in this way, the port monitor's current version number must be written to the standard output.

If the command fails for any reason during the execution of either of these tasks, no data should be written to standard output.

The Port Monitor/Service Interface

The interface between a port monitor and a service is determined solely by the service. Two mechanisms for invoking a service are presented here as examples.

New Service Invocations

The first interface is for services that are started anew with each request. This interface requires the port monitor to first `fork()` a child process. The child will eventually become the designated service by performing an `exec()`. Before the `exec()` happens, the port monitor may take some port monitor-specific action; however, one action that must occur is the interpretation of the per-service configuration script, if one is present. This is done by calling the library routine `doconfig()`.

Standing Service Invocations

The second interface is for invocations of services that are actively running. To use this interface, a service must have one end of a stream pipe open and be prepared to receive connections through it.

Port Monitor Requirements

To implement a port monitor, several generic requirements must be met. This section summarizes these requirements. In addition to the port monitor itself, an administrative command must be supplied.

Initial Environment

When a port monitor is started, it expects an initial execution environment in which:

- It has no file descriptors open
- It is not be a process group leader
- It has an entry in `/etc/utmp` of type `LOGIN_PROCESS`
- An environment variable, `ISTATE`, is set to “enabled” or “disabled” to indicate the port monitor’s correct initial state
- An environment variable, `PMTAG`, is set to the port monitor’s assigned tag
- The directory that contains the port monitor’s administrative files is its current directory
- The port monitor is able to create private files in the directory `/var/saf/`*`tag`*, where *tag* is the port monitor’s tag
- The port monitor is running with user id `0` (root)

Important Files

Relative to its current directory, the following key files exist for a port monitor.

Table D-3 Key Port Monitor Files

File	Description
`_config`	The port monitor’s configuration script. The port monitor configuration script is run by the SAC. The SAC is started by `init()` as a result of an entry in /etc/inittab that calls `sac()`.
`_pid`	The file into which the port monitor writes its process id.
`_pmtab`	The port monitor’s administrative file. This file contains information about the ports and services for which the port monitor is responsible.
`_pmpipe`	The FIFO through which the port monitor will receive messages from the `sac`.
`svctag`	The per-service configuration script for the service with the tag `svctag`.
`../_sacpipe`	The FIFO through which the port monitor will send messages to `sac()`.

Port Monitor Responsibilities

A port monitor is responsible for performing the following tasks in addition to its port monitor function:

- Write its process id into the file `_pid` and place an advisory lock on the file
- Terminate gracefully on receipt of the signal `SIGTERM`.
- Follow the protocol for message exchange with the `sac`

A port monitor must perform the following tasks during service invocation:

- Create a `utmp` entry if the requested service has the "u" flag set in `_pmtab`

Note – Port monitors may ignore this flag if creating a `utmp` entry for the service does not make sense because of the manner in which the service is to be invoked. On the other hand, some services may not start properly unless `utmp` entries have been created for them.

- Interpret the per-service configuration script for the requested service, if it exists, by calling the `doconfig()` library routine

Configuration Files and Scripts

Interpreting Configuration Scripts With `doconfig()`

The library routine `doconfig()`, defined in `libnsl.so`, interprets the configuration scripts contained in the files `/etc/saf/`*`pmtag`*`/_sysconfig` (the per-system configuration file), and `/etc/saf/_config` (per-port monitor configuration files); and in `/etc/saf/`*`pmtag`*`/`*svctag* (per-service configuration files). Its syntax is:

```
# include <sac.h>
int doconfig (int fd, char (**script, long rflag);
```

script is the name of the configuration script; *fd* is a file descriptor that designates the stream to which stream manipulation operations are to be applied; *rflag* is a bitmask that indicates the mode in which *script* is to be interpreted. *rflag* may take two values, NORUN and NOASSIGN, which may be or'd. If *rflag* is zero, all commands in the configuration script are eligible to be interpreted. If *rflag* has the NOASSIGN bit set, the `assign` command is considered illegal and will generate an error return. If *rflag* has the NORUN bit set, the `run` and `runwait` commands are considered illegal and will generate error returns.

If a command in the script fails, the interpretation of the script ceases at that point and a positive integer is returned; this number indicates which line in the script failed. If a system error occurs, a value of `-1` is returned.

If a script fails, the process whose environment was being established should *not* be started.

In the example, `doconfig()` is used to interpret a per-service configuration script.

```
. . .
    if ((i = doconfig (fd, svctag, 0)) != 0){
    error ("doconfigfailedonline%dofscript%s",i,svctag);
}
```

The Per-System Configuration File

The per-system configuration file, `/etc/saf/_sysconfig`, is delivered empty. It may be used to customize the environment for all services on the system by writing a command script in the interpreted language described in this chapter and on the `doconfig(3N)` manpage. When the SAC is started, it calls the `doconfig()` function to interpret the per-system configuration script. The SAC is started when the system enters multiuser mode.

Per-Port Monitor Configuration Files

Per-port monitor configuration scripts (`/etc/saf/`*`pmtag`*`/_config`) are optional. They allow the user to customize the environment for any given port monitor and for the services that are available through the ports for which that port monitor is responsible. Per-port monitor configuration scripts are written in the same language used for per-system configuration scripts.

The per-port monitor configuration script is interpreted when the port monitor is started. The port monitor is started by the SAC after the SAC has itself been started and after it has run its own configuration script, `/etc/saf/_sysconfig`.

The per-port monitor configuration script may override defaults provided by the per-system configuration script.

Per-Service Configuration Files

Per-service configuration files allow the user to customize the environment for a specific service. For example, a service may require special privileges that are not available to the general user. Using the language described in the `doconfig(3N)` manpage, you can write a script that will grant or limit such special privileges to a particular service offered through a particular port monitor.

The per-service configuration may override defaults provided by higher-level configuration scripts. For example, the per-service configuration script may specify a set of STREAMS modules other than the default set.

The Configuration Language

The language in which configuration scripts are written consists of a sequence of commands, each of which is interpreted separately. The following reserved keywords are defined: `assign`, `push`, `pop`, `runwait`, and `run`. The comment character is #. Blank lines are not significant. No line in a command script may exceed 1024 characters.

`assign` *`variable=value`*

Used to define environment variables. *variable* is the name of the environment variable and *value* is the value to be assigned to it. The value assigned must be a string constant; no form of parameter substitution is available. *value* may be quoted. The quoting rules are those used by the shell for defining environment variables. `assign` will fail if space cannot be allocated for the new variable or if any part of the specification is invalid.

`push` *`module1`*`[, `*`module2`*`, `*`module3`*`, . . .]`

Used to push STREAMS modules onto the stream designated by *fd*. See the `doconfig(3N)`manpage. *module1* is the name of the first module to be pushed, *module2* is the name of the second module to be pushed, and so on. The command will fail if any of the named modules cannot be pushed. If a module cannot be pushed, the subsequent modules on the same command line will be ignored and modules that have already been pushed will be popped.

`pop [`*`module`*`]`

Used to pop STREAMS modules off the designated stream. If `pop` is invoked with no arguments, the top module on the stream is popped. If an argument is given, modules will be popped one at a time until the named module is at the top of the stream. If the named module is not on the designated stream, the stream is left as it was and the command fails. If *module* is the special keyword ALL, then all modules on the stream will be popped. Note that only modules above the topmost driver are affected.

`runwait` *`command`*

The runwait command runs a command and waits for it to complete. *command* is the path name of the command to be run. The command is run with /bin/sh -c prepended to it; shell scripts may thus be executed from configuration scripts. The runwait command will fail if *command* cannot be found or cannot be executed, or if *command* exits with a nonzero status.

run *command*

The run command is identical to runwait except that it does not wait for *command* to complete. *command* is the path name of the command to be run. run will not fail unless it is unable to create a child process to execute the command.

Although they are syntactically indistinguishable, some of the commands available to run and runwait are interpreter built-in commands. Interpreter built-ins are used when it is necessary to alter the state of a process within the context of that process. The doconfig interpreter built-in commands are similar to the shell special commands and, like these, they do not spawn another process for execution. See the sh(1)manpage. The initial set of built-in commands is:

cd ulimit umask

Printing, Installing, and Replacing Configuration Scripts

This section describes the form of the SAC and port monitor administrative commands used to install the three types of configuration scripts. Per-system and per-port monitor configuration scripts are administered using the sacadm command. Per-service configuration scripts are administered using the pmadm command.

Per-System Configuration Scripts

sacadm -G [-z *script*]

The -G option is used to print or replace the per-system configuration script. The -G option by itself prints the per-system configuration script. The -G option in combination with a -z option replaces /etc/saf/_sysconfig with the contents of the file *script*. Other combinations of options with a -G option are invalid.

Sample Per-System Configuration Script

The `_sysconfig` file in the example sets the time zone variable, `TZ`.

```
assign TZ=EST5EDT # set TZ
runwait echo SAC is starting > /dev/console
```

Per-Port Monitor Configuration Scripts

```
sacadm –g –p pmtag [ –z script ]
```

The `–g` option is used to print, install, or replace the per-port monitor configuration script. A `–g` option requires a `–p` option. The `–g` option with only a `–p` option prints the per-port monitor configuration script for port monitor *pmtag*. The `–g` option with a `–p` option and a `–z` option installs the file *script* as the per-port monitor configuration script for port monitor *pmtag*, or, if `/etc/saf/`*`pmtag`*`/_config` exists, it replaces `_config` with the contents of *script*. Other combinations of options with `–g` are invalid.

Sample Per-Port Monitor Configuration Script

In the hypothetical`_config` file in the figure, the command `/usr/bin/daemon` is assumed to start a daemon process that builds and holds together a STREAMS multiplexor. By installing this configuration script, the command can be executed just before starting the port monitor that requires it.

```
# build a STREAMS multiplexor
run /usr/bin/daemon
runwait echo $PMTAG is starting > /dev/console
```

Per-Service Configuration Scripts

```
pmadm –g –p pmtag –s svctag [ –z script ]
pmadm –g –s svctag –t type –z script
```

Per-service configuration scripts are interpreted by the port monitor before the service is invoked.

Note – The SAC interprets both its own configuration file, `_sysconfig`, and the port monitor configuration files. Only the per-service configuration files are interpreted by the port monitors.

The -g option is used to print, install, or replace a per-service configuration script. The -g option with a -p option and a -s option prints the per-service configuration script for service *svctag* available through port monitor *pmtag*. The -g option with a -p option, a -s option, and a -z option installs the per-service configuration script contained in the file *script* as the per-service configuration script for service *svctag* available through port monitor *pmtag*. The -g option with a -s option, a -t option, and a -z option installs the file *script* as the per-service configuration script for service *svctag* available through any port monitor of type *type*. Other combinations of options with -g are invalid.

Sample Per-Service Configuration Script

The following per-service configuration script does two things: It specifies the maximum file size for files created by a process by setting the process's `ulimit` to 4096. It also specifies the protection mask to be applied to files created by the process by setting `umask` to 077.

```
runwait ulimit 4096
runwait umask 077
```

Sample Port Monitor Code

Code Example D-1shows an example of a "null" port monitor that simply responds to messages from the SAC.

Code Example D-1 Sample Port Monitor

```
# include <stdio.h>
# include <unistd.h>
# include <fcntl.h>
# include <signal.h>
# include <sac.h>

char *getenv();
char Scratch[BUFSIZ]; /* scratch buffer */
char Tag[PMTAGSIZE]; /* port monitor's tag */
FILE *Fp; /* file pointer for log file */
FILE *Tfp; /* file pointer for pid file */
char State; /* portmonitor'scurrentstate*/

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

```
{
    char *istate;
    strcpy(Tag, getenv("PMTAG"));
/*
 * open up a log file in port monitor's private directory
 */
    sprintf(Scratch, "/var/saf/%s/log", Tag);
    Fp = fopen(Scratch, "a+");
    if (Fp == (FILE *)NULL)
        exit(1);
    log(Fp, "starting");
/*
 * retrieve initial state (either "enabled" or "disabled") and set
 * State accordingly
 */
    istate = getenv("ISTATE");
    sprintf(Scratch, "ISTATE is %s", istate);
    log(Fp, Scratch);
    if (!strcmp(istate, "enabled"))
        State = PM_ENABLED;
    else if (!strcmp(istate, "disabled"))
        State = PM_DISABLED;
    else {
        log(Fp, "invalid initial state");
        exit(1);
    }
    sprintf(Scratch, "PMTAG is %s", Tag);
    log(Fp, Scratch);
/*
 * set up pid file and lock it to indicate that we are active
 */
    Tfp = fopen("_pid", "w");
    if (Tfp == (FILE *)NULL) {
        log(Fp, "couldn't open pid file");
        exit(1);
    }
    if (lockf(fileno(Tfp), F_TEST, 0) < 0) {
        log(Fp, "pid file already locked");
        exit(1);
    }
    fprintf(Tfp, "%d", getpid());
    rewind(Tfp);
    log(Fp, "locking file");
    if (lockf(fileno(Tfp), F_LOCK, 0) < 0) {
        log(Fp, "lock failed");
        exit(1);
```

```
    }
/*
 * handle poll messages from the sac ... this function never returns
 */
    handlepoll();
    pause();
    fclose(Tfp);
    fclose(Fp);
}

handlepoll()
{
    int pfd; /* file descriptor for incoming pipe */
    int sfd; /* file descriptor for outgoing pipe */
    struct sacmsg sacmsg; /* incoming message */
    struct pmmsg pmmsg; /* outgoing message */
/*
 * open pipe for incoming messages from the sac
 */
    pfd = open("_pmpipe", O_RDONLY);
    if (pfd < 0) {
        log(Fp, "_pmpipe open failed");
        exit(1);
    }
/*
 * open pipe for outgoing messages to the sac
 */
    sfd = open("../_sacpipe", O_WRONLY);
    if (sfd < 0) {
        log(Fp, "_sacpipe open failed");
        exit(1);
    }
/*
 * start to build a return message; we only support class 1 messages
 */
    strcpy(pmmsg.pm_tag, Tag);
    pmmsg.pm_size = 0;
    pmmsg.pm_maxclass = 1;
/*
 * keep responding to messages from the sac
 */
    for (;;) {
        if (read(pfd, &sacmsg, sizeof(sacmsg)) != sizeof(sacmsg)) {
            log(Fp, "_pmpipe read failed");
            exit(1);
        }
```

```
/*
 * determine the message type and respond appropriately
 */
       switch (sacmsg.sc_type) {
           case SC_STATUS:
               log(Fp, "Got SC_STATUS message");
               pmmsg.pm_type = PM_STATUS;
               pmmsg.pm_state = State;
               break;
           case SC_ENABLE:
               /*note internal state change below*/
               log(Fp, "Got SC_ENABLE message");
               pmmsg.pm_type = PM_STATUS;
               State = PM_ENABLED;
               pmmsg.pm_state = State;
               break;
           case SC_DISABLE:
               /*noteinternalstatechangebelow*/
               log(Fp, "Got SC_DISABLE message");
               pmmsg.pm_type = PM_STATUS;
               State = PM_DISABLED;
               pmmsg.pm_state = State;
               break;
           case SC_READDB:
               /*
                * if this were a fully functional port monitor it
                * would read _pmtab here and take appropriate action
                */
               log(Fp, "Got SC_READDB message");
               pmmsg.pm_type = PM_STATUS;
               pmmsg.pm_state = State;
               break;
           default:
               sprintf(Scratch, "Got unknown message <%d>",
               sacmsg.sc_type);
               log(Fp, Scratch);
               pmmsg.pm_type = PM_UNKNOWN;
               pmmsg.pm_state = State;
               break;
       }
/*
 * send back a response to the poll
 * indicating current state
 */
       if (write(sfd, &pmmsg, sizeof(pmmsg)) != sizeof(pmmsg))
           log(Fp, "sanity response failed");
```

```
    }
}
/*
 * general logging function
 */
log(fp, msg)
    FILE *fp;
    char *msg;
{
    fprintf(fp, "%d; %sen", getpid(), msg);
    fflush(fp);
}
```

Code Example D-2 shows the sac.h header file.

Code Example D-2 sac.h Header File

```
/* length in bytes of a utmp id */
# define IDLEN 4
/* wild character for utmp ids */
# define SC_WILDC 0xff
/* max len in bytes for port monitor tag */
# define PMTAGSIZE 14
/*
 * values for rflag in doconfig()
 */
/* don't allow assign operations */
# define NOASSIGN 0x1
/* don't allow run or runwait operations */
# define NORUN 0x2
/*
 * message to SAC (header only). This header is forever fixed. The
 * size field (pm_size) defines the size of the data portion of the
 * message, which follows the header. The form of this optional data
 * portion is defined strictly by the message type (pm_type).
 */
struct pmmsg {
    char pm_type;               /* type of message */
    unchar pm_state;            /* current state of pm */
    char pm_maxclass;         /* max message class this port monitor
                                 understands */
    char pm_tag[PMTAGSIZE + 1]; /* pm's tag */
    int pm_size;                /* size of opt data portion */
};
/*
 * pm_type values
```

```
 */
# define PM_STATUS 1 /* status response */
# define PM_UNKNOWN 2 /* unknown message was received */
/*
 * pm_state values
 */
/*
 * Class 1 responses
 */
# define PM_STARTING 1   /* monitor in starting state */
# define PM_ENABLED 2    /* monitor in enabled state */
# define PM_DISABLED 3   /* monitor in disabled state */
# define PM_STOPPING 4   /* monitor in stopping state */
/*
 * message to port monitor
 */
struct sacmsg {
    int sc_size;         /* size of optional data portion */
    char sc_type;        /* type of message */
};
/*
 * sc_type values
 * These represent commands that the SAC sends to a port monitor.
 * These commands are divided into "classes" for extensibility. Each
 * subsequent "class" is a superset of the previous "classes" plus
 * the new commands defined within that "class". The header for all
 * commands is identical; however, a command may be defined such that
 * an optional data portion may be sent in addition to the header.
 * The format of this optional data piece is self-defining based on
 * the command. Important note: the first message sent by the SAC
will
 * always be a class 1 message. The port monitor response indicates
 * the maximum class that it is able to understand. Another note is
 * that port monitors should only respond to a message with an
 * equivalent class response (i.e. a class 1 command causes a class 1
 * response).
 */
/*
 * Class 1 commands (currently, there are only class 1 commands)
 */
# define SC_STATUS 1    /* status request *
# define SC_ENABLE 2    /* enable request */
# define SC_DISABLE 3   /* disable request */
# define SC_READDB 4    /* read pmtab request */
/*
 * 'errno' values for Saferrno, note that Saferrno is used by both
```

```
 * pmadm and sacadm and these values are shared between them
 */
# define E_BADARGS 1   /* bad args/ill-formed cmd line */
# define E_NOPRIV 2    /* user not priv for operation */
# define E_SAFERR 3    /* generic SAF error */
# define E_SYSERR 4    /* system error */
# define E_NOEXIST 5   /* invalid specification */
# define E_DUP 6       /* entry already exists */
# define E_PMRUN 7     /* port monitor is running */
# define E_PMNOTRUN 8  /* port monitor is not running */
# define E_RECOVER 9   /* in recovery */
```

Logic Diagram and Directory Structure

Figure D-1 is a logical diagram of the SAF. It illustrates how a single service access controller may spawn a number of port monitors on a per-system basis. This means that several monitors may be running concurrently, providing for the simultaneous operation of several different protocols.

Figure D-2 is the corresponding directory structure diagram. Following the diagram is a description of the files and directories.

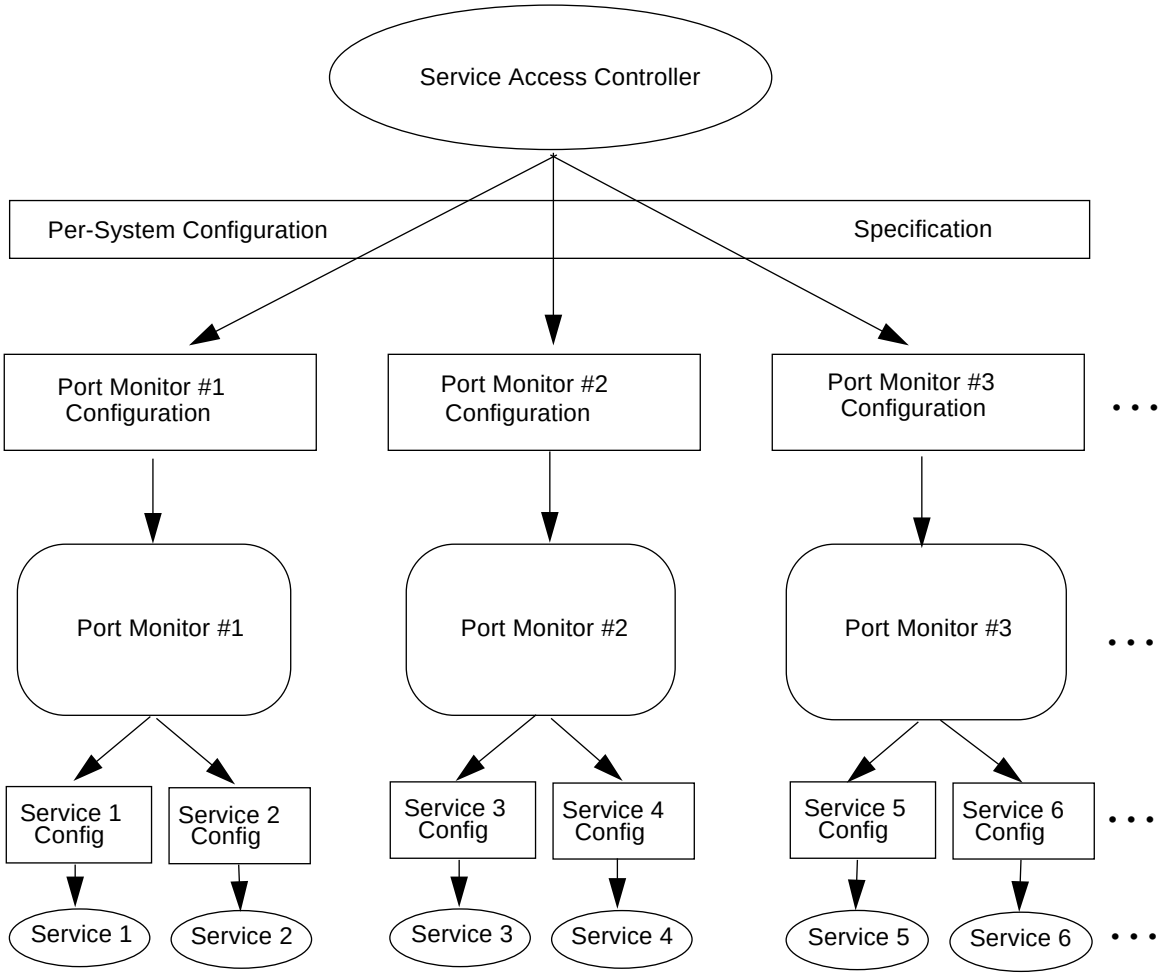


Figure D-1 SAF Logical Framework

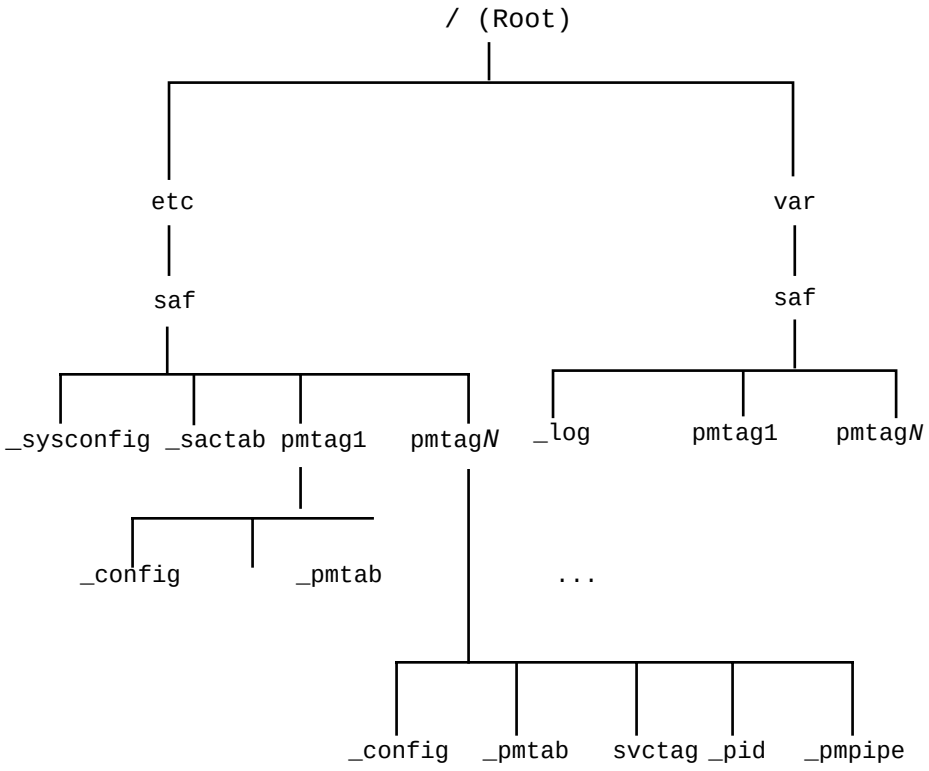


Figure D-2 SAF Directory Structure

/etc/saf/_sysconfig

The per-system configuration script.

/etc/saf/_sactab

The SAC's administrative file. Contains information about the port monitors for which the SAC is responsible.

/etc/saf/pmtag

The home directory for port monitor *pmtag*.

/etc/saf/pmtag/_config

The per-port monitor configuration script for port monitor *pmtag*.

/etc/saf/pmtag/_pmtab

Port monitor *pmtag*'s administrative file. Contains information about the services for which *pmtag* is responsible.

/etc/saf/pmtag/svctag

The file in which the per-service configuration script for service *svctag* (available through port monitor *pmtag*) is placed.

/etc/saf/pmtag/_pid

The file in which a port monitor writes its process id in the current directory and places an advisory lock on the file.

/etc/saf/pmtag/_pmpipe

The file in which the port monitor receives messages from the `sac` and `../_sacpipe` and sends return messages to the `sac`.

/var/saf/_log

The SAC's log file.

/var/saf/pmtag

The directory for files created by port monitor *pmtag*, for example its log file.

The `portmap` *Utility*

E

The `rpcbind` utility replaces the `portmap` utility available in previous releases of SunOS. This appendix is included to help you understand the history of port and network address resolution using the `portmap` utility.

SunOS 4.x RPC-based services use `portmap` as a system registration service. It manages a table of correspondences between ports (logical communications channels) and the services registered at them. It provides a standard way for a client to look up the TCP/IP or UDP/IP port number of an RPC program supported by the server.

System Registration Overview

For client programs to find distributed services on a network, they need a way to look up the network addresses of server programs. Network transport (protocol) services do not provide this function. Their task is to provide process-to-process message transfer across a network (that is, a message is sent to a transport-specific network address). A network address is a logical communications channel — by listening on a specific network address, a process receives messages from the network.

The way a process waits on a network address varies from one operating system to the next, but all provide mechanisms by which a process can synchronize its activity with arriving messages. Messages are not sent across networks to receiving processes, but rather to the network address at which receiving processes pick them up. Network addresses are valuable because they allow message receivers to be specified in a way that is independent of

the conventions of the receiving operating system. TI-RPC, being transport independent, makes no assumptions about the structure of a network address. It uses a universal addresses. This universal address is specified as a null-terminated string of characters. Such universal addresses are translated into local transport addresses by routines specific to each transport provider.

The `rpcbind` protocol defines a network service that provides a standard way for clients to look up the network address of any remote program supported by a server. Because it can be implemented on any transport, it provides a single solution to a general problem that works for all clients, all servers, and all networks.

`portmap` Protocol

The `portmap` program maps RPC program and version numbers to transport-specific port numbers. This program makes dynamic binding of remote programs possible.

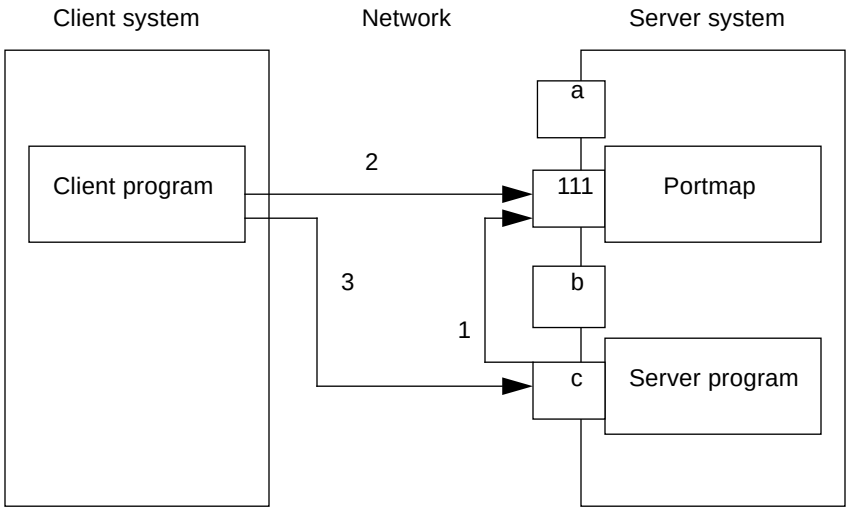


Figure E-1 Typical Portmap Sequence (For TCP/IP Only)

Figure E-1 shows the process:

1. The server registers with `portmap`
2. The client gets the server's port from `portmap`
3. The client calls the server.

The range of reserved port numbers is small and the number of potential remote programs is very large. By running only the port mapper on a well known port, the port numbers of other remote programs can be ascertained by querying the port mapper. In Figure E-1, a, 111, b, and c represent port numbers, where 111 is the assigned portmapper port number.

The port mapper also aids in broadcast RPC. A given RPC program usually has different port number bindings on different machines, so there is no way to directly broadcast to all of these programs. The port mapper, however, does have a fixed port number. So, to broadcast to a given program, the client actually sends its message to the port mapper located at the broadcast address. Each port mapper that receives the broadcast then calls the local service specified by the client. When `portmap` gets the reply from the local service, it returns the reply to the client. The `portmap` protocol specification is shown in Code Example E-1.

Code Example E-1 `portmap` Protocol Specification (in RPC Language)

```
const PMAP_PORT = 111;       /* portmapper port number */
 /*
  * A mapping of (program, version, protocol) to port number
  */
 struct pmap {
    unsigned int prog;
    unsigned int vers;
    unsigned int prot;
    unsigned int port;
 };
 /*
  * Supported values for the "prot" field
  */
 const IPPROTO_TCP = 6; /* protocol number for TCP/IP */
 const IPPROTO_UDP = 17; /* protocol number for UDP/IP */
 /*
  * A list of mappings
  */
 struct pmaplist {
```

```
   pmap map;
   pmaplist *next;
};
/*
 * Arguments to callit
 */
struct call_args {
   unsigned int prog;
   unsigned int vers;
   unsigned int proc;
   opaque args<>;
};
/*
* Results of callit
*/
struct call_result {
   unsigned int port;
   opaque res<>;
};
/*
* Port mapper procedures
*/
program PMAP_PROG {
   version PMAP_VERS {
       void
       PMAPPROC_NULL(void) = 0;
       bool
       PMAPPROC_SET(pmap) = 1;
       bool
       PMAPPROC_UNSET(pmap) = 2;
       unsigned int
       PMAPPROC_GETPORT(pmap) = 3;
       pmaplist
       PMAPPROC_DUMP(void) = 4;
       call_result
       PMAPPROC_CALLIT(call_args) = 5;
   } = 2;
} = 100000;
```

portmap Operation

portmap currently supports two protocols (UDP/IP and TCP/IP). portmap is contacted by talking to it on assigned port number 111 (SUNRPC (5)) on either of these protocols. The following is a description of each of the portmapper procedures.

PMAPPROC_NULL

This procedure does no work. By convention, procedure zero of any protocol takes no parameters and returns no results.

PMAPPROC_SET

When a program first becomes available on a machine, it registers itself with the local port map program. The program passes its program number “prog”, version number “vers”, transport protocol number “prot”, and the port “port” on which it receives service requests. The procedure refuses to establish a mapping if one already exists for the specified port and it is bound. If the mapping exists and the port is not bound, portmap unregisters the port and performs the requested mapping. The procedure returns TRUE if the procedure successfully established the mapping and FALSE otherwise. See also the pmap_set function in the rpc_soc(3N) manpage.

PMAPPROC_UNSET

When a program becomes unavailable, it should unregister itself with the port mapper program on the same machine. The parameters and results have meanings identical to those of PMAPPROC_SET. The protocol and port number fields of the argument are ignored. See also the pmap_unset function in the rpc_soc(3N) manpage.

PMAPPROC_GETPORT

Given a program number “prog”, version number “vers”, and transport protocol number “prot”, this procedure returns the port number on which the program is awaiting call requests. A port value of zeros means the program has not been registered. The “port” field of the argument is ignored. See also the pmap_getport function in the rpc_soc(3N) manpage.

PMAPPROC_DUMP

This procedure enumerates all entries in the port mapper's database. The procedure takes no parameters and returns a list of program, version, protocol, and port values. See also the `pmap_getmaps` function in the `rpc_soc(3N)` manpage.

PMAPPROC_CALLIT

This procedure allows a caller to call another remote procedure on the same machine without knowing the remote procedure's port number. It is intended for supporting broadcasts to arbitrary remote programs via the well-known port mapper's port. The parameters "prog," "vers," "proc," and the bytes of "args" are the program number, version number, procedure number, and parameters of the remote procedure. See also the `pmap_rmtcall` function in the `rpc_soc(3N)` manpage.

This procedure only sends a response if the procedure was successfully executed and is silent (no response) otherwise. It also returns the remote program's port number, and the bytes of results are the results of the remote procedure.

The port mapper communicates with the remote program using UDP/IP only.

Bibliography

1. Birrell, Andrew D. and Nelson, Bruce Jay; "Implementing Remote Procedure Calls"; *XEROX* CSL-83-7, October 1983.
2. Cheriton, D.; "VMTP: Versatile Message Transaction Protocol," Preliminary Version 0.3; Stanford University, January 1987.
3. Postel, J.; "Transmission Control Protocol - DARPA Internet Program Protocol Specification," *RFC* 793; Information Sciences Institute, September 1981.
4. Postel, J.; "User Datagram Protocol," *RFC* 768; Information Sciences Institute, August 1980.
5. Reynolds, J. & Postel, J.; "Assigned Numbers," *RFC* 923; Information Sciences Institute, October 1984.

Live RPC Code Examples F

This appendix contains copies of the complete live code modules used in the `rpcgen` and RPC chapters of this book. They are compilable as they are written and will run (unless otherwise noted to be pseudo-code or the like). They are provided for informational purposes only. SunSoft assumes no liability from their use.

▼ Directory Listing Program and Support Routines (`rpcgen`)

Code Example F-1 `rpcgen` Program: `dir.x`

```
/*
 * dir.x: Remote directory listing
 * protocol
 *
 * This source module is a rpcgen source module
 * used to demonstrate the functions of the rpcgen
 * tool.
 *
 * It is compiled with the rpcgen -h -T switches to
 * generate both the header (.h) file and the
 * accompanying data structures.
 *
 */
const MAXNAMELEN = 255;/*maxlengthofadirectoryentry*/

typedef string nametype<MAXNAMELEN>; /* directory entry */
typedef struct namenode *namelist;   /*linkinthelisting*/
```

```
/*
 * A node in the directory listing
 */
struct namenode {
   nametype name;       /* name of directory entry */
   namelist next;       /* next entry */
 };

/*
 * The result of a READDIR operation:
 *  a truly portable application would use an agreed upon list of
 *  error codes rather than, as this sample program does, rely upon
 *  passing UNIX errno's back. In this example the union is used to
 *   discriminate between successful and unsuccessful remote calls.
 */
union readdir_res switch (int errno) {
   case 0:
      namelist list; /*no error: return directory listing*/
   default:
      void;      /*error occurred: nothing else to return*/
};

/*
 * The directory program definition
 */
program DIRPROG {
   version DIRVERS {
      readdir_res
      READDIR(nametype) = 1;
   } = 1;
 } = 0x20000076;
```

Code Example F-2 Remote `dir_proc.c`

```
/*
 * dir_proc.c: remote readdir implementation
 */
#include <rpc/rpc.h>      /* Always needed */
#include <dirent.h>
#include "dir.h"          /* Created by rpcgen */

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

/* ARGSUSED1*/
```

```
readdir_res *
readdir_1(dirname,req)
   nametype *dirname;
   struct svc_req *req;
{
   DIR *dirp;
   struct dirent *d;
   namelist nl;
   namelist *nlp;
   static readdir_res res; /* must be static! */

   /*
    * Open directory
    */
   dirp = opendir(*dirname);
   if (dirp == (DIR *)NULL) {
      res.errno = errno;
      return (&res);
   }
   /*
    * Free previous result
    */
   xdr_free(xdr_readdir_res, &res);
   /*
    * Collect directory entries. Memory allocated here is freed by
    * xdr_free the next time readdir_1 is called.
    */

   nlp = &res.readdir_res_u.list;
   while (d = readdir(dirp)) {
      nl = *nlp = (namenode *) malloc(sizeof(namenode));
      if (nl == (namenode *) NULL) {
         res.errno = EAGAIN;
         closedir(dirp);
         return(&res);
      }
      nl->name = strdup(d->d_name);
      nlp = &nl->next;
   }
   *nlp = (namelist)NULL;
   /* Return the result */
   res.errno = 0;
   closedir(dirp);
   return (&res);
}
```

Code Example F-3 `rls.c` Client

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

#include <stdio.h>
#include <rpc/rpc.h>/* always need this */
#include "dir.h"    /* generated by rpcgen */

extern int errno;

main(argc, argv)
   int argc;
   char *argv[];
{
   CLIENT *cl;
   char *server;
   char *dir;
   readdir_res *result;
   namelist nl;

   if (argc != 3) {
      fprintf(stderr, "usage: %s host directory\n",
      argv[0]);
      exit(1);
   }
   server = argv[1];
    dir = argv[2];
/*
 * Create client "handle" used for calling MESSAGEPROG on the server
 * designated on the command line.
 */
   cl = clnt_create(server, DIRPROG, DIRVERS, "visible");
   if (cl == (CLIENT *)NULL) {
      clnt_pcreateerror(server);
      exit(1);
   }

   result = readdir_1(&dir, cl);
   if (result == (readdir_res *)NULL) {
      clnt_perror(cl, server);
      exit(1);
   }
```

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

   if (result->errno != 0) {
   /*
    * A remote system error occurred. Print error message and die.
    */
   }
   if (result->errno < sys_nerr)
      fprintf (stderr, "%s : %s\n", dir,
      sys_enlist[result->errno]);
      errno = result->errno;
      perror(dir);
      exit(1);
   }

/* Successfully got a directory listing. Print it out. */
   for(nl = result->readdir_res_u.list; nl != NULL; nl = nl->next) {
      printf("%s\n", nl->name);
   }
exit(0);
}
```

▼ Time Server Program (rpcgen)

Code Example F-4 rpcgen Program: time.x

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

#ifdef RPC_SVC
%int *
%timeget_1()
%{
%  static int thetime;
%
%  thetime = time(0);
%  return (&thetime);
%}
#endif
```

▼ Add Two Numbers Program (rpcgen)

Code Example F-5 rpcgen program: Add Two Numbers

```
/* This program contains a procedure to add 2 numbers to demonstrate
 * some of the features of the new rpcgen. Note that add() takes 2
 * arguments in this case.
 */
program ADDPROG {/* program number */
   version ADDVER {/* version number */
      int add ( int, int )  /* procedure */
         = 1;
   } = 1;
} = 199;
```

▼ Spray Packets Program (rpcgen)

Code Example F-6 rpcgen program: spray.x

```
/*
 * Copyright (c) 1987, 1991 by Sun Microsystems, Inc.
 */

/* from spray.x */

#ifdef RPC_HDR
#pragma ident "@(#)spray.h  1.2  91/09/17 SMI"
#endif

/*
 * Spray a server with packets
 * Useful for testing flakiness of network interfaces
 */

const SPRAYMAX = 8845;/* max amount can spray */

/*
 * GMT since 0:00, 1 January 1970
 */
struct spraytimeval {
   unsigned int sec;
   unsigned int usec;
};

/*
 * spray statistics
```

```
 */
struct spraycumul {
   unsigned int counter;
   spraytimeval clock;
};

/*
 * spray data
 */
typedef opaque sprayarr<SPRAYMAX>;

program SPRAYPROG {
   version SPRAYVERS {
      /*
       * Just throw away the data and increment the counter. This
       * call never returns, so the client should always time it out.
       */
      void
      SPRAYPROC_SPRAY(sprayarr) = 1;

      /*
       * Get the value of the counter and elapsed time since last
       * CLEAR.
       */
      spraycumul
      SPRAYPROC_GET(void) = 2;

      /*
       * Clear the counter and reset the elapsed time
       */
      void
      SPRAYPROC_CLEAR(void) = 3;
   } = 1;
} = 100012;
```

▼ Print Message Program With Remote Version

Code Example F-7 printmesg.c

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

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

```
{
   char *message;

   if (argc != 2) {
      fprintf(stderr, "usage: %s <message>\n", argv[0]);
      exit(1);
   }
   message = argv[1];
   if( !printmessage(message) ) {
      fprintf(stderr, "%s: couldn't print your message\n",
              argv[0]);
      exit(1);
   }
   printf("Message Delivered!\n");
   exit(0);
}

/* Print a message to the console. */

/*
 * Return a boolean indicating whether the message was actually
 * printed.
 */
printmessage(msg)
   char *msg;
{
   FILE *f;

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

Code Example F-8 Remote Version of `printmesg.c`

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

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

```
   CLIENT *cl;
   int *result;
   char *server;
   char *message;
   extern int sys_nerr;
   extern char *sys_errlist[];

   if (argc != 3) {
      fprintf(stderr,"usage: %s host messagen", 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.
 */
   cl = clnt_create(server, MESSAGEPROG, PRINTMESSAGEVERS,
                    "visible");
   if (cl == (CLIENT *)NULL) {
      /*
       * Couldn't establish connection with server.
       * Print error message and die.
       */
      clnt_pcreateerror(server);
      exit(1);
   }
   /* Call the remote procedure "printmessage" on the server */
   result = printmessage_1(&message, cl);
   if (result == (int *)NULL) {
   /*
    * An error occurred while calling the server.
    * Print error message and die.
    */
      clnt_perror(cl, server);
      exit(1);
   }
   /* Okay, we successfully called the remote procedure. */
   if (*result == 0) {
      /*
       * Server was unable to print our message.
       * Print error message and die.
```

```
        */
       fprintf(stderr,"%s"
   }
/* The message got printed on the server's console */
   printf("Message delivered to %s!\n", server);
   exit(0);
}
```

Code Example F-9 `rpcgen` Program: `msg.x`

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

Code Example F-10 `mesg_proc.c`

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

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

/*
 * Remote version of "printmessage"
 */
/*ARGSUSED1*/
int printmessage_1(msg, req)
   char **msg;
   struct svc_req *req;
{
   static int result; /* must be static! */
   FILE *f;

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

```
    fclose(f);
    result = 1;
    return (&result);
}
```

▼ Batched Code Example

Client Side

Code Example F-11 Batched Client Program

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

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

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

    timerclear(&total_timeout);
    while (scanf("%s", s) != EOF) {
        clnt_call(client, RENDERSTRING_BATCHED, xdr_wrapstring,
          &s,xdr_void, (caddr_t) NULL, total_timeout);
    }

    /* Now flush the pipeline */
    total_timeout.tv_sec = 20;
    clnt_stat = clnt_call(client, NULLPROC, xdr_void,
            (caddr_t) NULL, xdr_void, (caddr_t) NULL, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "rpc");
        exit(1);
    }
```

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

Server Side

Code Example F-12 Batched Server Program

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

void            windowdispatch();
main()
{
    int num;

    num = svc_create(windowdispatch, WINDOWPROG, WINDOWVERS,
          "CIRCUIT_V");
    if (num == 0) {
        fprintf(stderr, "can't create an RPC server\n");
        exit(1);
    }
    svc_run();                          /* Never returns */
    fprintf(stderr, "should never reach this point\n");
}

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

    switch (rqstp->rq_proc) {
        case NULLPROC:
            if (!svc_sendreply(transp, xdr_void, 0))
                fprintf(stderr, "can't reply to RPC call\n");
            return;
        case RENDERSTRING:
            if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                fprintf(stderr, "can't decode arguments\n");
                /* Tell caller an error occurred */
                svcerr_decode(transp);
                break;
            }
```

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

▼ Non-Batched Example

This example is included for reference only. It is a version of the batched client string rendering service, Code Example F-11, written in as a non-batched program.

Code Example F-13 Unbatched Version of Batched Client

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

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

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

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

Glossary

RPC Programming Terms

The following terms define the RPC concepts used throughout this manual.

client

A program or system that uses the services of a remote program or system.

client handle

A client process data structure that represents the binding of the client to a particular server's RPC program.

connection-oriented transport

See *stream transport*.

connectionless transport

See *datagram transport*.

datagram transport

Datagram transports have less overhead than connection-oriented transports but are considered less reliable and data transmissions are limited by buffer size.

deserialize

Convert data from XDR format to a machine-specific representation.

handle

An abstraction used by the service libraries to refer to a file or a file-like object such as a socket.

host

A computer (mainframe, mini, server, workstation, or personal computer) connected to a network.

MT hot

An interface is multithreaded hot if the library or call automatically creates threads.

MT safe

An interface is multithreaded safe if it can be called in a threaded environment. An MT-safe interface may be invoked concurrently for multiple threads.

network client

Usually client. A process that makes remote procedure calls to services.

network server

Usually server. A process that performs a network service. A server may support more than one version of a remote program to be forward compatible with changing protocols.

network service

A collection of one or more remote service programs.

`ping`

The `ping` service is used to verify activity on a remote system.

remote program

A program that implements one or more remote procedures.

RPC Language (RPCL)

A C-like programming language translated by the `rpcgen` compiler. RPCL is a superset of XDR Language.

RPC library

`libnsl`, specified to the link editor at compile time. Also known as the RPC package.

RPC protocol

The message-passing protocol that is the basis of the RPC package.

RPC/XDR

See *RPC Language*.

serialize

Converting data from a machine representation to XDR format.

server

A process that provides remote service to clients.

stream transport

Stream transport is considered reliable. It supports byte-stream deliveries of unlimited data size.

transport

The fourth layer of the Open Systems Interconnection (OSI) Reference Model.

transport handle

An abstraction used by the RPC libraries to refer to the transport's data structures.

TI-RPC

Transport-independent RPC. The version of RPC supported in SunOS 5.x.

TS-RPC

Transport-specific RPC. The version of RPC supported in SunOS 4.x. TS-RPC is also supported in SunOS 5.x.

universal address

A machine-independent representation of a transport address.

virtual circuit transport

See *stream transport*.

XDR Language

A data description language and data representation protocol.

Index

D

E

F

G

H

I

K

O

P

R

S

T

U

V

W

X