IRIX® Network Programming Guide

Document Number 007-0810-080

CONTRIBUTORS

Written by Susan Thomas, Jed Hartman, and Judith Radin, updated by Helen Vanderberg
Production by Linda Rae Sande and Carlos Miqueo
Engineering contributions by Dana Treadwell, Paul Fronberg, Carlin Otto, Mary Artibee, Bill Calvert, Andrew Cherenson, and Nelson Bolyard

This document uses material from chapters of the 4.3BSD *Programmer's Supplementary Documents* and from various Internet Request For Comment documents.

IRIX® Network Programming Guide
Document Number 007-0810-080

Contents

List of Examples xv

List of Figures xvii

List of Tables xix

About This Guide xxi
Audience for This Guide xxii
Typographic Conventions xxii
Chapter Summaries xxiii
Documentation Sources xxiv
Additional Reading xxv

1. Network Programming Overview 1
Introduction to IRIX Network Programming 2
The Internet Protocol Suite 2
Compiling BSD and RPC Programs 4

2. Sockets-based Communication 5
Sockets Basics 6
Socket Types 7
Stream Sockets 7
Datagram Sockets 7
Raw Sockets 8

Creating Sockets 8
Binding Local Names to a Socket 10
Establishing Socket Connections 12
Transferring Data 15
Discarding Sockets 17
Connectionless Sockets 17
I/O Multiplexing 19
Network Library Routines 22
Host Names 23
Network Names 24
Protocol Names 24
Service Names 25
Network Dependencies 25
Byte Ordering 27
The Client/Server Model 29
Connection-based Servers 29
Connection-based Clients 32
Connectionless Servers 34
Advanced Topics 38
Out-of-Band Data 38
Nonblocking Sockets 40
Interrupt-driven Sockets I/O 41
Signals and Process Groups 42
Pseudo-Terminals 44
Selecting Protocols 46
Address Binding 46
Socket Options 49
The inetd Daemon 51
Broadcasting 54
IP Multicasting 59
Sending IP Multicast Datagrams 60
Receiving IP Multicast Datagrams 62
Sample Multicast Program 64

3. Introduction to RPC Programming 67
Overview of Remote Procedure Calls 68
The Remote Procedure Call Model 68
RPC Transports and Semantics 70
Binding and Rendezvous Independence 71
RPC Message Identification and Authentication 71
The XDR Standard 72
The Layers of RPC 73
The Highest Layer 73
The Middle Layer 73
The Lowest Layer 74
The rpcgen Protocol Compiler 74
Assigning RPC Program Numbers 75
The Port Mapper Programs 78

4. Programming with rpcgen 79
Introduction to the rpcgen Compiler 80
Changing Local Procedures to Remote Procedures 81
Generating XDR Routines 89
The C Preprocessor 95
pcgen Programming Notes 96
Generating ANSI C Prototypes 96
Client-side Timeout Changes 96
Server-side Broadcast Handling 97
Other Information Passed to Server Procedures 97

5. RPC Programming Guide 103
The Layers of RPC 104
The Highest Layer of RPC 104
The Middle Layer of RPC 105
Passing Arbitrary Data Types 108
The Lowest Layer of RPC 111
More Information about the Server 111
More Information about the Client 114
Memory Allocation with XDR 117

Other RPC Features 118
Select on the Server Side 118
Broadcast RPC 119
Broadcast RPC Synopsis 120
Batching 121
Authentication 124
Client-side Authentication 125
Server-side Authentication 126
Using inetd 132
More Examples 133
Program Version Number 133
TCP 135
Callback Procedures 138

6. XDR and RPC Language Structure 143
XDR Language 144
Notational Conventions 144
Lexical Notes 145
Syntax Information 145
Syntax Notes 147
XDR Data Description Example 147
RPC Language 149
Definitions 149
Structures 149
Unions 150
Enumerations 151
Typedefs 152
Constants 152
Programs 152
Declarations 153
Special Cases 155

7. **XDR Programming Notes** 157
Overview of XDR Programming 158
The XDR Library 162
XDR Library Primitives 165
Number Filters 165
Floating-point Filters 165
Enumeration Filters 166
No Data 166
Constructed Data Type Filters 166
Strings 167
Byte Arrays 168
Arrays 168
Examples of Constructed Data Types 169
Opaque Data 171
Fixed-length Size Arrays 171
Discriminated Unions 172
Pointers 174
Pointer Semantics and XDR 175
Non-filter Primitives 175
XDR Operation Directions 176
XDR Stream Access 176
Standard I/O Streams 177
Memory Streams 177
Record (TCP/IP) Streams 177
XDR Stream Implementation 179
The XDR Object 179
Advanced Topics 181
Linked Lists 181

8. Transport Layer Interface 185
Introduction 186
Network Selection and Name-to-Address Mapping 186
OSI Reference Model 187
Overview of the Transport Interface 190
Modes of Service 191
Connection-Mode Service 192
Local Management 192
Connection Establishment 194
Data Transfer 195
Connection Release 195
Connectionless-Mode Service 196
State Transitions 197
Introduction to Connection-Mode Service 197
Local Management 197
The Client 199
The Server 201
Connection Establishment 204
The Client 205
Event Handling 207
The Server 208
Data Transfer 212
The Client 213
The Server 214
Connection Release 216
The Server 217
The Client 218

Introduction to Connectionless-Mode Service 219
Local Management 219
Data Transfer 222
Datagram Errors 224
A Read/Write Interface 225
write() 227
read() 227
close() 228
Advanced Topics 229
Asynchronous Execution Mode 229
Advanced Programming Example 230
State Transitions 237
Transport Interface States 237
Outgoing Events 237
Incoming Events 239
Transport User Actions 240
State Tables 240
Guidelines for Protocol Independence 243
Some Examples 244
Connection-Mode Client 244
Connection-Mode Server 246
Connectionless-Mode Transaction Server 250
Read/Write Client 252
Event-Driven Server 254
Error Messages 260

A. RPC Protocol Specification 263
RPC Protocol Requirements 264
Remote Programs and Procedures 264
Message Authentication 265
Other Uses of the RPC Protocol 266
Batching 266
Broadcast RPC 266
RPC Protocol Definition 267
Authentication Protocols 271
Null Authentication 271
AUTH_UNIX Authentication 271
Trusted UNIX Systems 272
Record Marking Standard 274
Port Mapper Program Protocol 274
Port Mapper Protocol Specification 275
Port Mapper Operation 277

B. XDR Protocol Specification 279
Basic Block Size 280
Block 280
XDR Data Types 280
Integers 280
Integer 281
Unsigned Integers 281
Unsigned Integer 281
Enumerations 281
Booleans 282
Hyper Integers and Hyper Unsigned 282
Hyper Integer or Unsigned Hyper Integer 282

Floating Points 282
Single-Precision Floating-Point 283
Double-Precision Floating Points 284
Double-Precision Floating-Point 284
Fixed-Length Opaque Data 285
Fixed-Length Opaque 285
Variable-Length Opaque Data 285
Variable-Length Opaque 286
Strings 286
String 287
Fixed-Length Arrays 287
Fixed-Length Array 287
Variable-Length Arrays 287
Counted Array 288
Structures 288
Discriminated Unions 288
Discriminated Union 289
Voids 289
Void 289
Constants 290
Typedefs 290
Optional Data 291
Areas for Future Enhancement 292
Common Questions about XDR 293

C. IRIX Name Service Implementation 295
Overview of UNS 296
UNS Programming Steps 296
UNS Library Routines 297
getXbyY() Routine 297
getXent() Routine 298
ns_lookup() Routine 298
ns_list() Routine 298
UNS Cache Files 299
UNS Name Service Daemon Operation 300
Name Service Configuration Files and Data Structures 300
Understanding the UNS Runtime Loop 302
Understanding UNS Utility Functions 303
How UNS Protocol Libraries Work 307
Library Init Routine 307
Library Lookup Routine 308
Library List Routine 308
Library Dump Routine 308
Library Verify Routine 309
Library Shake Routine 309
Files Callout Library 309
NIS Callout Library(Optional) 310
Nisserv Callout Library (Optional) 311
DNS Callout Library 311
MDBM Callout Library 312
Berkeley DB Callout Library 312
NDBM Callout Library 312
NFS Interface to UNS 313

Index 315

List of Examples

Example 2-1 A Remote-Login Client 26
Example 2-2 Flushing Terminal I/O on Receipt of Out-of-Band Data 39
Example 2-3 Asynchronous Notification of I/O Requests 42
Example 2-4 Using the SIGCHLD Signal 43
Example 2-5 Creating and Using a Pseudo-Terminal on IRIX 45
Example 8-1 The Connection-Mode Client Definitions and Local Management 199
Example 8-2 The Connection-Mode Server Definitions and Local Management 201
Example 8-3 Sending Data to a Client 214
Example 8-4 The Transaction Server Definitions and Local Management 220
Example 8-5 The Data Transfer Phase of a Connectionless-Mode Server 222
Example 8-6 An Advanced Server 230
Example 8-7 Processing an Incoming Event 233
Example 8-8 A Connection-Mode Client 245
Example 8-9 A Connection-Mode Server 246
Example 8-10 A Connectionless-Mode Transaction Server 250
Example 8-11 A Connection-Mode Read/Write Client 252
Example 8-12 A Connection-Mode Server 254

List of Figures

Figure 1-1 BSD Model of Network Layering 3
Figure 3-1 The Remote Procedure Call Model 69
Figure 8-1 OSI Reference Model 187
Figure 8-2 Transport Interface 190
Figure 8-3 Channel between User and Provider 192
Figure 8-4 Transport Connection 194
Figure 8-5 Listening and Responding Transport Endpoints 211

List of Tables

Table 2-1 Common *errno* values 13
Table 2-2 C Run-time Routines 27
Table 2-3 TTL Threshold Convention 60
Table 3-1 Some Registered RPC Programs 76
Table 3-2 RPC Program Number Assignment 77
Table 4-1 C Preprocessor Symbol Definition 95
Table 6-1 DR Data Encoding Examples 148
Table 8-1 Local Management Routines for the Transport Interface 193
Table 8-2 Routines for Establishing a Transport Connection 195
Table 8-3 Connection-Mode Data Transfer Routines 195
Table 8-4 Connection Release Routines 196
Table 8-5 Routines for Connectionless-Mode Data Transfer 196
Table 8-6 States Describing Transport Interface State Transitions 237
Table 8-7 Outgoing Events 238
Table 8-8 Incoming Events 239
Table 8-9 Common Local Management State Table 241
Table 8-10 Connectionless-Mode State Table 241
Table 8-11 Connection-Mode State Table 242
Table A-1 Port Mapper Procedures 277

About This Guide

The *IRIX Network Programming Guide* describes the network programming facilities available with the IRIX operating system.

IRIX implements the Internet Protocol (IP) suite and UNIX domain sockets using the BSD sockets mechanism and supports access to the underlying network media using raw sockets. It also implements the Transport Layer Interface (TLI) defined in ISO-OSI, using SVR4 STREAMS modules. IRIX does not support the Xerox NS protocol suite.

Note: Silicon Graphics does not encourage use of the TLI model; its inclusion is for compatibility with interfaces used by other vendors.

The networking software described in this guide is derived from the BSD UNIX release from the University of California, Berkeley; from the Sun Microsystems Remote Procedure Call (RPC) programming interface; and from UNIX System V, Release 4 from UNIX System Laboratories, Inc.

The *IRIX Network Programming Guide* is for programmers who want to develop network applications using the sockets interface, Sun RPC, or TLI. It explains the fundamental elements of each interface, including the libraries, routines, and other programming tools offered by each interface, and explains how to use them to develop IRIX network applications.

This introduction contains background information that you should read before proceeding. Topics include:

- the audience for this guide
- typographic conventions
- chapter summaries
- documentation sources
- additional reading

Audience for This Guide

This guide is for experienced programmers who intend to write applications that use network interfaces. Knowledge of the UNIX operating system, the C language, and general network theory is assumed.

Typographic Conventions

IRIX manual pages are referred to by name and section number, in this format:

name(sect)

where *name* is the name of a command, system call, or library routine, and *sect* is the section number where the entry resides. For example:

rpc(3R)

refers to the *rpc* manual page in section 3 of the IRIX manual pages (which is divided up into subsections such as 3N and 3R). To look at that manual page, enter the command:

```
% man 3 rpc
```

In syntax descriptions and examples, you will see these type conventions:

Bold is used for function names, system calls, options to commands, and for chapter, section, and table headings.

`Fixed-width` indicates sample code and system output.

`Fixed-width Bold` indicates user input.

Italic indicates IRIX filenames, command names, and arguments to be replaced with a value. Also used to indicate a new term used for the first time.

Chapter Summaries

This guide contains the following chapters:

- Chapter 1, “Network Programming Overview,” provides general information about IRIX network programming.
- Chapter 2, “Sockets-based Communication,” describes the BSD sockets interface.
- Chapter 3, “Introduction to RPC Programming,” provides background information about the RPC interface.
- Chapter 4, “Programming with rpcgen,” describes how to use the *rpcgen* compiler to write RPC applications. This chapter contains the complete source code for a working RPC service.
- Chapter 5, “RPC Programming Guide,” describes the details of the RPC programming interface. If you use *rpcgen*, it isn’t necessary to understand most of the information in this chapter.
- Chapter 6, “XDR and RPC Language Structure,” describes the structure and syntax of the RPC and XDR languages; it shows you how to write program interface definitions using RPC language.
- Chapter 7, “XDR Programming Notes,” contains technical notes about the XDR standard.
- Chapter 8, “Transport Layer Interface,” describes USL’s implementation of the ISO-OSI network interface to the transport layer.
- Appendix A, “RPC Protocol Specification,” describes the underlying details of the RPC protocol specification.
- Appendix B, “XDR Protocol Specification,” describes the underlying details of the XDR protocol specification.
- Appendix C, “IRIX Name Service Implementation,” describes the underlying details of the Unified Name Service.

Documentation Sources

This guide uses material from several sources:

- Deering, S. "Host Extensions for IP Multicasting." *Internet Request For Comment 1112*. Menlo Park, California: Network Information Center, SRI International, August 1989.
- Karels, Michael J., Chris Torek, James M. Bloom, et al. *4.3BSD UNIX System Manager's Manual*. Berkeley, California: University of California.
- Kirkpatrick, S., M. Stahl, and M. Recker. "Internet Numbers." *Internet Request For Comment 1166*. Menlo Park, California: Network Information Center, SRI International, July 1990.
- Leffler, Samuel J., Robert S. Fabry, William N. Joy, et al. "An Advanced 4.3BSD Interprocess Communication Tutorial." *4.3BSD UNIX Programmer's Supplementary Documents*, Volume 1. Berkeley, California: University of California.
- Lottor, M. "TCP Port Service Multiplexer (TCPMUX)." *Internet Request for Comment 1078*. Menlo Park, California: Network Information Center, SRI International, November 1988.
- Reynolds, J., and J. Postel. "Assigned Numbers." *Internet Request for Comment 1060*. Menlo Park, California: Network Information Center, SRI International, March 1990.
- Sechrest, Stuart. "An Introductory 4.3BSD Interprocess Communication Tutorial." *4.3BSD UNIX Programmer's Supplementary Documents*, Volume 1. Berkeley, California: University of California.
- Sun Microsystems. *eXternal Data Representation: Sun Technical Notes* (for RPC 4.0). Mountain View, California: Sun Microsystems, Inc.
- Sun Microsystems. *eXternal Data Representation Standard: Protocol Specification* (for RPC 4.0). Mountain View, California: Sun Microsystems, Inc.
- Sun Microsystems. *Remote Procedure Calls: Protocol Specification* (for RPC 4.0). Mountain View, California: Sun Microsystems, Inc.
- Sun Microsystems. *rpcgen Programming Guide* (for RPC 4.0). Mountain View, California: Sun Microsystems, Inc.
- UNIX System Laboratories. *Programmer's Guide: Networking Interfaces* (for SVR4.1). Englewood Cliffs, New Jersey: Prentice Hall, Inc.

Additional Reading

For additional information, you can consult your online manual pages and these documents:

- *IRIX Admin* manual set (available from SGI as an option).
- Comer, Douglas E. *Internetworking with TCP/IP,* Volume I, Second Edition. Prentice Hall, Inc., Englewood Cliffs, New Jersey (1991).
- Corbin, John R. *The Art of Distributed Applications*. Springer-Verlag, New York (1991).
- Kockan, Stephen G., and Wood, Patrick H., editors. *UNIX Networking*. Hayden Books, Indiana (1989).
- Stevens, W. Richard. *UNIX Network Programming*, Prentice Hall, Inc., Englewood Cliffs, New Jersey (1990).
- Stevens, W. Richard. *TCP/IP Illustrated*, Addison-Wesley Publishing Co.
- Schneier, Bruce. *Applied Cryptography, Second Edition*, John Wiley and Sons, New York (1996).

You can also find related information in Internet *Request For Comment* documents, available by anonymous *ftp* from the */rfc* directory at Government Systems, Inc. (IP number 192.112.36.5). For more information about using *ftp*, see *IRIS Essentials*.

Chapter 1

Network Programming Overview

The network programming facilities available with the IRIX operating system include the BSD sockets library, and the Sun Microsystems Remote Procedure Call (RPC) interface, and ISO-OSI's TLI. The programming interface you use depends on the requirements of the application you plan to develop.

This chapter introduces general concepts related to network programming. Topics include:

- an overview of network programming under IRIX
- a discussion of the Internet Protocol (IP) suite
- comments and caveats on compiling BSD and RPC programs

Introduction to IRIX Network Programming

The BSD program-to-program communication facility provides the *socket* abstraction. The sockets interface enables low-level access to network addressing and data transfer, and it provides the flexibility to accommodate diverse application requirements. The sockets interface also provides greater speed, simpler programming, and a wider base of platforms than TLI.

RPC implements a remote procedure call model, in which a procedure executing on a remote machine can be treated as a local procedure call by the calling application. RPC enables synchronous execution of procedure calls on remote hosts, provides transparent access to network facilities, and uses eXternal Data Representation (XDR) to ensure portability. (See Chapter 3, "Introduction to RPC Programming," for more information.)

The International Standards Organization (ISO) has developed a standard known as the Reference Model of Open Systems Interconnection (abbreviated as ISO-OSI, or the OSI Reference Model, or simply OSI). This model conceives of networking as being divided into seven layers. The interface between the fourth and fifth layers (that is, between the transport layer and the session layer) is known as the Transport Layer Interface (TLI); it provides a set of functions for applications to call to perform various network procedures.

Note: Silicon Graphics does not encourage use of the TLI model; its inclusion is for compatibility with interfaces used by other vendors.

The Internet Protocol Suite

A *protocol* is a set of rules, data formats, and conventions that regulate the transfer of data between participants in the communication.

The IRIX operating system implements the Internet Protocol (IP) suite. The IP suite is a collection of layered protocols developed by the U.S. Department of Defense Advanced Research Projects Agency (DARPA). The two most widely used IP protocols are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

TCP/IP provides a reliable means of transferring data between systems. TCP/IP messages are acknowledged by the receiver as they are received. UDP/IP provides a faster, low-overhead method of transferring data, but the receiver under UDP/IP does not provide the sender with any acknowledgment of messages received.

TCP creates a *virtual circuit*, a data path in which data blocks are guaranteed delivery to a target machine in the correct order. Messages are sent from the sender to the receiver until the receiver sends back a message saying that all the data blocks have been received in the correct order.

By using network applications built on top of the IP suite, you can interactively transfer files between computers, log in to remote computers, execute commands on remote computers, and send mail between users on different machines.

Figure 1-1 illustrates the BSD model of network layering.

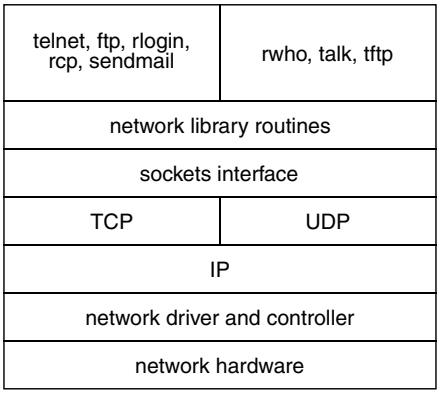


Figure 1-1 BSD Model of Network Layering

Compiling BSD and RPC Programs

Most BSD and RPC programs compile and link under IRIX without change. Some BSD and RPC programs, however, may require compiler options, linking with additional libraries, or even source code modification.

Since most BSD and Sun RPC programs were written before the ANSI C standard, you may need to compile with the **–cckr** command-line option to obtain traditional C semantics. For example:

```
% cc -cckr example.c -o example
```

If your program assumes that the *char* data type is signed, use the **–signed** option (most BSD programs assume signed characters, but the IRIX C compiler assumes unsigned characters by default). For example:

```
% cc -cckr -signed example.c -o example
```

The BSD library routines formerly in */usr/lib/libbsd.a* (in releases before IRIX 3.3) are now in the standard C library, which is linked in by default during compilation.

Note: In previous versions of IRIX, BSD header files were located in the */usr/include/bsd* directory. In more recent versions, such header files are in */usr/include* with the other header files. If your program contains **#include** statements to include header files of the form <*bsd/filename.h*>, you should remove the *bsd/* part of such filenames from the **#include** statement.

Several network library routines have NIS equivalents that used to be in the *libsun* library. In current versions of IRIX, *libsun* has been incorporated into *libc*.

IRIX provides UNIX System V, BSD, and POSIX signal handling mechanisms. BSD signals are obtained with the **–D_BSD_SIGNALS** compiler directive.

Chapter 2

Sockets-based Communication

This chapter describes the BSD sockets-based Inter-Process Communication (IPC) facilities available with the IRIX operating system.

Topics in this chapter include:

- the basic sockets communication model and IPC-related system calls
- network library routines used to construct distributed applications
- the client/server model that is used to develop applications, including examples of the two major types of servers
- advanced topics for sophisticated users, such as UDP/IP broadcasting and multicasting

Sockets Basics

A *socket* is the basic building block for program-to-program communication. A socket is an endpoint of communication to which a name can be bound. Each socket in use has a *type* and one or more associated processes. Sockets are typed according to their communication properties.

Three socket types are available:

- *Stream sockets* provide a bidirectional, reliable, sequenced, and unduplicated flow of message data.
- *Datagram sockets* support bidirectional data flow, but don't guarantee that the message data is sequenced, reliable, or unduplicated.
- *Raw sockets* give you access to the underlying communication protocols that support socket abstractions.

(All three socket types are described in "Socket Types" on page 7.)

The processes associated with a socket communicate through the socket. Sockets are presumed to communicate with sockets of the same type; however, nothing prevents communication between sockets of different types should the underlying communication protocols support it.

Sockets exist within *communication domains*. A domain dictates various properties of the socket. One such property is the scheme used to name sockets. For example, in the UNIX communication domain, sockets are named with UNIX pathnames; a socket, for example, may be named */dev/foo*.

Normally, sockets exchange data within the same domain. It may be possible to cross domain boundaries, but only if some translation process is performed.

The sockets facility supports three communication domains:

- The *UNIX domain* is used only for on-system communication.
- The *Internet domain* is used by processes that communicate using the Internet standard communication protocols IP/TCP/UDP.
- The *Raw domain* provides access to the link-level protocols of network interfaces (unique to IRIX).

The underlying communication facilities provided by each domain significantly influence the interface to the sockets facilities available to users, providing protocol-specific socket properties that may be set or changed by the user. For example, a socket operating in the UNIX domain can see a subset of the error conditions that are possible when operating in the Internet domain.

In general, there is one protocol for each socket type within each domain. The code that implements a protocol keeps track of the names that are bound to sockets, sets up connections, and transfers data between sockets, perhaps sending the data across a network. It is possible for several protocols, differing only in low-level details, to implement the same style of communication within a particular domain. Although it is possible to select which protocol should be used, for nearly all uses it is sufficient to request the default protocol.

Socket Types

This section describes the three socket types: stream sockets, datagram sockets, and raw sockets.

Stream Sockets

A *stream* socket provides a bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow and some additional signaling facilities, a pair of connected stream sockets provides an interface similar to that of a pipe. (In the UNIX domain, in fact, the semantics are identical.)

Note: Stream sockets should not be confused with STREAMS, the modularized driver interface on which TLI is built.

Datagram Sockets

A *datagram* socket supports the bidirectional flow of messages that are not necessarily sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket can find messages duplicated or in a different order. The data in any single message, however, is in the correct order, with no duplications, deletions, or changes.

An important characteristic of a datagram socket is that record boundaries in the data are preserved. Datagram sockets closely model facilities found in many packet-switched networks. However, datagram sockets provide additional facilities, including routing and fragmentation.

Routing is used to forward messages from one local network to another nearby or distant network. Dividing one large network into several smaller ones can improve network performance in each smaller network, improve security, and facilitate administration and troubleshooting.

Fragmentation divides large messages into pieces small enough to fit on the local medium. It allows application programs to use a single message size independent of the packet size limitations of the underlying networks.

Raw Sockets

A *raw* socket provides access to the underlying communication protocols that support socket abstractions. Raw sockets are normally datagram-oriented, though their exact characteristics depend on the interface provided by the protocol.

Raw sockets are not intended for the general user. They are provided for programmers interested in developing new communication protocols or for gaining access to some of the more esoteric facilities of an existing protocol.

Creating Sockets

To create a socket, use the **socket()** system call (see socket(2)):

```
#include <sys/types.h>
#include <sys/socket.h>
s = socket(domain, type, protocol);
```

This call creates a socket in the specified *domain*, of the specified *type*, using the specified *protocol*, and returns a descriptor (a small integer) that can be used in later system calls operating on sockets.

If *protocol* is not specified (a 0 value is given), a default protocol is used. The system selects from the protocols that make up the communication domain and that can be used to support the requested socket type.

The domain is specified as one of the manifest constants defined in the file <*sys/socket.h*>:

AF_UNIX	UNIX domain
AF_INET	Internet domain
AF_RAW	Raw domain

Note: AF indicates the *address family* (or format) to use in interpreting names.

The socket types are also defined in <*sys/socket.h*>, as SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW.

For example, to create a stream socket in the Internet domain, you could use this call:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This creates a stream socket in which underlying communication support is provided by the default protocol, TCP.

The default protocol should be correct for most situations. However, you can specify other protocols; see "Selecting Protocols" on page 46 for details.

To create a datagram socket for same-machine use, the call might be:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

To create a *drain* socket, which receives all packets that have a network-layer type-code or encapsulation not implemented by the kernel, use this call:

```
#include <net/raw.h>
s = socket(AF_RAW, SOCK_RAW, RAWPROTO_DRAIN);
```

For details about raw domain sockets, see the manual pages for raw(7F), snoop(7P), and drain(7P).

A **socket()** call can fail for several reasons, each of which sets the *errno* variable appropriately. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request can fail in response to a request for an unknown protocol (EPROTONOSUPPORT) or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

Binding Local Names to a Socket

A socket is created without a name. Until a name is bound to the socket, processes have no way to reference it, and, consequently, no messages can be received on it.

Communicating processes are bound by an association. An *association* is a temporary or permanent specification of a pair of communicating sockets.

In the Internet domain, an association is composed of local and foreign addresses, and local and foreign ports. The structure of Internet domain addresses is defined in the file <*netinet/in.h*>.

Internet addresses specify a host address (a 32-bit number) and a delivery slot, or port, on that machine. These ports are managed by the system routines that implement a particular protocol. Unlike UNIX domain socket names, Internet domain socket names are not entered into the filesystem and, therefore, do not have to be unlinked after the socket is closed.

When a message is exchanged between machines, it is first sent to the protocol routine on the destination machine. This routine interprets the address to determine to which socket the message should be delivered. Several different protocols may be active on the same machine, but, in general, they will not communicate with one another. As a result, different protocols are allowed to use the same port numbers. Thus, an Internet address is a triple address, including a protocol, the port, and the machine address.

An Internet association is identified by the tuple <*protocol, local address, local port, remote address, remote port*>. Duplicate tuples are not allowed. An association may be transient when using datagram sockets; the association actually exists during a **send()** operation.

In the UNIX domain, an association is composed of local and foreign pathnames (a *foreign pathname* is a pathname created by a foreign process, not a pathname on a foreign system). UNIX domain sockets need not always be bound to a name, but when they are bound, there may never be duplicate <*protocol, local pathname, foreign pathname*> tuples.

The pathnames may not refer to files already existing on the system. Like pathnames for normal files, they may be either absolute (for example, */dev/imaginary*) or relative (for example, *socket*). Because these names are used to allow processes to rendezvous, relative pathnames can pose difficulties and should be used with care.

When a name is bound into the name space, a file (inode) is allocated in the filesystem. If the inode is not deallocated, the name will continue to exist even after the bound socket is closed. This situation can cause subsequent runs of a program to find a name unavailable and can cause directories to fill up with these objects. You can remove names by calling **unlink()** (see unlink(2)) or by using the *rm* command.

Names in the UNIX domain are used only for rendezvous; they are not used for message delivery once a connection is established. Therefore, in contrast to the Internet domain, unbound sockets are not, and need not be, automatically given addresses when they are connected.

The **bind()** system call (see bind(2)) allows a process to specify half of an association, <*local address*, *local port*> (or <*local pathname*>), while the **connect()** and **accept()** system calls are used to complete a stream socket's association.

The form of the **bind()** system call is:

```
bind(s, name, namelen);
```

The bound name is a variable-length byte string that is interpreted by the supporting protocol(s). The interpretation of the bound name may vary from communication domain to communication domain (this is one of the properties that make up the domain).

In the UNIX domain, names contain a pathname and a family, which is always AF_UNIX. The following code fragment binds the name */tmp/foo* to a UNIX domain 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 a UNIX domain address, null bytes are not counted, which is why **strlen()** is used.

Note: In the current implementation of UNIX domain IPC under IRIX, the filename referred to in *addr.sun_path* is created as a socket in the system's file space. The caller must, therefore, have write permission in the directory where *addr.sun_path* is to reside, and this file should be deleted by the caller when it is no longer needed using the **unlink()** system call (see unlink(2)). Future versions of IRIX may not create this file.

In the Internet domain, binding names to sockets can be fairly complex. Fortunately, it usually isn't necessary to specifically bind an address and port number to a socket, because the **connect()** and **send()** calls automatically bind an appropriate address if they are used with an unbound socket. To bind an Internet address, use the **bind()** system call like this:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
bind(s, (struct sockaddr *)&sin, sizeof(sin));
```

Note: Selecting what to place in the address *sin* requires some discussion. See "Network Library Routines" on page 22 for information about formulating Internet addresses and the library routines used in name resolution.

Establishing Socket Connections

Stream socket connections are usually established asymmetrically, with one process a client and the other a server. When it offers its advertised services, the server binds a socket to a well-known address associated with the service and then passively listens on its socket. It is then possible for an unrelated process to rendezvous with the server.

Note: For details about datagram sockets, see "Connectionless Sockets" on page 17.

The client requests services from the server by initiating a connection to the server's socket. On the client side, the **connect()** call is used to initiate a connection. Using the UNIX domain, this might appear as:

```
struct sockaddr_un server;
 ...
connect(s, (struct sockaddr *)&server,
    strlen(server.sun_path) +
    sizeof(server.sun_family));
```

Using the Internet domain, this might appear as:

```
struct sockaddr_in server;
 ...
connect(s, (struct sockaddr *)&server, sizeof(server));
```

In the preceding examples, *server* contains either the UNIX pathname or the Internet address and port number of the server to contact. If the client process's socket is unbound at the time of the **connect()** call, the system will automatically select and bind a name to the socket if necessary. This is the way local addresses are usually bound to a socket.

The **connect()** call returns an error if the connection attempt was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server, and data transfer can begin.

When a connection attempt fails, an error is returned and the global variable *errno* is set to indicate the error. Table 2-1 lists some of the more common *errno* values.

Table 2-1 Common *errno* values

Value	Explanation
ETIMEDOUT	This error indicates that after failing to establish a connection for a period of time, the system stopped trying. It usually occurs because the destination host is down or because problems in the network resulted in lost transmissions.
ECONNREFUSED	This error indicates that the host has refused service. It usually occurs because a server process is not present at the requested port on the host. It may also indicate an explicit refusal due to access control.
EHOSTDOWN, ENETDOWN	These errors describe status information delivered to the client host by the underlying communication services.
EHOSTUNREACH, ENETUNREACH	These errors can occur either because the network or host is unknown (no route to the network or host is present) or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to determine if a network or host is down, in which case the system indicates that the entire network is unreachable.

To receive a client's connection, the server must perform two steps after binding its socket: it indicates that it is ready to listen for incoming connection requests, and then it accepts the connection.

To indicate that a socket is ready to listen for incoming connection requests, use the **listen()** call (see listen(2)):

```
listen(s, 5);
```

The second parameter of the **listen()** call specifies the maximum number of outstanding connections that can be queued awaiting acceptance by the server process; this number is limited by the system and is a value that is intended to catch flagrant abuses of system resources. If a connection is requested while the queue is full, the connection is not refused, but the individual messages that make up the request are ignored. This gives a busy server time to make room in its pending connection queue while the client retries the connection request. If the connection returns with the ECONNREFUSED error, the client will be unable to determine whether the server is up.

It is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the **listen()** call is limited to a very large value, (currently 1000). Applications should limit the backlog parameter to a value consistent with a server's usage.

With a socket marked as listening, a server can accept a connection by using the **accept()** system call (see accept(2)):

```
struct sockaddr_in from;
int fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

Note: For the UNIX domain, *from* would be declared as a *struct sockaddr_un*, but the rest of this example would remain the same. The examples that follow describe only Internet-domain routines.

A new descriptor is returned on receipt of a connection (along with a new socket). To identify the client, a server can supply a buffer for the client socket's name. The server initializes the value-result parameter *fromlen* to indicate how much space is associated with *from*. The parameter is then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter can be a null pointer.

The **accept()** call normally blocks. That is, **accept()** will not return until a connection is available or the system call is interrupted by a signal to the program. Furthermore, a program cannot indicate it will accept connections from only a specific individual or individuals. It is up to the program to consider whom the connection is from and close down the connection if it does not wish to speak to the remote program. If the server program wants to accept connections on more than one socket, or wants to avoid blocking on the accept call, there are alternatives; see "The Client/Server Model" on page 29 for details.

Transferring Data

IRIX has several system calls for reading and writing information. The simplest calls are **read()** and **write()** (see read(2) and write(2)). They take as arguments a descriptor, a pointer to a buffer containing the data, and the size of the data:

```
char buf [100];
 ...
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

The descriptor may indicate a file or a connected socket. "Connected" can mean either a connected stream socket or a datagram socket for which a **connect()** call has provided a default destination. The **write()** call requires a connected socket, since no destination is specified in the parameters of the system call. The **read()** call can be used for either a connected or an unconnected socket. These calls are, therefore, quite flexible and may be used to write applications that do not require assumptions about the source of their input or the destination of their output.

The **readv()** and **writev()** calls (see read(3) and write(3)) (for read and write *vector*) are variations of the **read()** and **write()** calls, which allow the source and destination of the input and output to use several separate buffers, while retaining the flexibility to handle both files and sockets.

Sometimes it's necessary to send high-priority data over a connection that may have unread low-priority data at the other end. For example, a user interface process may be interpreting commands and sending them on to another process through a stream connection. The user interface may have filled the stream with as-yet-unprocessed requests when the user types a command to cancel all outstanding requests. Rather than have the high-priority data wait to be processed after the low-priority data, it is possible to send it as *out-of-band* (OOB) data. OOB data is specific to stream sockets and is discussed in "Out-of-Band Data" on page 38.

The **send()** and **recv()** calls (see send(2) and recv(2)) are similar to **read()** and **write()**, but they allow options, including sending and receiving OOB information:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

These calls are used only with connected sockets; specifying a descriptor for a file will result in an error.

While **send()** and **recv()** are virtually identical to **write()** and **read()**, the addition of the *flags* argument is important. The flags are defined in *<sys/socket.h>* and can have nonzero values if one or more of the following are required:

MSG_PEEK look at data without reading

MSG_OOB send/receive out-of-band data

MSG_DONTROUTE
send data without routing packets

To preview data, specify MSG_PEEK with a **recv()** call. The **recv()** call allows a process to read data without removing the data from the stream. That is, the next **read()** or **recv()** call applied to the socket will return the data previously previewed.

One use of this facility is to read ahead in a stream to determine the size of the next item to be read. The option to have data sent in outgoing packets without routing is used only by the routing table management process.

To send datagrams, one must be allowed to specify the destination. The call **sendto()** (see sendto(2)) takes a destination address as an argument and is therefore used for sending datagrams. The call **recvfrom()** (see recvfrom(2)) is often used to read datagrams, since this call returns the address of the sender, if it is available, along with the data. If the identity of the sender does not matter, one may use **read()** or **recv()**.

Finally, there is a pair of calls that allow you to send and receive messages from multiple buffers (the sender must specify the address of the recipient). These are **sendmsg()** and **recvmsg()** (see sendmsg(2) and recvmsg(2)). These calls are actually quite general and have other uses, including, in the UNIX domain, the transmission of a file descriptor from one process to another.

Discarding Sockets

A socket is discarded by closing the descriptor; use the **close()** system call (see close(2)):

```
close(s);
```

If data is associated with a socket that promises reliable delivery (for example, a stream socket) when a close takes place, the system will continue trying to transfer the data. However, after a period of time, undelivered data is discarded. Should you have no use for any pending data, perform a **shutdown()** on the socket prior to closing it:

```
shutdown(s, how);
```

The value *how* is 0 if you do not want to read data, 1 if no more data will be sent, or 2 if no data is to be sent or received.

Connectionless Sockets

The sockets described so far follow a connection-oriented model. However, connectionless interactions, typical of the datagram facilities found in contemporary packet-switched networks, are also supported. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as described in "Creating Sockets" on page 8. If a particular local address is needed, the **bind()** operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data is first sent.

To send data, use the **sendto()** system call:

```
sendto(s, buf, buflen, flags, (struct sockaddr *)&to,
       sizeof(to));
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as described for the **send()** call (see "Transferring Data" on page 15). The *to* value indicates the destination address. On an unreliable datagram interface, errors probably will not be reported to the sender. When information is present locally to recognize a message that cannot be delivered (for instance when a network is unreachable), the call will return –1 and the global variable *errno* will contain an error number.

To receive messages on an unconnected datagram socket, use the **recvfrom()** call:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&from,
         &fromlen);
```

Once again, the value-result parameter, *fromlen*, initially contains the size of the *from* buffer and is modified on return to indicate the actual size of the address from which the datagram was received. If you don't care who the sender is, use 0 for the *&from* and *&fromlen* parameters.

In addition to **sendto()** and **recvfrom()**, datagram sockets can use the **connect()** call to associate a socket with a specific destination address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user.

Only one connected address is permitted for each socket at a time; a second **connect()** will change the destination address, and a **connect()** to a "null" address (family AF_UNSPEC) will cause a disconnection.

Connection requests on datagram sockets return immediately, because the request simply results in the system recording the peer's address. Connection requests on a stream socket, however, do not return immediately; the request initiates the establishment of an end-to-end connection. (The **accept()** and **listen()** calls are not used with datagram sockets.)

While a datagram socket is connected, errors from recent **send()** calls can be returned asynchronously. These errors can be reported on subsequent operations on the socket or by using a special socket option, SO_ERROR, with **getsockopt()** that can be used to interrogate the error status. A **select()** for reading or writing will return true when an error indication has been received. The next operation will return the error, and the error status is cleared. For additional details about datagram sockets, see "Advanced Topics" on page 38.

I/O Multiplexing

You can multiplex I/O requests among multiple sockets and/or files by using the **select()** call:

```
#include <sys/time.h>
#include <sys/types.h>
 ...
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

The **select()** call takes three sets of pointers as arguments:

- one for the set of file descriptors on which the caller wants to read data
- one for the set of file descriptors on which data is to be written
- one for which exceptional conditions are pending (out-of-band data is the only exceptional condition currently implemented)

If you are not interested in certain conditions (that is, read, write, or exceptions), the corresponding argument to the **select()** call should be a null pointer.

Each set is a structure containing an array of long integer bit masks. The size of the array is set by the definition FD_SETSIZE. The array must be long enough to hold one bit for each FD_SETSIZE file descriptor.

The set should be zeroed before use. To clear the set mask, use this macro:

```
FD_ZERO(&mask)
```

To add and remove the file descriptor *fd* in the set mask, use these macros:

```
FD_SET(fd, &mask)
FD_CLR(fd, &mask)
```

The parameter *nfds* in the **select()** call specifies the range of file descriptors (one plus the value of the largest descriptor) to be examined in a set.

You can specify a timeout value if the selection will not last more than a predetermined period of time. If the fields in *timeout* are set to 0, the selection takes the form of a poll, returning immediately. If *timeout* is a null pointer, the selection will block indefinitely. To be more specific, a return takes place only when a descriptor is selectable or when a signal is received by the caller, interrupting the system call.

The **select()** call normally returns the number of file descriptors selected. If the **select()** call returns because the timeout expires, the value 0 is returned. If the **select()** call terminates because of an error or interruption, –1 is returned with the error number in *errno*, and with the file descriptor masks unchanged.

For a successful return, the three sets will indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a file descriptor in a select mask can be tested with this macro:

```
FD_ISSET(fd, &mask)
```

This macro returns a nonzero value if *fd* is a member of the set *mask*, and 0 if it is not.

To check for read readiness on a socket to be used with an **accept()** call, use **select()** followed by the FD_ISSET(*fd*, *&mask*) macro. If FD_ISSET returns a nonzero value, indicating permission to read, then a connection is pending on the socket.

For example, to read data from two sockets, *s1* and *s2*, as the data becomes available and with a one-second timeout, this code might be used:

```
#include <sys/time.h>
#include <sys/types.h>
 ...
fd_set read_template; struct timeval wait;
 ...
for (;;) {
    wait.tv_sec = 1;        /* one second */
    wait.tv_usec = 0;
    FD_ZERO(&read_template);
    FD_SET(s1, &read_template);
    FD_SET(s2, &read_template);
    nb = select(FD_SETSIZE, &read_template, (fd_set *) 0,
                (fd_set *) 0, &wait);
    if (nb <= 0) {
        /*
         *  An error occurred during the select, or
         *  the select timed out.
         */
    }
    if (FD_ISSET(s1, &read_template)) {
        /* Socket #1 is ready to be read from. */
    }
    if (FD_ISSET(s2, &read_template)) {
        /* Socket #2 is ready to be read from. */
    }
}
```

Note: In 4.2BSD, the arguments to **select()** were pointers to integers instead of pointers to *fd_set*s. This type of call will still work, as long as the largest file descriptor is numerically less than the number of bits in an integer (that is, 32). However, the methods illustrated above should be used in all current programs.

The **select()** call provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions are possible through the use of the SIGIO and SIGURG signals.

Network Library Routines

When you use the IPC facilities in a distributed environment, programs need to locate and construct network addresses. This section discusses the library routines you can use to manipulate Internet network addresses.

Locating a service on a remote host requires many levels of mapping before client and server can communicate. A service is assigned a name, such as *login server*, that humans can easily understand. This name, and the name of the peer host, must then be translated into network addresses. Finally, the address is used to locate a physical location and route to the service.

The specifics of these mappings can vary among network architectures. For instance, it is desirable that a network not require a host to have a name indicating its physical location to a client host. Instead, underlying services in the network can discover the actual location of the host at the time the client host wants to communicate.

This ability to have hosts named in a location-independent manner can induce overhead in connection establishment, as a discovery process must take place, but it allows a host to be physically mobile. The host does not have to notify its clients of its current location.

Standard routines are provided for these mappings:

- host names to network addresses
- network names to network numbers
- protocol names to protocol numbers
- service names to port numbers

Routines also indicate the appropriate protocol to use to communicate with the server process. The file <*netdb.h*> must be included when using any of these routines.

Host Names

The *hostent* data structure provides Internet host Name-to-Address Mapping:

```
struct    hostent {
    char  *h_name;       /* official name of host */
    char  **h_aliases;   /* alias list */
    int   h_addrtype;    /* host address type (eg AF_INET) */
    int   h_length;      /* length of address */
    char  **h_addr_list; /* list of addresses, */
                         /* null-terminated */
};
/* first address, in network byte order, for backward
 * compatibility: */
#define h_addr    h_addr_list[0]
```

The routine **gethostbyname()** takes an Internet host name and returns a *hostent* structure, while the routine **gethostbyaddr()** maps Internet host addresses into a *hostent* structure (see gethostbyname(3N) and gethostbyaddr(3N)).

These routines return the official name of the host and its public aliases, along with the address type (family) and a null-terminated list of variable-length addresses. The list of addresses is required because a host may have many addresses and the same name. The *h_addr* definition is provided for backward compatibility and is defined as the first address in the list of addresses in the *hostent* structure.

The database for these calls is provided either by the */etc/hosts* file—see hosts(4)—or by using the Internet domain name server, *named* (see named(1M)). The database can also come from the NIS, if you have the NFS option. Because of the differences in these databases and their access protocols, the information returned can differ. When using the host table or NIS versions of **gethostbyname()**, the call returns only one address but includes all listed aliases. When using the name server version, the calls can return alternate addresses, but they will not provide any aliases other than the one given as the argument.

Network Names

The *netent* data structure defines the Network-Name-to-Network-Number Mapping used with the **getnetbyname()**, **getnetbynumber()**, and **getnetent()** routines (see getnetbyname(3N), getnetbynumber(3N), and getnetent(3N)):

```
/*
 * Assumption here is 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;       /* network number, host byte order */
};
```

These routines are the network counterparts to the host routines described in the preceding section. The routines extract their information from */etc/networks* or from the NIS if the NFS option is installed.

Protocol Names

The *protoent* data structure defines the protocol Name-to-Number Mapping used with the routines **getprotobyname()**, **getprotobynumber()**, and **getprotoent()** (see getprotobyname(3N), getprotobynumber(3N), and getprotoent(3N)):

```
struct   protoent {
    char  *p_name;      /* official protocol name */
    char  **p_aliases;  /* alias list */
    int   p_proto;      /* protocol number */
};
```

The routines extract their information from */etc/protocols* or from the NIS if the NFS option is installed.

Service Names

A service is expected to reside at a specific port and employ a particular communication protocol. This view is consistent with the Internet domain but is inconsistent with other network architectures. Furthermore, a service can reside on multiple ports. If it does, the higher-level library routines will have to be bypassed or extended. Services available are obtained from the file */etc/services* or from the NIS if the NFS option is installed.

The *servent* structure defines the service Name-to-Port-Number Mapping:

```
struct   servent {
    char  *s_name;     /* official service name */
    char  **s_aliases; /* alias list */
    int   s_port;      /* port #, network byte order */
    char  *s_proto;    /* protocol to use */
};
```

The routine **getservbyname()** (see getservbyname(3N)) maps service names to a *servent* structure by specifying a service name and, optionally, a qualifying protocol.

The following returns the service specification for a TELNET server using any protocol:

```
sp = getservbyname("telnet", (char *) 0);
```

This returns only the TELNET server that uses the TCP protocol:

```
sp = getservbyname("telnet", "tcp");
```

The routines **getservbyport()** and **getservent()** also provide service mappings (see getservbyport(3N) and getservent(3N)). The **getservbyport()** routine has an interface similar to that provided by getservbyname—you specify an optional protocol name to qualify lookups.

Network Dependencies

With the support routines already described, an Internet application program rarely has to deal directly with addresses. This allows services to operate as much as possible in a network-independent fashion. However, purging all network dependencies is difficult. As long as the user must supply network addresses when naming services and sockets, some network dependency is required in a program. For example, the normal code included in client programs, such as the remote login program, takes the form shown in Example 2-1:

Example 2-1 A Remote-Login Client

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    struct sockaddr_in server;
    struct servent *sp;
    struct hostent *hp;
    int s;
    ...
    sp = getservbyname("login", "tcp");
    if (sp == NULL) {
        fprintf(stderr,
                "rlogin: tcp/login: unknown service\n");
        exit(1);
    }
    hp = gethostbyname(argv[1]);
    if (hp == NULL) {
        fprintf(stderr,
                "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
    }
    bzero((char *)&server, sizeof (server));
    bcopy(hp->h_addr, (char *)&server.sin_addr,
          hp->h_length);
    server.sin_family = hp->h_addrtype;
    server.sin_port = sp->s_port;
    s = socket(hp->h_addrtype, SOCK_STREAM, 0);
    if (s < 0) {
        perror("rlogin: socket");
        exit(3);
    }
    ...
    /* Connect does the bind() for us */
    if (connect(s, (struct sockaddr *)&server,
        sizeof (server)) < 0) {
        perror("rlogin: connect");
        exit(4);
    }
}
```

Note: To make the remote login program independent of the Internet protocols and addressing scheme, the program would have to have a layer of routines that masked the network-dependent aspects from the mainstream login code. For the current facilities available in the system, this does not appear worthwhile.

Byte Ordering

In addition to the address-related database routines, several other routines are available to simplify manipulation of names and addresses. Table 2-2 summarizes the routines that manipulate variable-length byte strings and handle byte swapping of network addresses and values.

Table 2-2 C Run-time Routines

Call	Synopsis
bcmp(s1, s2, n)	Compare byte strings; 0 if same, not 0 otherwise.
bcopy(s1, s2, n)	Copy *n* bytes from *s1* to *s2*.
bzero(base, n)	Zero-fill *n* bytes starting at base.
htonl(val)	(*host-to-network-long*) Convert 32-bit quantity from host to network byte order.
htons(val)	(*host-to-network-short*) Convert 16-bit quantity from host to network byte order.
ntohl(val)	(*network-to-host-long*) Convert 32-bit quantity from network to host byte order.
ntohs(val)	(*network-to-host-short*) Convert 16-bit quantity from network to host byte order.

The format of the socket address is specified, in part, by standards within the Internet domain. The specification includes the order of the bytes in the address (called the network byte order). Addresses supplied to system calls must be in network byte order; values returned by the system also have this ordering. Because machines differ in the internal representation of integers, examining an address as returned by **getsockname()** or **getservbyname()** (see getsockname(2) or getservbyname(3N)) may result in a misinterpretation. To use the number, it is necessary to call the routine **ntohs()** to convert the number from the network representation to the host's representation. For example:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines that have "big-endian" byte ordering, such as the IRIS, the *ntohs* is a null operation. On machines with "little-endian" ordering, such as the VAX™, this results in a swapping of bytes. Another routine exists to convert a short integer from the host format to the network format, called **htons()**; the **ntohl()** and **htonl()** routines exist for long integers. Any protocol that transfers integer data between machines with different byte orders should use these routines. The library routines that return network addresses and ports provide them in network order so that they can simply be copied into the structures provided to the system.

The Client/Server Model

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme, client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server. (See "Establishing Socket Connections" on page 12 for details.) This section examines the interactions between client and server, and considers some of the problems in developing client and server applications.

The client and server require a well-known set of conventions before service can be rendered (and accepted). This set of conventions constitutes a protocol that must be implemented at both ends of a connection. The protocol can be symmetric or asymmetric. In a symmetric protocol, either side can play the master or slave role. In an asymmetric protocol, one side is always the master, and the other is the slave. An example of a symmetric protocol is TELNET, which is used in Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet File Transfer Protocol (FTP). Regardless of whether the protocol is symmetric or asymmetric, when it accesses a service there is a "server process" and a "client process."

A server process normally listens at a well-known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such a time the server process "wakes up" and services the client, performing actions the client requests.

Alternative schemes that use a server to provide a service can eliminate a flock of server processes clogging the system while remaining dormant most of the time. For Internet servers in BSD-based systems, this scheme has been implemented via *inetd,* the so-called "Internet super-server." The *inetd* daemon listens at a variety of ports, determined at startup by reading a configuration file. When a connection is requested to a port on which *inetd* is listening, *inetd* executes the appropriate server program to handle the client. With this method, clients are unaware that an intermediary such as *inetd* has played any part in the connection. The *inetd* daemon is described in more detail in "Advanced Topics" on page 38.

Connection-based Servers

Most servers are accessed at well-known Internet addresses. The remote login server's main loop takes the form shown in this sample code:

```
main(int argc, char **argv)
{
    int f;
    struct sockaddr_in from;
    struct servent *sp;

    sp = getservbyname("login", "tcp");
    if (sp == NULL) {
        fprintf(stderr,
            "rlogind: tcp/login: unknown service\n");
        exit(1);
    }
    ...
#ifndef DEBUG
    /* Disassociate server from controlling terminal */
    ...
#endif
    /* Restricted port -- see "Address Binding" */
    from.sin_port = sp->s_port;
    ...
    f = socket(AF_INET, SOCK_STREAM, 0);
    ...
    if (bind(f, (struct sockaddr *) &from,
             sizeof (from)) < 0) {
        syslog(LOG_ERR, "rlogind: bind: %m");
        exit(1);
    }
    ...
    listen(f, 5);
    for (;;) {
        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;
        }
        if (fork() == 0) {            /* child */
            close(f);
            doit(g, &from);
        }
        close(g);                /* parent */
    }
}
```

The first step taken by the server is to look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
    fprintf(stderr, "rlogind: tcp/login: unknown service\n");
    exit(1);
}
```

The result of the **getservbyname()** call is used in later portions of the code to define the well-known Internet port where the server listens for service requests (indicated by a connection).

The second step taken by the server is to disassociate from the controlling terminal of its invoker:

```
_daemonize(0, -1, -1, -1);
```

The **_daemonize()** function does the common work needed to put a program into the background or to make a program into a daemon. This generally includes **fork()**ing a new process, closing most files, and releasing the controlling terminal. See the daemonize(3) manual page for details.

The server is protected from receiving signals delivered to the process group of the controlling terminal. Note, however, that once a server has disassociated itself, it can no longer send reports of errors to a terminal and must log errors via **syslog()**.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The **bind()** call is required to ensure that the server listens at its expected location. Note that the remote login server listens at a restricted port number and must therefore be run with a user ID of *root*. This concept of a "restricted port number" is specific to BSD-based systems; see "Address Binding" on page 46 for more information.

The main body of the loop is shown in this example:

```
for (;;) {
    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;
    }
    if (fork() == 0) {    /* Child */
        close(f);
        doit(g, &from);
    }
    close(g);             /* Parent */
}
```

An **accept()** call blocks the server until a client requests service. This call could return a failure status if interrupted by a signal such as SIGCHLD. Therefore, the return value from **accept()** is checked to ensure that a connection has actually been established, and an error report is logged via **syslog()** if an error has occurred.

With a connection established, the server then **fork()**s a child process and invokes the main body of the remote login protocol processing. Note that the socket used by the parent for queuing connection requests is closed in the child, while the socket created as a result of **accept()** is closed in the parent. The address of the client is also handed to the **doit()** routine, because the routine requires it in authenticating clients.

Connection-based Clients

The client side of the remote login service was described in “Network Dependencies” on page 25. The separate, asymmetric roles of the client and server show clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Consider the steps taken by the client remote login process. As in the server process, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
    fprintf(stderr, "rlogin: tcp/login: unknown service\n");
    exit(1);
}
```

Then the **gethostbyname()** call looks up the destination host:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
    fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
    exit(2);
}
```

Next, a connection is established to the server at the requested host and the remote login protocol is started. The address buffer is cleared and is then filled in with the Internet address of the remote host and the port number of the login process on the remote host:

```
bzero((char *)&server, sizeof (server));
bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created and a connection is initiated:

```
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);
}
```

Note that **connect()** implicitly performs a **bind()** call in this case, since *s* is unbound.

Connectionless Servers

While connection-based services are the norm, some services are based on the use of datagram sockets. The *rwho* service is an example. It provides users with status information for hosts connected to a local area network. This service is predicated on the ability to broadcast information to all hosts connected to a particular network.

A user on any machine running the *rwho* server can find out the current status of a machine with the *ruptime* program (see ruptime(1C)). For example, *ruptime* might generate this output:

```
dali      up    2+06:28,    9 users, load 1.04, 1.20, 1.65
breton    down  0:24
manray    up    3+06:18,    0 users, load 0.03, 0.03, 0.05
magritte  up    1+00:43,    2 users, load 0.22, 0.09, 0.07
```

Status information for each host is periodically broadcast by *rwho* server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

The use of broadcast for such a task is inefficient, as all hosts must process each message, whether or not they are using an *rwho* server. Unless such a service is sufficiently universal and frequently used, the expense of periodic broadcasts outweighs the simplicity. However, on a very small network (for example, one dedicated to a computation engine and several display engines) broadcast works well because all services are universal.

Note: Multicasting reduces the load on host machines and is an alternative to broadcasting. Setting up multicast sockets is described in “IP Multicasting” on page 59.

The *rwho* server, in a simplified form, is shown in this code sample:

```
main()
{
    ...
    sp = getservbyname("who", "udp");
    from.sin_addr.s_addr = htonl(INADDR_ANY);
    from.sin_port = sp->s_port;
    ...
    s = socket(AF_INET, SOCK_DGRAM, 0);
    ...
    on = 1;
    if (setsockopt(s, SOL_SOCKET, SO_BROADCAST,
                  &on, sizeof(on)) < 0) {
        syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
        exit(1);
    }
    bind(s, (struct sockaddr *)&from, sizeof (from));
    ...
    signal(SIGALRM, onalrm);
    onalrm();
    for (;;) {
        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 < 0 && 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: malformed 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);
            ...
            /*undo header byte swapping before writing to file*/
            wd.wd_sendtime = ntohl(wd.wd_sendtime);
            ...
            (void) time(&wd.wd_recvtime);
            (void) write(whod, (char *)&wd, cc);
            (void) close(whod);
        }
}
```

The server performs two separate tasks. The first task is to receive status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the *rwho* port are interrogated to make sure they were sent by another *rwho* server process. They are then time-stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error, because an *rwho* server can be down while a host is actually up.

The second task performed by the server is to supply host status information. This task involves periodically acquiring system status information, packaging it in a message, and broadcasting it on the local network for other *rwho* servers to hear. The supply function is triggered by a timer and runs off a signal.

Deciding where to transmit the resultant packet is somewhat problematical. Status information must be broadcast on the local network. For networks that do not support broadcast, another scheme must be used. One possibility is to enumerate the known neighbors (based on the status messages received from other *rwho* servers). This method requires some bootstrapping information, because a server will have no idea what machines are its neighbors until it receives status messages from them. Therefore, if all machines on a network are freshly booted, no machine will have any known neighbors and thus will never receive, or send, any status information. This problem also occurs in the routing table management process in propagating routing status information.

The standard solution is to inform one or more servers of known neighbors and request that the servers always communicate with these neighbors. If each server has at least one neighbor supplied to it, status information can then propagate through a neighbor to hosts that are not directly neighbors.

If the server is able to support networks that provide a broadcast capability, as well as those that do not, networks with an arbitrary topology can share status information. However, network loops can cause problems. That is, if a host is connected to multiple networks, it will receive status information from itself. This situation can lead to an endless, wasteful exchange of information.

Software operating in a distributed environment should not have any site-dependent information compiled into it. To achieve this, each host must have a separate copy of the server, making server maintenance difficult. The BSD model attempts to isolate host-specific information from applications by providing system calls that return the necessary information. An example of such a call is **gethostname()** (see gethostname(2)), which returns the host's "official" name. In addition, an *ioctl* call can find the collection of networks to which a host is directly connected. Furthermore, a local network broadcasting mechanism has been implemented at the sockets level.

Combining these features lets a process broadcast on any directly connected local network that supports the notion of broadcasting in a site-independent manner. The system decides how to propagate status information in the case of *rwho,* or more generally in broadcasting. Such status information is broadcast to connected networks at the sockets level, where the connected networks have been obtained via the appropriate *ioctl* calls. The specifics of this kind of broadcasting are discussed in the next section, "Advanced Topics."

Advanced Topics

For most users of the sockets interface, the mechanisms already described will suffice in constructing distributed applications. However, you might need to use some of the more advanced features described in this section.

Out-of-Band Data

Stream sockets can accommodate "out-of-band" data. Out-of-band data is transmitted on a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data is delivered to the user independently of normal data. For stream sockets, 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 can be pending delivery to the user at any one time.

For communication protocols that support only in-band signaling (that is, the urgent data is delivered in sequence with the normal data), the system extracts the data from the normal data stream and stores it separately. You can choose between receiving urgent data in sequence and receiving it out of sequence, without having to buffer all the intervening data.

It is possible to "peek" (via MSG_PEEK) at out-of-band data. If the socket has a process group, SIGURG 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 via the appropriate *fcntl* call as described for SIGIO (see "Interrupt-driven Sockets I/O" on page 41). If multiple sockets can have out-of-band data awaiting delivery, a *select* call for exceptional conditions can be used to determine which sockets have such data pending. Neither the signal nor the *select* indicates the actual arrival of the out-of-band data, only notification of pending data.

In addition to the information passed, a logical mark is placed in the data stream to indicate 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 flushes pending output from the remote process(es), all data up to the mark in the data stream is discarded.

To send an out-of-band message, the MSG_OOB flag is supplied to a **send()** or **sendto()** call. To receive out-of-band data, MSG_OOB should be indicated when performing a **recvfrom()** or **recv()** call. To find out if the read pointer is currently pointing at the mark in the data stream, use the SIOCATMARK *ioctl*:

```
int yes;
ioctl(s, SIOCATMARK, &yes);
```

If the value *yes* is a 1 on return, the next *read* will return data after the mark. Otherwise (assuming out-of-band data has arrived), the next *read* will provide data sent by the client prior to transmission of the out-of-band signal. Example 2-2 shows the routine used in the remote login process to flush output on receipt of an interrupt or quit signal. It reads the normal data up to the mark (to discard it) and then reads the out-of-band byte.

Example 2-2 Flushing Terminal I/O on Receipt of Out-of-Band Data

```
#include <stdio.h>
#include <termios.h>        /* POSIX-style */
#include <sys/ioctl.h>
#include <sys/socket.h>

oob()
{
    int  mark;
    char waste[BUFSIZ];
    /* Flush local terminal output */
    tcflush(1, TCOFLUSH);
    for (;;) {
        if (ioctl(rem, SIOCATMARK, &mark) < 0) {
            perror("ioctl");
            break;
        }
        if (mark)
            break;
        (void) read(rem, waste, sizeof (waste));
    }
    if (recv(rem, &mark, 1, MSG_OOB) < 0) {
        perror("recv");
        ...
    }
    ...
}
```

A process can also read the out-of-band data without first reading up to the mark. Reading the out-of-band data in this way 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, the TCP protocol used to implement 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 will return an error of EWOULDBLOCK. Worse, there may be so much in-band data in the input buffer that normal flow control prevents the sender from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data for the urgent data to be delivered.

Certain programs that use multiple bytes of urgent data and must handle multiple urgent signals—for example, *telnet* (see telnet(1C))—need to retain the position of urgent data within the stream. This treatment is available as a sockets-level option, SO_OOBINLINE; see setsockopt(2) for usage. With this option, the position of urgent data (the "mark") is retained, but the urgent data immediately follows the mark within 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 is lost.

Nonblocking Sockets

Programs that cannot wait for a socket operation to be completed should use nonblocking sockets. I/O requests on nonblocking sockets return with an error if the request cannot be satisfied immediately.

Once a socket has been created with the **socket()** call, it can be marked as nonblocking by **fcntl()** as follows:

```
#include <fcntl.h>
 ...
int s;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
if (fcntl(s, F_SETFL, FNDELAY) < 0)  {
    perror("fcntl F_SETFL, FNDELAY");
    exit(1);
}
...
```

When performing nonblocking I/O on sockets, check for the error EAGAIN (stored in the global variable *errno*). This error occurs when an operation would normally block, but the socket it was performed on is nonblocking. In particular, **accept()**, **connect()**, **send()**, **recv()**, **read()**, and **write()** can all return EAGAIN, and processes should be prepared to deal with this return code.

Note: In previous releases of IRIX, the error EWOULDBLOCK was sometimes returned instead of EAGAIN. In the current release of IRIX, EWOULDBLOCK is defined as EAGAIN for source compatibility.

If an operation such as a **send()** cannot be completed, but partial writes are sensible (for example, when using a stream socket), data that can be sent immediately is processed, and the return value indicates the amount that was actually sent.

Interrupt-driven Sockets I/O

The SIGIO signal allows a process to be notified when a socket (or more generally, a file descriptor) has data waiting to be read. Use of the SIGIO facility requires three steps:

1. The process must use a *signal* call to set up a SIGIO signal handler.
2. The process must set the process ID or process group ID (see "Signals and Process Groups" on page 42) to receive notification of pending input either to its own process ID or to the group ID of its process group (the default process group of a socket is group zero). To do this, the process uses an *fcntl* call.
3. The process uses another *fcntl* call to enable asynchronous notification of pending I/O requests. Example 2-3 shows sample code that enables a process to receive information on pending I/O requests as they occur for a socket *s*. With the addition of a handler for SIGURG, this code can be used to prepare for receipt of SIGURG signals.

Example 2-3 Asynchronous Notification of I/O Requests

```
#include <signal.h>
#include <fcntl.h>

...
int    io_handler();
...
main()
{
    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);
    }

    /* Allow receipt of asynchronous I/O signals */
    if (fcntl(s, F_SETFL, FASYNC) < 0) {
        perror("fcntl F_SETFL, FASYNC");
        exit(1);
    }
}
io_handler()
{
...
}
```

Signals and Process Groups

Due to the existence of the SIGURG and SIGIO signals, each socket has an associated process number. This value is initialized to zero, but it can be redefined at a later time with the F_SETOWN *fcntl*, as was done in the previous code for SIGIO. To set the socket's process ID for signals, positive arguments should be given to the *fcntl* call. To set the socket's process group for signals, negative arguments should be passed to *fcntl*. Note that the process number indicates either the associated process ID or the associated process group; it is impossible to specify both at the same time. A similar *fcntl*, F_GETOWN, is available for determining the current process number of a socket.

Another useful signal you can use when constructing server processes is SIGCHLD, which is delivered to a process when any child processes have changed state. Normally, servers use SIGCHLD to "reap" child processes that have exited, without explicitly awaiting their termination or periodic polling for exit status. For example, the remote login server loop shown in "Connection-based Servers" on page 29 can be augmented, as shown in Example 2-4.

Example 2-4 Using the SIGCHLD Signal

```
#include <signal.h>

int reaper();
...
main()
{
    ...
    signal(SIGCHLD, reaper);
    listen(f, 5);
    for (;;) {
        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 <sys/wait.h>
reaper()
{
    union wait status;

    while (wait3(&status, WNOHANG, 0) > 0) {
        ;    /* no-op */
    }
}
```

If the parent server process fails to reap its children, a large number of "zombie" processes can be created.

Pseudo-Terminals

Many programs will not function properly without a terminal for standard input and output. Since sockets do not provide the semantics of terminals, it is often necessary to have a process communicate over the network through a *pseudo-terminal.* A pseudo-terminal is actually a pair of devices, master and slave, that allows a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo-terminal is supplied as input to a process reading from the master side, while data written on the master side is processed as terminal input for the slave. In this way, the process manipulating the master side of the pseudo-terminal has control over the information read and written on the slave side, as if it were manipulating the keyboard and reading the screen on a real terminal. The purpose of this abstraction is to preserve terminal semantics over a network connection. The slave side appears as a normal terminal to any process reading from or writing to it.

For example, the remote login server uses pseudo-terminals for remote login sessions. A user logging in to a machine across the network gets a shell with a slave pseudo-terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends a character that causes a remote machine to flush terminal output, the pseudo-terminal generates a control message for the server process. The server then sends an out-of-band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

Under IRIX, the name of the slave side of a pseudo-terminal has this syntax:

`/dev/ttyq`*x*

In this syntax, *x* is a number in the range 0 through 99. The master side of a pseudo-terminal is the generic device */dev/ptc*.

Creating a pair of master and slave pseudo-terminals is straightforward. The master half of a pseudo-terminal is opened first. The slave side of the pseudo-terminal is then opened and is set to the proper terminal modes if necessary. The process then *fork*s. The child closes the master side of the pseudo-terminal and *exec*s the appropriate program. Meanwhile, the parent closes the slave side of the pseudo-terminal and begins reading and writing from the master side.

The sample code in Example 2-5 illustrates making use of pseudo-terminals. This code assumes that a connection on a socket *s* exists, connected to a peer that wants a service of some kind, and that the process has disassociated itself from any previously controlling terminal.

Example 2-5 Creating and Using a Pseudo-Terminal on IRIX

```
#include <sys/sysmacros.h>
#include <fcntl.h>
#include <syslog.h>

int master, slave;
struct stat stb;
char line[sizeof("/dev/ttyqxxx")];
master = open("/dev/ptc", O_RDWR | O_NDELAY);
if (master < 0 || fstat(master, &stb) < 0) {
    syslog(LOG_ERR, "All network ports in use");
    exit(1);
}
sprintf(line, "/dev/ttyq%d", minor(stb.st_rdev));
/* Put in separate process group, disassociate
   controlling terminal. */
setsid();

slave = open(line, O_RDWR);    /* Open slave side */
if (slave < 0) {
    syslog(LOG_ERR, "Cannot open slave pty %s", line);
    exit(1);
}
pid = fork();
if (pid < 0) {
    syslog(LOG_ERR, "fork: %m");
    exit(1);
}
if (pid > 0) {        /* Parent */
    close(slave);
    ...
} else {            /* Child */
    close(master);
    dup2(slave, 0);
    dup2(slave, 1);
    dup2(slave, 2);
    if (slave > 2)
        (void) close(slave);
    ...
}
```

Selecting Protocols

If the third argument to the **socket()** call is 0, **socket()** will select 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. However, when using raw sockets to communicate directly with lower-level protocols or hardware interfaces, the protocol argument can be important for setting up de-multiplexing. For example, raw sockets in the Internet family can be used to implement a new protocol above IP, and the socket will receive packets only for the protocol specified.

To obtain a particular protocol, determine the protocol number as defined within the communication domain. For the Internet domain, you can use one of the library routines described in "Network Library Routines" on page 22. For example, you can use **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 call results in a socket *s* using a stream-based connection, but with a protocol type of *newtcp* instead of the default *tcp.*

Address Binding

Binding addresses to sockets in the Internet domain can be fairly complex. These associations are composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system.

Through the **bind()** system call, a process can specify half of an association, the <*local address, local port*> part, while the *connect* and *accept* calls are used to complete a socket's association by specifying the <*foreign address, foreign port*> part. Since the association is created in two steps, the association uniqueness requirement could be violated unless care is taken.

Furthermore, user programs do not always know the proper values to use for the local address and local port, since a host can reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding in the Internet domain, a wildcard address is provided. When an address is specified as INADDR_ANY (a manifest constant defined in <*netinet/in.h*>), the system interprets the address as "any valid address."

For example, to bind a specific port number to a socket but leave the local address unspecified, the following code might be used:

```
#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));
```

Sockets with wildcarded local addresses can 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 addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as above, the process will be able to accept connection requests that are addressed to 128.32.0.4 or 10.0.0.78. For a server process to allow only hosts on a given network to connect to it, it would bind whichever of the server's addresses were on the appropriate network.

Similarly, a local port can be left unspecified (specified as zero), in which case the system selects an appropriate port number for it. For example, to bind a specific local address to a socket but leave the local port number unspecified, use this code:

```
hp = gethostbyname(hostname);
if (hp == NULL) {
   ...
}
bcopy(hp->h_addr, (char *) sin.sin_addr, hp->h_length);
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The system selects the local port number based on two criteria:

- On BSD systems, Internet ports between 512 and 1023 (IPPORT_RESERVED – 1) are reserved for privileged users; Internet ports above IPPORT_USERRESERVED (5000) are reserved for nonprivileged servers; and Internet ports between IPPORT_RESERVED and IPPORT_USERRESERVED are used by the system for assignment to clients.
- The port number may not be bound to another socket.

To find a free Internet port number in the privileged range, the *rresvport* library routine can be used as follows to return a stream socket with a privileged port number:

```
int lport = IPPORT_RESERVED - 1;
int s;
 ...
s = rresvport(&lport);
if (s < 0) {
    if (errno == EAGAIN)
        fprintf(stderr, "socket: all ports in use");
    else
        perror("rresvport: socket");
    ...
}
```

The restriction on allocating ports allows processes executing in a "secure" environment to perform authentication based on the originating address and port number. For example, the *rlogin* command (see rlogin(1C)) lets users log in across a network without being asked for a password, under two conditions:

- The name of the system the user is logging in from is in the file */etc/hosts.equiv* on the system being logged in to (or the system name and the user name are in the user's *.rhosts* file in the user's home directory).
- The user's *rlogin* process is coming from a privileged port on the machine from which the user is logging in.

The port number and network address of the machine the user is logging in from can be determined either by the *from* result of the **accept()** call or from the **getpeername()** call.

The algorithm used by the system to select port numbers can be unsuitable for an application, because the algorithm creates associations in a two-step process. For example, FTP specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. The system disallows binding the same local address and port number to a socket if a previous data connection's socket still exists. To override the default port selection algorithm, the following 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 that are already in use can be bound. Binding local addresses does not violate the uniqueness requirement, because the system still checks at connect time to make sure 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.

Socket Options

You can use the **setsockopt()** and **getsockopt()** system calls to set and get a number of options on sockets. These options include marking a socket for broadcasting, not routing, lingering on closing, and so on. In addition, you can specify protocol-specific options for IP and TCP, as described in ip(7P) and tcp(7P), and in "IP Multicasting" on page 59.

The general form of the **setsockopt()** and **getsockopt()** calls is:

```
setsockopt(s, level, optname, optval, optlen);
getsockopt(s, level, optname, optval, optlen);
```

The parameters have these meanings:

- *s* is the socket on which the option is to be applied.
- *level* specifies the protocol layer on which the option is to be applied; in most cases, *level* is the sockets level, indicated by the symbolic constant SOL_SOCKET, defined in <*sys/socket.h*>.
- *optname* specifies the actual option, a symbolic constant that is also defined in <*sys/socket.h*>.
- *optval* points to the value of the option (in most cases, whether the option is to be turned on or off).
- *optlen* points to the length of the value of the option. For **getsockopt**, *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval* and modified upon return to indicate the actual amount of storage used.

For example, sometimes it's useful to determine the type (stream or datagram) of an existing socket. Programs under *inetd* (described in "The inetd Daemon" on page 51) may need to perform this task. You can do so via the SO_TYPE socket option and the **getsockopt** call, shown in this code:

```
#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) {
    perror("getsockopt");
    ...
}
```

After the **getsockopt** call, *type* will be set to the value of the socket type, as defined in <*sys/socket.h*>. For example, if the socket were a datagram socket, *type* would have the value corresponding to SOCK_DGRAM.

The inetd Daemon

When a single daemon listens for requests for many daemons, instead of having each daemon listen for its own requests, the number of idle daemons is reduced and the implementation of each daemon is simplified.

The *inetd* daemon handles three types of service:

- A standard service, which has a well-known port assigned to it and is listed in */etc/services* or the NIS *services* map—see services(4). It may be a service that implements an official Internet standard or is a BSD UNIX-specific service.
- An RPC service, which uses the Sun RPC calls as the transport; such services are listed in */etc/rpc* or the NIS *rpc* map—see rpc(4).
- A TCPMUX service, which is nonstandard and does not have a well-known port assigned to it. TCPMUX services are invoked from *inetd* when a program connects to the *tcpmux* well-known port and specifies the service name. This is useful for adding locally developed servers.

The *inetd* daemon is invoked at boot time. It examines the file */usr/etc/inetd.conf* to determine the servers it will listen for. Once this information has been read and a pristine environment created, *inetd* proceeds to create one socket for each service it is to listen for, binding the appropriate port number to each socket.

The *inetd* daemon performs a **select()** on these sockets for **read()** availability, waiting for a process to request a connection to the service corresponding to that socket. The *inetd* daemon then performs an **accept()** on the socket in question, **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 making use of *inetd* are considerably simplified, because *inetd* takes care of most of the IPC work required in establishing a connection. The server invoked by *inetd* expects the socket connected to its client on file descriptors 0 and 1, and can immediately perform any operations such as **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. However, for server programs that handle multiple services or protocols, *inetd* allocates socket descriptors to protocols based on lexicographic order of service and protocol name.

For example, the RPC mount daemon, *rpc.mountd,* has two entries in *inetd.conf* for its TCP and UDP ports. When invoked by *inetd*, the TCP socket is on descriptor 0, and UDP is on 1.

When writing servers under *inetd,* you can use the **getpeername** call to return the address of the peer (process) connected on the other end of the socket. For example, to log a client's Internet address in "dot notation" (for example, 128.32.0.4), you might use the following code:

```
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));
}
```

While the **getpeername** call is especially useful when writing programs to run with *inetd,* it can be used by stand-alone servers.

Standard TCP services are assigned unique well-known port numbers in the range of 0 to 255. These ports are of a limited number and are typically only assigned to official Internet protocols. The TCPMUX service, as described in RFC-1078, allows you to add locally developed protocols without needing an official TCP port assignment.

The protocol used by TCPMUX is simple: a TCP client connects to a foreign host on TCP port 1. It sends the service name followed by a carriage-return/ line-feed **<Ctrl>-F**. The server replies with a single character indicating positive (+) or negative (–) acknowledgment, immediately followed by an optional message of explanation, terminated with a **<Ctrl>-F**. If the reply was positive, the selected protocol begins; otherwise, the connection is closed. In the IRIX system, the TCPMUX service is built into *inetd;* that is, *inetd* listens on TCP port 1 for requests for TCPMUX services listed in *inetd.conf.*

The following code is an example TCPMUX server and its *inetd.conf* entry:

```
#include <sys/types.h>
#include <stdio.h>

main()
{
    time_t t;
    printf("+Go\r\n");
    fflush(stdout);
    time(&t);
    printf("%d = %s", t, ctime(&t));
    fflush(stdout);
}
```

More sophisticated servers may want to do additional processing before returning the positive or negative acknowledgment.

The *inetd.conf* entry is:

```
tcpmux/current_time stream tcp nowait guest /d/curtime curtime
```

The following portion of the client code handles the TCPMUX handshake:

```
char line[BUFSIZ];
FILE *fp;
 ...
/* Use stdio for reading data from the server */
fp = fdopen(sock, "r");
if (fp == NULL) {
    fprintf(stderr, "Can't create file pointer\n");
    exit(1);
}
/* Send service request */
sprintf(line, "%s\r\n", "current_time");
if (write(sock, line, strlen(line)) < 0) {
    perror("write");
    exit(1);
}
```

```
/* Get ACK/NAK response from the server */
if (fgets(line, sizeof(line), fp) == NULL) {
    if (feof(fp)) {
        die();
    } else {
        fprintf(stderr, "Error reading response\n");
        exit(1);
    }
}
/* Delete <CR> */
if ((lp = index(line, '\r')) != NULL) {
    *lp = ' ';
}

switch (line[0]) {
    case '+':
        printf("Got ACK: %s\n", &line[1]);
        break;
    case '-':
        printf("Got NAK: %s\n", &line[1]);
        exit(0);
    default:
        printf("Got unknown response: %s\n", line);
        exit(1);
}
/* Get rest of data from the server */
while ((fgets(line, sizeof(line), fp)) != NULL) {
    fputs(line, stdout);
}
```

Broadcasting

Using a datagram socket, you can send broadcast packets on many networks supported by the system. The network itself 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. Consequently, the ability to send broadcast packets has been limited to sockets explicitly marked to allow broadcasting. Broadcast is typically used for one of two reasons: to find a resource on a local network without prior knowledge of its address or to send information to all accessible neighbors.

Note: Multicasting is an alternative to broadcasting. See "IP Multicasting" on page 59 for information about setting up multicast sockets.

To send a broadcast message, create a datagram socket:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

Mark the socket to allow broadcasting:

```
int on = 1;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on));
```

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 destination address of the broadcast message depends on the network(s). The Internet domain supports a shorthand notation for broadcast on the local network, the address INADDR_BROADCAST (defined in *<netinet/in.h>*).

Determining the list of addresses for all reachable neighbors requires knowledge of the networks to which the host is connected. Since this information should be obtained in a host-independent fashion and may be impossible to derive, IRIX provides a method for retrieving this information from the system data structures.

The SIOCGIFCONF *ioctl* call returns the interface configuration of a host in the form of a single *ifconf* structure. This structure contains a data area that is made up of an array of *ifreq* structures, one for each network interface to which the host is connected.

These structures are defined in *<net/if.h>*, as shown in this example:

```
struct ifconf {
    ifc_len    /* size of associated buffer */
    union {
        caddr_t  ifcu_buf;
        struct   ifreq *ifcu_req;
    } ifc_ifcu;
};

/* Buffer address */
#define ifc_buf   ifc_ifcu.ifcu_buf
```

```
/* Array of structures returned */
#define ifc_req   ifc_ifcu.ifcu_req

#define IFNAMSIZ        16
struct  ifreq {

        /* Interface name, e.g. "en0" */
        char    ifr_name[IFNAMSIZ];
        union {
                struct  sockaddr ifru_addr;
                struct  sockaddr ifru_dstaddr;
                struct  sockaddr ifru_broadaddr;
                short   ifru_flags;
                int     ifru_metric;
                /* MIPS ABI - unused by BSD */
                char    ifru_data[1];
                char    ifru_enaddr[6];         /* MIPS ABI */
                char    ifru_oname[IFNAMSIZ];   /* MIPS ABI */
                struct  ifstats ifru_stats;

                /* Trusted IRIX */
                struct {
                        caddr_t ifruv_base;
                        int     ifruv_len;
                }       ifru_vec;
        } ifr_ifru;
};

/* Address */
#define ifr_addr        ifr_ifru.ifru_addr

/* Other end of p-to-p link */
#define ifr_dstaddr     ifr_ifru.ifru_dstaddr

/* Broadcast address */
#define ifr_broadaddr   ifr_ifru.ifru_broadaddr

/* Flags */
#define ifr_flags       ifr_ifru.ifru_flags

/* Metric */
#define ifr_metric      ifr_ifru.ifru_metric

/* For use by interface */
#define ifr_data        ifr_ifru.ifru_data
```

```
/* Ethernet address */
#define ifr_enaddr      ifr_ifru.ifru_enaddr

/* Other interface name */
#define ifr_oname       ifr_ifru.ifru_oname

/* Statistics */
#define ifr_stats       ifr_ifru.ifru_stats

/* Trusted IRIX */
#define ifr_base        ifr_ifru.ifru_vec.ifruv_base
#define ifr_len         ifr_ifru.ifru_vec.ifruv_len
```

The following call obtains the interface configuration:

```
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* will contain one *ifreq* structure for each network to which the host is connected, and *ifc.ifc_len* will have been modified to reflect the number of bytes used by the *ifreq* structure.

Each structure has an associated set of interface flags that tell whether the network corresponding to that interface is up or down, point-to-point or broadcast, and so on. The SIOCGIFFLAGS *ioctl* retrieves these flags for an interface specified by an *ifreq* structure:

```
struct ifreq *ifr;
struct sockaddr dst;

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 family other than the one 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_POINTTOPOINT)) == 0) {
            continue;
    }
```

Once you retrieve the flags, retrieve the broadcast address. For broadcast networks, retrieval is done via the SIOCGIFBRDADDR *ioctl.* For point-to-point networks, the address of the destination host is obtained with SIOCGIFDSTADDR:

```
if (ifr->ifr_flags & IFF_POINTTOPOINT) {
    if (ioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
        ...
    }
    bcopy((char *) ifr->ifr_dstaddr, (char *) &dst,
          sizeof (ifr->ifr_dstaddr));

} else if (ifr->ifr_flags & IFF_BROADCAST) {
    if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
        ...
    }
    bcopy((char *) ifr->ifr_broadaddr, (char *) &dst,
          sizeof (ifr->ifr_broadaddr));
}
```

After the appropriate *ioctls* get the broadcast or destination address (now in *dst*), use the **sendto()** call:

```
    sendto(s, buf, buflen, 0, (struct sockaddr *)&dst,
           sizeof (dst));
```

In the above loop, one **sendto()** occurs for every interface the host is connected to that supports broadcast or point-to-point addressing. For a process to send only broadcast messages on a given network, use code similar to that outlined above, but the loop needs to find the correct destination address.

Received broadcast messages contain the sender's address and port, since datagram sockets are bound before a message is allowed to go out.

IP Multicasting

Multicasting is the transmission of an IP datagram to a host group, a set of zero or more hosts identified by a single IP destination address. A multicast datagram is delivered to all members of its destination host group with the same best-efforts reliability as regular unicast IP datagrams; that is, the datagram is not guaranteed to arrive intact at all members of the destination group or in the same order relative to other datagrams.

The membership of a host group is dynamic; that is, hosts may join and leave groups at any time. There is no restriction on the location or number of members in a host group. A host may be a member of more than one group at a time. A host need not be a member of a group to send datagrams to it.

A host group may be permanent or transient. A permanent group has a well-known, administratively assigned IP address. It is the address, not the membership of the group, that is permanent; at any time a permanent group may have any number of members, even zero. Those IP multicast addresses that are not reserved for permanent groups are available for dynamic assignment to transient groups, which exist as long as they have members.

In general, a host cannot assume that datagrams sent to any host group address will reach only the intended hosts, or that datagrams received as a member of a transient host group are intended for the recipient. Misdirected delivery must be detected at a level above IP, using higher-level identifiers or authentication tokens. Information transmitted to a host group address should be encrypted or governed by administrative routing controls if the sender is concerned about unwanted listeners.

Note: This RFC-1112 level-2 implementation of IP multicasting is experimental and subject to change in order to track future BSD UNIX releases. In particular, there may be changes in the way a process overrides the default interface for sending multicast datagrams and for joining multicast groups. This ability to override the default interface is intended mainly for routing daemons; normal applications should not be concerned with specific interfaces.

IP multicasting is currently supported only on AF_INET sockets of type SOCK_DGRAM and SOCK_RAW, and only on subnetworks for which the interface driver has been modified to support multicasting. The standard Ethernet, FDDI, and SLIP interfaces on the IRIS support multicasting. (Older versions of ENP-10 Ethernet interfaces may require an upgrade; see the *IRIX Admin* manual set for details.)

The next sections describe how to send and receive multicast datagrams.

Sending IP Multicast Datagrams

To send a multicast datagram, specify an IP multicast address in the range 224.0.0.0 to 239.255.255.255 as the destination address in a **sendto()** call.

The definitions required for the multicast-related socket options are found in *<netinet/in.h>*. All IP addresses are passed in network byte order.

By default, IP multicast datagrams are sent with a time-to-live (TTL) of 1, which prevents them from being forwarded beyond a single subnetwork. A new socket option allows the TTL for subsequent multicast datagrams to be set to any value from 0 to 255, in order to control the scope of the multicasts:

```
u_char ttl;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl,
           sizeof(ttl));
```

Multicast datagrams with a TTL of 0 will not be transmitted on any subnetwork but may be delivered locally if the sending host belongs to the destination group and if multicast loopback has not been disabled on the sending socket. Multicast datagrams with a TTL greater than 1 may be delivered to more than one subnetwork if there is at least one multicast router attached to the first-hop subnetwork. To provide meaningful scope control, the multicast routers support the notion of TTL thresholds, which prevent datagrams with less than a certain TTL from traversing certain subnetworks.

The thresholds enforce the convention shown in Table 2-3.

Table 2-3 TTL Threshold Convention

Scope	Initial TTL
Restricted to the same host	0
Restricted to the same subnetwork	1
Restricted to the same site	32
Restricted to the same region	64
Restricted to the same continent	128
Unrestricted	255

"Sites" and "regions" are not strictly defined, and sites may be further subdivided into smaller administrative units, as a local matter.

An application may choose an initial TTL other than one listed in Table 2-3. For example, an application might perform an expanding-ring search for a network resource by sending a multicast query, first with a TTL of 0, and then with larger and larger TTLs, until a reply is received, perhaps using the TTL sequence 0, 1, 2, 4, 8, 16, 32.

The multicast router *mrouted* (see mrouted(1M)) refuses to forward any multicast datagram with a destination address between 224.0.0.0 and 224.0.0.255, inclusive, regardless of its TTL. This range of addresses is reserved for the use of routing protocols and other low-level topology discovery or maintenance protocols, such as gateway discovery and group membership reporting.

The address 224.0.0.0 is guaranteed not to be assigned to any group, and 224.0.0.1 is assigned to the permanent group of all IP hosts (including gateways). This assignment convention is used to address all multicast hosts on the directly connected network. There is no multicast address (or any other IP address) for all hosts on the total Internet. The addresses of other well-known, permanent groups are published in the "Assigned Numbers" RFC (*Internet Request for Comment 1060*).

Each multicast transmission is sent from a single network interface, even if the host has more than one multicast-capable interface. (If the host is also serving as a multicast router, a multicast may be *forwarded* to interfaces other than the originating interface, provided that the TTL is greater than 1.) The default interface to be used for multicasting is the primary network interface on the system. A socket option is available to override the default for subsequent transmissions from a given socket:

```
struct in_addr addr;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &addr,
           sizeof(addr));
```

where *addr* is the local IP address of the desired outgoing interface. An address of INADDR_ANY may be used to revert to the default interface. The local IP address of an interface can be obtained via the SIOCGIFCONF *ioctl.* To determine if an interface supports multicasting, fetch the interface flags via the SIOCGIFFLAGS *ioctl* and see if the IFF_MULTICAST flag is set. (Normal applications should not need to use this option; it is intended primarily for multicast routers and other system services specifically concerned with Internet topology.) The SIOCGIFCONF and SIOCGIFFLAGS *ioctl*s are described in "Broadcasting" on page 54.

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP layer for local delivery. Another socket option gives the sender explicit control over whether or not subsequent datagrams are looped back:

```
u_char loop;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop,
           sizeof(loop));
```

In this example, *loop* is set to 0 to disable loopback, and set to 1 to enable loopback. This option improves performance for applications that may have no more than one instance on a single host (such as a router daemon) by eliminating the overhead of receiving their own transmissions. In general, *loop* should not be used by applications for which there may be more than one instance on a single host (such as a conferencing program) or for which the sender does not belong to the destination group (such as a time-querying program).

A multicast datagram sent with an initial TTL greater than 1 may be delivered to the sending host on a different interface from that on which it was sent if the host belongs to the destination group on that other interface. The loopback control option has no effect on such delivery.

Receiving IP Multicast Datagrams

Before a host can receive IP multicast datagrams, it must become a member of one or more IP multicast groups. A process can ask the host to join a multicast group by using this socket option:

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq,
           sizeof(mreq))
```

mreq is defined in this structure:

```
struct ip_mreq {
    struct in_addr imr_multiaddr; /*multicast group to join*/
    struct in_addr imr_interface; /*interface to join on*/
}
```

Every membership is associated with a single interface, and it is possible to join the same group on more than one interface. *imr_interface* should be INADDR_ANY to choose the default multicast interface or one of the host's local addresses to choose a particular (multicast-capable) interface. Up to IP_MAX_MEMBERSHIPS (currently 20) memberships may be added on a single socket.

To drop a membership, use

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq,
           sizeof(mreq));
```

where *mreq* contains the same values as used to add the membership. The memberships associated with a socket are also dropped when the socket is closed or the process holding the socket is killed. However, more than one socket may claim a membership in a particular group, and the host will remain a member of that group until the last claim is dropped.

The memberships associated with a socket do not necessarily determine which datagrams are received on that socket. Incoming multicast packets are accepted by the kernel IP layer if any socket has claimed a membership in the destination group of the datagram; however, delivery of a multicast datagram to a particular socket is based on the destination port (or protocol type for raw sockets), just as with unicast datagrams. To receive multicast datagrams sent to a particular port, it is necessary to bind to that local port, leaving the local address unspecified (that is, INADDR_ANY).

More than one process may bind to the same SOCK_DGRAM UDP port if the **bind()** call is preceded by:

```
int on = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &on, sizeof(on));
```

In this case, every incoming multicast or broadcast UDP datagram destined to the shared port is delivered to all sockets bound to the port. For backward compatibility reasons, this does not apply to incoming unicast datagrams. Unicast datagrams are never delivered to more than one socket, regardless of how many sockets are bound to the datagram's destination port. SOCK_RAW sockets do not require the SO_REUSEPORT option to share a single IP protocol type.

Note: A final multicast-related extension is independent of IP: two new *ioctl*s, SIOCADDMULTI and SIOCDELMULTI, are available to add or delete link-level (for example, Ethernet) multicast addresses accepted by a particular interface. The address to be added or deleted is passed as a *sockaddr* structure of family AF_UNSPEC, within the standard *ifreq* structure.

These *ioctl*s are used for protocols other than IP and require superuser privileges. A link-level multicast address added via SIOCADDMULTI is not automatically deleted when the socket used to add it goes away; it must be explicitly deleted. It is inadvisable to delete a link-level address that may be in use by IP.

Sample Multicast Program

The following program sends or receives multicast packets. If invoked with one argument, it sends a packet containing the current time to an arbitrarily chosen multicast group and UDP port. If invoked with no arguments, it receives and prints these packets. Start it as a sender on just one host and as a receiver on all the other hosts.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <stdio.h>

#define EXAMPLE_PORT    6000
#define EXAMPLE_GROUP    "224.0.0.250"

main(argc)
    int argc;
{
    struct sockaddr_in addr;
    int    addrlen, fd, cnt;
    struct ip_mreq mreq;
    char message[50];

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        perror("socket");
        exit(1);
    }
```

```
    bzero(&addr, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(EXAMPLE_PORT);
    addrlen = sizeof(addr);
        if (argc > 1) {    /* Send */
        addr.sin_addr.s_addr = inet_addr(EXAMPLE_GROUP);
        while (1) {
            time_t t = time(0);
            sprintf(message, "time is %-24.24s", ctime(&t));
            cnt = sendto(fd, message, sizeof(message), 0,
                         &addr, addrlen);
            if (cnt < 0) {
                perror("sendto");
                exit(1);
            }
            sleep(5);
        }
    } else {        /* Receive */
        if (bind(fd, &addr, sizeof(addr)) < 0) {
            perror("bind");
            exit(1);
        }
        mreq.imr_multiaddr.s_addr = inet_addr(EXAMPLE_GROUP);
        mreq.imr_interface.s_addr = htonl(INADDR_ANY);
        if (setsockopt(fd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
            &mreq, sizeof(mreq)) < 0) {
            perror("setsockopt mreq");
            exit(1);
        }
        while (1) {
            cnt = recvfrom(fd, message, sizeof(message), 0,
                           &addr, &addrlen);
                if (cnt < 0) {
                perror("recvfrom");
                exit(1);
                } else if (cnt == 0) {
                break;
            }
            printf("%s: message = \"%s\"\n",
                   inet_ntoa(addr.sin_addr), message);
        }
    }
}
```

Chapter 3

Introduction to RPC Programming

Remote procedure calls are a high-level communication paradigm that allows programmers to write network applications using procedure calls that hide the details of the underlying network. RPC implements a client/server system without requiring that callers be aware of the underlying network.

This chapter introduces the RPC programming interface, which enables an application to make procedure calls to remote machines using architecture-independent mechanisms. This portability is achieved by using eXternal Data Representation (XDR) data-encoding to resolve byte-ordering differences and the port mapper program to locate and invoke a requested procedure.

Topics in this chapter include:

- an overview of remote procedure calls, including the RPC model, the RPC protocol, and RPC message authentication
- the XDR standard
- the layers of RPC
- the *rpcgen* protocol compiler
- assigning RPC program numbers
- the port mapper programs

Overview of Remote Procedure Calls

Programs that communicate over a network need a paradigm for communication. For example, a low-level mechanism might send a signal when incoming packets arrive, causing a network signal handler to execute. With the remote procedure call paradigm, a client makes a procedure call to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, sends back the reply, and the procedure call returns to the client.

In this context, a *server* is a machine where some number of network services are implemented. A *service* is a collection of one or more remote programs. A remote program implements one or more remote procedures; the procedures, their parameters, and results are documented in the specific program's protocol specification. Network *clients* are pieces of software that initiate remote procedure calls to services. A server may support more than one version of a remote program in order to be forward-compatible with changing protocols.

The Remote Procedure Call Model

The remote procedure call model is similar to the local procedure call model. With the local model, the caller places arguments to a procedure in a well-specified location (such as a result register) and transfers control to the procedure. When the caller eventually regains control, it extracts the results of the procedure from the well-specified location and continues execution.

The remote procedure call model operates in a similar fashion, except control winds through two processes: the caller's process and a server's process. That is, the caller process sends a message to the server process and waits (blocks) for a reply message. The call message contains the procedure's parameters (among other things), and the reply message contains the procedure's results (among other things). When the reply message returns, the caller extracts the results of the procedure and resumes execution.

On the server side, a process is dormant as it waits for the arrival of a call message. When a reply arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then waits for the arrival of the next call message.

Note that in the remote procedure call model, only one of the two processes is active at any given time. However, this scenario is given only as an example. The RPC protocol (see "RPC Transports and Semantics" on page 70) makes no restrictions on concurrency, and other scenarios are possible. For example, an implementation may choose to have asynchronous RPC calls, so 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 the server can be free to receive other requests.

Figure 3-1 illustrates the remote procedure call model.

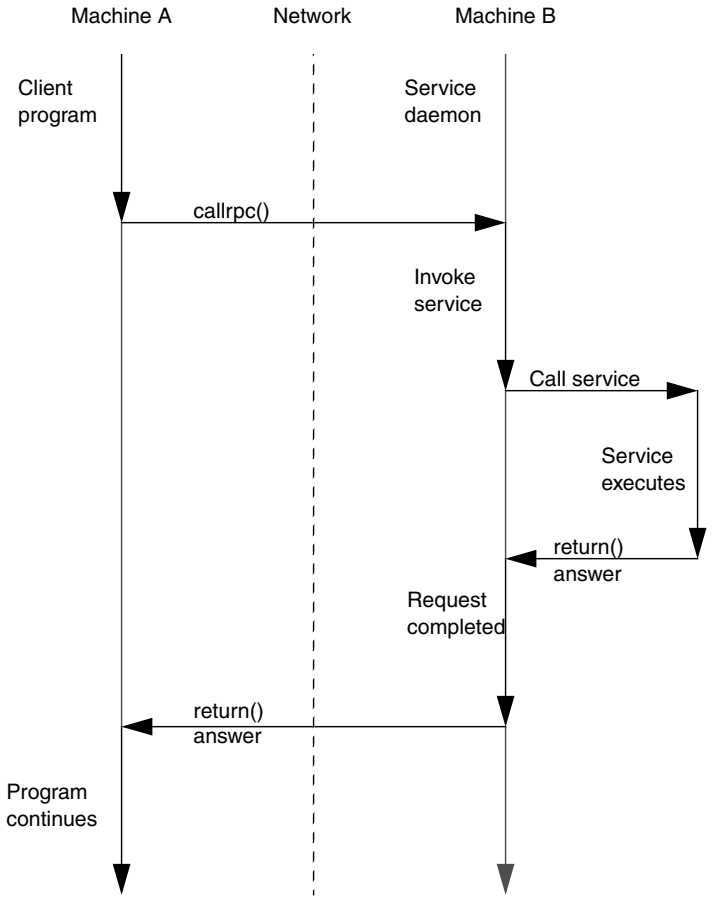


Figure 3-1 The Remote Procedure Call Model

RPC Transports and Semantics

The RPC package is implemented using the RPC protocol, a message protocol specified using XDR language. The RPC protocol is independent of transport protocols; that is, RPC does not care how a message is passed from one process to another; the protocol is concerned only with the specification and interpretation of messages.

RPC does not try to implement reliability; the application must be aware of the type of transport protocol underneath RPC. If the application knows it's running on top of a reliable transport (such as TCP/IP), most of the work is already done. If the application is running on top of an unreliable transport (such as UDP/IP), however, it must implement its own retransmission and timeout policy, because the RPC layer does not provide this service.

To ensure 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 what happens when RPC runs on top of an unreliable transport. If an application retransmits RPC messages after short timeouts and receives no reply, all it can infer is that the procedure was executed zero or more times. If it receives a reply, it can infer that the procedure was executed at least once.

A server may wish to ensure some degree of execute-at-most-once semantics and remember previously granted requests from a client and not grant them again. A server can do this by taking advantage of the transaction ID that is packaged with every RPC request.

The transaction ID is used primarily by the client RPC layer to match replies to requests. However, a client application may choose to reuse its previous transaction ID when retransmitting a request. The server application, knowing this fact, may choose to remember this ID after granting a request and not regrant requests with the same ID in order to achieve some degree of execute-at-most-once semantics. The server is not allowed to examine this ID in any other way except as a test for equality.

On the other hand, if the application uses a reliable transport, it can infer from a reply message that the procedure was executed exactly once. If it receives no reply message, however, it cannot assume the remote procedure was not executed. Note that even with a connection-oriented protocol such as TCP, an application still needs timeouts and reconnection to handle server crashes.

Additional transport possibilities exist for datagram- or connection-oriented protocols. On IRIX, RPC is currently implemented on top of both TCP/IP and UDP/IP transports.

Binding and Rendezvous Independence

The act of binding a client to a service is not part of the RPC specification. This important and necessary function is left up to some higher-level software. (The software may use RPC itself; see "Port Mapper Program Protocol" in Appendix A for more information.)

Implementors 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, using RPC to accomplish its task.

RPC Message Identification and Authentication

The RPC protocol provides the fields necessary for a client to identify itself to a service and vice versa. Authentication and identity-based access control mechanisms, on the other hand, are not provided and must be added to provide security.

Identification is the means to present or assert an identity that is recognizable to the receiver; it provides no proof that the identity is valid. In UNIX, typing a user name at the login prompt is identification, as is putting a UID and a GID into an AUTH_UNIX credential for RPC.

Authentication provides the actions to verify the truth of the asserted identity. To continue with the examples above, the password is used by the UNIX login program to verify the user's identity, and the AUTH_DES or AUTH_KERB protocols (not provided with Silicon Graphics' RPC) provide authentication in RPC. The action the password program performs is to compare what the user types to the encrypted copy that resides on the system. AUTH_DES and AUTH_KERB use encryption-based user authentication.

Silicon Graphics' RPC message authentication mechanism, which consists of the AUTH_UNIX and AUTH_NONE protocols, does not provide authentication, as defined above. When these protocols are used, the application must provide the authentication. An example of how to extend RPC to include an authentication procedure, as provided in the *rlogin* program, is shown in "Server-side Authentication" in Chapter 5.

Once the client is identified and verified, access control can be implemented. Access control is the mechanism that provides permission to allow the requests made by the user to be granted, based upon the user's authentic identity. Access control is not provided in RPC and must be supplied by the application.

Several different authentication protocols are supported. A field in the RPC header indicates which protocol is being used.

For information about specific authentication protocols, see "Authentication Protocols" in Appendix A.

The XDR Standard

RPC assumes the existence of XDR, a set of library routines that allow a C programmer to describe arbitrary data structures in a machine-independent fashion. XDR is useful for transferring data between diverse computer architectures and has been used to communicate data between such diverse machines as the IRIS, Sun, VAX, IBM PC, and Cray computers.

XDR enables RPC to handle arbitrary data structures, regardless of a machine's byte order or structure layout conventions, by converting the data structures to XDR before sending them over the wire. Any program running on any machine can use XDR to create portable data by translating its local representation into the XDR representation; similarly, any program running on any machine can read portable data by translating the XDR standard representations into its local equivalents. This process of converting from a particular machine representation to XDR format is called *serializing*, and the reverse process is called *deserializing* (see Chapter 7, "XDR Programming Notes," for details).

XDR uses the XDR language to describe data formats (see Chapter 6, "XDR and RPC Language Structure"). Protocols such as Sun RPC and NFS use XDR to describe their data format.

The XDR language lets you describe intricate data formats in a concise manner. The alternative—using graphical representations (an informal language)—quickly becomes incomprehensible when faced with complexity. The XDR language is similar to the C language, but it is not a programming language and can only be used to describe data.

XDR fits into the ISO presentation layer and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference is that XDR uses implicit typing, while X.409 uses explicit typing.

The Layers of RPC

This section provides a brief overview of the RPC layers. For programming details about each layer, see Chapter 5, "RPC Programming Guide."

RPC is divided into three layers: the highest layer, the middle layer, and the lowest layer.

The Highest Layer

The highest layer of RPC is transparent to the operating system, the machine, and the network upon which it is run. It's probably best to think of this level as a way of using RPC, rather than as a part of "RPC proper." Programmers who write RPC routines should (almost) always make this layer available to others by using a simple C front end that entirely hides the networking.

For example, at this level, a program can make a call to the C routine **rnusers()**, which returns the number of users on a remote machine. Users are not explicitly aware of using RPC—they simply call a procedure, just as they would call **malloc()**.

The Middle Layer

The middle layer of RPC is really RPC proper and consists of routines used for most applications. In the middle layer, the user simply makes remote procedure calls to routines on other machines, without considering details about the socket interface, the UNIX system, or other low-level implementation mechanisms. For example, the middle layer allows RPC to pass the "hello world" test.

RPC calls are made with the **registerrpc()**, **callrpc()**, and **svc_run()** routines. **registerrpc()** and **callrpc()** are the most fundamental: **registerrpc()** obtains a unique system-wide procedure-identification number, and **callrpc()** actually executes a remote procedure call. In the middle layer, a call to **rnusers()** is implemented by using these two routines.

Note: The middle layer of RPC is rarely used in serious programming due to its inflexibility (simplicity). It does not allow timeout specifications or the choice of transport; it does not allow UNIX process control or flexibility in the case of errors; and it does not support multiple methods of call authentication. Although programmers rarely need all of these controls, one or two are sometimes necessary.

The Lowest Layer

In the lowest layer of RPC, the programmer has control over the hidden details and can write more-sophisticated applications that alter the defaults of the routines. At this layer, programmers can explicitly manipulate sockets used for transmitting RPC messages.

Programs written at this level are most efficient, but efficiency is rarely an issue, because RPC clients and servers rarely generate heavy network loads; if possible, this level should be avoided.

Note: This guide only describes the interface to C, but you can make remote procedure calls from any language. And, although this guide describes RPC when it is used to communicate between processes on different machines, it works just as well for communication between different processes on the same machine.

The rpcgen Protocol Compiler

Programming applications that use RPC can be difficult, especially when you are writing XDR routines that convert procedure arguments and results into their network format and vice versa. The *rpcgen* compiler helps automate the process of writing RPC applications. Using *rpcgen* decreases development time that would otherwise be spent coding and debugging low-level routines. With *rpcgen*, the compiler does most of the dirty work; the programmer need only debug the main features of the application, rather than spend time debugging network interface code.

rpcgen accepts remote program interface definitions written in the RPC language (see Chapter 6, "XDR and RPC Language Structure," for more information) and produces C language output for RPC programs. This output consists of a stub version of the client routines, a server skeleton, XDR filter routines for parameters and results, a header file that contains common definitions, and ANSI C prototyped stub routines.

You can compile and link *rpcgen*'s output files using standard techniques. Then after writing the server procedures, you can link the server procedures with the server skeletons to produce an executable server program.

To use a remote program, the programmer writes an ordinary main program that makes local procedure calls to the client skeletons. Linking the main program with the skeletons creates an executable program.

Like other compilers, *rpcgen* provides an escape hatch that lets programmers mix low-level code with high-level code. In speed-critical applications, handwritten routines can be linked with the *rpcgen* output without any difficulty. In addition, *rpcgen* output can be used as a starting point; you can rewrite the code as necessary.

For details about *rpcgen*, see Chapter 4, "Programming with rpcgen."

Assigning RPC Program Numbers

An RPC call message has three fields, which uniquely identify the procedure to be called. These fields include:

- the remote program's RPC version number
- the remote procedure number
- the remote program number

The version field of the call message identifies which version of the RPC protocol the caller is using. Because most new protocols evolve into better, stable, and mature protocols, a version field of the call message identifies which version of the protocol the caller is using. Version numbers make it possible for old and new protocols to communicate through the same server process.

The procedure number identifies the procedure to be called. This number is documented in the specific 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()** (see "Remote Programs and Procedures" in Appendix A for more information).

Program numbers are administered by a central authority (such as Sun Microsystems). Once you have a program number, you can implement your remote program. Table 3-1 lists some of the currently registered programs.

Table 3-1 Some Registered RPC Programs

RPC Number	Program	Description
100000	PMAPPROG	port mapper
100001	RSTATPROG	remote stats
100002	RUSERSPROG	remote users
100003	NFSPROG	NFS
100004	YPPROG	NIS
100005	MOUNTPROG	mount daemon
100006	DBXPROG	remote dbx
100007	YPBINDPROG	ypbind server
100008	WALLPROG	shutdown msg
100009	YPPASSWDPROG	yppasswd server
100010	ETHERSTATPROG	ether stats
100012	SPRAYPROG	spray packets
100017	REXECPROG	remote execution
100020	LOCKPROG	local lock manager
100021	NETLOCKPROG	network lock manager
100023	STATMON1PROG	status monitor 1
100024	STATMON2PROG	status monitor 2
100026	BOOTPARAMPROG	boot parameters service
100028	YPUPDATEPROG	ypupdate server
100029	KEYSERVEPROG	key server
100036	PWDAUTHPROG	password authorization

RPC program numbers are assigned in groups of 0x20000000 (536870912) according to the categories in Table 3-2.

Table 3-2 RPC Program Number Assignment

Number	Assignment
0x0 - 0x1fffffff	Defined by Sun
0x20000000 - 0x3fffffff	Defined by user
0x40000000 - 0x5fffffff	Transient
0x60000000 - 0x7fffffff	Reserved
0x80000000 - 0x9fffffff	Reserved
0xa0000000 - 0xbfffffff	Reserved
0xc0000000 - 0xdfffffff	Reserved
0xe0000000 - 0xffffffff	Reserved

Sun Microsystems administers the first group of numbers. The second group 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.

To register a protocol specification, write to:

RPC Administrator
Sun Microsystems
2550 Garcia Avenue
Mountain View, CA 94043

Make sure to include a compilable *rpcgen* "*.x*" file (see Chapter 4) describing your protocol. In return, you will be given a unique program number.

You can find the RPC program numbers and protocol specifications of standard Sun RPC services in the include files in */usr/include/rpcsvc*. These services, however, constitute only a small subset of those that have been registered.

The Port Mapper Programs

The port mappers (see portmap(1M) or rpcbind(1M)) are servers that convert RPC program numbers into universal addresses (IP port numbers). Either *portmap* or *rpcbind* must be running in order to make RPC calls. If *rpcbind* is installed, it runs by default.

When an RPC server is started, it tells the port mapper the port number it is listening to and what RPC program numbers it is prepared to serve. When a client wants to make an RPC call to a given program number, it first checks the port mapper on the server machine to determine the port number where RPC packets should be sent.

Chapter 4

Programming with rpcgen

This chapter describes the *rpcgen* protocol compiler, which helps automate the process of writing RPC applications. When you use *rpcgen*, the compiler does most of the dirty work; you need only debug the main features of the application instead of spending time debugging network interface code.

Topics in this chapter include:

- converting local procedures into remote procedures
- generating XDR routines
- the C preprocessor
- *rpcgen* programming notes

Note: For a general introduction to RPC, see Chapter 3, “Introduction to RPC Programming.” For details about programming RPC without *rpcgen*, see Chapter 5, “RPC Programming Guide.”

Introduction to the rpcgen Compiler

The *rpcgen* protocol compiler accepts remote program interface definitions written in RPC language and produces C language output for RPC programs. (See Chapter 6, "XDR and RPC Language Structure," for details about writing program interface definitions using RPC language.) This C output includes:

- skeleton versions of the client routines
- a server skeleton
- XDR filter routines for parameters and results
- a header file that contains common definitions
- ANSI C prototyped stub routines (optional)

The client skeletons interface with the RPC library and "hide" the network from its caller. Similarly, the server skeleton hides the network from server procedures that are to be invoked by remote clients.

The programmer writes server procedures, using any language that observes C language calling conventions, and links them with the server skeleton generated by *rpcgen* to produce an executable server program. To use a remote program, the programmer writes an ordinary main program that makes local procedure calls to the client skeletons produced by *rpcgen*.

Note: At present, the main program must be written in C or C++.

Linking the main program with *rpcgen*'s skeletons creates an executable program. Options to *rpcgen* let you suppress stub generation, specify the transport to be used by the server stub, pass flags to *cpp*, or choose a different preprocessor. See rpcgen(1) for details.

Changing Local Procedures to Remote Procedures

Assume you have an application that runs on a single machine and you want to convert it to run over a network. The following code sample demonstrates the conversion for a program that prints a message to the console:

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

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

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

And then, of course:

```
% cc printmsg.c -o printmsg
% printmsg "Hello, there"
Message Delivered!
%
```

If **printmessage()** were turned into a remote procedure, it could be called from anywhere in the network. It would be nice to be able to simply insert a keyword such as *remote* in front of a procedure to turn it into a remote procedure. Unfortunately, you have to live within the constraints of the C language, since it existed long before RPC did. But even without language support, it's not very difficult to make a procedure remote.

In general, it's necessary to figure out what the types are for all procedure inputs and outputs. In this case, there is a procedure, **printmessage()**, that takes a string as input and returns an integer as output. Knowing this, you can write a protocol specification in RPC language that describes the remote version of **printmessage()**:

```
/*
 * msg.x: Remote message printing protocol
 */
program MESSAGEPROG {
    version MESSAGEVERS {
        int PRINTMESSAGE(string) = 1;
    } = 1;
} = 99;
```

Remote procedures are part of remote programs, so an entire remote program was declared here that contains the single procedure PRINTMESSAGE. This procedure was declared to be in version 1 of the remote program. No null procedure (procedure 0) is necessary, because *rpcgen* generates it automatically.

Note: Notice that everything is declared with all capital letters. This is not required, but it is a good convention to follow.

Notice that the argument type is *string* and not *char **. This is because *char ** is ambiguous in C. Programmers usually intend it to mean a null-terminated string of characters, but it could also represent a pointer to a single character or a pointer to an array of characters. In RPC language, a null-terminated string is unambiguously called a *string*.

Next, define the remote procedure itself. The following example implements the PRINTMESSAGE procedure declared above:

```
/*
 *  msg_proc.c: implementation of the remote
 *  procedure "printmessage"
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <rpc/rpc.h>  /* Required. */
#define _RPCGEN_SVC   /*Selects server function prototypes.*/
#include "msg.h"      /* This will be generated by rpcgen. */

/*  Remote version of "printmessage" */
int *printmessage_1(msg, UNUSED)
/* UNUSED specified for prototype agreement */
char **msg;
struct svc_req *UNUSED;
{
    static int result;  /* must be static! */

    FILE *f;

    f = fopen("/dev/console", "w");
    if (f == NULL) {            /* failure! */
        result = 0;
        return (&result);
    }

    fprintf(f, "%s\n", *msg);   /* success! */
    fclose(f);
    result = 1;
    return (&result);
}
```

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

- **printmessage_1()** takes a pointer to a string instead of a string itself, which is true of all remote procedures; they always take pointers to their arguments rather than the arguments themselves.
- **printmessage_1()** returns a pointer to an integer instead of returning an integer itself. This is also generally true of remote procedures: they return a pointer to their results.
- **printmessage_1()** has *_1* appended to its name. In general, all remote procedures called by *rpcgen* are named using the following rule: the name in the program definition (here PRINTMESSAGE) is converted to all lowercase letters and an underscore (_) is appended to it, followed by the version number (here, 1).

Finally, declare the main client program that will call the remote procedure:

```
/*
 * rprintmsg.c: remote version of "printmsg.c"
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <rpc/rpc.h>  /* Required. */
#define _RPCGEN_CLNT  /*selects client function prototypes*/
#include "msg.h"      /* This will be generated by rpcgen. */

void main(argc, argv)
int argc;
char **argv;
{
    CLIENT *cl;
    int *result;
    char *server;
    char *message;

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

    /* save values of command line arguments */
    server = argv[1];
    message = argv[2];
```

```
    /* Create client "handle" used for calling MESSAGEPROG
     * on the server designated on the command line. We tell
     * the RPC package to use "tcp" when contacting the
     * server.
     */
    cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS,
                     "tcp");
    if (cl == NULL) {
        /* Couldn't establish connection with the server.
         * Print error message and exit.
         */
        clnt_pcreateerror(server);
        exit(1);
    }

    cl->cl_auth = authunix_create_default();

    /* Call the remote procedure "printmessage" on the
     * server */
    result = printmessage_1(&message, cl);
    if (result == NULL) {
        /*
         * An error occured while calling the server.
         * Print error message and exit.
         */
        clnt_perror(cl, server);
        exit(1);
    }

    /* Okay, we've *called( the server; now, did it print
     * the message? */
    if (*result == 0) {
        /*  The server was unable to print our message.
         *  Print error message and exit.
         */
        fprintf(stderr,
                "%s: %s couldn't print your message\n",
                argv[0], server);
        exit(1);
    }
    /* The message was printed on the server's console */
    printf("Message delivered to %s!\n", server);
    exit(0);
}
```

There are two things to note:

- A client *handle* is created using the RPC library routine **clnt_create()**. This client handle will be passed to the stub routines that call the remote procedure.
- 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.

Here's how to put the pieces together:

```
% rpcgen -P msg.x
% cc rprintmsg.c msg_clnt.c -o rprintmsg
rprintmsg.c:
msg_clnt.c:
% cc msg_proc.c msg_svc.c -o msg_server
msg_proc.c:
msg_svc.c:
%
```

Note: The command-line option **–lsun** used to be required to compile these programs, but it should no longer be used because *libsun* has been incorporated into *libc*.

Two programs were compiled: the client program *rprintmsg* and the server program *msg_server*. Before compilation, *rpcgen* was used to fill in the missing pieces. The following explains what *rpcgen* did with the input file *msg.x*:

- *rpcgen* created a header file called *msg.h* that contained *#define*s for MESSAGEPROG, MESSAGEVERS, and PRINTMESSAGE for use in the other modules.
- *rpcgen* created client stub routines in the *msg_clnt.c* file. In this case, there is only one, the **printmessage_1()** that was referred to from the *printmsg* client program. The name of the output file for client stub routines is always formed in this way: if the name of the input file is *foo.x,* the client stubs output file is called *foo_clnt.c*.
- *rpcgen* created the server program that calls **printmessage_1()** in *msg_proc.c*. This server program is named *msg_svc.c*. The rule for naming the server output file is similar to the previous one: for an input file called *foo.x*, the output server file is named *foo_svc.c*.

The following shows the contents of the file *msg_svc.c*, as generated by *rpcgen* from the file *msg.x*. Note that all registerable RPC server functions take the same parameters as the function **messageprog_1(),** shown in this example.

```
/* msg_svc.c
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <bstring.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpc/pmap_clnt.h>
#define _RPCGEN_SVC
#include "msg.h"

static void messageprog_1(struct svc_req *, SVCXPRT *);

main(void)
{
    register SVCXPRT *transp;

    (void) pmap_unset(MESSAGEPROG, MESSAGEVERS);

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
        fprintf(stderr, "cannot create udp service.");
        exit(1);
    }
    if (!svc_register(transp, MESSAGEPROG, MESSAGEVERS, messageprog_1, IPPROTO_UDP)) {
        fprintf(stderr, "unable to register (MESSAGEPROG, MESSAGEVERS, udp).");
        exit(1);
    }

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
        fprintf(stderr, "cannot create tcp service.");
        exit(1);
    }
    if (!svc_register(transp, MESSAGEPROG, MESSAGEVERS, messageprog_1, IPPROTO_TCP)) {
        fprintf(stderr, "unable to register (MESSAGEPROG, MESSAGEVERS, tcp).");
        exit(1);
    }

    svc_run();
    fprintf(stderr, "svc_run returned");
    exit(1);
    /* NOTREACHED */
}
```

```
static void
messageprog_1(struct svc_req *rqstp, SVCXPRT *transp)
{
    union __svcargun {
        char *printmessage_1_arg;
    } argument;
    xdrproc_t xdr_argument, xdr_result;
    void *result;
    typedef void *(*__svcproc_t)(union __svcargun *, struct svc_req *);
    __svcproc_t local;

    switch (rqstp->rq_proc) {
    case NULLPROC:
        (void) svc_sendreply(transp, (xdrproc_t)xdr_void, (char *)NULL);
        return;

    case PRINTMESSAGE:
        xdr_argument = (xdrproc_t)xdr_wrapstring;
        xdr_result = (xdrproc_t)xdr_int;
        local = (__svcproc_t) printmessage_1;
        break;

    default:
        svcerr_noproc(transp);
        return;
    }
    bzero((char *)&argument, sizeof(argument));
    if (!svc_getargs(transp, xdr_argument, &argument)) {
        svcerr_decode(transp);
        return;
    }
    result = (*local)(&argument, rqstp);
    if (result != NULL && !svc_sendreply(transp, xdr_result, result)) {
        svcerr_systemerr(transp);
    }
    if (!svc_freeargs(transp, xdr_argument, &argument)) {
        fprintf(stderr, "unable to free arguments");
        exit(1);
    }
    return;
}
```

Now you're ready to have some fun. For this example, the local machine is called *bonnie* and the remote machine is called *clyde.* First, copy the server to a remote machine and run it:

```
clyde% msg_server &
```

Note: Server processes are run in the background because they never exit.

Next, on the local machine (*bonnie*), print a message on the remote machine's console:

```
bonnie% rprintmsg clyde "Hello, clyde"
Message delivered to clyde!
bonnie%
```

The message will print on *clyde*'s console. You can print a message on anybody's console (including your own) with this program if you are able to copy the server to that person's machine and run it.

Generating XDR Routines

The previous example demonstrated the automatic generation of client and server RPC code. You can also use *rpcgen* to generate XDR routines; that is, the routines necessary to convert local data structures into network format and vice versa. This example presents a complete RPC service, a remote directory listing service; *rpcgen* is used to generate stub routines and to generate the XDR routines.

This code is an example of a protocol description file:

```
/* dir.x: Remote directory listing protocol */
const MAXNAMELEN = 255; /*maximum length of directory entry*/
typedef string nametype<MAXNAMELEN>;  /* directory entry */
typedef struct namenode *namelist;    /* a link in 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. */
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;
} = 76;
```

Note: Define types (such as *readdir_res* in the example above) by using the *struct, union,* and *enum* keywords; these keywords should not be used in subsequent declarations of variables of those types. For example, if you define a union *foo,* you should declare using only *foo* and not *union foo.* In fact, *rpcgen* compiles RPC unions into C structures; it is an error to declare them using the *union* keyword.

Running *rpcgen –P* on *dir.x* creates four output files. Three are the same as before: a header file, client stub routines, and a server skeleton. The fourth output file consists of the XDR routines necessary for converting the data types you declared into XDR format and vice versa. These routines are output in the file *dir_xdr.c*.

This example implements the READDIR procedure:

```
/* dir_proc.c: remote readdir implementation */
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <rpc/rpc.h>
#include <sys/dir.h>
#include "dir.h"

readdir_res *readdir_1(dirname, UNUSED)
/* UNUSED specified for prototype agreement */
nametype *dirname;
struct svc_req *UNUSED;
{
    DIR *dirp;
    struct direct *d;
    namelist nl;
    namelist *nlp;
    static readdir_res res;     /* must be static! */

    /* Open directory */
    dirp = opendir(*dirname);
    if (dirp == NULL) {
        res.errno = errno;
        return (&res);
    }
    /* Free previous result */
    xdr_free(xdr_readdir_res, &res);

    /* Collect directory entries. Memory allocated here
     * will be freed by xdr_free next time readdir_1 is
     * called
     */
    nlp = &res.readdir_res_u.list;
    while (d = readdir(dirp)) {
        nl = *nlp = (namenode *) malloc(sizeof(namenode));
        nl->name = strdup(d->d_name);
        nlp = &nl->next;
    }
    *nlp = NULL;

    /* Return the result */
    res.errno = 0;
    closedir(dirp);
    return (&res);
}
```

This example shows the client-side program to call the server:

```
/*
 * rls.c: Remote directory listing client
 */
#include <stdio.h>
#include <errno.h>
#include <rpc/rpc.h>   /* always need this */
#define _RPCGEN_CLNT   /*selects client function prototypes*/
#include "dir.h"       /* will be generated by rpcgen */

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);
    }
    /* Remember what command line arguments refer to */
    server = argv[1];
    dir = argv[2];
    /* Create client "handle" used for calling
     * MESSAGEPROG on the server designated on the
     * command line. We tell the RPC package to use the
     * "tcp" protocol when contacting the server.
     */
    cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");
    if (cl == NULL) {
        /* Couldn't establish connection with server.
         * Print error message and close up shop.
         */
        clnt_pcreateerror(server);
        exit(1);
    }
```

```
    /* Call the remote procedure readdir() on the server */
    result = readdir_1(&dir, cl);
    if (result == NULL) {
        /* An error occurred while calling the server.
         * Print error message and exit.
         */
        clnt_perror(cl, server);
        exit(1);
    }
    /* Okay, the remote procedure was called successfully. */

    if (result->errno != 0) {
        /* A remote system error occurred. Print error
         * message and exit.
         */
        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);
}
```

Finally, compile everything and run the server:

```
bonnie% rpcgen -P dir.x
bonnie% cc rls.c dir_clnt.c dir_xdr.c -o rls
rls.c:
dir_clnt.c:
dir_xdr.c:
bonnie% cc dir_svc.c dir_proc.c dir_xdr.c -o dir_svc
dir_svc.c:
dir_proc.c:
dir_xdr.c:
bonnie% dir_svc &
```

Now run the client from another machine:

```
clyde% rls bonnie /usr/pub
.
..
apseqnchar
cateqnchar
eqnchar
psceqnchar
terminals
clyde%
```

You can test the client program and the server procedure together as a single program by linking them to each other, rather than linking to the client and server stubs. The procedure calls will be executed as ordinary local procedure calls, and the program can be debugged with a local debugger such as *dbx*. When the program is working, the client program can be linked to the client stub produced by *rpcgen*, and the server procedures can be linked to the server stub produced by *rpcgen*.

Note that if you link the programs in this way, you may want to comment out calls to RPC library routines, and have client-side routines call server routines directly.

The C Preprocessor

The C preprocessor is run on all input files before they are compiled, so all preprocessor directives are legal within a *.x* file.

Four symbols may be defined, depending on which output file is being generated. These symbols are listed in Table 4-1.

Table 4-1 C Preprocessor Symbol Definition

Symbol	Usage
RPC_CLNT	for client stub output
RPC_HDR	for header file output
RPC_SVC	for server skeleton output
RPC_XDR	for XDR routine output

rpcgen also does some preprocessing of its own. Any line that begins with a percent sign (%) is passed directly into the output file, without any interpretation of the line. The following example demonstrates the preprocessing features:

```
/* time.x: Remote time protocol */
program TIMEPROG {
        version TIMEVERS {
            unsigned int TIMEGET(void) = 1;
        } = 1;
} = 44;

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

Note: The percent (%) feature is not generally recommended, since there is no guarantee that the compiler will put the output where you intended.

pcgen Programming Notes

This section describes ANSI C prototypes, timeout changes, broadcast on the server side, and information passed to server procedures.

Generating ANSI C Prototypes

To generate prototyped XDR and stub function declarations and definitions suitable for ANSI C, use the **–P** option to *rpcgen*—see rpcgen(1). The prototypes for the client and server-side stubs are different; their declarations in the generated header file are conditionally compiled with the value _RPCGEN_CLNT or _RPCGEN_SVC. If you write your own client or server code, you must define the appropriate value in your source files before including the generated header file.

For instance, in the remote message example from the "Changing Local Procedures to Remote Procedures" section, the file for client code uses:

```
#define _RPCGEN_CLNT
#include "msg.h"
```

and the file for server code uses:

```
#define _RPCGEN_SVC
#include "msg.h"
```

Client-side Timeout Changes

RPC sets a default timeout of 25 seconds for RPC calls when **clnt_create()** is used. This timeout may be changed using **clnt_control()**. This code fragment demonstrates the use of **clnt_control()**:

```
struct timeval tv;
CLIENT *cl;
cl = clnt_create("somehost", SOMEPROG, SOMEVERS, "tcp");
if (cl == NULL) {
    exit(1);
}
/* change timeout to 1 minute */
tv.tv_sec = 60;
tv.tv_usec = 0;
clnt_control(cl, CLSET_TIMEOUT, &tv);
```

Server-side Broadcast Handling

When a procedure is known to be called via broadcast RPC, it is usually wise for the server not to reply unless it can provide some useful information to the client. This prevents the network from being flooded by useless replies.

To prevent the server from replying, a remote procedure can return NULL as its result, and the server code generated by *rpcgen* will detect the NULL and not send out a reply.

The next example shows a simple procedure that replies only if it thinks it is an NFS server. It assumes an NFS client won't have this file, which may not be valid.

```
void *reply_if_nfsserver(void)
{
    char notnull;  /* just here so you can use its address */
    if (access("/etc/exports", F_OK) < 0) {
        return (NULL);    /* prevent RPC from replying */
    }
    /*return non-null pointer so RPC will send out a reply*/
    return ((void *)&notnull);
}
```

Note that if a procedure returns type *void **, it must return a non-NULL pointer if it wants RPC to reply to it.

Other Information Passed to Server Procedures

Server procedures will often want to know more about an RPC call than just its arguments. For example, getting authentication information is important to procedures that want to implement some level of security.

This extra information is actually supplied to the server procedure as a second argument, as shown in the following example. The previous **printmessage_1()** procedure has been rewritten to allow only root users to print a message to the console:

```
/*
 *  msg_proc.c: implementation of the remote
 *  procedure "printmessage"
 */
#include <stdio.h>
#include <syslog.h>
#include <pwd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <rpc/rpc.h>  /* Required. */
#define _RPCGEN_SVC   /*Selects server function prototypes.*/
#include "msg.h"      /* This will be generated by rpcgen. */

#define MAX_LOG_MESSAGE 160

int printmessage_1_client_ok(struct svc_req *rqstp);

/*  Remote version of "printmessage" */
int *printmessage_1(msg, rq)
char **msg;
struct svc_req *rq;
{
    static int result;    /* Must be static or external */
    FILE *f;
    struct authunix_parms *aup;

    /* perform authentication checks on client credentials */
    if (! printmessage_1_client_ok(rq)) {
        result = 0;
        return (&result);
    }

    /*  Same code as before. */
    f = fopen("/dev/console", "w");
    if (f == NULL) {            /* failure! */
        result = 0;
        return (&result);
    }
    fprintf(f, "%s\n", *msg);   /* success! */
    fclose(f);
    result = 1;
    return (&result);
}
```

```
static int logging_successful_requests = 1;

/* This routine attempts to verify that the client user is
 * authorized access on the server host.  A true value is
 * returned to indicate that the client user is not authorized.
 * Otherwize the value returned is false. */
int
printmessage_1_client_ok(struct svc_req *rqstp)
{
    SVCXPRT *   transp  = rqstp->rq_xprt;
    char *      user    = NULL;
    uid_t       uid;
    struct authunix_parms *unix_cred;
    struct hostent *host_entry = NULL;
    struct passwd  *passwd_entry = NULL;
    char        log_message[MAX_LOG_MESSAGE];

    static u_long peer_addr        = 0;
    static u_long client_host_addr = 0;
    static char * client_host      = NULL;

    switch (rqstp->rq_cred.oa_flavor) {
    case AUTH_UNIX:
        unix_cred = (struct authunix_parms *)rqstp->rq_clntcred;
        uid = unix_cred->aup_uid;
        break;
    case AUTH_NULL:
    default:            /* invalid credentials */
        sprintf(log_message, "Rejected request, "
                 "invalid credentials, type %d", rqstp->rq_cred.oa_flavor);
        syslog(LOG_NOTICE | LOG_AUTH, log_message);
        return 0;
    }

    if (transp->xp_raddr.sin_port >= IPPORT_RESERVED) {
        sprintf(log_message, "Rejected request, "
                 "non-priviledged port %d", transp->xp_raddr.sin_port);
        syslog(LOG_NOTICE | LOG_AUTH, log_message);
        return 0;
    }
```

```
/* Determine the client host name and address. */
if (peer_addr != transp->xp_raddr.sin_addr.s_addr) {
    host_entry = gethostbyaddr(&transp->xp_raddr.sin_addr,
                               sizeof(struct in_addr),
                               AF_INET);
    if (host_entry == NULL) {
        sprintf(log_message, "Rejected request, "
                "unknown client host at address 0x%08x",
                transp->xp_raddr.sin_addr);
        syslog(LOG_NOTICE | LOG_AUTH, log_message);
        return 0;
    }
    peer_addr = transp->xp_raddr.sin_addr.s_addr;
    if (client_host != NULL) {
        free(client_host);
    }
    client_host = strdup(host_entry->h_name);
    client_host_addr = *(u_long *) host_entry->h_addr;
}

/* Determine the user name. */
passwd_entry = getpwuid(uid);
if (passwd_entry == NULL) {
    sprintf(log_message, "Rejected request, "
            "unknown uid %d from host %s",
            uid, client_host);
    syslog(LOG_NOTICE | LOG_AUTH, log_message);
    return 0;
}
user = strdup(passwd_entry->pw_name);
if (passwd_entry->pw_passwd != NULL &&
    *passwd_entry->pw_passwd != '\0' &&
    ruserok(client_host, 1, user, "root") < 0) {
    sprintf(log_message, "Rejected request by %s at %s",
            user, client_host);
    syslog(LOG_NOTICE | LOG_AUTH, log_message);
    free(user);
    return 0;
}
```

```
    if (logging_successful_requests) {
        if (user != NULL) {
            sprintf(log_message, "Granted request by %s at %s",
                    user, client_host);
        } else {
            sprintf(log_message, "Granted request by uid %d at %s",
                    uid, client_host);
        }
        syslog(LOG_INFO | LOG_AUTH, log_message);
    }
    if (user != NULL) {
        free(user);
    }
    return 1;
} /* printmessage_1_client_ok */
```

Chapter 5

RPC Programming Guide

This chapter is for programmers who want to write network applications using RPC. For most applications, you can use the *rpcgen* compiler, thus avoiding the need to understand much of the information in this chapter. (Chapter 4, "Programming with rpcgen," contains the source for a working RPC service, which uses *rpcgen* to generate XDR routines and client and server stubs.)

Topics in this chapter include:

- programming in each RPC layer
- RPC features such as broadcast, batching, and authentication
- examples of other uses of RPC

Note: For a general introduction to RPC, see Chapter 3, "Introduction to RPC Programming." For information about XDR and RPC language, see Chapter 6, "XDR and RPC Language Structure." For a description of the RPC Protocol, see Appendix A, "RPC Protocol Specification." For details about the routines described in this chapter, see rpc(3R).

The Layers of RPC

This section presents detailed information about programming in the three RPC layers (see Chapter 3, "Introduction to RPC Programming," for background information about the RPC layers).

The Highest Layer of RPC

The highest layer of RPC is transparent to the operating system, machine, and network upon which it is run and consists of RPC library-based services. Suppose you're writing a program that needs to know how many users are logged into a remote machine.

You can do this by calling the RPC library routine **rnusers()**, as shown in this code fragment:

```
/*
 * howmany.c
 */

#include <stdio.h>

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

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

RPC library routines in C, such as **rnusers()**, are included in the DSO *librpcsvc.so*. (For more information about DSOs, see the *IRIX System Programming Guide*.) Thus, you can compile the above program with *cc*:

```
% cc howmany.c -lrpcsvc -o howmany
```

Note: See "Compiling BSD and RPC Programs" in Chapter 1 for other compiling hints.

The Middle Layer of RPC

The middle layer of RPC consists of routines used for most applications. In this layer, the user can make remote procedure calls to routines on other machines without considering details about the socket interface, the UNIX system, or other low-level implementation mechanisms.

The simplest interface, which explicitly makes RPC calls, uses the **callrpc()** and **registerrpc()** functions. Another way to determine the number of remote users is shown in this example, which can be compiled in the same way as the previous example:

```
/*
 * howmany2.c
 */

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

main(int argc, char **argv)
{
    unsigned long nusers;
    int stat;
    if (argc != 2) {
        fprintf(stderr, "usage: howmany2 hostname\n");
        exit(1);
    }

    if (stat = callrpc(argv[1], RUSERSPROG, RUSERSVERS,
                       RUSERSPROC_NUM, xdr_void, 0,
                       xdr_u_long, &nusers) != 0) {
        clnt_perrno(stat);
        exit(1);
    }
    printf("%d users on %s\n", nusers, argv[1]);
    exit(0);
}
```

Each RPC procedure is uniquely defined by a program number, version number, and procedure number (see "Assigning RPC Program Numbers" in Chapter 3 for details). The program number specifies a group of related remote procedures, each of which has a different procedure number. Each program also has a version number, so when a minor change is made to a remote service (such as adding a new procedure), a new program number doesn't have to be assigned.

The simplest way to make a remote procedure call is with the **callrpc()** routine. **callrpc()** has eight parameters:

- The first parameter is the name of the remote server machine.
- The next three parameters identify the procedure to be called and consist of the program, version, and procedure numbers.
- The fifth parameter is an XDR filter.
- The sixth parameter is an argument to be encoded and passed to the remote procedure.
- The seventh parameter is a filter for decoding the results returned by the remote procedure.
- The last parameter is a pointer to the place where the procedure's results are to be stored.

Multiple arguments and results are handled by embedding them in structures. If **callrpc()** completes successfully, it returns zero; otherwise, it returns a nonzero value. The return codes (of type cast into an integer) are found in <*rpc/clnt.h*>.

Since data types may be represented differently on different machines, **callrpc()** needs both the type of the RPC argument and a pointer to the argument itself (and similarly for the result). For RUSERSPROC_NUM, the return value is an *unsigned long*. So, **callrpc()** has *xdr_u_long* as its first return parameter, which says that the result is of type *unsigned long*, and *&nusers* as its second return parameter, which is a pointer to where the long result will be placed. Since RUSERSPROC_NUM takes no argument, the argument parameter of **callrpc()** is *xdr_void*.

After trying several times to deliver a message, if **callrpc()** gets no answer, it returns with an error code. The delivery mechanism is UDP. Methods for adjusting the number of retries or for using a different protocol require you to use the lowest layer of the RPC library (see "The Lowest Layer of RPC" on page 111).

The remote server procedure corresponding to the preceding example might look like this:

```
void *nuser(indata)
char *indata;
{
    static int nusers;
    /* Code here to compute the number of users
     * and place result in variable nusers.
     */
    return ((void *)&nusers);
}
```

It takes one argument, which is a pointer to the input of the remote procedure call (ignored in the example), and it returns a pointer to the result.

Normally, a server registers all of the RPC calls it plans to handle and then goes into an infinite loop waiting to service requests. In this example, there is only a single procedure to register, so the main body of the server looks like this:

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

void *nuser();

main()
{
    registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
                nuser, xdr_void, xdr_u_long);
    svc_run();                    /* never returns */
    fprintf(stderr, "Error: svc_run returned!\n");
    exit(1);
}
```

The **registerrpc()** routine establishes what C procedure corresponds to each RPC procedure number. The first three parameters—RUSERSPROG, RUSERSVERS, and RUSERSPROC_NUM—are the program, version, and procedure numbers of the remote procedure to be registered; *nuser* is the name of the C procedure implementing it; and *xdr_void* and *xdr_u_long* are the XDR filters for the remote procedure's arguments and results, respectively. (Multiple arguments or multiple results are passed as structures.)

Only the UDP transport mechanism can use **registerrpc()**; thus, **registerrpc()** is always safe in conjunction with calls generated by **callrpc()**.

Note: The UDP transport mechanism can only deal with arguments and results that are less than 8 kilobytes long.

After registering the local procedure, the server program's main procedure calls **svc_run()**, the RPC library's remote procedure dispatcher. It is this function that calls the remote procedures in response to RPC call messages. Note that the dispatcher takes care of decoding remote procedure arguments and encoding results, using the XDR filters specified when the remote procedure was registered.

Passing Arbitrary Data Types

In the previous example, the RPC call passes a single *unsigned long*. RPC can handle arbitrary data structures, regardless of different machines' byte order or structure-layout conventions, by always converting them to XDR before sending them over the network. (The process of converting from a particular machine representation to XDR format is called *serializing*, and the reverse is called *deserializing*.)

The *type* field parameters passed to **callrpc()** and **registerrpc()** can be built-in procedures like **xdr_u_long()** or user-supplied procedures. XDR has the following built-in *type* routines that can be used with **callrpc()** and **registerrpc()**:

```
xdr_int()       xdr_u_int()       xdr_enum()
xdr_long()      xdr_u_long()      xdr_bool()
xdr_short()     xdr_u_short()     xdr_wrapstring()
xdr_char()      xdr_u_char()
```

Note that the routine **xdr_string()** exists but cannot be used with **callrpc()** and **registerrpc()**, which pass only two parameters to their XDR routines. **xdr_wrapstring()** has only two parameters and is thus okay; it calls **xdr_string()**.

This is an example of a user-defined *type* routine:

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

If you want to send and receive this structure, call **callrpc()** like this:

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_simple,
        &simple, xdr_simple, &simple);
```

In this case, **xdr_simple()** is written as:

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

xdr_simple(XDR *xdrsp, struct simple *simplep)
{
    if (!xdr_int(xdrsp, &simplep->a))
        return(0);
    if (!xdr_short(xdrsp, &simplep->b))
        return(0);
    return(1);
}
```

An XDR routine returns a nonzero value (which means "true" in C) if it completes successfully; zero otherwise.

Note: This section gives only a few examples of implementing XDR. For more information, see Chapter 7, "XDR Programming Notes."

In addition to the built-in routines, XDR has these prefabricated building blocks:

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

To send a variable array of integers, you might package them as a structure:

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

Next, you could make an RPC call something like this:

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_varintarr,
        &arr, xdr_varintarr, &arr);
```

In this case, **xdr_varintarr()** is defined as:

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

This routine takes as parameters the XDR handle, a pointer to the array, a pointer to the size of the array, the maximum allowable array size, the size of each array element, and an XDR routine for handling each array element.

If the size of the array is known in advance, you can use **xdr_vector()**, which serializes fixed-length arrays:

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

XDR always converts quantities to four-byte multiples when it is serializing. Thus, if either of the preceding examples involved characters instead of integers, each character would occupy 32 bits, which is the reason for the **xdr_bytes()** routine. **xdr_bytes()** is like **xdr_array()**, except it packs characters; **xdr_bytes()** has four parameters, similar to the first four parameters of **xdr_array()**.

For null-terminated strings, there is also the **xdr_string()** routine. **xdr_string()** is the same as **xdr_bytes()** without the *length* parameter. When serializing, the string length is taken from **strlen()**; when deserializing, a null-terminated string is created.

In this final example of the middle layer, a call is made to the previously written **xdr_simple()**, as well as to the built-in functions **xdr_string()** and **xdr_reference()**:

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

xdr_finalexample(XDR *xdrsp, struct finalexample *finalp)
{
    int i;
    if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
        return (0);
    if (!xdr_reference(xdrsp, &finalp->simplep,
                       sizeof(struct simple), xdr_simple))
        return (0);
    return (1);
}
```

Note that you could just as easily call **xdr_simple()** instead of **xdr_reference()**.

The Lowest Layer of RPC

The lowest layer of RPC is used for more-sophisticated applications. In this section, you'll see how to change defaults by using the lowest layer of the RPC library.

Note: This section assumes that you are familiar with socket-related concepts and the socket library (see Chapter 2, "Sockets-based Communication").

You may need to use the lowest layer of RPC in one of the following instances:

- To use TCP (the highest layer uses UDP, which restricts RPC calls to 8 kilobytes of data). Using TCP permits calls to send long streams of data (see "TCP" on page 135).
- To allocate and free memory while serializing or deserializing with XDR routines. There is no call at the highest level to let you free memory explicitly. See "Memory Allocation with XDR" on page 117.
- To perform authentication on either the client or server side, by supplying credentials or verifying them. See "Authentication" on page 124.

More Information about the Server

There are a number of assumptions built into **registerrpc()**. One is that you are using the UDP datagram protocol. Another is that you don't want to do anything unusual while deserializing, since deserialization is automatic and occurs before the user's server routine is called.

The server for the following program is written using the lowest layer of RPC, which does not make these assumptions:

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

main()
{
    SVCXPRT *transp;
    void nuser();
```

```
    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL){
        fprintf(stderr, "can't create an RPC server\n");
        exit(1);
    }
    pmap_unset(RUSERSPROG, RUSERSVERS);
    if (!svc_register(transp, RUSERSPROG, RUSERSVERS, nuser,
                      IPPROTO_UDP)) {
        fprintf(stderr, "can't register RUSER service\n");
        exit(1);
    }
    svc_run();  /* never returns */
    fprintf(stderr, "should never reach this point\n");
    exit(1);
}

void nuser(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{
    unsigned long nusers;

    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can't reply to RPC call\n");
            exit(1);
        }
        return;
    case RUSERSPROC_NUM:
        /* Code here to compute the number of users and
         * assign to the variable nusers
         */
        if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
            fprintf(stderr, "can't reply to RPC call\n");
            exit(1);
        }
        return;
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

In this example, the server gets a transport handle, which is used for sending RPC messages. **registerrpc()** uses **svcudp_create()** to get a UDP handle. If you require a reliable protocol, call **svctcp_create()** instead. If the argument to **svcudp_create()** is RPC_ANYSOCK, the RPC library creates a socket on which to send out RPC calls. Otherwise, **svcudp_create()** expects its argument to be a valid socket number.

If you specify your own socket, it can be bound or unbound. If it is bound to a port by the user, the port numbers of **svcudp_create()** and **clntudp_create()** (the low-level client routine) must match.

If you specify RPC_ANYSOCK for a socket, the RPC library routines will open sockets. Otherwise, they will expect the caller to do so. The **svcudp_create()** and **clntudp_create()** routines will cause RPC library routines to bind their sockets if they are not bound already.

A service may choose to register its port number with the local port mapper service. This is done by specifying a nonzero protocol number in **svc_register()**. Incidentally, a client can discover the server's port number by consulting the port mapper on the server's machine. This can be done automatically by specifying a zero port number in **clntudp_create()** or **clnttcp_create()**.

After creating a SVCXPRT, the next step is to call **pmap_unset()** so that if the *nusers* server crashed earlier, any previous trace of it is erased before restarting. More precisely, **pmap_unset()** erases the entry for RUSERS from the port mapper's tables.

Finally, you associate the program number for *nusers* with the procedure **nuser()**. The final argument to **svc_register()** is normally the protocol being used, which in this case is IPPROTO_UDP. Notice that unlike **registerrpc()**, there are no XDR routines involved in the registration process. In addition, registration is done on the program level rather than the procedure level.

The user routine **nuser()** must call and dispatch the appropriate XDR routines based on the procedure number. Note that two things are handled by **nuser()** that **registerrpc()** handles automatically:

- A simple test to detect whether a remote program is running: call procedure NULLPROC (currently zero), which returns with no arguments.
- A check for invalid procedure numbers. If one is detected, **svcerr_noproc()** is called to handle the error.

The user service routine serializes the results and returns them to the RPC caller via **svc_sendreply()**. Its first parameter is the SVCXPRT handle, the second parameter is the XDR routine, and the third parameter is a pointer to the data to be returned.

Not illustrated previously is how a server handles an RPC program that passes data. For example, we could add a procedure RUSERSPROC_BOOL, which has an argument *nusers*, and returns TRUE or FALSE, depending on whether the number of users logged in is equal to *nusers*. The procedure looks something like this:

```
case RUSERSPROC_BOOL: {
    int bool;
    unsigned nuserquery;

    if (!svc_getargs(transp, xdr_u_int, &nuserquery) {
        svcerr_decode(transp);
        return;
    }
    /* Insert code here to set nusers = number of users */
    if (nuserquery == nusers)
        bool = TRUE;
    else
        bool = FALSE;
    if (!svc_sendreply(transp, xdr_bool, &bool)) {
        fprintf(stderr, "can't reply to RPC call\n");
        exit(1);
    }
    return;
}
```

The relevant routine is **svc_getargs()**, which takes a SVCXPRT handle, the XDR routine, and a pointer to where the input is to be placed as arguments.

More Information about the Client

When you use **callrpc()**, you have no control over the RPC delivery mechanism or the socket used to transport the data. To illustrate how the lowest layer of RPC lets you adjust these parameters, consider the following code sample, which calls the *nusers* service:

```
/*
 * howmany3.c
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.h>
```

```
#include <rpcsvc/rusers.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

main(int argc, char **argv)
{
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    unsigned long nusers;

    if (argc != 2) {
        fprintf(stderr, "usage: howmany3 hostname\n");
        exit(1);
    }
    if ((hp = gethostbyname(argv[1])) == NULL) {
        herror(argv[1]);
        exit(1);
    }
    pertry_timeout.tv_sec = 3;
    pertry_timeout.tv_usec = 0;
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
          hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port =  0;
    if ((client = clntudp_create(&server_addr, RUSERSPROG,
         RUSERSVERS, pertry_timeout, &sock)) == NULL) {
        clnt_pcreateerror("clntudp_create");
        exit(1);
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client,RUSERSPROC_NUM,xdr_void,0,
                          xdr_u_long,&nusers, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "rpc");
        exit(1);
    }

    printf("%d users on %s\n", nusers, argv[1]);
```

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

The low-level version of **callrpc()** is **clnt_call()**, which takes a CLIENT pointer rather than a host name. The parameters to **clnt_call()** are a CLIENT pointer, the procedure number, the XDR routine for serializing the argument, a pointer to the argument, the XDR routine for deserializing the return value, a pointer to where the return value will be placed, and the time in seconds to wait for a reply.

The CLIENT pointer is encoded with the transport mechanism. **callrpc()** uses UDP; thus it calls **clntudp_create()** to get a CLIENT pointer. To specify TCP/IP, use **clnttcp_create()**.

The parameters to **clntudp_create()** are the server address, the program number, the version number, a timeout value (how long to wait before trying again), and a pointer to a socket. The final argument to **clnt_call()** is the total time to wait for a response. Thus, the number of tries is the **clnt_call()** timeout divided by the **clntudp_create()** timeout.

Note that the **clnt_destroy()** call always deallocates the space associated with the CLIENT handle. It closes the socket associated with the CLIENT handle, however, only if the RPC library opened it. If the socket was opened by the user, it stays open. This makes it possible, in cases where there are multiple client handles using the same socket, to destroy one handle without closing the socket that other handles are using.

To make a stream connection, the call to **clntudp_create()** is replaced with a call to **clnttcp_create()***:*

```
clnttcp_create(&server_addr, prognum, versnum, &socket,
               inputsize, outputsize);
```

There is no timeout argument; instead, the receive and send buffer sizes must be specified. When the **clnttcp_create()** call is made, a TCP connection is established. All RPC calls using that CLIENT handle use this connection. The server side of an RPC call using TCP has **svcudp_create()** replaced by **svctcp_create()**:

```
transp = svctcp_create(RPC_ANYSOCK, 0, 0);
```

The last two arguments to **svctcp_create()** are send and receive sizes, respectively. If 0 is specified for either argument, the system chooses a reasonable default.

Memory Allocation with XDR

In addition to input and output, XDR routines do memory allocation. For this reason, the second parameter of **xdr_array()** is a pointer to an array, rather than the array itself. If it is NULL, **xdr_array()** allocates space for the array and returns a pointer to it, putting the size of the array in the third argument. For example, consider the following XDR routine, **xdr_chararr1()**, which deals with a fixed array of bytes with length SIZE:

```
xdr_chararr1(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:

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

In this case, *chararr* has already allocated space.

If you want XDR to do the allocation, you have to rewrite the routine; for example:

```
xdr_chararr2(XDR *xdrsp, char **chararrp)
{
    int len;
    len = SIZE;
    return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

The RPC call might then look like this:

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

Note that after being used, the character array can be freed with **svc_freeargs()**, which will not attempt to free any memory if the variable indicating memory is NULL. For example, in the routine **xdr_finalexample()** (described in "Passing Arbitrary Data Types" on page 108), if *finalp->string* is NULL, it is not freed. The same is true for *finalp->simplep*.

To summarize, each XDR routine is responsible for serializing, deserializing, and freeing memory. When an XDR routine is called from **callrpc()**, the serializer is used. When called from **svc_getargs()**, the deserializer is used. When called from **svc_freeargs()**, the memory deallocator is used.

When building simple examples like the ones in this section, a user doesn't have to worry about the three modes. Chapter 7, "XDR Programming Notes," provides examples of more sophisticated XDR routines that determine which of the three modes they are in order to function correctly.

Other RPC Features

This section discusses some other aspects of RPC that can be useful to RPC programmers.

Select on the Server Side

Suppose a process is processing RPC requests while performing some other activity. If the other activity involves periodically updating a data structure, the process can set an alarm signal before calling **svc_run()**. But if the other activity involves waiting on a file descriptor, the **svc_run()** call won't work. The following is the code for **svc_run()***:*

```
void svc_run()
{
    fd_set readfds;
    int dtbsz = getdtablesize();
    for (;;) {
        readfds = svc_fdset;
        switch (select(dtbsz, &readfds, NULL,NULL,NULL)) {
        case -1:
            if (errno == EINTR)
                continue;
            perror("select");
            return;
        case 0:
            break;
        default:
            svc_getreqset(&readfds);
        }
    }
}
```

You can bypass **svc_run()** and call **svc_getreqset()** yourself. All you need to know are the file descriptors of the sockets associated with the programs you are waiting on. Thus, you can have your own **select()** that waits on both the RPC socket and your own descriptors. Note that *svc_fdset* is a bit mask of all the file descriptors that RPC is using for services. It can change whenever *any* RPC library routine is called, because descriptors are constantly being opened and closed, such as for TCP connections.

Broadcast RPC

You cannot do broadcast RPC without a port mapper, which converts RPC program numbers into UDP or TCP port numbers; see portmap(1M) or rpcbind(1M) for more information.

The main differences between broadcast RPC and normal RPC calls are:

- Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answers from each responding machine).
- Broadcast RPC can be supported only by packet-oriented (connectionless) transport protocols, such as UDP/IP.
- The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC never knows.
- All broadcast messages are sent to the port mapper port. Thus, only services that register themselves with their port mapper are accessible via the broadcast RPC mechanism.
- Broadcast requests are limited in size to the Maximum Transfer Unit (MTU) of the local network. For Ethernet, the MTU is 1500 bytes. For FDDI, the MTU is 4352 bytes.

Broadcast RPC Synopsis

The following is the synopsis of broadcast RPC:

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

enum clnt_stat  clnt_stat;
    . . .
clnt_stat = clnt_broadcast(prognum, versnum, procnum,
            inproc, in, outproc, out, eachresult)
        u_long    prognum;     /* program number */
        u_long    versnum;     /* version number */
        u_long    procnum;     /* procedure number */
        xdrproc_t inproc;      /* xdr routine for args */
        void      *in;         /* pointer to args */
        xdrproc_t outproc;      /* xdr routine for results */
        void      *out;        /* pointer to results */
        bool_t  (*eachresult)();
                          /* call with each result gotten */
clnt_stat = clnt_broadcast_exp(prognum, versnum, procnum,
                               inproc, in, outproc, out,
                               eachresult,inittime,waittime)
        /* first eight parameters same as above. */
        int    inittime;        /* initial wait period */
        int    waittime;        /* total wait period */
```

The procedure **eachresult()** is called each time a valid result is obtained. It returns a boolean that indicates whether or not the client wants more responses:

```
bool_t done;

done = eachresult(resultsp, raddr)
       void *resultsp;
       struct sockaddr_in *raddr;
       /* address of machine that sent response */
```

If *done* is TRUE, broadcasting stops and **clnt_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. Use **clnt_broadcast_exp()** to control the initial and total waiting intervals. To interpret *clnt_stat* errors, feed the error code to **clnt_perrno()**.

Batching

The RPC architecture is designed so that a client sends a call to a server and waits for a reply that the call succeeded. Clients do not compute while servers are processing a call, which is inefficient if the client does not want or need an acknowledgment for every message sent. RPC batch facilities make it possible for clients to continue computing while waiting for a response.

Batching occurs when RPC messages are placed in a "pipeline" of calls to a server. Batching assumes that:

- Each RPC call in the pipeline does not require a response from the server, and the server does not send a response message.
- The pipeline of calls is transported on a reliable byte stream transport such as TCP/IP.

Since the server does not respond to every call, the client can generate new calls in parallel, with the server executing previous calls. In addition, the TCP/IP implementation can buffer many calls and send them to the server in a single **write** system call.

This overlapped execution greatly decreases the inter-process communication overhead of the client and server processes, and the total elapsed time of a series of calls.

Since the batched calls are buffered, the client should eventually do a nonbatched call to flush the pipeline.

The following is a (contrived) example of batching. Assume that a string-rendering service (such as a window system) has two similar calls: one renders a string and returns void results, while the other renders a string and remains silent. The service (using the TCP/IP transport) could look something like this:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/windows.h>  /* assumes this files exists
                              * and defines all the
                              * necessary constants.
                              */

void windowdispatch();
```

```
main()
{
    SVCXPRT *transp;

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
        fprintf(stderr, "can't create an RPC server\n");
        exit(1);
    }
    pmap_unset(WINDOWPROG, WINDOWVERS);
    if (!svc_register(transp, WINDOWPROG, WINDOWVERS,
                      windowdispatch, IPPROTO_TCP)) {
        fprintf(stderr, "can't register WINDOW service\n");
        exit(1);
    }
    svc_run();            /* never returns */
    fprintf(stderr, "should never reach this point\n");
    exit(1);
}

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");
            exit(1);
        }
        return;
    case RENDERSTRING:
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "can't decode arguments\n");
            /* tell caller it messed up */
            svcerr_decode(transp);
            break;
        }
        /* Code here to render the string s */
        if (!svc_sendreply(transp, xdr_void, NULL)) {
            fprintf(stderr, "can't reply to RPC call\n");
            exit(1);
        }
        break;
```

```
    case RENDERSTRING_BATCHED:
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "can't decode arguments\n");
            /* We are silent in 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);
}
```

Of course, the service could have one procedure that takes the string and a boolean to indicate whether or not the procedure should respond.

In order for a client to take advantage of batching, the client must perform RPC calls on a TCP-based transport, and the actual calls must have these attributes:

- The result's XDR routine must be zero (NULL).
- The RPC call's timeout must be zero.

The following is an example of a client that uses batching to render a collection of strings; the batching is flushed when the client gets a null string:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/windows.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

main(int argc, char **argv)
{
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    char buf[1000], *s = buf;
```

```
    if ((client = clnttcp_create(&server_addr,
         WINDOWPROG, WINDOWVERS, &sock, 0, 0)) == NULL) {
        perror("clnttcp_create");

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

    /* Now flush the pipeline */

    total_timeout.tv_sec = 20;
    clnt_stat = clnt_call(client, NULLPROC, xdr_void, NULL,
                          xdr_void, NULL, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "rpc");
        exit(1);
    }
    clnt_destroy(client);
    exit(0);
}
```

Since the server does not send a message, the clients cannot be notified of any failures that occur. Therefore, clients are on their own when it comes to handling errors.

Authentication

In the examples presented so far, the caller never identifies itself to the server, and the server never requires an ID from the caller. Clearly, some network services, such as a network filesystem, require stronger security.

The Silicon Graphics RPC authentication subsystem provides two protocols: AUTH_NONE and AUTH_UNIX. AUTH_UNIX provides only for identification of the client in the RPC call, while AUTH_NONE, the default, turns the subsystem off.

Therefore, for these two protocols, no authentication is actually performed by the RPC code or library code prior to calling the program's implementation function.

Note: For these two protocols, authentication is entirely the responsibility of the server application program. Applications should not rely on the veracity of the identification information in the AUTH_UNIX protocol without first taking steps to authenticate. An example of how to implement an authentication procedure is provided in "Server-side Authentication."

The authentication subsystem of the RPC package is open-ended. That is, other types of authentication are easy to support. However, this section deals only with the AUTH_UNIX authentication type, which is the only supported type other than AUTH_NONE.

Client-side Authentication

In this example, a caller creates a new RPC client handle:

```
clnt = clntudp_create(address, prognum, versnum, wait, sockp)
```

The appropriate transport instance defaults to the associated authentication protocol:

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

The RPC client can choose to use the AUTH_UNIX protocol by setting *clnt->cl_auth* after creating the RPC client handle:

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

This code causes each RPC call associated with *clnt* to carry with it this AUTH_UNIX protocol credentials structure:

```
/*
* AUTH_UNIX protocol credentials
*/
struct authunix_parms {
    u_long aup_time;        /* credentials creation time */
    char   *aup_machname;   /* host name where client is */
    int    aup_uid;         /* client's UNIX effective uid */
    int    aup_gid;         /* client's current group ID */
    u_int  aup_len;         /* element length of aup_gids */
    int    *aup_gids;       /* array of groups user in */
};
```

These fields are set by **authunix_create_default**() when you invoke the appropriate system calls.

Since the RPC user created the AUTH_UNIX protocol structure, he or she is responsible for destroying it with:

```
auth_destroy(clnt->cl_auth);
```

You should use this call in all cases to conserve memory.

Server-side Authentication

Server-side authentication is necessary in cases where security needs to be enforced. Since by default, no authentication is provided, this section provides an example of how to implement an authentication of the UID that is passed in the AUTH_UNIX protocol.

Consider the fields of a request handle passed to a service dispatch routine:

```
/*
 * An RPC Service request
 */
struct svc_req {
  u_long  rq_prog;        /* service program number */
  u_long  rq_vers;        /* service protocol vers num */
  u_long  rq_proc;        /* desired procedure num */
  struct opaque_auth rq_cred; /* raw credentials from wire */
  caddr_t rq_clntcred;    /* read only credentials */
};
```

The *rq_cred* is mostly opaque, except for one field of interest—the style or flavor of authentication credentials:

```
/*
 * Authentication info.  Mostly opaque to the programmer.
 */
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 request's *rq_cred* is well formed. Thus, the service implementor should inspect the request's *rq_cred.oa_flavor* to determine which style of authentication protocol the caller used. The service implementor may also want to inspect the other fields of *rq_cred* if the style is not one supported by the RPC package.
- The request's *rq_clntcred* field is either NULL or points to a well-formed structure that corresponds to a supported style of authentication protocol. Only the AUTH_UNIX protocol is currently supported, so it is strongly recommended that the program check that the value of *rq_cred.oa_flavor* is AUTH_UNIX before *rq_clntcred* is cast to a pointer to the *authunix_parms* structure. If the authentication protocol in the client request differs from the protocol used in the server's *rq_cred.oa_flavor* value, errant behavior and possible security holes could result.

Note: In both cases, the authentication protocol does not verify the veracity of the request; it checks only that the structure format is adhered to.

The following example implements authentication by first checking that the protocol used in the authentication credential (*rq_cred.oa_flavor*) is AUTH_UNIX. If it is, and since AUTH_UNIX provides no authentication, further checks are made to verify that the identity provided in the client identity structure, *rq_clntcred,* is valid. If either of these checks fails, the function **svcerr_weakauth**() is called to alert the server that security is not being enforced.

```
#include <stdio.h>
#include <syslog.h>
#include <pwd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <rpc/rpc.h>  /* Required. */
#include <utmp.h>
#include <rpcsvc/rusers.h>

#define MAX_LOG_MESSAGE 160

int nuser_client_ok(struct svc_req *rqstp, SVCXPRT *transp);
void nuser();
```

```
main()
{
    SVCXPRT *transp;
    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL){
        fprintf(stderr, "can't create an RPC server\n");
        exit(1);
    }
    pmap_unset(RUSERSPROG, RUSERSVERS);
    if (!svc_register(transp, RUSERSPROG, RUSERSVERS, nuser,
                      IPPROTO_UDP)) {
        fprintf(stderr, "can't register RUSER service\n");
        exit(1);
    }
    svc_run();  /* never returns */
    fprintf(stderr, "should never reach this point\n");
    exit(1);
}

void
nuser(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{
    uid_t uid;
    unsigned long nusers;
    struct authunix_parms *unix_cred;

    /* we don't care about authentication for the null procedure */
    if (rqstp->rq_proc == NULLPROC) {
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can't reply to RPC call\n");
            exit(1);
        }
        return;
    }
    /* now get the uid */
    switch (rqstp->rq_cred.oa_flavor) {
    case AUTH_UNIX:
        /* perform authentication checks on client credentials */
        if (! nuser_client_ok(rqstp, transp)) {
            svcerr_weakauth(transp);
            return;
        }
        /* passed test */
```

```
            unix_cred = (struct authunix_parms *)rqstp->rq_clntcred;
            uid = unix_cred->aup_uid;
            break;
        case AUTH_NULL:
        default:
            svcerr_weakauth(transp);
            return;
        }
        switch (rqstp->rq_proc) {
        case RUSERSPROC_NUM:
            /* make sure the caller is allowed to call this procedure.  */
            if (uid == 16) {
                svcerr_systemerr(transp);
                return;
            }
            /* code here to compute the number of users and put
             * in variable nusers
             */
            if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
                fprintf(stderr, "can't reply to RPC call\n");
                exit(1);
            }
            return;
        default:
            svcerr_noproc(transp);
            return;
        }
    } /* nuser */

    static int logging_successful_requests = 1;

    /* This routine attempts to verify that the client user is
     * authorized access on the server host.  A value of 0 is
     * returned to indicate that the client user is not authorized.
     * Otherwize the value returned is 1. */
    int
    nuser_client_ok(struct svc_req *rqstp, SVCXPRT *transp)
    {
        uid_t uid;
        char *user = NULL;
        struct authunix_parms *unix_cred;
        struct hostent *host_entry = NULL;
        struct passwd *passwd_entry;
        char log_message[MAX_LOG_MESSAGE];
```

```
static u_long peer_addr = 0;
static char *client_host = NULL;
static u_long client_host_addr = 0;

if (transp->xp_raddr.sin_port >= IPPORT_RESERVED) {
    sprintf(log_message, "Rejected request, "
             "non-priviledged port %d", transp->xp_raddr.sin_port);
    syslog(LOG_NOTICE | LOG_AUTH, log_message);
    return 0;
}
/* Determine the client host name and address. */
if (peer_addr != transp->xp_raddr.sin_addr.s_addr) {
    host_entry = gethostbyaddr(&transp->xp_raddr.sin_addr,
                               sizeof(struct in_addr),
                               AF_INET);
    if (host_entry == NULL) {
        sprintf(log_message, "Rejected request, "
                 "unknown client host at address 0x%08x",
                 transp->xp_raddr.sin_addr);
        syslog(LOG_NOTICE | LOG_AUTH, log_message);
        return 0;
    }
    peer_addr = transp->xp_raddr.sin_addr.s_addr;
    if (client_host != NULL) {
        free(client_host);
    }
    client_host = strdup(host_entry->h_name);
    client_host_addr = *(u_long *) host_entry->h_addr;
}

/* Determine the user name. */
unix_cred = (struct authunix_parms *)rqstp->rq_clntcred;
uid = unix_cred->aup_uid;
passwd_entry = getpwuid(uid);
if (passwd_entry == NULL) {
    sprintf(log_message, "Rejected request, "
             "unknown uid %d from host %s",
             uid, client_host);
    syslog(LOG_NOTICE | LOG_AUTH, log_message);
    return 0;
}
```

```
    user = strdup(passwd_entry->pw_name);
    if (passwd_entry->pw_passwd != NULL &&
        *passwd_entry->pw_passwd != '\0' &&
        ruserok(client_host, uid == 0, user, user) < 0) {
        sprintf(log_message, "Rejected request by %s at %s",
                 user, client_host);
        syslog(LOG_NOTICE | LOG_AUTH, log_message);
        free(user);
        return 0;
    }

    if (logging_successful_requests) {
        if (user != NULL) {
            sprintf(log_message, "Granted request by %s at %s",
                     user, client_host);
        } else {
            sprintf(log_message, "Granted request by uid %d at %s",
                     uid, client_host);
        }
        syslog(LOG_INFO | LOG_AUTH, log_message);
    }
    if (user != NULL) {
        free(user);
    }
    return 0;
} /* nuser_client_ok */
```

Several points should be noted:

- It is customary not to check the authentication parameters associated with the NULLPROC (procedure number zero).
- If the authentication parameter's type is not suitable for your service, you should call **svcerr_weakauth()**.
- The service protocol itself should return status for access denied; in the case of our example, the protocol does not have such a status, so we call the service primitive **svcerr_systemerr()** instead.

The last point underscores the relationship between the RPC authentication package and the services—RPC with the AUTH_UNIX protocol is concerned only with identification and not with individual services' authentication or access control. The services themselves must implement these policies, and they must reflect these policies as return statuses in their protocols.

It is important to recognize that the AUTH_ UNIX credential is passed in the clear across the network and can be easily modified or counterfeited. There are no checks performed on the AUTH_ UNIX credential except to make sure it is correctly formatted.

Services that provide functions that require root permissions, and accept requests bearing AUTH_UNIX credentials, should take steps to perform authentication of the information in the AUTH_UNIX credential before relying on that information for access control decisions.

- Limit the service to a reserved port. This requires that the originator have sufficient privilege to create a port with an address less than 1024.
- Verify that the source IP address of the RPC request is from the machine named in the RPC AUTH_UNIX credential. The name corresponding to the source IP address should be in the list of addresses for the name.
- Ensure that the user name/UID is known to the local system. This requires that the originator be in the network name service.
- Consider using **ruserok()** (see ruserok(3N)) or an equivalent functionality to limit access to a list of known hosts.

Further authentication such as encryption, digital signatures, Kerberos, and time stamps can be incorporated into the body of the RPC request. Consult references on network security such as *Applied Cryptography, Second Edition*, for more examples.

Using inetd

An RPC server can be started from *inetd*. Call the service creation routine as follows (since *inetd* passes a socket as file descriptor 0):

```
transp = svcudp_create(0);    /* For UDP */
transp = svctcp_create(0,0,0);/* For listener TCP sockets */
transp = svcfd_create(0,0,0); /* For connected TCP sockets */
```

In addition, you should call **svc_register()** as:

```
svc_register(transp, PROGNUM, VERSNUM, service, 0);
```

The final flag is 0, since the program will already be registered by *inetd*.

Remember that if you want to exit from the server process and return control to *inetd*, you must explicitly exit, since **svc_run()** never returns.

Entries in */usr/etc/inetd.conf* for RPC services should be in one of these two formats:

```
p_name/version dgram  rpc/udp wait user server args
p_name/version stream rpc/tcp wait user server args
```

In these entries, *p_name* is the symbolic name of the program as it appears in rpc(4); *server* is the program implementing the server; and *version* is the version number of the service. By convention, the first argument must be the program's name. For more information about *inetd,* see inetd(1M).

If the same program handles multiple versions, the version number can be a range. For example:

```
rstatd/1-2 dgram rpc/udp wait root /usr/etc/rpc.rstatd rstatd
```

For server programs that handle multiple services or protocols, *inetd* allocates socket descriptors to protocols based on lexicographic order of service and protocol names.

More Examples

The examples in this section illustrate a program version number, TCP use, and a callback procedure.

Program Version Number

By convention, the first version number of program PROG is PROGVERS_ORIG, and the most recent version is PROGVERS. Suppose there is a new version of the *user* program that returns an *unsigned short* rather than a *long.* If we name this version RUSERSVERS_SHORT, a server that wants to support both versions does a double register:

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

Both versions can be handled by the same C procedure:

```
nuser(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");
            exit(1);
        }
        return;

    case RUSERSPROC_NUM:
        /* Code here to compute the number of users and
         * assign it to the variable nusers
         */
        nusers2 = 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:
           if (!svc_sendreply(transp,xdr_u_short,&nusers2)) {
                fprintf(stderr,"can't reply to RPC call\n");
            }
            break;
        }
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

TCP

This example is essentially *rcp*. The initiator of the RPC **snd()** call sends its standard input to the server **rcv()**, which prints it on standard output. The RPC call uses TCP. This example also illustrates an XDR procedure that behaves differently on serialization than on deserialization:

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

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

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

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

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

    if (argc < 2) {
        fprintf(stderr, "usage: %s server-name\n", argv[0]);
        exit(-1);
    }
    if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC_FP,
         RCPVERS, xdr_rcp, stdin, xdr_void, 0)) != 0) {
        clnt_perrno(err);
        fprintf(stderr, " can't make RPC call\n");
        exit(1);
    }
    exit(0);
}

callrpctcp(host, prognum, procnum, versnum, inproc, in,
           outproc, out)
char *host;
int prognum;
int procnum;
int versnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
{
    struct sockaddr_in server_addr;
    int socket = RPC_ANYSOCK;
    enum clnt_stat clnt_stat;
    struct hostent *hp;
    register CLIENT *client;
    struct timeval total_timeout;
```

```
    if ((hp = gethostbyname(host)) == NULL) {
        herror(host);
        return (-1);
    }
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
          hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = 0;
    if ((client = clnttcp_create(&server_addr, prognum,
         versnum, &socket, BUFSIZ, BUFSIZ)) == NULL) {
        perror("rpctcp_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);
    return (int)clnt_stat;
}

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

main()
{
    register SVCXPRT *transp;
    int rcp_service(), xdr_rcp();
    if ((transp = svctcp_create(RPC_ANYSOCK, 1024, 1024))
        == NULL) {
        fprintf(stderr, "svctcp_create: error\n");
        exit(1);
    }
    pmap_unset(RCPPROG, RCPVERS);
    if (!svc_register(transp, RCPPROG, RCPVERS, rcp_service,
                      IPPROTO_TCP)) {
        fprintf(stderr, "svc_register: error\n");
        exit(1);
    }
    svc_run();              /* never returns */
    fprintf(stderr, "svc_run should never return\n");
    exit(1);
}
```

```
rcp_service(rqstp, transp)
register struct svc_req *rqstp;
register SVCXPRT *transp;
{
    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (svc_sendreply(transp, xdr_void, 0) == 0) {
            fprintf(stderr, "err: rcp_service\n");
            return(1);
        }
        return;
    case RCPPROC_FP:
        if (!svc_getargs(transp, xdr_rcp, stdout)) {
            svcerr_decode(transp);
            return(1);
        }
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can't reply\n");
            return(1);
        }
        return(0);
    default:
        svcerr_noproc(transp);
        return(1);
    }
}
```

Callback Procedures

In some cases (for example, in remote debugging), it's useful for a server to become a client and make an RPC callback to its client process.

For example, if the client is a window system program and the server is a debugger running on a remote machine, when the user clicks a mouse button in the debugging window, the click is converted to a debugger command, and an RPC call is made to the server (where the debugger is actually running) telling it to execute that command.

When the debugger hits a breakpoint, however, the roles are reversed. The debugger wants to make an RPC call to the window program to notify the user of the breakpoint.

To do an RPC callback, you need a program number on which to make the RPC call (see "Assigning RPC Program Numbers" in Chapter 3 for more information). Because the callback will be a dynamically generated program number, it should be in the transient range, 0x40000000—0x5fffffff.

The **gettransient()** routine returns a valid program number in the transient range and registers it with the port mapper (see "The Port Mapper Programs" in Chapter 3 for more information). The program talks only to the port mapper running on the same machine as the **gettransient()** routine itself.

The call to **pmap_set()** is a test-and-set operation. **pmap_set()** indivisibly tests whether a program number has already been registered; if the number has not been registered, **pmap_set()** reserves it. Upon return, the *sockp* argument will contain a socket that can be used as the argument to a **svcudp_create()** or **svctcp_create()** call.

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

gettransient(proto, vers, sockp)
int proto, vers, *sockp;
{
    static int prognum = 0x40000000;
    int s, len, socktype;
    struct sockaddr_in addr;
    switch(proto) {
        case IPPROTO_UDP:
            socktype = SOCK_DGRAM;
            break;
        case IPPROTO_TCP:
            socktype = SOCK_STREAM;
            break;
        default:
            fprintf(stderr, "unknown protocol type\n");
            return(0);
    }
    if (*sockp == RPC_ANYSOCK) {
        if ((s = socket(AF_INET, socktype, 0)) < 0) {
            perror("socket");
            return(0);
        }
        *sockp = s;
    } else
```

```
        s = *sockp;
    addr.sin_addr.s_addr = 0;
    addr.sin_family = AF_INET;
    addr.sin_port = 0;
    len = sizeof(addr);
         /* may be already bound, so don't check for error*/
    bind(s, &addr, len);
    if (getsockname(s, &addr, &len)< 0) {
        perror("getsockname");
        return(0);
    }
    while (!pmap_set(prognum++, vers, proto,
                     ntohs(addr.sin_port)))
        continue;
    return (prognum-1);
}
```

Note: The call to **ntohs()** is necessary to ensure that the port number in *addr.sin_port*, which is in *network* byte order, is passed in *host* byte order (as **pmap_set()** expects). See byteorder(3N) for more information about network address conversion from network to host byte order.

The following programs illustrate how to use the **gettransient()** routine. The client makes an RPC call to the server, passing it a transient program number. Next, the client waits to receive a callback from the server at that program number. The server registers the program EXAMPLEPROG so it can receive the RPC call informing it of the callback program number. Then, at some random time (on receiving a SIGALRM signal in this example), it sends a callback RPC call, using the program number it received earlier.

```
/* client */
#include <stdio.h>
#include <rpc/rpc.h>

void callback();
char hostname[256];

main()
{
    int x, ans, s;
    SVCXPRT *xprt;

    gethostname(hostname, sizeof(hostname));
    s = RPC_ANYSOCK;
    x = gettransient(IPPROTO_UDP, 1, &s);
```

```
    fprintf(stderr, "client gets prognum %d\n", x);
    if ((xprt = svcudp_create(s)) == NULL) {
        fprintf(stderr, "rpc_server: svcudp_create\n");
        exit(1);
    }
    /* protocol is 0 - gettransient does registering */
    (void) svc_register(xprt, x, 1, callback, 0);
    ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
          EXAMPLEPROC_CALLBACK, xdr_int, &x, xdr_void, 0);
    if ((enum clnt_stat) ans != RPC_SUCCESS) {
        fprintf(stderr, "call:");
        clnt_perrno(ans);
        fprintf(stderr, "\n");
    }
    svc_run();
    fprintf(stderr,
            "Error: svc_run shouldn't have returned\n");
    exit(1);
}

void callback (rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch (rqstp->rq_proc) {
    case 0:
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "err: exampleprog\n");
            return(1);
        }
        return(0);
    case 1:
        if (!svc_getargs(transp, xdr_void, 0)) {
            svcerr_decode(transp);
            return(1);
        }
        fprintf(stderr, "client got callback\n");
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "err: exampleprog");
            return(1);
        }
    }
}
```

```
/* server */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>

char *getnewprog();
char hostname[256];
void docallback();
int pnum;           /*program number for callback routine */

main()
{
    gethostname(hostname, sizeof(hostname));
    registerrpc(EXAMPLEPROG, EXAMPLEVERS,
                EXAMPLEPROC_CALLBACK, getnewprog, xdr_int,
                xdr_void);
    fprintf(stderr, "server going into svc_run\n");
    signal(SIGALRM, docallback);
    alarm(10);
    svc_run();
    fprintf(stderr,
            "Error: svc_run shouldn't have returned\n");
    exit(1);
}

char *getnewprog(pnump)
char *pnump;
{
    pnum = *(int *)pnump;
    return NULL;
}

void docallback()
{
    int ans;
    ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0,
                  xdr_void, 0);
    if (ans != 0) {
        fprintf(stderr, "server: %s", clnt_sperrno(ans));
    }
}
```

Chapter 6

XDR and RPC Language Structure

This chapter describes the XDR and RPC languages. Topics include:

- XDR language structure, syntax, and examples
- RPC language structure, syntax, and examples

Note: For an overview of the relationship between the RPC and XDR languages and the RPC interface, see Chapter 3, “Introduction to RPC Programming.” For information about *rpcgen*, see Chapter 4, “Programming with rpcgen.” For information about RPC programming, see Chapter 5, “RPC Programming Guide.” For technical details about XDR, see Chapter 7, “XDR Programming Notes.” For complete specifications of the RPC and XDR protocols, see Appendix A, “RPC Protocol Specification,” and Appendix B, “XDR Protocol Specification.”

XDR Language

This section describes the components of the XDR language.

Notational Conventions

This specification uses an extended Backus-Naur Form notation for describing the XDR language. This notation has the following characteristics:

- These are the special characters:

  ```
  |  (  )  [  ]  "  *
  ```

- Terminal symbols are strings of any characters surrounded by double quotes (`" "`).
- Nonterminal symbols are strings of non-special characters.
- Alternative items are separated by a vertical bar (`|`).
- Optional items are enclosed in brackets (`[]`).
- Items are grouped by enclosing them in parentheses (`( )`).
- An asterisk (`*`) following an item means zero or more occurrences of that item.

For example, consider this pattern:

```
"a" "very" ("," "very")* ["cold" "and"]  "rainy" ("day" | "night")
```

An infinite number of strings match this pattern; for example:

```
"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

This section discusses some lexical features of the XDR language:

- Comments begin with `/*` and end with `*/`. For example:

  ```
  /* comment */
  ```

- White space separates items and is otherwise ignored.
- An identifier is a letter followed by an optional sequence of letters, digits, or an underscore (_). The case of identifiers is not ignored.
- A constant is a sequence of one or more decimal digits, optionally preceded by a minus sign (–).

Syntax Information

This section describes XDR language syntax:

```
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"
    | "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

This section provides additional information about XDR language syntax:

- The following keywords cannot be used as identifiers:

```
bool        double      opaque    typedef
case        enum        string    union
const       float       struct    unsigned
default     hyper       switch    void
```

- 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.
- Constant and type identifiers within the scope of a specification are in the same name space and must be declared uniquely within this scope.
- Similarly, variable names must be unique within the scope of *struct* and *union* declarations. Nested *struct* and *union* declarations create new scopes.
- The discriminant of a *union* must be of a type that evaluates to an integer. That is, *int*, *unsigned int*, *bool*, an enumerated type, or any **typedef** type that evaluates to one of these is legal. Also, the **case** values must be one of the legal values of the discriminant. Finally, a **case** value may not be specified more than once within the scope of a *union* declaration.

XDR Data Description Example

The following is a short XDR data description of a "file" that you might use to transfer files from one machine to another:

```
const MAXUSERNAME = 32;    /* max length of a user name */
const MAXFILELEN = 65535;  /* max length of a file */
const MAXNAMELEN = 255;    /* max length of a filename */

/* 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 interpreter<MAXNAMELEN>;/*program interpreter*/
};

/* 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 that a user named *jean* wants to store a LISP program *sillyprog*, which contains just the data "(quit)." The file would be encoded as shown in Table 6-1.

Table 6-1 DR Data Encoding Examples

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 6	ypro	More filename characters
12	67 00 00 00	g...	The last filename character plus 3 zero-bytes of fill
16	00 00 00 02	....	File kind is EXEC = 2
20	00 00 00 04	....	Length of interpreter name = 4
24	6c 69 73 70	lisp	Interpreter name
28	00 00 00 04	....	Length of owner name = 4
32	6a 65 61 6e	jean	Owner name

Table 6-1 (continued) DR Data Encoding Examples

Offset	Hex Bytes	ASCII	Description
36	00 00 00 06	....	Length of data = 6
40	28 71 75 69	(qui	File data bytes
44	74 29 00 00	t)..	More data plus 2 zero-bytes of fill

RPC Language

RPC language is an extension of the XDR language; the sole extension is the addition of the *program* type.

Definitions

An RPC language file consists of a series of definitions:

```
definition-list:
    definition ";"
    definition ";" definition-list
```

It recognizes six types of definition:

```
definition:
    enum-definition
    struct-definition
    union-definition
    typedef-definition
    const-definition
    program-definition
```

Structures

An XDR structure is declared almost exactly like its C counterpart:

```
struct-definition:
    "struct" struct-ident "{"
        declaration-list
    "}"
```

```
declaration-list:
    declaration ";"
    declaration ";" declaration-list
```

For example, the following code is an XDR structure for a two-dimensional coordinate and the C structure into which it is compiled in the output header file:

```
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. Using **typedef** allows you to use *coord* instead of *struct coord* when declaring items.

Unions

XDR unions are discriminated unions, and they look different from C unions. They are more analogous to Pascal variant records than they are to C unions:

```
union-definition:
    "union" union-ident "switch" "(" declaration ")" "{"
        case-list
    "}"
case-list:
    "case" value ":" declaration ";"
    "default" ":" declaration ";"
    "case" value ":" declaration ";" case-list
```

The next example shows a type that might be returned as the result of a read data operation. If no error, return a block of data. Otherwise, return nothing.

```
union read_result switch (int errno) {
case 0:
    opaque data[1024];
default:
    void;
};
```

This code is compiled into:

```
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 name as the type name, except for the trailing *_u*.

Enumerations

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
```

The XDR *enum* and the C *enum* are compiled into:

```
enum colortype {          enum colortype {
    RED = 0,                  RED = 0,
    GREEN = 1,     -->        GREEN = 1,
    BLUE = 2                  BLUE = 2
};                        };
                          typedef enum colortype colortype;
```

Typedefs

An XDR **typedef** has the same syntax as a C **typedef**:

```
typedef-definition:
    "typedef" declaration
```

The following example defines a *fname_type* used to declare filename strings that have a maximum length of 255 characters:

```
typedef string fname_type<255>; --> typedef char *fname_type;
```

Constants

XDR constants are symbolic constants. They may be used wherever an integer constant is used; for example, in array-size specifications:

```
const-definition:
     "const" const-ident "=" integer
```

For example, the following defines a constant DOZEN equal to 12:

```
const DOZEN = 12;  -->  #define DOZEN 12
```

Programs

RPC programs are declared using this 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
```

This example shows the time protocol, revisited:

```
/*
 * time.x: Get or set the time. Time is represented as
 * number of seconds since 0:00, January 1, 1970.
 */
program TIMEPROG {
    version TIMEVERS {
        unsigned int TIMEGET(void) = 1;
        void TIMESET(unsigned) = 2;
    } = 1;
} = 44;
```

This file compiles into **#define**s in the output header file:

```
#define TIMEPROG 44
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

Declarations

There are four kinds of declaration in XDR:

- Simple declarations are like simple C declarations:

  ```
  simple-declaration:
      type-ident variable-ident
  ```

  For example:

  ```
  colortype color;    --> colortype color;
  ```

- Fixed-array declarations are like array declarations in C:

  ```
  fixed-array-declaration:
      type-ident variable-ident "[" value "]"
  ```

For example:

```
colortype palette[8];    --> colortype palette[8];
```

- Variable-array declarations have no explicit syntax in C, so XDR invents its own 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 */
```

- Since variable-length arrays have no explicit syntax in C, these declarations are actually compiled into structs. For example, the *heights* declaration gets 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's name is the same as the name of the declared XDR variable.
- Pointer declarations are made in XDR exactly as they are in C. You can't really send pointers over the network, but you can use XDR pointers to send recursive data types such as lists and trees. The type is actually called *optional data*, not *pointer*, in XDR language.

```
pointer-declaration:
    type-ident "*" variable-ident
```

For example:

```
listitem *next;    -->    listitem *next;
```

Special Cases

There are a few exceptions to the rules described in the preceding sections:

- Booleans

  C has no built-in boolean type. However, the RPC library does have a boolean type called *bool_t* that is either TRUE or FALSE. Things declared as type *bool* in XDR language are compiled into *bool_t* in the output header file. For example:

```
bool married;    -->    bool_t married;
```

- Strings

  C has no built-in string type, but instead uses the null-terminated *char* * convention. In XDR language, strings are declared using the *string* keyword and are compiled into *char* *s in the output header file. The number contained in the angle brackets specifies the maximum number of characters allowed in the string (not counting the NULL character). The maximum size may be left off, indicating a string of arbitrary length. For example:

```
string name<32>;      -->    char *name;
string longname<>;    -->    char *longname;
```

- Opaque data

  Opaque data is used in RPC and XDR to describe untyped data; that is, just sequences of arbitrary bytes. Opaque data may be declared as either a fixed- or a variable-length array. For example:

```
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 occur in only two places: *union* definitions and program definitions (as the argument or the result of a remote procedure).

Chapter 7

XDR Programming Notes

XDR is the backbone of Sun's RPC package—the data for remote procedure calls is transmitted using the XDR standard. This chapter is based on Sun's technical notes about the implementation of the XDR standard. (For a complete specification of the XDR protocol, see Appendix B, "XDR Protocol Specification.")

Note: Most programmers (especially RPC programmers) will only need the information in three sections of this chapter: "Number Filters" on page 165, "Floating-point Filters" on page 165, and "Enumeration Filters" on page 166.

Topics in this chapter include:

- overview of XDR programming
- XDR library routines and primitives
- XDR operation directions
- XDR stream access
- defining new streams and data types
- advanced topics

Overview of XDR Programming

XDR's approach to standardizing data representations is *canonical*. That is, XDR defines a single byte order (big-endian), 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 data representations to the equivalent 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 on 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, the local representations of the new machine are also irrelevant to existing programs running on other machines; such programs can immediately read portable data produced by the new machine, because such data conforms to the canonical standard that it already understands.

There are strong precedents for XDR's canonical approach. For example, TCP/IP, UDP/IP, 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 this disadvantage 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, when compared to the alternative.

The time spent converting to and from a canonical representation is insignificant, especially in networking 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 an image of 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 deallocated as well. Similarly, to receive a tree image, 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. Every machine pays the cost of traversing and copying data structures, regardless of whether conversion is required.

In networking applications, communication 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.

Consider the *writer* and *reader* programs.

The *writer* program looks like this:

```
#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);
}
```

The *reader* program looks like this:

```
#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 *writer* and *reader* programs appear to be portable, because they pass *lint* checking, and they exhibit the same behavior when executed on different hardware architectures—an IRIS-4D and a VAX.

Piping the output of the *writer* program to the *reader* program produces identical results on both machines:

```
IRIS% writer | reader
0 1 2 3 4 5 6 7
VAX% writer | reader
0 1 2 3 4 5 6 7
```

With the advent of local area networks and Berkeley's 4.2BSD UNIX came the concept of "network pipes"—a process produces data on one machine, and a second process consumes data on another machine. You can construct a network pipe with *writer* and *reader*. The next example shows the results if *writer* produces data on an IRIS, and *reader* consumes data on a VAX:

```
IRIS% writer | rsh vax reader
0 16777216 33554432 50331648 67108864 83886080 100663296 117440512
IRIS%
```

You can obtain identical results by executing *writer* on the VAX and *reader* on the IRIS. These results occur because the byte ordering of long integers differs between the VAX and the IRIS, even though word size is the same.

Note: The 16777216 is 2^{24}—when the order of 4 bytes is reversed, the 1 that started in the zeroth bit winds up 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()** system calls with calls to the XDR routine **xdr_long()**, a filter that knows the standard representation of a long integer in its external form.

This is the revised version of the *writer* program:

```
#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);
}
```

This is a revised version of the *reader* program:

```
#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);
}
```

When the revised programs are executed on an IRIS, on a VAX, and from an IRIS to a VAX, the results are:

```
IRIS% writer | reader
0 1 2 3 4 5 6 7

VAX% writer | reader
0 1 2 3 4 5 6 7

IRIS% writer | rsh vax reader
0 1 2 3 4 5 6 7
```

Dealing with integers is 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. Pointers are convenient to use, but they have no meaning outside the machine where they are defined.

Note: On IRIX systems, C programs that want access to XDR routines should include the *<rpc/rpc.h>* header file, which contains all necessary interfaces to the XDR system. Since the default C DSO contains all the XDR routines, you don't need to indicate any special libraries on the compilation line in order to use XDR. See "Compiling BSD and RPC Programs" in Chapter 1 for additional compiling hints.

The XDR Library

The XDR library not only solves data portability problems, it also lets you write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it makes sense to use the XDR library, even when 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.

Examine the *reader* and *writer* programs more closely. 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; therefore, use

xdrstdio_create(). The parameters to XDR stream creation routines vary according to their function. In our 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 the 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 (that is, 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 shown in this example:

```
xdr_xxx(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 **xdr_xxx()** routine, 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 the software engineering of portable data. The idea is to call the same routine for either operation—which almost guarantees that serialized data can also be deserialized. One routine is used by both producer and consumer of networked data. This direction independence 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. (See "XDR Operation Directions" on page 176 for details.)

For a slightly more complicated example, assume that a person's gross assets and liabilities are to be exchanged among processes, and assume that these values are important enough to warrant their own data type:

```
struct gnumbers {
    long g_assets;
    long g_liabilities;
};
```

A 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 chapter uses the following definitions:

```
#define bool_t   int
#define TRUE     1
#define FALSE    0
```

Keeping these conventions in mind, **xdr_gnumbers()** can be rewritten like this:

```
xdr_gnumbers(XDR *xdrs, struct gnumbers *gp)
{
    return (xdr_long(xdrs, &gp->g_assets) &&
        xdr_long(xdrs, &gp->g_liabilities));
}
```

This chapter uses both coding styles.

XDR Library Primitives

This section gives a synopsis of each XDR primitive, including basic data types, constructed data types, and XDR utilities. The interface to these primitives and utilities is defined in the include file *<rpc/xdr.h>*, automatically included by *<rpc/rpc.h>*.

Number Filters

The XDR library provides primitives to translate between numbers and their corresponding external representations. Primitives cover the set of numbers in:

```
[signed, unsigned] [short, int, long]
```

The eight primitives are:

```
bool_t xdr_char(XDR *xdrs, char *cp);
bool_t xdr_u_char(XDR *xdrs, unsigned char *ucp);
bool_t xdr_int(XDR *xdrs, int *ip);
bool_t xdr_u_int(XDR *xdrs, unsigned *up);
bool_t xdr_long(XDR *xdrs, long *lip);
bool_t xdr_u_long(XDR *xdrs, u_long *lup);
bool_t xdr_short(XDR *xdrs, short *sip);
bool_t xdr_u_short(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's floating-point types:

```
bool_t xdr_float(XDR *xdrs, float *fp);
bool_t xdr_double(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 enumeration 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 one (TRUE) or zero (FALSE).

```
#define bool_t   int
#define FALSE    0
#define TRUE     1
#define enum_t   int

bool_t xdr_enum(XDR *xdrs, enum_t *ep);
bool_t xdr_bool(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*. The routines return TRUE if they complete successfully, and FALSE otherwise.

No Data

Occasionally, an XDR routine must be supplied to the RPC system, even when no data is passed or required. The XDR library provides this routine:

```
bool_t xdr_void(XDR *xdrs, void *vp); /*always returns TRUE*/
```

Constructed Data Type Filters

Constructed or compound data type primitives require more parameters and perform more complicated functions than the primitives already discussed. 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 deallocate memory. The XDR operation XDR_FREE is used for this purpose.

To review, the three XDR directional operations are:

- XDR_ENCODE
- XDR_DECODE
- XDR_FREE

Strings

In 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 the string 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 **xdr_string()** routine:

```
bool_t xdr_string(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 filename may be no longer than 255 characters.)

The routine returns FALSE if the number of characters exceeds *maxlength*, and TRUE if it doesn't.

Note: Keep *maxlength* small. If it is too big, you can overrun the heap, since **xdr_string()** will call **malloc()** for space.

The behavior of **xdr_string()** is similar to the behavior of other routines discussed in this chapter. The XDR_ENCODE operation 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 non-NULL, the XDR package assumes that a target area has been allocated

that 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.

Byte Arrays

Variable-length byte arrays are often preferable to strings. Byte arrays differ from strings in several ways:

- The length of the array (the byte count) is explicitly located in an unsigned integer.
- The byte sequence is not terminated by a null character.
- The external representation of the bytes in the array is the same as their internal representation.

The **xdr_bytes()** primitive converts byte arrays between their internal and external representations:

```
bool_t xdr_bytes(XDR *xdrs, char **bpp, u_int *lp,
                 u_int maxlength);
```

The usage of the *xdrs*, *bpp*, and *maxlength* parameters is identical to their usage in **xdr_string()**. 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 provides a primitive for handling arrays of arbitrary elements. **xdr_bytes()** 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(XDR *xdrs, char **ap, u_int *lp,
                 u_int maxlength, u_int elementsize,
                 xdrproc_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; *elementsize* 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.

Examples of Constructed Data Types

Before defining more constructed data types, consider the examples in this section.

Example A

A user on a networked machine can be identified by:

- the machine name, such as *krypton*; see gethostname(2)
- the user's user ID; see geteuid(2)
- the group numbers to which the user belongs; see getgroups(2)

A structure with this information and its associated XDR routine could be coded like this:

```
struct netuser {
    char    *nu_machinename;
    int     nu_uid;
    u_int   nu_glen;
    int     *nu_gids;
};

#define NLEN 255     /* machine names must < 256 chars */
#define NGRPS 20     /* user can't belong to > 20 groups */

bool_t
xdr_netuser(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));
}
```

Example B

A party of network users could be implemented as an array of *netuser* structure. This is the declaration and its associated XDR routines:

```
struct party {
    u_int p_len;
    struct netuser *p_nusers;
};
#define PLEN 500     /* max number of users in a party */

bool_t
xdr_party(XDR *xdrs, struct party *pp)
{
    return (xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
                      sizeof (struct netuser), xdr_netuser));
}
```

Example C

The well-known parameters to **main()**, *argc* and *argv,* can be combined into a structure, and an array of instances of this structure can make up a history of commands. The declarations and XDR routines might look like this:

```
struct cmd {
    u_int c_argc;
    char **c_argv;
};

struct history {
    u_int h_len;
    struct cmd *h_cmds;
};

#define NCMDS  75   /* history is no more than 75 commands */
#define ALEN 1000   /* args cannot be > 1000 chars */
#define NARGC 100   /* command  cannot have > 100 args */

bool_t
xdr_wrap_string(XDR *xdrs, char **sp)
{
    return (xdr_string(xdrs, sp, ALEN));
}
```

```
bool_t
xdr_cmd(XDR *xdrs, struct cmd *cp)
{
    return (xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
                      sizeof (char *), xdr_wrap_string));
}

bool_t
xdr_history(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 **xdr_wrap_string()** routine is needed to package the **xdr_string()** routine, because the implementation of **xdr_array()** only passes two parameters to the array element description routine; **xdr_wrap_string()** supplies the third parameter to **xdr_string()**.

Opaque Data

In some protocols, handles are passed from a server to a 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. In other words, handles are opaque. The **xdr_opaque()** primitive is used to describe fixed-size opaque bytes.

```
bool_t xdr_opaque(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 is not machine portable.

Fixed-length Size Arrays

The XDR library does not provide a primitive for fixed-length arrays; the primitive **xdr_array()** is for variable-length arrays.

Example A could be rewritten to use fixed-size arrays, as shown in this code:

```
#define NLEN 255
/* machine names must be shorter than 256 chars */
#define NGRPS 20
```

```
/* user cannot be a member of more than 20 groups */
struct netuser {
    char *nu_machinename;
    int nu_uid;
    int nu_gids[NGRPS];
};

bool_t
xdr_netuser(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_gi , NGRPS, sizeof(int),
                    xdr_int)) {
        return (FALSE);
    }
    return (TRUE);
}
```

Discriminated Unions

The XDR library supports discriminated unions, C unions, 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(XDR *xdrs, enum_t *dscmp, char *unp,
                 struct xdr_discrim *arms,
                 xdrproc_t defaultarm);
```

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*, 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, the *defaultarm* procedure is called if it is non-NULL; otherwise, the routine returns FALSE.

Example D

Suppose the type of a union is 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;
};
```

The following constructs and XDR procedure (de)serialize the discriminated union:

```
struct xdr_discrim u_tag_arms[4] = {
    { INTEGER, xdr_int },
    { GNUMBERS, xdr_gnumbers }
    { STRING, xdr_wrap_string },
    { __dontcare__, NULL }
    /* always terminate arms with a NULL xdr_proc */
}

bool_t
xdr_u_tag(XDR *xdrs, struct u_tag *utp)
{
    return (xdr_union(xdrs, &utp->utype, &utp->uval,
                      u_tag_arms, NULL));
}
```

The routine **xdr_gnumbers()** was described in "The XDR Library" on page 162. The routine **xdr_wrap_string()** was described in Example C. The *defaultarm* 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 need not be sorted.

It is worth pointing out that the values of the discriminant may be sparse, although in this example they are not. It is always good practice to assign explicit 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.

Pointers

In C language it is often convenient to put pointers to a structure within another structure. The **xdr_reference()** primitive makes it easy to serialize, deserialize, and free these referenced structures:

```
bool_t xdr_reference(XDR *xdrs, char **pp, u_int ssize,
                     xdrproc_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 you are 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.

Note: **xdr_reference()** and **xdr_array()** are not interchangeable external representations of data.

Example E

Suppose there's a structure containing a person's name and a pointer to a *gnumbers* structure containing the person's gross assets and liabilities. This example demonstrates this construct:

```
struct pgn {
    char *name;
    struct gnumbers *gnp;
};
```

The corresponding XDR routine for this structure is:

```
bool_t
xdr_pgn(XDR *xdrs, struct pgn *pp)
{
    if (xdr_string(xdrs, &pp->name, NLEN) &&
        xdr_reference(xdrs, &pp->gnp,
                      sizeof(struct gnumbers), xdr_gnumbers))
        return(TRUE);
    return(FALSE);
}
```

Pointer Semantics and XDR

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 you are serializing data will most likely cause a memory fault and, on the UNIX system, a core dump.

xdr_pointer() correctly handles NULL pointers. For more information about its use, see "Linked Lists" on page 181.

Non-filter Primitives

XDR streams can be manipulated with the primitives discussed in this section.

```
u_int xdr_getpos(XDR *xdrs);
```

```
bool_t xdr_setpos(XDR *xdrs, u_int pos);
```

```
xdr_destroy(XDR *xdrs);
```

The routine **xdr_getpos()** returns an unsigned integer that describes the current position in the data stream.

Note: In some XDR streams, the returned value of **xdr_getpos()** is meaningless; the routine returns –1 in this case (though –1 should be a legitimate value).

The **xdr_setpos()** routine sets a stream position to *pos*.

Note: 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.

XDR Operation Directions

You can 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. Programmers are not encouraged to take advantage of this information. Therefore, no example is presented here. However, an example in "Linked Lists" on page 181 demonstrates the usefulness of the *xdrs->x_op* field.

XDR 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, TCP/IP connections and UNIX files, and memory. "XDR Stream Implementation" on page 179 documents the XDR object and how to make new XDR streams when they are required.

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 streams part of rpc */

void
xdrstdio_create(XDR *xdrs, FILE *fp, enum xdr_op x_op);
```

The **xdrstdio_create()** routine 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(XDR *xdrs, char *addr, u_int len,
              enum xdr_op x_op);
```

The **xdrmem_create()** routine initializes an XDR stream in local memory. The memory is pointed to by parameter *addr*; *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 UDP/IP implementation of RPC uses **xdrmem_create()**. Complete call or result messages are built in memory before calling the **sendto()** system call.

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 4.3BSD connection interface.

```
#include <rpc/rpc.h>   /* xdr streams are part of the
                        * rpc library */

xdrrec_create(XDR *xdrs, u_int sendsize, u_int recvsize,
              void *iohandle,
              int (*readproc) (void *, void *, u_int),
              int (*writeproc) (void *, void *, u_int));
```

The routine **xdrrec_create()** provides an XDR stream interface that allows for bidirectional, arbitrarily long sequences 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), predetermined defaults are used. When a buffer needs to be filled or flushed, the routine **readproc()** or **writeproc()** is called, respectively.

These routines are much like the **read()** and **write()** system calls. However, the first parameter to each routine is the opaque parameter *iohandle*. The other two parameters (*buf* and *nbytes*) and the results (byte count) are identical to the system routines.

If *xxx* is **readproc()** or **writeproc()**, it has this form:

```
/*
 * Returns the actual number of bytes transferred.
 * -1 is an error.
 */
int xxx(char *iohandle, char *buf, int len, int nbytes);
```

The XDR stream provides a means for delimiting records in the byte stream. The primitives specific to record streams are:

```
bool_t
xdrrec_endofrecord(XDR *xdrs, bool_t flushnow);

bool_t
xdrrec_skiprecord(XDR *xdrs);

bool_t
xdrrec_eof(XDR *xdrs);
```

(See "Advanced Topics" on page 181 for the implementation details of delimiting records in a stream.)

The **xdrrec_endofrecord()** routine causes the current outgoing data to be marked as a record. If the parameter *flushnow* is TRUE, the stream's **writeproc()** will be called; otherwise, **writeproc()** will be called when the output buffer has been filled.

The **xdrrec_skiprecord()** routine 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 no data remains in the stream's input buffer, the **xdrrec_eof()** routine returns TRUE; that is, 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

This structure defines the interface to an XDR stream:

```
enum xdr_op { XDR_ENCODE = 0, XDR_DECODE = 1, XDR_FREE = 2 };

typedef struct {
    enum xdr_op x_op;       /* operation; fast added param */
    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.

Note: The **x_inline()** routine is for cycle squeezers. Use of the resulting buffer is not data portable. Programmers should avoid 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(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 IRIX routines **htonl()** and **ntohl()** can be helpful in accomplishing this task. Appendix B, "XDR Protocol Specification," defines the standard representation of numbers.

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.

These routines return TRUE if they succeed, and FALSE otherwise. They have identical parameters:

```
bool_t xxxlong(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 creation routine.

Advanced Topics

This section describes additional techniques for passing data structures; for example, linked lists (of arbitrary lengths). Unlike the simpler examples already presented in this chapter, the examples in this section are written using both the XDR C library routines and the XDR data description language.

Linked Lists

Example E (see "Pointers" on page 174) presented a C data structure and its associated XDR routines for an individual's gross assets and liabilities. The example is duplicated here:

```
struct gnumbers {
    long g_assets;
    long g_liabilities;
};

bool_t
xdr_gnumbers(XDR *xdrs, struct gnumbers *gp)
{
    if (xdr_long(xdrs, &(gp->g_assets)))
        return (xdr_long(xdrs, &(gp->g_liabilities)));
    return (FALSE);
}
```

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 or not the object has terminated. Unfortunately, if the object continues, the *gn_next* field is also the address of where it continues. The link addresses do not carry any useful information when the object is serialized.

The XDR data description of this linked list is described by the recursive type declaration of *gnumbers_list*:

```
typedef union switch (boolean) {
    case TRUE: struct {
        struct gnumbers current_element;
        gnumbers_list rest_of_list;
    };
    case FALSE: struct {};
} gnumbers_list;
```

In this description, the boolean indicates whether there is more data following it. If the boolean is FALSE, then it is the last data field of the structure. If TRUE, it is followed by a *gnumbers* structure and (recursively) by a *gnumbers_list* (the rest of the object). Note that the C declaration has no boolean explicitly declared (although the *gn_next* field implicitly carries the information), while the XDR data description has no pointer explicitly declared.

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 above XDR union:

```
bool_t
xdr_gnumbers_node(XDR *xdrs, gnumbers_node *gn)
{
    return(xdr_gnumbers(xdrs, &gn->gn_numbers) &&
           xdr_gnumbers_list(xdrs, &gp->gn_next));
}

bool_t
xdr_gnumbers_list(XDR *xdrs, gnumbers_list *gnp)
{
    return(xdr_pointer(xdrs, gnp,
                       sizeof(struct gnumbers_node),
                       xdr_gnumbers_node));
}
```

The unfortunate side effect of XDRing a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list due to the recursion. The following routine collapses the above two mutually recursive routines into a single, nonrecursive routine:

```
bool_t
xdr_gnumbers_list(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 the 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 XDR's the *more_data* field of the XDR union. If there isn't any more data, 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. You set this value 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 longer valid. You can't free *gnp* in this way for all directions, though, because in the XDR_DECODE direction the value of *gnp* won't be set until the next statement.

Next, XDR the data in the node using the **xdr_reference()** primitive. **xdr_reference()** is like **xdr_pointer()** (used earlier), but it does not send over the boolean indicating whether there is more data. Use it instead of **xdr_pointer()**, because you have already XDR'd this information.

Notice that the XDR routine passed is not the same type as an element in the list. The routine passed is **xdr_gnumbers()**, for XDR'ing *gnumbers*, but each element in the list is actually of type *gnumbers_node*. You don't pass **xdr_gnumbers_node()**, because it is recursive, but instead use **xdr_gnumbers()**, which XDR's all of the nonrecursive part. Note that this trick will work 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, update *gnp* to point to the next item in the list. If the direction is XDR_FREE, 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 is less likely to blow the C stack. It will also run more efficiently, since a lot 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.

Chapter 8

Transport Layer Interface

This chapter provides detailed information, with various examples, on X/Open's Transport Layer Interface. This chapter describes the more important and common facilities of TLI, but is not meant to be exhaustive.

Note: Silicon Graphics does not encourage use of the TLI model; its inclusion is for compatibility with interfaces used by other vendors. The sockets interface, with its ease of use and compatibility with industry standardization is the preferred interface.

Topics covered in this chapter include:

- an introduction to TLI
- a brief discussion of Network Selection and Name-to-Address Mapping
- a description of the OSI Reference Model
- a summary of the basic services available to Transport Interface users
- an introduction to connection-mode (virtual circuit) communication
- an introduction to connectionless-mode (datagram) communication
- how to use **read()** and **write()**
- advanced topics such as asynchronous event handling and processing of multiple simultaneous connect requests
- the allowable state transitions associated with the Transport Interface
- necessary guidelines for developing software that can be run without change over any transport protocol developed for TLI
- a full listing of each programming example given (in fragmentary form) elsewhere in the chapter
- error message enhancements provided for XTI compatibility

Introduction

The Transport Layer Interface is a programming interface to the transport layer of ISO's Open Systems Interconnection Reference Model. It is a subset of the X/Open Transport Interface (XTI), and is implemented within the STREAMS framework. TLI is media- and protocol-independent; it allows applications to run across any transport protocol that supports the interface.

Network Selection and Name-to-Address Mapping facilities have been added to TLI to provide a means of guaranteeing media and protocol independence for transport applications. Network Selection and Name-to-Address Mapping allow network applications to acquire transport-specific information in a transport-independent way.

The following discussion assumes that you have a working knowledge of IRIX, C language programming, and data communication concepts. You should also be familiar with ISO-OSI before reading this chapter.

Network Selection and Name-to-Address Mapping

If TLI applications are to be media- and protocol-independent, they require an understanding of Network Selection and Name-to-Address Mapping facilities. The Network Selection routines in *libnsl.so* provide a standard interface to the networks available in any environment. Name-to-Address Mapping allows applications to translate transport-specific addresses. The following manual pages give more information on these topics:

getnetconfig() describes the *libnsl.so* routines that manipulate the network configuration administrative file, *netconfig*. For more information, see getnetconfig(3N).

getnetpath() describes the routines that manipulate the NETPATH variable, allowing you to specify which networks in the *netconfig* file to try. For more information, see getnetpath(3N).

netconfig() describes the network configuration database file. For more information, see netconfig(4).

netdir() describes the Name-to-Address Mapping library functions. For more information, see netdir(3N).

Note: *libnsl*, the Network Selection library, should not be confused with *libnls*, the network license server library.

OSI Reference Model

This section discusses the Reference Model to place the Transport Interface in perspective. The Reference Model partitions networking functions into seven layers, as depicted in Figure 8-1.

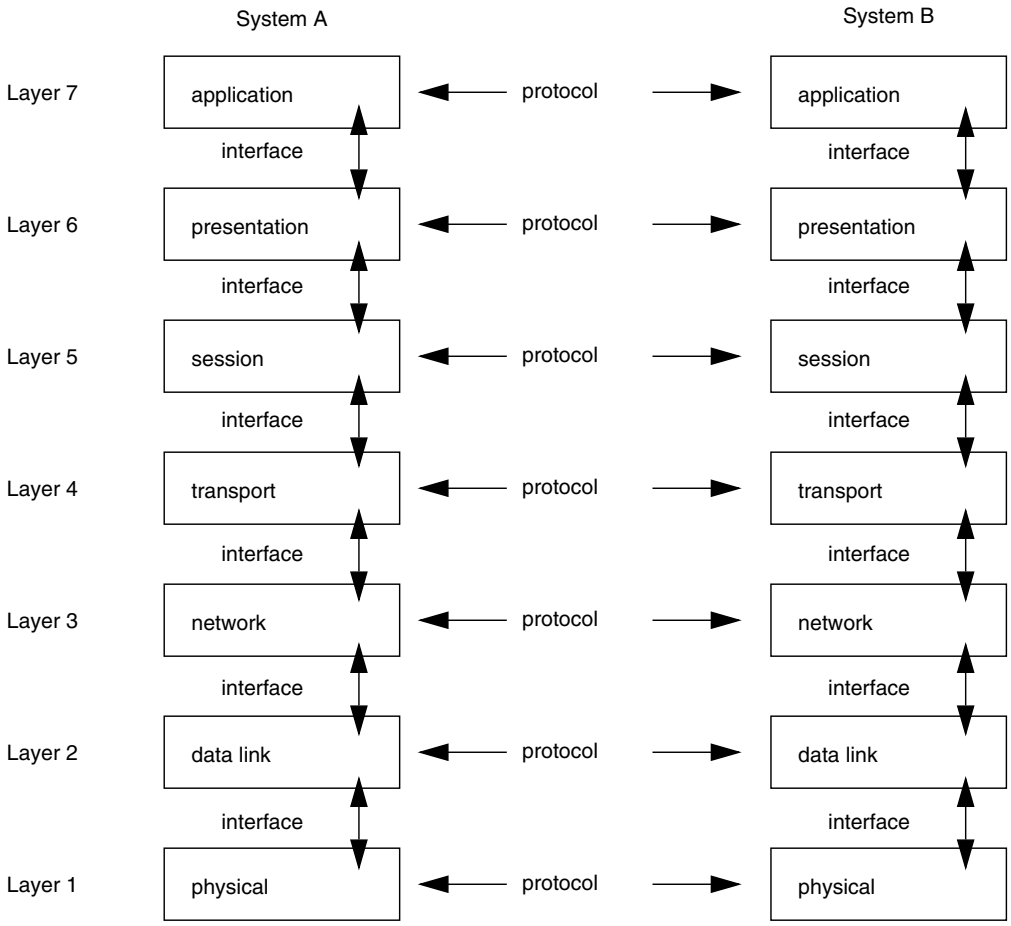


Figure 8-1 OSI Reference Model

Layer 1	The physical layer is responsible for the transmission of raw data over a communication medium.
Layer 2	The data-link layer provides the exchange of data between network layer entities. It detects and corrects any errors that may occur in the physical layer transmission.
Layer 3	The network layer manages the operation of the network. In particular, it is responsible for the routing and management of data exchange between transport layer entities within the network.
Layer 4	The transport layer provides transparent data transfer services between session layer entities, thereby relieving them of concerns about how to achieve reliable and cost-effective transfer of data.
Layer 5	The session layer provides the services needed by presentation layer entities that enable them to organize and synchronize their dialogue and manage their data exchange.
Layer 6	The presentation layer manages the representation of information that application layer entities either communicate or reference in their communication.
Layer 7	The application layer serves as the window between corresponding application processes that are exchanging information.

A basic principle of the Reference Model is that each layer provides services needed by the next higher layer in a way that frees the upper layer from concern about how these services are provided. This approach simplifies the design of each particular layer.

Industry standards have been defined (or are being defined) at each layer of the Reference Model. Two standards are defined at each layer: one that specifies an interface to the services of the layer (to be used when interacting with other layers) and one that defines the protocol by which services are provided (used by each instance of the current layer, as shown in Figure 8-1). A service interface standard at any layer frees users of the service from details of how that layer's protocol is implemented, or even which protocol is used to provide the service.

The transport layer is important because it is the lowest layer in the Reference Model that provides the basic service of reliable, end-to-end data transfer needed by applications and higher-layer protocols. In doing so, this layer hides the topology and characteristics of the underlying network from its users. More important, however, the transport layer defines a set of services common to layers of many contemporary protocol suites, including the International Standards Organization (ISO) protocols, the Transmission Control Protocol and Internet Protocol (TCP/IP) of the Internet, and the Systems Network Architecture (SNA).

A transport service interface, then, enables applications and higher layer protocols to be implemented without knowledge of the underlying protocol stack. That is a principal goal of the Transport Interface. Also, because an inherent characteristic of the transport layer is that it hides details of the physical medium being used, the Transport Interface offers both protocol and medium independence to networking applications and higher layer protocols.

The Transport Interface was modeled after the industry standard ISO Transport Service Definition (ISO 8072). As such, it is intended for those applications and protocols that require transport services. Because the Transport Interface provides reliable data transfer, and because its services are common to several protocol suites, many networking applications will find these services useful.

The Transport Interface is implemented as a user library using the STREAMS input/output mechanism. Therefore, many services available to STREAMS applications are also available to users of the Transport Interface. These services will be highlighted throughout this guide. For detailed information about STREAMS, refer to any generic UNIX SVR4 document set.

Overview of the Transport Interface

This section presents a high-level overview of the services of the Transport Interface, which supports the transfer of data between two user processes. Figure 8-2 illustrates the Transport Interface.

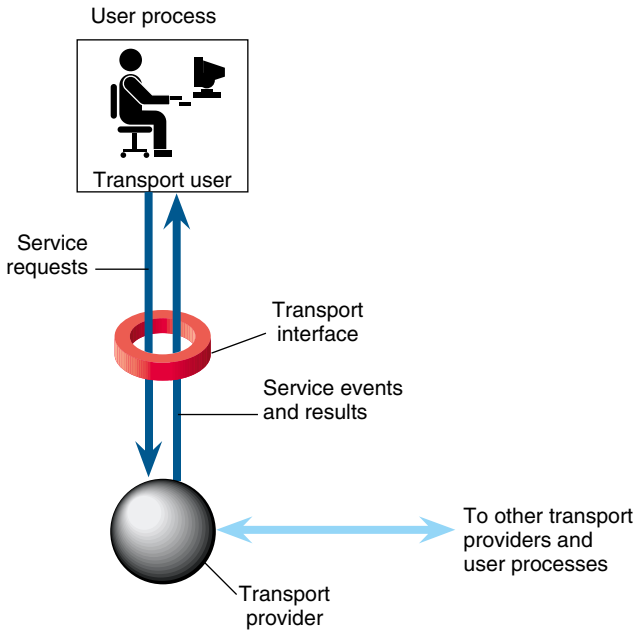


Figure 8-2 Transport Interface

The transport provider is the entity that provides the services of the Transport Interface, and the transport user is the entity that requires these services. An example of a transport provider is ISO 8073 (the OSI transport protocol), while a transport user can be a networking application or session layer protocol.

The transport user accesses the services of the transport provider by issuing the appropriate service requests. One example is a request to transfer data over a connection. Similarly, the transport provider notifies the user of various events, such as the arrival of data on a connection.

The Network Services Library includes a set of functions that support the services of the Transport Interface for user processes.

These functions enable a user to make requests to the provider and process incoming events. Programs using the Transport Interface can link the appropriate routines from the Network Services Library by using the **–lnsl** command-line option to *cc*.

Modes of Service

The Transport Interface provides two modes of service: connection mode and connectionless mode.

Connection mode is circuit-oriented and enables the transmission of data over an established connection in a reliable, sequenced manner. It also provides an identification procedure that avoids the overhead of address resolution and transmission during the data transfer phase. This service is attractive for applications that require relatively long-lived, datastream-oriented interactions. Connection-mode service is analogous to BSD's "stream sockets" (as opposed to datagram sockets), which provide a stream of data instead of isolated data units.

Connectionless mode, by contrast, is message-oriented and supports data transfer in self-contained units with no logical relationship required among multiple units. This service requires only a preexisting association between the peer users involved, which determines the characteristics of the data to be transmitted. This mode corresponds to sending datagrams in the sockets paradigm. All the information required to deliver a unit of data (for example, the destination address) is presented to the transport provider, together with the data to be transmitted, in one service access (which need not relate to any other service access). Each unit of data transmitted is entirely self-contained, like datagrams transmitted through BSD datagram sockets. Connectionless-mode service is attractive for applications that:

- involve short-term request/response interactions
- exhibit a high level of redundancy
- are dynamically reconfigurable
- do not require guaranteed, in-sequence delivery of data

Connection-Mode Service

The connection-mode transport service is characterized by four phases:

- local management
- connection establishment
- data transfer
- connection release

Local Management

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, as illustrated in Figure 8-3. Each channel between a transport user and transport provider is a unique endpoint of communication, and is called the transport endpoint. The **t_open()** routine (see t_open(3)) enables a user to choose a particular transport provider that supplies the connection-mode service, and establishes the transport endpoint.

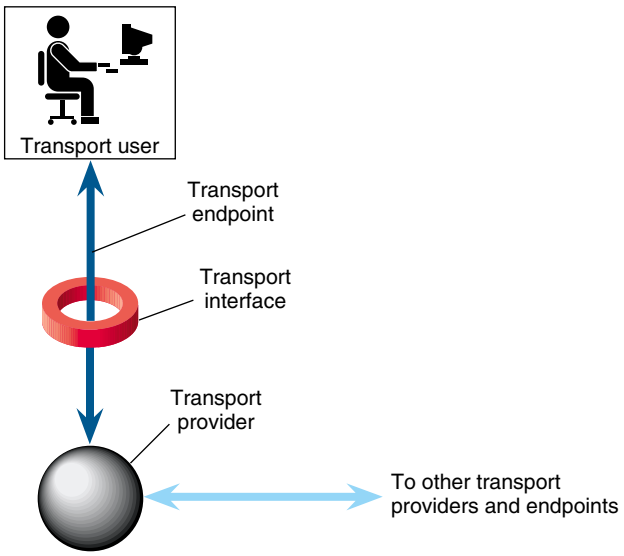


Figure 8-3 Channel between User and Provider

Another necessary local function for each user is to establish an identity with the transport provider. Each user is identified by a transport address. More accurately, a transport address is associated with each transport endpoint, and one user process can manage several transport endpoints. In connection-mode service, one user requests a connection to another user by specifying that user's address. The structure of a transport address is defined by the address space of the transport provider. An address can be anything from a simple character string (such as "file_server") to 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 can be assigned to each transport endpoint by **t_bind()**.

In addition to **t_open()** and **t_bind()**, several routines are available to support local operations. Table 8-1 summarizes all local management routines of the Transport Interface.

Table 8-1 Local Management Routines for the Transport Interface

Routine	Description
t_alloc()	Allocates Transport Interface data structures
t_bind()	Binds a transport address to a transport endpoint
t_close()	Closes a transport endpoint
t_error()	Prints a Transport Interface 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

Connection Establishment

The connection establishment phase enables two users to create a connection, or virtual circuit, between them, as demonstrated in Figure 8-4.

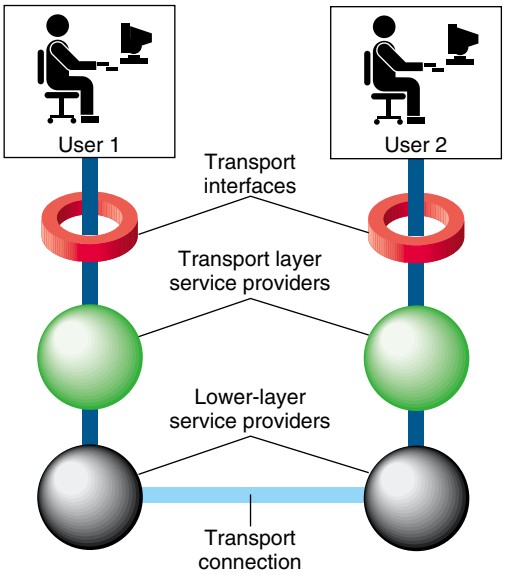


Figure 8-4 Transport Connection

This phase is illustrated in the following description of a client/server relationship. The client application and the server application are users of their respective transport providers. One user, the server, typically advertises some service to a group of users, and then listens for requests from those users. When a client requires the service, the client attempts to connect itself to the server using the server's advertised transport address. The **t_connect()** routine (see t_connect(3N)) initiates the connect request. One argument to **t_connect()**, the transport address, identifies the server the client wishes to access. The server is notified of each incoming request using **t_listen()** and can call **t_accept()** (see t_listen(3N) and t_accept(3N)) to accept the client's request for access to the service. If the request is accepted, the transport connection is established.

Table 8-2 summarizes all routines available for establishing a transport connection.

Table 8-2 Routines for Establishing a Transport Connection

Routine	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()	Retrieves an indication of a connect request from another transport user
t_rcvconnect()	Completes connection establishment if **t_connect()** was called in asynchronous mode (see "Advanced Topics" on page 229)

Data Transfer

The data transfer phase enables users to transfer data in both directions over an established connection. Two routines, **t_snd()** and **t_rcv()**, send and receive data over this connection. All data sent by a user is guaranteed to be delivered to the user on the other end of the connection in the order in which it was sent. Table 8-3 summarizes the connection-mode data transfer routines.

Table 8-3 Connection-Mode Data Transfer Routines

Routine	Description
t_rcv()	Retrieves data that has arrived over a transport connection
t_snd()	Sends data over an established transport connection

Connection Release

The connection release phase allows you to break an established connection. When you decide that a conversation should end, you can request that the provider release the transport connection. Two types of connection release are supported by the Transport Interface. The first is an abortive release, which directs the transport provider to release the connection immediately. Any previously sent data that has not yet reached the other transport user can be discarded by the transport provider. The **t_snddis()** routine initiates this abortive disconnect, and **t_rcvdis()** processes the incoming indication for an abortive disconnect.

All transport providers must support the abortive release procedure. In addition, some transport providers can also support an orderly release facility that enables users to terminate communication gracefully with no data loss. The functions **t_sndrel()** and **t_rcvrel()** support this capability. Table 8-4 summarizes the connection release routines.

Table 8-4 Connection Release Routines

Routine	Description
t_rcvdis()	Returns an indication of an aborted connection, including a reason code and user data
t_rcvrel()	Returns an indication that the other transport user has requested an orderly release of a connection
t_snddis()	Aborts a connection or rejects a connect request
t_sndrel()	Requests the orderly release of a connection

Connectionless-Mode Service

The connectionless-mode transport service is characterized by two phases: local management and data transfer. The local management phase defines the same local operations described above for the connection-mode service.

The data transfer phase enables a user to transfer data units (sometimes called datagrams) to the specified peer user. Each data unit must be accompanied by the transport address of the receiver. Two routines, **t_sndudata()** and **t_rcvudata()**, support this message-based data transfer facility, while another routine, **t_rcvuderr()**, allows retrieval of error messages. (For more information, see t_sndudata(3N), t_rcvudata(3N), and t_rcvuderr(3N).) Table 8-5 summarizes all routines associated with connectionless-mode data transfer.

Table 8-5 Routines for Connectionless-Mode Data Transfer

Routine	Description
t_rcvudata()	Retrieves a message sent by another transport user
t_rcvuderr()	Retrieves error information associated with a previously sent message
t_sndudata()	Sends a message to the specified destination user

State Transitions

The Transport Interface has two components:

- the library routines that provide the transport services to users
- the state transition rules that define the sequence in which the transport routines can be invoked

The state transition rules can be found in the state tables under "State Transitions" on page 237. The state tables define the legal sequence of library calls based on state information and the handling of events. These events include user-generated library calls, as well as provider-generated event indications.

Note: Any user of the Transport Interface must completely understand all possible state transitions before writing software using the interface.

Introduction to Connection-Mode Service

This section describes the connection-mode service of the Transport Interface. As discussed in the previous section, the connection-mode service can be illustrated using a client/server paradigm. The important concepts of connection-mode service are presented using two programming examples. The examples are related: Example 8-1 illustrates how a client establishes a connection to a server and then communicates with it, while Example 8-2 shows the server's side of the interaction. All code-fragment examples discussed in this chapter are presented as complete programs in "Some Examples" on page 244.

In the examples, the client establishes a connection with a server process. The server then transfers a file to the client. The client, in turn, receives the data from the server and writes it to its standard output file.

Local Management

Before the client and server can establish a transport connection, each must first establish a local channel (the transport endpoint) to the transport provider using **t_open()** and establish its identity (or address) using **t_bind()**.

The set of services supported by the Transport Interface may not be implemented by all transport protocols. Each transport provider has a set of characteristics associated with it that determines the services it offers and the limits associated with those services. This information is returned to the user by **t_open()** and consists of the following:

addr	maximum size of a transport address
options	maximum bytes of protocol-specific options that can be passed between the transport user and transport provider
tsdu	maximum message size that can be transmitted
etsdu	maximum expedited data message size that can be sent over a transport connection
connect	maximum number of bytes of user data that can be passed between users during connection establishment
discon	maximum number of bytes of user data that can be passed between users during the abortive release of a connection
servtype	type of service supported by the transport provider

The three service types (*servtype*) defined by the Transport Interface are as follows:

T_COTS	The transport provider supports connection-mode service but does not provide the optional orderly release facility.
T_COTS_ORD	The transport provider supports connection-mode service with the optional orderly release facility.
T_CLTS	The transport provider supports connectionless-mode service. Only one such service can be associated with the transport provider identified by **t_open()**.

Note: **t_open()** returns the default provider characteristics associated with a transport endpoint. However, some characteristics can change after an endpoint has been opened. This occurs if the characteristics are associated with negotiated options (option negotiation is described later in this section). For example, if the support of expedited data transfer is a negotiated option, the value of this characteristic can change. **t_getinfo()** can be called to retrieve the current characteristics of a transport endpoint.

Once a user establishes a transport endpoint with the chosen transport provider, it must establish its identity. As mentioned earlier, **t_bind()** does this by binding a transport address to the transport endpoint. In addition, for servers, this routine informs the transport provider that the endpoint will be used to listen for incoming connect indications, also called connect requests.

An optional facility, **t_optmgmt()** (see t_optmgmt(3N)), is also available during the local management phase. It enables a user to negotiate the values of protocol options with the transport provider. Each transport protocol is expected to define its own set of negotiable protocol options, which can include such information as Quality-of-Service parameters. Because of the protocol-specific nature of options, only applications written for a particular protocol environment are expected to use this facility.

The Client

The local management requirements of the example client and server are used to discuss details of these facilities following each example. The following are the definitions needed by the client program, followed by its necessary local management steps:

Example 8-1 The Connection-Mode Client Definitions and Local Management

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>

#define SRV_ADDR  1     /* server's well-known address */

void main()
{
    int fd;
    int nbytes;
    int flags = 0;
    char buf[1024];
    struct t_call *sndcall;
    extern int t_errno;
if ((fd = t_open("/dev/ticotsord", O_RDWR, NULL)) < 0) {
        t_error("t_open failed");
        exit(1);
    }

    if (t_bind(fd, NULL, NULL) < 0) {
        t_error("t_bind failed");
        exit(2);
    }
```

The first argument to **t_open()** is the pathname of a filesystem node that identifies the transport protocol that supplies the transport service. In this example, */dev/ticotsord* is a STREAMS clone device node that identifies a generic, connection-based transport protocol (see clone(7)). The *clone* device finds an available minor device of the transport provider for the user. It is opened for both reading and writing, as specified by the *O_RDWR* flag passed as the second argument. The third argument can be used to return the service characteristics of the transport provider to the user. This information is useful when writing protocol-independent software (discussed in "Guidelines for Protocol Independence" on page 243). For simplicity, the client and server in this example ignore this information and assume 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_COTS_ORD service type, and the example uses the orderly release facility to release the connection.
- User data cannot be passed between users during either connection establishment or abortive release.
- The transport provider does not support protocol-specific options.

Because these characteristics are not needed by the user, NULL is specified in the third argument to **t_open()**. If the user were to require a service other than T_COTS_ORD, another transport provider would be opened. An example of the T_CLTS service invocation is presented in "Introduction to Connectionless-Mode Service" on page 219.

The return value of **t_open()** is an identifier for the transport endpoint that is used by all subsequent Transport Interface function calls. This identifier is actually a file descriptor obtained by opening the transport protocol file (see open(2)). The significance of this fact is highlighted in "A Read/Write Interface" on page 225.

After the transport endpoint is created, the client calls **t_bind()** to assign an address to the endpoint. The first argument identifies the transport endpoint. The second argument describes the address the user would like to bind to the endpoint, and the third argument is set on return from **t_bind()** to specify the address that the provider bound.

The address associated with a server's transport endpoint is important, because that is the address used by all clients to access the server. However, the typical client does not care what its own address is, because no other process tries to access it. That is the case in this example, where the second and third arguments to **t_bind()** are set to NULL. A NULL second argument directs the transport provider to choose an address for the user. A NULL third argument specifies that the user does not care what address was assigned to the endpoint.

If either **t_open()** or **t_bind()** fails, the program calls **t_error()** (see t_error(3N)) to print an appropriate error message to *stderr*. If any Transport Interface routine fails, the global integer *t_errno* is assigned a transport error value. A set of error values is defined (in <*tiuser.h*>) for the Transport Interface, and **t_error()** prints an error message corresponding to the value in *t_errno*. This routine is analogous to **perror()**, which prints an error message based on the value of *errno* (see perror(3)). If the error associated with a transport function is a system error, *t_errno* is set to TSYSERR, and *errno* is set to the appropriate value.

The Server

The server in Example 8-2 must take similar local management steps before communication can begin. The server must establish a transport endpoint through which it listens for connect indications. The necessary definitions and local management steps are shown in Example 8-2.

Example 8-2 The Connection-Mode Server Definitions and Local Management

```
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>

#define DISCONNECT -1
#define SRV_ADDR  1   /* server's well-known address */

int conn_fd;             /* connection established here */
extern int t_errno;

void main()
{
    int listen_fd;        /* listening transport endpoint */
    struct t_bind *bind;
    struct t_call *call;

    if ((listen_fd = t_open("/dev/ticotsord", O_RDWR, NULL))
        < 0) {
        t_error("t_open failed for listen_fd");
        exit(1);
    }
```

```
    /*
     * By assuming that the address is an integer value,
     * this program may not run over another protocol.
     */

    if ((bind = (struct t_bind *)t_alloc(listen_fd,
        T_BIND, T_ALL))== 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 (*(int *)bind->addr.buf != SRV_ADDR)  {
        fprintf(stderr, "t_bind bound wrong address\n");
        exit(4);
    }
```

As with the client, the first step is to call **t_open()** to establish a transport endpoint with the desired transport provider. This endpoint, *listen_fd*, is used to listen for connect indications. Next, the server must bind its well-known address to the endpoint. This address is used by each client to access the server. The second argument to **t_bind()** requests that a particular address be bound to the transport endpoint. This argument points to a *t_bind* structure with the following format:

```
struct t_bind {
    struct netbuf addr;
    unsigned qlen;
}
```

addr describes the address to be bound, and *qlen* specifies the maximum outstanding connect indications that can arrive at this endpoint. All Transport Interface structure and constant definitions are found in <*tiuser.h*>.

The address is specified using a *netbuf* structure that contains the following members:

```
struct netbuf {
    unsigned int maxlen;
    unsigned int len;
    char *buf;
}
```

buf points to a buffer containing the data, *len* specifies the number of bytes of data in the buffer, and *maxlen* specifies the maximum number of bytes the buffer can hold (and need only be set when data is returned to the user by a Transport Interface routine). For the *t_bind* structure, the data pointed to by *buf* identifies a transport address. The structure of addresses is likely to vary between protocol implementations under the Transport Interface; the *netbuf* structure is intended to support any address structure.

If the value of *qlen* is greater than 0, the transport endpoint can be used to listen for connect indications. In such cases, **t_bind()** directs the transport provider to begin queueing connect indications destined for the bound address immediately. Furthermore, the value of *qlen* specifies the maximum outstanding connect indications the server wishes to process. The server must respond to each connect indication, either accepting or rejecting the request for connection. An outstanding connect indication is one to which the server has not yet responded. Often, a server fully processes a single connect indication and responds to it before receiving the next indication. When this occurs, a value of 1 is appropriate for *qlen*. However, some servers may wish to retrieve several connect indications before responding to any of them. In such cases, *qlen* specifies the maximum number of outstanding indications the server processes. An example of a server that manages multiple outstanding connect indications is presented in "Advanced Topics" on page 229.

t_alloc() is called to allocate the *t_bind* structure needed by **t_bind()**. **t_alloc()** takes three arguments. The first is a file descriptor that references a transport endpoint. This is used to access the characteristics of the transport provider (see t_open(3N)). The second argument identifies the appropriate Transport Interface structure to be allocated. The third argument specifies which, if any, *netbuf* buffers should be allocated for that structure. T_ALL specifies that all *netbuf* buffers associated with the structure should be allocated, and causes the *addr* buffer to be allocated in this example. The size of this buffer is determined from the transport provider characteristic that defines the maximum address size. The *maxlen* field of this *netbuf* structure is set to the size of the buffer allocated by **t_alloc()**. The use of **t_alloc()** helps ensure the compatibility of user programs with future releases of the Transport Interface.

The server in this example processes connect indications one at a time, so *qlen* is set to 1. The address information is then assigned to the newly allocated *t_bind* structure. This *t_bind* structure is passed to **t_bind()** as both the second and third arguments; as the second argument, it contains information for **t_bind()**, while as the third argument, it returns information to the user.

On return, the *t_bind* structure contains whatever address was bound to the transport endpoint. If the provider can't bind the requested address (perhaps because it's already bound to another transport endpoint), it returns another appropriate address.

Note: Each transport provider manages its address space differently. Some transport providers can allow a single transport address to be bound to several transport endpoints, while others can require a unique address per endpoint. The Transport Interface supports either choice. Based on its address management 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 given transport endpoint.

The server must check the bound address to ensure that it is the one previously advertised to clients. Otherwise, the clients are unable to reach the server.

If **t_bind()** succeeds, the provider begins queueing connect indications, entering the next phase of communication, connection establishment.

Connection Establishment

The connection establishment procedures highlight the distinction between clients and servers. The Transport Interface imposes a different set of procedures in this phase for each type of transport user. The client starts the connection establishment procedure by requesting a connection to a particular server using **t_connect()**. The server is then notified of the client's request by calling **t_listen()**. The server can either accept or reject the client's request. It calls **t_accept()** to establish the connection, or calls **t_snddis()** to reject the request. The client is notified of the server's decision when **t_connect()** completes. For more information, see t_connect(3N), t_listen(3N), t_accept(3N), and t_snddis(3N).

The Transport Interface supports two facilities during connection establishment that are not necessarily supported by all transport providers:

- The ability to transfer data between the client and server when establishing the connection.

  The client can send data to the server when it requests a connection. This data is passed to the server by **t_listen()**. Similarly, 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 can transfer during connect establishment.

- The negotiation of protocol options.

  The client can specify protocol options that it would like the transport provider and/or the other user to support. The Transport Interface supports both local and remote option negotiation. As discussed earlier, option negotiation is inherently a protocol-specific function. If you want protocol-independent software, you should not use this facility (see "Guidelines for Protocol Independence" on page 243).

The Client

Continuing with the client/server example, the steps needed by the client to establish a connection are as follows:

```
/* Since it assumes that the address is an integer value,
 * this program may not run over another protocol.
 */
if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL) {
    t_error("t_alloc failed");
    exit(3);
}
sndcall->addr.len = sizeof(int);
*(int *)sndcall->addr.buf = SRV_ADDR;

if (t_connect(fd, sndcall, NULL) < 0) {
    t_error("t_connect failed for fd");
    exit(4);
}
```

The **t_connect()** call establishes the connection with the server. The first argument to **t_connect()** identifies the transport endpoint through which the connection is established, and the second argument identifies the destination server. This argument is a pointer to a *t_call* structure with 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* can be used to specify protocol-specific options that the client would like to associate with the connection, and *udata* identifies user data that can be sent with the connect request to the server. The sequence field has no meaning for **t_connect()**.

t_alloc() is called to allocate the *t_call* structure dynamically. Once allocated, the appropriate values are assigned. In this example, no options or user data items are associated with the **t_connect()** call, but the server's address must be set. The third argument to **t_alloc()** is set to T_ADDR to specify that an appropriate *netbuf* buffer should be allocated for the address. The server's address is then assigned to *buf*, and *len* is set accordingly.

The third argument to **t_connect()** can be used to return information to the user about the newly established connection, and can be used to retrieve any user data sent by the server in its response to the connect request. It is set to NULL by the client here to indicate that this information is not needed. The connection is established on the successful return of **t_connect()**. If the server rejects the connect request, **t_connect()** fails and sets *t_errno* to TLOOK.

Event Handling

The TLOOK error has special significance in the Transport Interface. TLOOK notifies the user if a Transport Interface routine is interrupted by an unexpected asynchronous transport event on the given transport endpoint. As such, TLOOK does not report an error with a Transport Interface routine, but the normal processing of that routine is not performed because of the pending event. The events defined by the Transport Interface are listed as follows:

T_LISTEN — A request for a connection, called a connect indication, has arrived at the transport endpoint.

T_CONNECT — The confirmation of a previously sent connect request, called a connect confirmation, has arrived at the transport endpoint. The confirmation is generated when a server accepts a connect request.

T_DATA — User data has arrived at the transport endpoint.

T_EXDATA — Expedited user data has arrived at the transport endpoint. Expedited data is discussed later in this section.

T_DISCONNECT — A notification that the connection was aborted or that the server rejected a connect request, called a disconnect indication, has arrived at the transport endpoint.

T_ERROR — A notification that a fatal error has occurred.

T_UDERR — A notification of an error in a previously sent datagram, called a unitdata error indication, has arrived at the transport endpoint (see “Introduction to Connectionless-Mode Service” on page 219).

T_ORDREL — A request for the orderly release of a connection, called an orderly release indication, has arrived at the transport endpoint.

It is possible in some states to receive one of several asynchronous events, as described in the state tables of “State Transitions” on page 237. The **t_look()** routine enables a user to determine what event has occurred if a TLOOK error is returned. The user can then process that event accordingly. In the example, if a connect request is rejected, the event passed to the client is a disconnect indication. The client exits if its request is rejected.

The Server

Returning to the example, when the client calls **t_connect()**, a connect indication is generated on the server's listening transport endpoint. The steps required by the server to process the event are discussed below. For each client, the server accepts the connect request and spawns a server process to manage the connection as follows:

```
if ((call = (struct t_call *)t_alloc(listen_fd, T_CALL,
                                     T_ALL)) == NULL) {
    t_error("t_alloc of t_call structure failed");
    exit(5);
}
while (1) {
    if (t_listen(listen_fd, call) < 0) {
        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 loops forever, processing each connect indication. First, the server calls **t_listen()** to retrieve the next connect indication. When one 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 value of that endpoint. *conn_fd* is a global variable that identifies the transport endpoint where the connection is established. Because the connection is accepted on an alternate endpoint, the server can continue listening for connect indications on the endpoint that was bound for listening. If the call is accepted without error, **run_server()** spawns a process to manage the connection.

The server allocates a *t_call* structure to be used by **t_listen()**. The third argument to **t_alloc()**, T_ALL, specifies that all necessary buffers should be allocated for retrieving the caller's address, options, and user data. As mentioned earlier, the transport provider in this example does not support the transfer of user data during connection establishment, and also does not support any protocol options. Therefore, **t_alloc()** does not allocate buffers for the user data and options. It must, however, allocate a buffer large enough to store the address of the caller. **t_alloc()** determines the buffer size from the *addr* characteristic returned by **t_open()**. The *maxlen* field of each *netbuf* structure is set to the size of the buffer allocated by **t_alloc()** (*maxlen* is 0 for the user data and options buffers).

Using the *t_call* structure, the server calls **t_listen()** to retrieve the next connect indication. If one is currently available, it is returned to the server immediately. Otherwise, **t_listen()** blocks until a connect indication arrives.

The Transport Interface supports an asynchronous mode for these routines, which prevents a process from blocking. This feature is discussed in "Advanced Topics" on page 229.

When a connect indication arrives, the server calls **accept_call()** to accept the client's request, as follows:

```
int accept_call(listen_fd, call)
int listen_fd;
struct t_call *call;
{
    int resfd;

    if ((resfd = t_open("/dev/ticotsord", O_RDWR, NULL))
        < 0) {
        t_error("t_open for responding fd failed");
        exit(7);
    }

    if (t_bind(resfd, NULL, NULL) < 0) {
        t_error("t_bind for responding fd failed");
        exit(8);
    }
    if (t_accept(listen_fd, resfd, call) < 0) {
        if (t_errno == TLOOK) {  /* must be a disconnect */
            if (t_rcvdis(listen_fd, NULL) < 0) {
                t_error("t_rcvdis failed for listen_fd");
                exit(9);
            }
            if (t_close(resfd) < 0) {
                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() takes two arguments:

- *listen_fd* identifies the transport endpoint where the connect indication arrived
- *call* is a pointer to a *t_call* structure that contains all information associated with the connect indication.

The server first establishes another transport endpoint by opening the *clone* device node of the transport provider and binding an address. As with the client, a NULL value is passed to **t_bind()** to specify that the user does not care what address is bound by the provider. The newly established transport endpoint, *resfd*, is used to accept 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. A connection can be accepted on the listening endpoint, but this 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 associated with the connect indication. 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 manages multiple outstanding connect indications. "Advanced Topics" presents an example of this situation. Also, the *t_call* structure should identify protocol options the user has requested and user data that can be passed to the client. Because the transport provider in this example does not support protocol options or the transfer of user data during connection establishment, the *t_call* structure returned by **t_listen()** can be passed without change to **t_accept()**.

For simplicity in the example, the server exits if either the **t_open()** or **t_bind()** call fails. **exit()** closes the transport endpoint associated with *listen_fd*, causing the transport provider to pass a disconnect indication to the client that requested the connection. This disconnect indication notifies the client that the connection was not established; **t_connect()** fails, setting *t_errno* to TLOOK.

t_accept() can fail if an asynchronous event has occurred on the listening transport endpoint before the connection is accepted, and *t_errno* is set to TLOOK. The state transition table in "State Transitions" on page 237 shows that the only event that can occur in this state with only one outstanding connect indication is a disconnect indication. This event can occur if the client decides to undo the connect request it had previously sent. If a disconnect indication arrives, the server must retrieve the disconnect indication using **t_rcvdis()**. This routine takes a pointer to a *t_discon* structure as an argument, which is used to retrieve information associated with a disconnect indication.

In this example, however, the server does not care to retrieve this information, so it sets the argument to NULL. After receiving the disconnect indication, **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 indications.

Figure 8-5 illustrates how the server establishes connections.

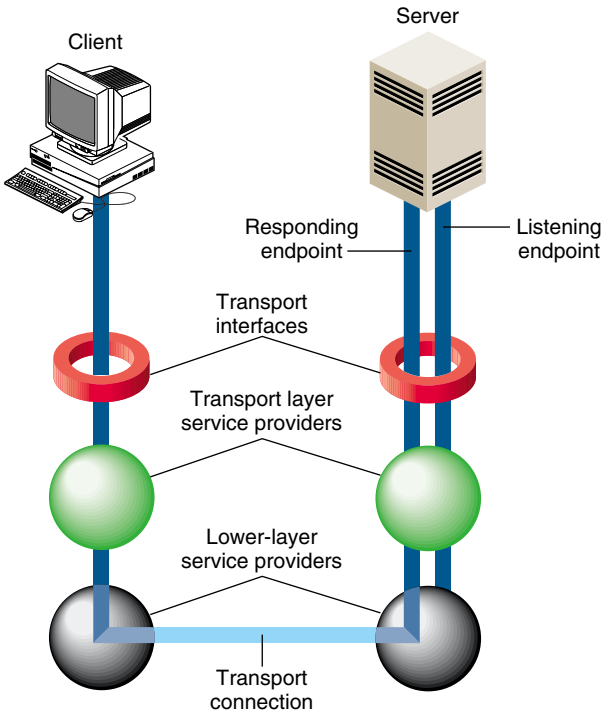


Figure 8-5 Listening and Responding Transport Endpoints

The transport connection is established on the newly created responding endpoint, and the listening endpoint is freed to retrieve further connect indications.

Data Transfer

Once the connection is established, both the client and server can begin transferring data over the connection using **t_snd()** and **t_rcv()**. The Transport Interface does not differentiate the client from the server from this point on. Either user can send and receive data or release the connection. The Transport Interface guarantees reliable, sequenced delivery of data over an existing connection.

Two classes of data can be transferred over a transport connection: normal data and expedited data.

Expedited data is typically associated with urgent information. The exact semantics of expedited data are subject to the interpretations of the transport provider. Furthermore, not all transport protocols support the notion of an expedited data class (see t_open(3N)).

All transport protocols support the transfer of data in byte stream mode, where *byte stream* implies no concept of message boundaries on data that's transferred over a connection. However, some transport protocols support the preservation of message boundaries over a transport connection. This service is supported by the Transport Interface, but protocol-independent software must not rely on its existence.

The message interface for data transfer is supported by a special flag of **t_snd()** and **t_rcv()** called T_MORE. The messages, called Transport Service Data Units (TSDU), can be transferred between two transport users as distinct units. The maximum TSDU size is a characteristic of the underlying transport protocol. This information is available to the user from **t_open()** and **t_getinfo()**. Because the maximum size can be large (possibly unlimited), the Transport Interface allows a user to transmit a message in multiple units.

To send a message in multiple units over a transport connection, the user must set the T_MORE flag on every **t_snd()** call except the last. This flag specifies that the user will send more data associated with the message in a subsequent call to **t_snd()**. The last message unit should be transmitted with T_MORE turned off to specify that this is the end of the TSDU.

Similarly, a TSDU can be passed in multiple units to the receiving user. Again, if **t_rcv()** returns with the T_MORE flag set, the user should continue calling **t_rcv()** to retrieve the remainder of the message. The last unit in the message is identified by a call to **t_rcv()** that does not set T_MORE.

Note: The T_MORE flag implies nothing about how the data can be packaged below the Transport Interface or how the data can be delivered to the receiver. Each transport protocol, and each implementation of that protocol, can 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 will deliver the data in a single unit to the remote transport user. Similarly, a message transmitted in two message units can be delivered in a single unit to the remote transport user. The message boundaries can only be preserved by noting the value of the T_MORE flag on **t_snd()** and **t_rcv()**. This guarantees that the receiving user sees a message with the same contents and message boundaries as that sent by the sender.

The Client

Continuing with the client/server example, the server transfers a log file to the client over the transport connection. The client receives this data and writes it to its standard output file. A byte stream interface is used by the client and server, where message boundaries (that is, the T_MORE flag) are ignored. The client receives data using the following instructions:

```
while ((nbytes = t_rcv(fd, buf, 1024, &flags)) != -1) {
    if (fwrite(buf, 1, nbytes, stdout) < 0) {
        fprintf(stderr, "fwrite failed\n");
        exit(5);
    }
}
```

The client continuously calls **t_rcv()** to process incoming data. If no data is currently available, **t_rcv()** blocks until data arrives. **t_rcv()** retrieves the available data up to 1024 bytes, which is the size of the client's input buffer, and returns the number of bytes received. The client then writes this data to standard output and continues. The data transfer phase completes when **t_rcv()** fails. **t_rcv()** fails if an orderly release or disconnect indication arrives, as discussed later in this section. If the **fwrite()** call (see fwrite(3S)) fails for any reason, the client exits, closing the transport endpoint. If the transport endpoint is closed (either by **exit()** or **t_close()**) during the data transfer phase, the connection is aborted and the other user receives a disconnect indication.

The Server

Looking now at the other side of the connection, the server manages its data transfer by spawning a child process to send the data to the client. The parent process then loops back to listen for further connect indications.

run_server() is called by the server to spawn this child process as shown in Example 8-3.

Example 8-3 Sending Data to a Client

```
void 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 indication - normal exit */
    exit(0);
 }

int 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) < 0) {
            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) < 0) {
            t_error("t_close failed for listen_fd");
            exit(22);
        }
        if ((logfp = fopen("logfile", "r")) == NULL) {
            perror("cannot open logfile");
            exit(23);
        }
        signal(SIGPOLL, connrelease);
        if (ioctl(conn_fd, I_SETSIG, S_INPUT) < 0) {
            perror("ioctl I_SETSIG failed");
            exit(24);
        }
    /* was disconnect there? */
        if (t_look(conn_fd) != 0) {
            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) < 0) {
                t_error("t_snd failed");
                exit(26);
            }
```

After the **fork()**, the parent process returns to the main processing loop and listens for further connect indications. Meanwhile, the child process manages the newly established transport connection. If the **fork()** call fails, **exit()** closes the transport endpoint associated with *listen_fd*, sending a disconnect indication 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 that data to the client using **t_snd()**. *buf* points to the start of the data buffer, and *nbytes* specifies the number of bytes to be transmitted. The fourth argument can contain one of the two optional flags as follows:

- T_EXPEDITED specifies that the data is expedited.
- T_MORE defines message boundaries when transmitting messages over a connection.

Neither flag is set by the server in this example.

If the user floods the transport provider with data, the provider can exert back pressure to provide flow control. In such cases, **t_snd()** blocks until the flow control is relieved, and then resumes its operation. **t_snd()** does not complete until *nbyte* bytes have been passed to the transport provider.

The **t_snd()** routine does not look for a disconnect indication (showing that the connection was broken) before passing data to the provider. Also, because the data traffic flows in one direction, the user never looks for incoming events. If the connection is aborted, the user should be notified since data can be lost. The user can invoke **t_look()**, which checks for incoming events before each **t_snd()** call. A more efficient solution is presented in Example 8-3. The STREAMS I_SETSIG **ioctl()** enables a user to request a signal when a given event occurs (see streamio(5) and signal(2)). S_INPUT causes a signal to be sent to the user if any input arrives on the Stream referenced by *conn_fd*. If a disconnect indication arrives, the signal catching routine (**connrelease()**) prints an error message and then exits.

If the data traffic flowed in both directions in this example, the user would not have to monitor the connection for disconnects. If the client alternated **t_snd()** and **t_rcv()** calls, it could rely on **t_rcv()** to recognize an incoming disconnect indication.

Connection Release

At any point during data transfer, either user can release the transport connection and end the conversation. As mentioned earlier, two forms of connection release are supported by the Transport Interface:

- Abortive release breaks a connection immediately and can result in the loss of any data that has not yet reached the destination user.

  Either user can call **t_snddis()** to generate an abortive release. Also, the transport provider can abort a connection if a problem occurs below the Transport Interface. **t_snddis()** enables a user to send data to the receiver when aborting a connection. Although the abortive release is supported by all transport providers, the ability to send data when aborting a connection is not.

When a user receives notification of the aborted connection, **t_rcvdis()** must be called to retrieve the disconnect indication. This call returns a reason code that identifies why the connection was aborted, and returns any user data that accompanied the disconnect indication (if the abortive release was initiated by the other user). This reason code is specific to the underlying transport protocol and should not be interpreted by protocol-independent software.

- Orderly release gracefully terminates a connection and guarantees that no data is lost.

All transport providers must support the abortive release procedure, but orderly release is an optional facility that is not supported by all transport protocols.

The Server

The client/server example in this section assumes that the transport provider supports the orderly release of a connection. When all the data has been transferred by the server, the connection can be released as follows:

```
        if (t_sndrel(conn_fd) < 0) {
            t_error("t_sndrel failed");
            exit(27);
        }
        pause();
    /* until orderly release indication arrives */
    }
```

The orderly release procedure consists of two steps by each user. The first user to complete data transfer can initiate a release using **t_sndrel()**, as illustrated in the example. This routine informs the client that no more data will be sent by the server. When the client receives this indication, it can continue sending data back to the server if desired. When all data has been transferred, however, the client must also call **t_sndrel()** to indicate that it is ready to release the connection. The connection is released only after both users have requested an orderly release and received the corresponding indication from the other user.

In this example, data is transferred in one direction from the server to the client, so the server does not expect to receive data from the client after it has initiated the release procedure. Thus, the server simply calls **pause()** (see pause(2))after initiating the release. Eventually, the client responds with its orderly release request, which generates a signal that is caught by **connrelease()**. Remember that the server earlier issued an I_SETSIG **ioctl()** call to generate a signal on any incoming event. Since the only possible Transport Interface events that can occur in this situation are a disconnect indication or an orderly release indication, **connrelease()** terminates normally when the orderly release indication arrives. The **exit()** call in **connrelease()** closes the transport endpoint, freeing the bound address for another user. If a user process wants to close a transport endpoint without exiting, it can call **t_close()**.

The Client

The client's view of connection release is similar to that of the server. As mentioned earlier, the client continues to process incoming data until **t_rcv()** fails. If 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) < 0) {
        t_error("t_rcvrel failed");
        exit(6);
    }
    if (t_sndrel(fd) < 0) {
        t_error("t_sndrel failed");
        exit(7);
    }
    exit(0);
}
t_error("t_rcv failed");
exit(8);
```

When an event occurs on the client's transport endpoint, the client checks whether the expected orderly release indication has arrived. If so, it proceeds with the release procedures by calling **t_rcvrel()** to process the indication and **t_sndrel()** to inform the server that it is also ready to release the connection. At this point the client exits, closing its transport endpoint.

Because not all transport providers support the orderly release facility just described, users may have to use the abortive release facility provided by **t_snddis()** and **t_rcvdis()**. However, steps must be taken by each user to prevent data loss. For example, a special byte pattern can be inserted in the datastream to indicate the end of a conversation. There are many possible routines for preventing data loss. Each application and high-level protocol must choose an appropriate routine given the target protocol environment and requirements.

Introduction to Connectionless-Mode Service

This section describes the connectionless-mode service of the Transport Interface. Connectionless-mode service is appropriate for short-term request/response interactions, such as transaction-processing applications. Data is transferred in self-contained units with no logical relationship required among multiple units.

The connectionless-mode service is described using a transaction server as an example. This server waits for incoming transaction queries, and processes and responds to each query.

Local Management

Just as with connection-mode service, the transport users must do appropriate local management steps before transferring data. A user must choose the appropriate connectionless service provider using **t_open()** and establish its identity using **t_bind()**. See the t_open(3N) man page or the other "Local Management" section of this chapter (under "Introduction to Connection-Mode Service") for information about what **t_open()** returns.

t_optmgmt() can be used to negotiate protocol options associated with the transfer of each data unit. As with the connection-mode service, each transport provider specifies the options, if any, that it supports. Option negotiation is therefore a protocol-specific activity.

The definitions and local management calls needed by the transaction server are shown in Example 8-4:

Example 8-4 The Transaction Server Definitions and Local Management

```
#include <stdio.h>
#include <fcntl.h>
#include <tiuser.h>

#define SRV_ADDR  2     /* server's well-known address */

void 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/ticlts", O_RDWR, NULL)) < 0) {
        t_error("unable to open /dev/provider");
        exit(1);
    }

    if ((bind = (struct t_bind *)t_alloc(fd, T_BIND,
                T_ADDR)) == 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) < 0)  {
        t_error("t_bind failed");
        exit(3);
    }

    /*
     * is the bound address correct?
     */
```

```
if (*(int *)bind->addr.buf != SRV_ADDR)
{
    fprintf(stderr, "t_bind bound wrong address\n");
    exit(4);
}
```

The local management steps should look familiar by now. The server establishes a transport endpoint with the desired transport provider using **t_open()**. Each provider has an associated service type, so the user can 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()** in the same way as the users in the connection-mode example, by setting the third argument to NULL. For simplicity, 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 also binds a transport address to the endpoint so that potential clients can identify and access the server. A *t_bind* structure is allocated using **t_alloc()**, and the *buf* and *len* fields of the address are set accordingly.

One important difference between the connection-mode server and this connectionless-mode server is that the *qlen* field of the *t_bind* structure has no meaning for connectionless-mode service, since all users are capable of receiving datagrams once they have bound an address. The Transport Interface defines an inherent client/server relationship between two users while establishing a transport connection in the connection-mode service. However, no such relationship exists in the connectionless-mode service. It is the context of this example, not the Transport Interface, that defines one user as a server and another 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 can be sent or received over that endpoint. Each outgoing message is accompanied by the address of the destination user. In addition, the Transport Interface enables a user to specify protocol options that should be associated with the transfer of the data unit (for example, transit delay). As discussed earlier, each transport provider defines the set of options, if any, that can accompany a datagram. When the datagram is passed to the destination user, the associated protocol options can be returned as well.

The sequence of calls in Example 8-5 illustrates the data transfer phase of the connectionless-mode server:

Example 8-5 The Data Transfer Phase of a Connectionless-Mode Server

```
if ((ud = (struct t_unitdata *)t_alloc(fd, T_UNITDATA,
                                       T_ALL)) == NULL)  {
        t_error("t_alloc of t_unitdata structure failed");
        exit(5);
}

if ((uderr = (struct t_uderr *)t_alloc(fd, T_UDERROR,
                                       T_ALL)) == NULL)  {
        t_error("t_alloc of t_uderr structure failed");
        exit(6);
}

    while (1)  {
        if (t_rcvudata(fd, ud, &flags) < 0)  {
            if (t_errno == TLOOK)  {
                /* Error on previously sent datagram */
                if (t_rcvuderr(fd, uderr) < 0)  {
                    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 < 0)  {
            t_error("t_sndudata failed");
            exit(9);
        }
    }
}
query()
{
    /* Merely a stub for simplicity */
}
```

The server must first allocate a *t_unitdata* structure for storing datagrams, 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* identifies any protocol options associated with the transfer of the datagram, and *udata* holds the data itself. The *addr*, *opt*, and *udata* fields must all be allocated with buffers large enough to hold any possible incoming values. As described in the previous section, the T_ALL argument to **t_alloc()** ensures this and sets the *maxlen* field of each *netbuf* structure accordingly. Because the provider does not support protocol options in this example, no options buffers are allocated, and *maxlen* is set to zero in the *netbuf* structure for options. The server also allocates a *t_uderr* structure for processing any datagram errors, as discussed later in this section.

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()** retrieves the next available incoming datagram. If none is currently available, **t_rcvudata()** blocks, waiting for a datagram to arrive. The second argument of **t_rcvudata()** identifies the *t_unitdata* structure in which the datagram should be stored.

The third argument, *flags*, must point to an integer variable and can be set to T_MORE on return from **t_rcvudata()** to specify that the user's *udata* buffer was not large enough to store the full datagram. In this case, subsequent calls to **t_rcvudata()** retrieve the remainder of the datagram. Because **t_alloc()** allocates a *udata* buffer large enough to store the maximum datagram size, the transaction server does not have to check the value of *flags*.

If a datagram is received successfully, the transaction server calls the **query()** routine to process the request. This routine stores the 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 used as the destination address by **t_sndudata()**.

When the response is ready, **t_sndudata()** is called to return the response to the client. The Transport Interface prevents a user from flooding the transport provider with datagrams using the same flow control mechanism described for the connection-mode service. In such cases, **t_sndudata()** blocks until the flow control is relieved, and then resumes its operation.

Datagram Errors

If the transport provider cannot process a datagram that was passed to it by **t_sndudata()**, it returns a unit data error event, T_UDERR, to the user. This event includes the destination address and options associated with the datagram, plus a protocol-specific error value that describes what could be wrong with the datagram. The reason a datagram could not be processed is protocol-specific. One reason can be that the transport provider could not interpret the destination address or options. Each transport protocol is expected to specify all reasons why it is unable to process a datagram.

Note: The unit data error indication is not necessarily intended to indicate success or failure in delivering the datagram to the specified destination. The transport protocol decides how the indication is used. Remember, the connectionless service does not guarantee reliable delivery of data.

The transaction server is notified of this error event when it attempts to receive another datagram. In this case, **t_rcvudata()** fails, 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 to **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;
 }
```

addr and *opt* identify the destination address and protocol options as specified in the bad datagram, and *error* is a protocol-specific error code that specifies why the provider could not process the datagram. The transaction server prints the error code and then continues by entering the processing loop again.

A Read/Write Interface

A user may wish to establish a transport connection and then **exec()** (see exec(2)) an existing user program such as *cat* to process the data as it arrives over the connection. However, existing programs use **read()** and **write()** for their input/output needs. The Transport Interface does not directly support a read/write interface to a transport provider, but one is available with IRIX. This interface enables a user to issue **read()** and **write()** calls over a transport connection that is in the data transfer phase. This section describes the read/write interface to the connection-mode service of the Transport Interface. This interface is not available with the connectionless-mode service.

The read/write interface is presented using the client example of "Introduction to Connection-Mode Service" with some minor modifications. The clients are identical until the data transfer phase is reached. At that point, this client uses the read/write interface and *cat* to process incoming data. **cat()** can be run without change over the transport connection. Only the differences between this client and that of the example in "Introduction to Connection-Mode Service" are shown below:

```
#include <stropts.h>
    ...
    /*
     * Same local management and connection
     * establishment steps.
     */
    ...
```

```
    if (ioctl(fd, I_PUSH, "tirdwr") < 0) {
        perror("I_PUSH of tirdwr failed");
        exit(5);
    }

    close(0);
    dup(fd);
    execl("/usr/bin/cat", "/usr/bin/cat", 0);
    perror("execl of /usr/bin/cat failed");
    exit(6);
}
```

The client invokes the read/write interface by pushing the *tirdwr* (see tirdwr(7)) module onto the Stream associated with the transport endpoint where the connection was established (see I_PUSH in streamio(5)). This module converts the Transport Interface above the transport provider into a pure read/write interface. With the module in place, the client calls **close()** and **dup()** (see close(2) and dup(2)) to establish the transport endpoint as its standard input file, and uses */usr/bin/cat* to process the input. Because the transport endpoint identifier is a file descriptor, the facility for **dup()**ing the endpoint is available to users.

Because the Transport Interface uses STREAMS, the facilities of this character input/output mechanism can be used to provide enhanced user services. By pushing the *tirdwr* module above the transport provider, the user's interface is effectively changed. The semantics of **read()** and **write()** must be followed, and message boundaries are not preserved.

Note: The *tirdwr* module can only be pushed onto a Stream when the transport endpoint is in the data transfer phase. Once the module is pushed, the user cannot call any Transport Interface routines. If a Transport Interface routine is invoked, *tirdwr* generates a fatal protocol error, EPROTO, on that Stream, rendering it unusable. Furthermore, if the user pops the *tirdwr* module off the Stream (see I_POP in streamio(5)), the transport connection is aborted.

The exact semantics of **write()**, **read()**, and **close()** using *tirdwr* are described below. To summarize, *tirdwr* enables a user to send and receive data over a transport connection using **read()** and **write()**. This module translates all Transport Interface indications into the appropriate actions. The connection can be released with the **close()** system call.

write()

The user can transmit data over the transport connection using **write()**. The *tirdwr* module passes data through to the transport provider. However, if a user attempts to send a zero-length data packet, which the STREAMS mechanism allows, *tirdwr* discards the message. If the transport connection is aborted (for example, because the other user aborts the connection using **t_snddis()**), a STREAMS hangup condition is generated on that Stream, and further **write()** calls fail and set *errno* to ENXIO. The user can still retrieve any available data after a hangup.

read()

read() can be used to retrieve data that has arrived over the transport connection. The *tirdwr* module passes data through to the user from the transport provider. However, any other event or indication passed to the user from the provider is processed by *tirdwr* as follows:

- **read()** cannot process expedited data because it cannot distinguish expedited data from normal data for the user. If an expedited data indication is received, *tirdwr* generates a fatal protocol error, EPROTO, on that Stream. This error causes further system calls to fail. You should therefore not communicate with a process that is sending expedited data.
- If an abortive disconnect indication is received, *tirdwr* discards it and generates a hangup condition on that Stream. Subsequent **read()** calls retrieve any remaining data, and then **read()** returns 0 for all further calls (indicating end-of-file).
- If an orderly release indication is received, *tirdwr* discards the indication and delivers a zero-length message to the user. As described in read(2), this notifies the user of end-of-file by returning 0.
- If any other Transport Interface indication is received, *tirdwr* generates a fatal protocol error, EPROTO, on that Stream. This causes further system calls to fail. If a user pushes *tirdwr* onto a Stream after the connection has been established, no indication is generated.

close()

With *tirdwr* on a Stream, the user can send and receive data over a transport connection for the duration of that 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* takes the following actions:

- If an orderly release indication was previously received by *tirdwr*, an orderly release request is passed to the transport provider to complete the orderly release of the connection. The user who initiated the orderly release procedure receives the expected indication when data transfer completes.
- If a disconnect indication was previously received by *tirdwr*, no special action is taken.
- If neither an orderly release indication nor a disconnect indication 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 indication 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, but *tirdwr* handles an orderly release properly if it is initiated by the user on the other side of a transport connection. If the client described in this section is communicating with the server program in "Introduction to Connection-Mode Service," that server terminates the transfer of data with an orderly release request. The server then waits for the corresponding indication from the client. At that point, the client exits and the transport endpoint is closed. When the file descriptor is closed, as explained in the first bulleted item above, *tirdwr* initiates the orderly release request from the client's side of the connection. This generates the indication that the server is expecting, and the connection is released properly.

Advanced Topics

This section presents the following important concepts of the Transport Interface that have not been covered in the previous section:

- An optional nonblocking (asynchronous) mode for some library calls
- An advanced programming example that defines a server supporting multiple outstanding connect indications and operating in an event-driven manner

Asynchronous Execution Mode

Many Transport Interface library routines can block waiting for an incoming event or the relaxation of flow control. However, some time-critical applications should not block for any reason. Similarly, an application may wish to do local processing while waiting for some asynchronous transport interface event.

Support for asynchronous processing of Transport Interface events is available to applications using a combination of the STREAMS asynchronous features and the nonblocking mode of the Transport Interface library routines. Earlier examples in this chapter have illustrated the use of the **poll()** system call and the I_SETSIG **ioctl()** command for processing events asynchronously.

In addition, any Transport Interface routine that can block while waiting for some event can be run in a special nonblocking mode. For example, **t_listen()** normally blocks waiting for a connect indication. However, a server can periodically poll a transport endpoint for existing connect indications by calling **t_listen()** in the nonblocking (or asynchronous) mode. The asynchronous mode is enabled by setting O_NDELAY or O_NONBLOCK on the file descriptor. These can be set as a flag on **t_open()** or by calling **fcntl()** (see fcntl(2)) before calling the Transport Interface routine. **fcntl()** can be used to enable or disable this mode at any time. All programming examples in this chapter use the default synchronous processing mode.

O_NDELAY or O_NONBLOCK affect each Transport Interface routine differently. To determine the exact semantics of O_NDELAY or O_NONBLOCK for a particular routine, see the relevant manual pages.

Advanced Programming Example

The example in Example 8-6 demonstrates two important concepts. The first is a server's ability to manage multiple outstanding connect indications. The second is an illustration of the ability to write event-driven software using the Transport Interface and the system call interface.

The server example in Example 8-6 is capable of supporting only one outstanding connect indication, but the Transport Interface supports the ability to manage multiple outstanding connect indications. One reason a server might wish to receive several simultaneous connect indications is to impose a priority scheme on each client. A server can retrieve several connect indications, and then accept them in an order based on a priority associated with each client. A second reason for handling several outstanding connect indications is that the single-threaded scheme has some limitations. Depending on the implementation of the transport provider, it is possible that while the server is processing the current connect indication, other clients will find it busy. If, however, multiple connect indications can be processed simultaneously, the server will be found to be busy only if the maximum allowed number of clients attempt to call the server simultaneously.

The server example in Example 8-6 is *event-driven: th*e process polls a transport endpoint for incoming Transport Interface events, and then takes the appropriate actions for the current event. The example demonstrates the ability to poll multiple transport endpoints for incoming events.

The definitions and local management functions needed by the example in Example 8-6 are similar to those of the server example in Example 8-4.

Example 8-6 An Advanced Server

```
#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 indications */
struct t_call  *calls[NUM_FDS][MAX_CONN_IND];

void main()
{
    struct pollfd pollfds[NUM_FDS];
    struct t_bind *bind;
    int i;

    /*
     * Only opening and binding one transport endpoint,
     * but more could be supported
     */
    if ((pollfds[0].fd = t_open("/dev/ticotsord", O_RDWR,
                                NULL)) < 0) {
        t_error("t_open failed");
        exit(1);
    }

    if ((bind = (struct t_bind *)t_alloc(pollfds[0].fd,
        T_BIND, T_ALL)) == 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 (t_bind(pollfds[0].fd, bind, bind) < 0)  {
        t_error("t_bind failed");
        exit(3);
    }

    /* Was the correct address bound? */
    if (*(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 (see poll(2)) that polls the transport endpoint for incoming data. Notice that 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 the above code.

An important aspect of this server is that it sets *qlen* to a value greater than 1 for **t_bind()**. This specifies that the server is willing to handle multiple outstanding connect indications. Remember that the earlier examples single-threaded the connect indications and responses. The server accepted the current connect indication before retrieving additional connect indications. This example, however, can retrieve up to MAX_CONN_IND connect indications at one time before responding to any of them. The transport provider can negotiate the value of *qlen* downward if it cannot support MAX_CONN_IND outstanding connect indications.

Once the server has bound its address and is ready to process incoming connect requests, it performs the following:

```
    pollfds[0].events = POLLIN;
    while (1) {
        if (poll(pollfds, NUM_FDS, -1) < 0)  {
            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 Transport Interface events. The server then enters an infinite loop, in which it **poll()**s the transport endpoint(s) for events, and then processes those events as they occur.

The **poll()** call blocks indefinitely, waiting for an incoming event. On return, each entry (corresponding to each transport endpoint) is checked for an existing event. If *revents* is set to 0, no event has occurred on that endpoint. In this case, the server continues to the next transport endpoint. If *revents* is set to POLLIN, an event does exist on the endpoint. In this case, **do_event()** is called to process the event. If *revents* contains any other value, an error must have occurred on the transport endpoint, and the server exits.

For each iteration of the loop, if any event is found on the transport endpoint, **service_conn_ind()** is called to process any outstanding connect indications. However, if another connect indication is pending, **service_conn_ind()** saves the current connect indication and responds to it later. This routine is explained shortly. If an incoming event is discovered, the routine in Example 8-7 is called to process it.

Example 8-7 Processing an Incoming Event

```
do_event(slot, 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] == NULL)
                break;
        }

        if ((calls[slot][i] = (struct t_call *)t_alloc(fd,
            T_CALL, T_ALL)) == NULL)   {
            t_error("t_alloc of t_call structure failed");
            exit(11);
        }

        if (t_listen(fd, calls[slot][i]) < 0)  {
            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) < 0)  {
            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((char*)calls[slot][i], T_CALL);
                calls[slot][i] = NULL;
            }
        }
        t_free(discon, T_DIS);
        break;
    }
}
```

This routine takes a number, *slot*, and a file descriptor, *fd*, as arguments. *slot* is used as an index into the global array *calls*. This array contains an entry for each polled transport endpoint, where each entry consists of an array of *t_call* structures that hold incoming connect indications for that transport endpoint. The value of *slot* is used to identify the transport endpoint.

do_event() calls **t_look()** to determine the Transport Interface event that has occurred on the transport endpoint specified by *fd*. If a connect indication (T_LISTEN event) or disconnect indication (T_DISCONNECT event) has arrived, the event is processed. Otherwise, the server prints an appropriate error message and exits.

For connect indications, **do_event()** scans the array of outstanding connect indications looking for the first free entry. A *t_call* structure is then allocated for that entry, and the connect indication is retrieved using **t_listen()**. There must always be at least one free entry in the connect indication array, because the array is large enough to hold the maximum number of outstanding connect indications as negotiated by **t_bind()**. The processing of the connect indication is deferred until later.

If a disconnect indication arrives, it must correspond to a previously received connect indication. This occurs if a client attempts to undo a previous connect request. In this case, **do_event()** allocates a *t_discon* structure to retrieve the relevant disconnect information. This structure has the following members:

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

udata identifies any user data that might have been sent with the disconnect indication, *reason* contains a protocol-specific disconnect reason code, and *sequence* identifies the outstanding connect indication that matches this disconnect indication.

Next, **t_rcvdis()** is called to retrieve the disconnect indication. The array of connect indications for *slot* is then scanned for one that contains a sequence number that matches the *sequence* number in the disconnect indication. When the connect indication is found, it is freed and the corresponding entry is set to NULL.

As mentioned earlier, if any event is found on a transport endpoint, **service_conn_ind()** is called to process all currently outstanding connect indications associated with that endpoint as follows:

```
service_conn_ind(slot, fd)
{
    int i;

    for (i = 0; i < MAX_CONN_IND; i++) {
        if (calls[slot][i] == NULL)
            continue;
```

```
        if ((conn_fd = t_open("/dev/ticotsord", O_RDWR,
                              NULL)) < 0) {
            t_error("open failed");
            exit(14);
        }
        if (t_bind(conn_fd, NULL, NULL) < 0) {
            t_error("t_bind failed");
            exit(15);
        }
        if (t_accept(fd, conn_fd, calls[slot][i]) < 0) {
            if (t_errno == TLOOK) {
                t_close(conn_fd);
                return;
            }
            t_error("t_accept failed");
            exit(16);
        }
        t_free((char*)calls[slot][i], T_CALL);
        calls[slot][i] = NULL;

        run_server(fd);
    }
}
```

For the given slot (the transport endpoint), the array of outstanding connect indications is scanned. For each indication, the server opens a responding transport endpoint, binds an address to the endpoint, and then accepts the connection on that endpoint. If another event (connect indication or disconnect indication) arrives before the current indication is accepted, **t_accept()** fails and sets *t_errno* to TLOOK.

Note: The user cannot accept an outstanding connect indication if any pending connect indication events or disconnect indication events exist on that transport endpoint.

If this error occurs, the responding transport endpoint is closed and **service_conn_ind()** returns immediately (saving the current connect indication for later processing). This causes the server's main processing loop to be entered, and the new event is discovered by the next call to **poll()**. In this way, multiple connect indications can be queued by the user.

Eventually, all events are processed, and **service_conn_ind()** is able to accept each connect indication in turn. Once the connection is established, the **run_server()** routine used by the server in "Introduction to Connection-Mode Service" is called to manage the data transfer.

State Transitions

These tables describe all state transitions associated with the Transport Interface. First, however, the states and events are described.

Transport Interface States

Table 8-6 defines the states used to describe the Transport Interface state transitions.

Table 8-6 States Describing Transport Interface State Transitions

State	Description	Service Type
T_UNINIT	Uninitialized—initial and final state of interface	T_COTS, T_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	T_COTS, T_COTS_ORD
T_DATAXFER	Data transfer	T_COTS, T_COTS_ORD
T_OUTREL	Outgoing orderly release (waiting for orderly release indication)	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 8-7 correspond to the return of the specified transport routines, where these routines send a request or response to the transport provider.

In the table, some events (such as *accept*n) 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 indications

fd file descriptor of the current transport endpoint

resfd file descriptor of the transport endpoint where a connection is accepted

Table 8-7 Outgoing Events

Event	Description	Service Type
open	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
close	Successful return of **t_close()**	T_COTS, T_COTS_ORD, T_CLTS
sndudata	Successful return of **t_sndudata()**	T_CLTS
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 indication arriving on the transport endpoint	T_COTS, T_COTS_ORD
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

Incoming Events

The incoming events correspond to the successful return of the specified routines, where these routines retrieve 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 user transfers a connection to another transport endpoint. This event occurs on the endpoint that is being passed the connection, despite the fact that no Transport Interface routine is issued on that endpoint. *pass_conn* is included in the state tables to describe the behavior when a user accepts a connection on another transport endpoint.

In Table 8-8, the *rcvdis* events are distinguished by the context in which they occur. The context is based on the value of *ocnt*, which is the count of outstanding connect indications on the transport endpoint.

Table 8-8 Incoming Events

Event	Description	Service Type
rcvudata	Successful return of **t_rcvudata()**	T_CLTS
rcvuderr	Successful return of **t_rcvuderr()**	T_CLTS
rcvconnect	Successful return of **t_rcvconnect()**	T_COTS, T_COTS_ORD
listen	Successful return of **t_listen()**	T_COTS, T_COTS_ORD
rcv	Successful return of **t_rcv()**	T_COTS, T_COTS_ORD
rcvdis1	Successful return of **t_rcvdis()** with ocnt <= 0	T_COTS, T_COTS_ORD
rcvdis2	Successful return of **t_rcvdis()** with 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
pass_conn	Receive a passed connection	T_COTS, T_COTS_ORD

Transport User Actions

In the state tables that follow, some state transitions are accompanied by a list of actions the transport user must take. These actions are represented by the notation [*x*], where *x* is a mnemonic for the specific action:

[0]	Set the count of outstanding connect indications to zero.
[+]	Increment the count of outstanding connect indications.
[-]	Decrement the count of outstanding connect indications.
[–>]	Pass a connection to another transport endpoint as indicated in **t_accept()**.

State Tables

Table 8-9, Table 8-10, and Table 8-11 describe the Transport Interface state transitions. Given a current state and an event, the transition to the next state is shown, as well as any actions that must be taken by the transport user (indicated by [*x*]). The state is that of the transport provider as seen by the transport user.

To see what the next state will be in a given situation, find the table cell at the intersection of the column headed by the current state and the row labeled with the current incoming or outgoing event. An empty cell represents a state/event combination that is invalid. Along with the next state, each cell can indicate one or more actions from among those listed in the previous section. The transport user must take the specific actions in the order specified in the state table.

The following should be understood when studying the state tables:

- The **t_close()** routine is referenced in the state tables (see *close* event in Table 8-9) but can be called from any state to close a transport endpoint. If **t_close()** is called when a transport address is bound to an endpoint, the address is unbound. Also, if **t_close()** is called when the transport connection is still active, the connection is aborted.
- If a transport user issues a routine out of sequence, the transport provider recognizes this and the routine fails, setting *t_errno* to TOUTSTATE. The state does not change.

- If any other transport error occurs, the state does not change unless explicitly stated on the manual page for that routine. The exception to this is a TLOOK or TNODATA error on **t_connect()**, as described in Table 8-7 under "connect2." The state tables assume correct use of the Transport Interface.
- The support routines **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.

Here are the state-transition tables: one for common local management steps; one for data transfer in connectionless mode; and one for connection establishment, connection release, and data transfer in connection mode.

Table 8-9 Common Local Management State Table

	T_UNINIT	T_UNBND	T_IDLE
open	T_UNBND		
bind		T_IDLE[0]	
optmgmt			T_IDLE
unbind			T_UNBND
close		T_UNINIT	

Table 8-10 Connectionless-Mode State Table

	T_IDLE
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

Table 8-11 Connection-Mode State Table

	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER	T_OUTREL	T_INREL
connect1	T_DATAXFER					
connect2	T_OUTCON					
rcvconnet		T_DATAXFR				
listen	T_INCON [+]		T_INCON [+]			
accept1			T_DATAXFER [-]			
accept2			T_IDLE [-] [–>]			
accept3			T_INCON [-] [–>]			
snd				T_DATAXFR		T_INRL
rcv				T_DATAXFR	T_OUTRL	
snddis1		T_IDLE	T_IDLE [-]	T_IDLE	T_IDLE	T_IDLE
snddis2			T_INCON [-]			
rcvdis1		T_IDLE		T_IDLE	T_IDLE	T_IDLE
rcvdis2			T_IDLE [-]			
rcvdis3			T_INCON [-]			
sndrel				T_OUTREL		T_IDLE
rcvrel				T_INREL	T_IDLE	
pass_conn	T_DATAXFER					

Guidelines for Protocol Independence

By defining a set of services common to many transport protocols, the Transport Interface provides many opportunities for protocol independence. However, there exist some transport protocols that do not support all of the services supported by the Transport Interface. If software must be run in a variety of protocol environments, only the common services should be accessed.

The following guidelines highlight services that may not be common to all transport protocols:

- In the connection-mode service, the concept of a transport service data unit (TSDU) may not be supported by all transport providers. The user should make no assumptions about the preservation of logical data boundaries across a connection. If messages must be transferred over a connection, a protocol should be implemented above the Transport Interface to support message boundaries.
- Protocol- and implementation-specific service limits are returned by the **t_open()** and **t_getinfo()** routines. These limits are useful when allocating buffers to store protocol-specific transport addresses and options. It is the responsibility of the user to access these limits and then adhere to the limits throughout the communication process.
- User data should not be transmitted with connect requests or disconnect requests (see t_connect(3N) and t_snddis(3N)). Not all transport protocols support this capability.
- The buffers in the *t_call* structure used for **t_listen()** must be large enough to hold any information passed by the client during connection establishment. The server should use the T_ALL argument to **t_alloc()**, which determines the maximum buffer sizes needed to store the address, options, and user data for the current transport provider.
- The user program should not look at or change options that are associated with any Transport Interface routine. These options are specific to the underlying transport protocol. The user should not pass options with **t_connect()** or **t_sndudata()**. In such cases, the transport provider uses default values. Also, a server should use the options returned by **t_listen()** when accepting a connection.

- Protocol-specific addressing issues should be hidden from the user program. A client should not specify any protocol address on **t_bind()**, but instead should allow the transport provider to assign an appropriate address to the transport endpoint. Similarly, a server should retrieve its address for **t_bind()** in such a way that it does not require knowledge of the transport provider's address space. Such addresses should not be hard-coded into a program. A name server procedure could be useful in this situation, but the details for providing this service are outside the scope of the Transport Interface. The reason codes associated with **t_rcvdis()** are protocol-dependent. The user should not interpret this information if protocol independence is important.
- The error codes associated with **t_rcvuderr()** are protocol-dependent. The user should not interpret this information if protocol independence is a concern.
- The names of devices should not be hard-coded into programs, because the device node identifies a particular transport provider and is not protocol independent.
- The optional orderly release facility of the connection-mode service (provided by **t_sndrel()** and **t_rcvrel()**) should not be used by programs targeted for multiple protocol environments. This facility is not supported by all connection-based transport protocols.

Some Examples

This section contains examples of complete client and server programs mentioned earlier in the chapter.

Connection-Mode Client

The code in Example 8-8 represents the connection-mode client program described in "Introduction to Connection-Mode Service."

This client establishes a transport connection with a server, and then receives data from the server and writes that data to the client's standard output. The connection is released using the orderly release facility of the Transport Interface. This client communicates with each of the connection-mode servers presented in the guide.

Example 8-8 A Connection-Mode Client

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>

#define SRV_ADDR 1   /* server's well-known address */

void main()
{
    int fd;
    int nbytes;
    int flags = 0;
    char buf[1024];
    struct t_call *sndcall;
    extern int t_errno;

    if ((fd = t_open("/dev/ticotsord", O_RDWR, NULL)) < 0) {
        t_error("t_open failed");
        exit(1);
    }

    if (t_bind(fd, NULL, NULL) < 0) {
        t_error("t_bind failed");
        exit(2);
    }

    /* By assuming that the address is an integer value,
     * this program may not run over another protocol. */

    if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL,
        T_ADDR)) == NULL) {
        t_error("t_alloc failed");
        exit(3);
    }
    sndcall->addr.len = sizeof(int);
    *(int *)sndcall->addr.buf = SRV_ADDR;

    if (t_connect(fd, sndcall, NULL) < 0) {
        t_error("t_connect failed for fd");
        exit(4);
    }
    while ((nbytes = t_rcv(fd, buf, 1024, &flags)) != -1){
        if (fwrite(buf, 1, nbytes, stdout) < 0){
            fprintf(stderr, "fwrite failed\n");
```

```
            exit(5);
        }
    }
    if ((t_errno == TLOOK)  &&  (t_look(fd) == T_ORDREL)) {
        if (t_rcvrel(fd) < 0) {
            t_error("t_rcvrel failed");
            exit(6);
        }
        if (t_sndrel(fd) < 0) {
            t_error("t_sndrel failed");
            exit(7);
        }
        exit(0);
    }
    t_error("t_rcv failed");
    exit(8);
}
```

Connection-Mode Server

The code in Example 8-9 represents the connection-mode server program described in "Introduction to Connection-Mode Service." This server establishes a transport connection with a client, and then transfers a log file to the client on the other side of the connection. The connection is released using the orderly release facility of the Transport Interface. The connection-mode client presented earlier communicates with this server.

Example 8-9 A Connection-Mode Server

```
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>

#define DISCONNECT -1
#define SRV_ADDR  1  /* server's well-known address */

int conn_fd;         /* connection established here */
extern int t_errno;

void main()
{
    int listen_fd;   /* listening transport endpoint */
```

```
    struct t_bind *bind;
    struct t_call *call;
    if ((listen_fd = t_open("/dev/ticotsord", O_RDWR, NULL))
        < 0) {
        t_error("t_open failed for listen_fd");
        exit(1);
    }
    /*
     * By assuming that the address is an integer value,
     * this program may not run over another protocol.
     */
    if ((bind = (struct t_bind *)t_alloc(listen_fd, T_BIND,
        T_ALL)) == 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 (*(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\n");
        exit(4);
    }
    if ((call = (struct t_call *)t_alloc(listen_fd, T_CALL,
                                         T_ALL)) == NULL) {
        t_error("t_alloc of t_call structure failed");
        exit(5);
    }
    while (1) {
        if (t_listen(listen_fd, call) < 0) {
            t_error("t_listen failed for listen_fd");
            exit(6);
        }
        if ((conn_fd = accept_call(listen_fd, call))
            != DISCONNECT)
            run_server(listen_fd);
    }
}
```

```
int accept_call(listen_fd, call)
int listen_fd; struct t_call *call;
{
    int resfd;

    if ((resfd = t_open("/dev/ticotsord", O_RDWR, NULL))
        < 0) {
        t_error("t_open for responding fd failed");
        exit(7);
    }
    if (t_bind(resfd, NULL, NULL) < 0) {
        t_error("t_bind for responding fd failed");
        exit(8);
    }
    if (t_accept(listen_fd, resfd, call) < 0) {
        if (t_errno == TLOOK) {  /* must be a disconnect */
            if (t_rcvdis(listen_fd, NULL) < 0) {
                t_error("t_rcvdis failed for listen_fd");
                exit(9);
            }
            if (t_close(resfd) < 0) {
                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);
}

void 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 indication - normal exit */
    exit(0);
 }
```

```
int 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) < 0) {
            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) < 0) {
            t_error("t_close failed for listen_fd");
            exit(22);
        }

        if ((logfp = fopen("logfile", "r")) == NULL) {
            perror("cannot open logfile");
            exit(23);
        }

        signal(SIGPOLL, connrelease);
        if (ioctl(conn_fd, I_SETSIG, S_INPUT) < 0) {
            perror("ioctl I_SETSIG failed");
            exit(24);
        }
        if (t_look(conn_fd) != 0) {
            /* disconnect wasn't 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) < 0) {
                t_error("t_snd failed");
                exit(26);
            }

        if (t_sndrel(conn_fd) < 0) {
            t_error("t_sndrel failed");
            exit(27);
        }
        pause(); /*until orderly release indication arrives*/
    }
}
```

Connectionless-Mode Transaction Server

The code in Example 8-10 represents the connectionless-mode transaction server program described in "Introduction to Connectionless-Mode Service."

This server waits for incoming datagram queries, and then processes each query and sends a response.

Example 8-10 A Connectionless-Mode Transaction Server

```
#include <stdio.h>
#include <fcntl.h>
#include <tiuser.h>

#define SRV_ADDR  2   /* server's well-known address */

void 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/ticlts", O_RDWR, NULL)) < 0) {
        t_error("unable to open /dev/provider");
        exit(1);
    }
```

```
if ((bind = (struct t_bind *)t_alloc(fd, T_BIND,
                                        T_ADDR)) == 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) < 0) {
    t_error("t_bind failed");
    exit(3);
}

/* is the bound address correct? */
if (*(int *)bind->addr.buf != SRV_ADDR) {
    fprintf(stderr, "t_bind bound wrong address\n");
    exit(4);
}

if ((ud = (struct t_unitdata *)t_alloc(fd, T_UNITDATA,
    T_ALL)) == NULL) {
    t_error("t_alloc of t_unitdata structure failed");
    exit(5);
}
if ((uderr = (struct t_uderr *)t_alloc(fd, T_UDERROR,
                                        T_ALL)) == NULL) {
    t_error("t_alloc of t_uderr structure failed");
    exit(6);
}

while (1)
    {
    if (t_rcvudata(fd, ud, &flags) < 0) {
        if (t_errno == TLOOK) {
            /* Error on previously sent datagram */
            if (t_rcvuderr(fd, uderr) < 0) {
                t_error("t_rcvuderr failed");
                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 < 0) {
            t_error("t_sndudata failed");
            exit(9);
        }
    }
}

query()
{
    /* Merely a stub for simplicity */
}
```

Read/Write Client

The code in Example 8-11 represents the connection-mode read/write client program described in "A Read/Write Interface." This client establishes a transport connection with a server, and then uses *cat* to retrieve the data sent by the server and write that data to the client's standard output. This client communicates with each of the connection-mode servers presented in the guide.

Example 8-11 A Connection-Mode Read/Write Client

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>
#include <stropts.h>

#define SRV_ADDR  1   /* server's well-known address */

void main()
{
    int fd;
    int nbytes;
```

```
    int flags = 0;
    char buf[1024];
    struct t_call *sndcall;
    extern int t_errno;

    if ((fd = t_open("/dev/ticotsord", O_RDWR, NULL)) < 0) {
        t_error("t_open failed");
        exit(1);
    }

    if (t_bind(fd, NULL, NULL) < 0) {
        t_error("t_bind failed");
        exit(2);
    }

     /* By assuming that the address is an integer value,
      * this program may not run over another protocol. */

    if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL,
        T_ADDR)) == NULL) {
        t_error("t_alloc failed");
        exit(3);
    }

    sndcall->addr.len = sizeof(int);
    *(int *)sndcall->addr.buf = SRV_ADDR;

    if (t_connect(fd, sndcall, NULL) < 0)
{
        t_error("t_connect failed for fd");
        exit(4);
    }
    if (ioctl(fd, I_PUSH, "tirdwr") < 0) {
        perror("I_PUSH of tirdwr failed");
        exit(5);
    }

    close(0);
    dup(fd);

    execl("/usr/bin/cat", "/usr/bin/cat", 0);
    perror("execl of /usr/bin/cat failed");
    exit(6);
}
```

Event-Driven Server

The code in Example 8-12 represents the connection-mode server program described in "Advanced Topics." This server manages multiple connect indications in an event-driven manner. Either connection-mode client presented earlier communicates with this server.

Example 8-12 A Connection-Mode Server

```
#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 indications */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];

void main()
{
    struct pollfd pollfds[NUM_FDS];
    struct t_bind *bind;
    int i;

    / * Only opening and binding one transport endpoint,
      * but more could be supported  */

    if ((pollfds[0].fd = t_open("/dev/ticotsord", O_RDWR,
        NULL)) < 0) {
        t_error("t_open failed");
        exit(1);
    }
```

```
    if ((bind = (struct t_bind *)t_alloc(pollfds[0].fd,
        T_BIND, T_ALL)) == 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 (t_bind(pollfds[0].fd, bind, bind) < 0) {
        t_error("t_bind failed");
        exit(3);
    }

    /* Was the correct address bound? */
    if (*(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\n");
        exit(4);
    }

    pollfds[0].events = POLLIN;

    while (1) {
        if (poll(pollfds, NUM_FDS, -1) < 0) {
            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);
            }
        }
    }
}
```

```
do_event(slot, 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] == NULL)
                break;
        }
        if ((calls[slot][i] = (struct t_call *)t_alloc(fd,
            T_CALL, T_ALL)) == NULL)  {
            t_error("t_alloc of t_call structure failed");
            exit(11);
        }
        if (t_listen(fd, calls[slot][i]) < 0) {
            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) < 0) {
```

```
            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((char*)calls[slot][i], T_CALL);
                calls[slot][i] = NULL;
            }
        }
        t_free((char*)discon, T_DIS);
        break;
    }
 }
service_conn_ind(slot, fd)
{
    int i;
    for (i = 0; i < MAX_CONN_IND; i++) {
        if (calls[slot][i] == NULL)
            continue;
        if ((conn_fd = t_open("/dev/ticotsord", O_RDWR,
                              NULL)) < 0){
            t_error("open failed");
            exit(14);
        }

        if (t_bind(conn_fd, NULL, NULL) < 0)  {
            t_error("t_bind failed");
            exit(15);
        }
        if (t_accept(fd, conn_fd, calls[slot][i]) < 0) {
            if (t_errno == TLOOK) {
                t_close(conn_fd);
                return;
            }
            t_error("t_accept failed");
            exit(16);
        }
        t_free((char*)calls[slot][i], T_CALL);
        calls[slot][i] = NULL;

        run_server(fd);
    }
 }
```

```
void 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 indication - normal exit */
    exit(0);
}

int 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) < 0) {
            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) < 0) {
            t_error("t_close failed for listen_fd");
            exit(22);
        }
        if ((logfp = fopen("logfile", "r")) == NULL) {
            perror("cannot open logfile");
            exit(23);
        }

        signal(SIGPOLL, connrelease);
        if (ioctl(conn_fd, I_SETSIG, S_INPUT) < 0) {
            perror("ioctl I_SETSIG failed");
            exit(24);
        }
        /* disconnect already there? */
        if (t_look(conn_fd) != 0) {
            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) < 0){
                t_error("t_snd failed");
                exit(26);
            }

        if (t_sndrel(conn_fd) < 0) {
            t_error("t_sndrel failed");
            exit(27);
        }
        pause();
    /* until orderly release indication arrives */
    }
}
```

Error Messages

The following errors have been added to TLI for X/Open Transport Interface (XTI) compatibility.

Note: XTI has changed the names specified by *struct_type* by appending _STR to the end. For example, to allocate a *t_bind* structure, the argument *struct_type* is T_BIND_STR instead of T_BIND. The old names will continue to be supported. **t_free()** supports the new names for the *struct_type* argument.

Upon failure, **t_accept()** sets *t_errno* to TBADADDR if the specified protocol address was in an incorrect format or contained illegal information.

Upon failure, **t_alloc()** sets *t_errno* to TNOSTRUCTYPE if the *struct_type* parameter is invalid.

Upon failure, **t_bind()** sets *t_errno* to TADDRBUSY if the requested address is in use and the transport provider can't allocate an new address.

t_look() includes two new events, and an existing event was removed. The T_ERROR event was removed because it can be handled by setting *t_errno* to TSYSERR. The new events, T_GODATA and T_GOEXDATA, are returned to indicate that flow-control restrictions on normal data flow (T_GODATA) or expedited data flow (T_GOEXDATA) have been lifted and data may be sent again.

Upon failure, **t_open()** sets *t_errno* to:

TBADFLAG if *oflag* is invalid.

TBADNAME if *name* is not a valid transport provider.

Upon failure, **t_rcv()**, **t_rcvconnect()**, **t_rcvdis()**, **t_rcvrel()**, and **t_rcvudata()** set *t_errno* to TOUTSTATE if the function was issued in the wrong sequence on this transport endpoint.

Upon failure, **t_snd()** sets *t_errno* to:

TBADFLAG	if *oflag* is invalid.
TLOOK	if an asynchronous event has occurred on this transport endpoint and requires immediate attention.
TOUTSTATE	if the function was issued in the wrong sequence on this transport endpoint.

Upon failure, **t_sndrel()** and **t_sndudata()** set *t_errno* to:

TLOOK	if an asynchronous event has occurred on this transport endpoint and requires immediate attention.
TOUTSTATE	if the function was issued in the wrong sequence on this transport endpoint.

Appendix A

RPC Protocol Specification

This chapter describes the RPC protocol, a message protocol that is specified with the XDR language and is used in implementing Sun's RPC package.

This chapter assumes you are familiar with both RPC and XDR, as described in this guide. It does not attempt to justify RPC or its uses. The casual user of RPC need not be familiar with the information in this chapter.

Topics in this chapter include:

- RPC protocol requirements
- RPC protocol definition
- authentication protocols
- RPC record-marking standard
- port mapper program protocol

Note: For details about RPC programming, see Chapter 5, "RPC Programming Guide." For information about the structure and syntax of XDR and RPC language, see Chapter 6, "XDR and RPC Language Structure."

RPC Protocol Requirements

The RPC protocol provides:

- unique specification of a procedure to be called
- provisions for matching response messages to request messages
- provisions for authenticating the caller to the service and vice versa

Besides these requirements, features that detect the following are worth supporting because of protocol roll-over errors, implementation bugs, user error, and network administration:

- RPC protocol mismatches
- remote program protocol version mismatches
- protocol errors (such as errors in specifying a procedure's parameters)
- reasons why remote authentication failed
- any other reasons why the desired procedure was not called

Remote Programs and Procedures

An 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 some central authority (see "Assigning RPC Program Numbers" in Chapter 3 for details). Once you have a program number, you can implement your remote program.

The version field of the call message identifies the version of the RPC protocol being used by the caller. Because most new protocols evolve into better, stable, and mature protocols, a version field identifies which version of the protocol the caller is using. Version numbers make it possible to speak old and new protocols through the same server process.

The procedure number identifies the procedure to be called. Such numbers are documented in the specific 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 actual RPC message protocol could also change. Therefore, the call message also has the RPC version number in it; this field must be two (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 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 situation is caused by a disagreement about the protocol between client and service.)

Message Authentication

The RPC protocol provides the fields necessary for a client to identify itself to a service and vice versa. 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 as the following *opaque* type:

```
enum auth_flavor {
    AUTH_NULL    = 0,
    AUTH_UNIX    = 1,
    AUTH_SHORT   = 2
/* and more to be defined */
};
struct opaque_auth {
    auth_flavor flavor;
    opaque body<400>;
};
```

In simple English, any *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 is specified by individual, independent authentication protocol specifications. (See "Authentication Protocols" on page 271 for definitions of the various authentication protocols.)

If authentication parameters are rejected, the response message contains information stating why they were rejected.

Other Uses of the RPC Protocol

The intended use of the RPC 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. Sun currently uses the RPC message protocol for the following two (non-RPC) protocols: batching (or pipelining) and broadcast RPC. These two protocols are discussed (but not defined) next.

Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server; batching uses reliable byte stream protocols (such as TCP/IP) for its transport. 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 terminated by a legitimate RPC in order to flush the pipeline (with positive acknowledgment).

Broadcast RPC

In broadcast RPC-based protocols, the client sends a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses unreliable, packet-based protocols (such as UDP/IP) as its transport. Servers that support broadcast protocols respond only when the request is successfully processed, and they are silent in the face of errors. Broadcast RPC uses the Port Mapper RPC service to achieve its semantics. See "Port Mapper Program Protocol" on page 274 for more information.

RPC Protocol Definition

This section defines the RPC protocol in the XDR data description language. The message is defined in a top-down style.

Note that this is an XDR specification, not C code:

```
enum msg_type {
    CALL = 0,
    REPLY = 1
};
/*
 * A reply to a call message can take 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 successfully executed */
    PROG_UNAVAIL   =1,  /* remote machine exports program */
    PROG_MISMATCH  =2,  /* remote can't support version num*/
    PROC_UNAVAIL   =3   /* prog can’t support procedure */
    GARBAGE_ARGS   =4   /* remote can't figure out params */
};
/*
 * Reasons why a call message was rejected:
 */
enum reject_stat {
    RPC_MISMATCH = 0,   /* RPC version number not 2 */
    AUTH_ERROR   = 1    /* caller not authenticated on */
                        /* remote */
};

/*
 * Why authentication failed:
 */
```

```
enum auth_stat {
  AUTH_BADCRED      = 1, /* bad credentials (seal broken) */
  AUTH_REJECTEDCRED = 2, /* have client begin new session */
  AUTH_BADVERF      = 3, /* bogus verifier (seal broken) */
  AUTH_REJECTEDVERF = 4, /* verifier expired or replayed */
  AUTH_TOOWEAK      = 5, /* rejected for security reasons */
};

/*
 * 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, and the
 * procedure within the remote program to be called.  These
 * fields are followed by 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 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.
 * Note: there could be an error even though the request
 * was accepted.  The first field is an authentication
 * verifier which 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_ARGS arms of the union are
 * void.  The PROG_MISMATCH arm specifies the lowest and
 * highest version numbers of the remote program that are
 * 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 because of two reasons: either
 * the server is not running a compatible version of the
 * RPC protocol (RPC_MISMATCH), or the server refused to
 * authenticate the caller (AUTH_ERROR).  In the case of
 * an RPC version mismatch, the server returns the lowest and
 * highest supported RPC version numbers. In the case of
 * refused authentication, the 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;
};
```

Authentication Protocols

As previously stated, authentication parameters are opaque but open-ended to the rest of the RPC protocol. This section defines some "flavors" of authentication in this implementation. Other sites are free to invent new authentication types, with the same rules of flavor number assignment as those for program number assignment.

Null Authentication

RPC calls are often made when the caller doesn't know its authentication parameters, and the server doesn't care. In this case, the *auth_flavor* value (the discriminant of the *opaque_auth*'s union) of the RPC message's credentials, verifier, and response verifier is AUTH_NULL(0). The bytes of the *opaque_auth*'s body are undefined. It is recommended that the opaque length be zero.

AUTH_UNIX Authentication

The caller of a remote procedure may want to identify itself as it is identified on a trusted UNIX system. The value of the *credential*'s discriminant of an RPC call message is AUTH_UNIX (1). The bytes of the *credential*'s opaque body encode the following structure:

```
struct auth_unix {
    unsigned int stamp;
    string machinename<255>;
    unsigned int uid;
    unsigned int gid;
    unsigned int gids<16>;
};
```

The *stamp* is an arbitrary ID that the caller machine may generate. The *machinename* is the name of the caller's machine (such as *krypton*). The *uid* is the caller's effective user ID. The *gid* is the caller's effective group ID. The *gid* is a counted array of groups that contain the caller as a member. The *verifier* accompanying the credentials should be of AUTH_NULL (defined in the previous section).

The value of the discriminate of the "response verifier" received in the reply message from the server may be AUTH_NULL or AUTH_SHORT(2). In the case of AUTH_SHORT, the bytes of the *response verifier*'s string encode an opaque structure. This new opaque structure may now be passed to the server instead of the original AUTH_UNIX flavor credentials. The server keeps a cache that maps shorthand opaque structures (passed back via 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 due to an authentication error. The reason for the failure will be AUTH_REJECTEDCRED. At this point, the caller may want to try the original AUTH_UNIX style of credentials.

Note: In an open environment, extra checks should be performed against the source and identity of the originator before accepting the credential values.

Trusted UNIX Systems

Authentication is based on the premise that one multi-user UNIX system should be able to accept and rely upon the user and group identification information from a trusted source. The criteria for such trust between two systems are as follows:

- Both systems are administered securely. This includes practices such as:
  - using passwords on all accounts, especially root.
  - ensuring all setuid-root programs and daemons that run as root are *trustworthy*, that is, they do not lie about their UID and are not easily fooled.
  - protecting system files (the kernel) through file system permissions.
- Both systems share a common set of user UIDs and GIDs, such as is implemented with NIS.

Systems that adhere to the criteria in the first bulleted item above are considered *equivalent*, and are typically named in the file */etc/hosts.equiv*.

As a result of following these criteria, a UNIX system believes the content of a credential is authentic if comes from a trusted and trustworthy source. A UNIX system attempts to assure itself that the credential has come from such a trusted source if:

- The packet is not self-inconsistent about its source (the host name in the credential maps to the IP source address of the packet).
- The packet bears a source address that maps into a list of trusted or equivalent hosts. Assuming that the list has been properly maintained, this assures the program that the source system is a UNIX system, with *privileged ports*.
- The credential information (that is, the UIDs and GIDs) are all known to the local system. This attempts to catch information from hosts that do not have equivalent (common) sets of users' UIDs and GIDs.
- The packet came from a privileged port on the source system. Given that the source system was a UNIX host, this implies that the packet came from a process running as root, which is trustworthy.

All of these actions constitute authentication, not access control. They attempt to answer the question "should we trust the identity information in this credential?" not the question "is the identity in this credential entitled to perform the requested RPC function?"

This entire premise of authentication based on trust of equivalent systems is dated. Personal computers and UNIX workstations that are individually administered are far less likely to be worthy of the level of trust suggested here than were the large multi-user systems that were kept in locked computer rooms, and whose root passwords were known only to a few trusted system administrators, so common years ago. Most typical modern UNIX workstation environments simply don't meet the criteria for *equivalent systems* any more.

Personal computers do not have root accounts, and no privileged ports. Any user on a personal computer can send packets from a port number that would be a privileged port if it were a UNIX system.

So, accepting an AUTH_UNIX RPC request from a system not known to be UNIX is risky.

Record Marking Standard

When RPC messages are passed on top of a byte stream protocol (such as TCP/IP), it is necessary, or at least desirable, to delimit one message from another in order to detect and possibly recover from user protocol errors. This is called record marking (RM). This implementation of RPC uses this RM/TCP/IP transport for passing RPC messages on TCP streams. One RPC message fits into one RM record.

A record is composed of one or more record fragments. A record fragment is a 4-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 from highest to lowest. The number encodes two values—a boolean, which indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment), and a 31-bit unsigned binary value, which 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. (Note that this record specification is not in XDR standard form.)

Port Mapper Program Protocol

The port mapper program maps RPC program and version numbers to transport-specific port numbers, which enables dynamic binding of remote programs.

This mapping is desirable because the range of reserved port numbers is very small and the number of potential remote programs is very large. By running only the port mapper on a reserved port, the port numbers of other remote programs can be ascertained by querying the port mapper.

The port mapper also aids in broadcast RPC. A given RPC program will usually have 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. 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 picks up the broadcast then calls the local service specified by the client. When the port mapper gets the reply from the local service, it sends the reply back to the client.

Port Mapper Protocol Specification

The following specifies the Port Mapper Protocol (in RPC language):

```
const PMAP_PORT = 111;      /* portmapper port number */

/*
 * A mapping of (program, version, protocol) to port number
 */
struct mapping {
    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 {
    mapping 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(mapping)       = 1;
        bool
        PMAPPROC_UNSET(mapping)     = 2;
        unsigned int
        PMAPPROC_GETPORT(mapping)   = 3;
        pmaplist
        PMAPPROC_DUMP(void)         = 4;
        call_result
        PMAPPROC_CALLIT(call_args)  = 5;
    } = 2;
} = 100000;
```

Port Mapper Operation

The port mapper program currently supports two protocols (UDP/IP and TCP/IP). The port mapper is contacted by talking to it on assigned port number 111 (*sunrpc* in */etc/services*) on either of these protocols.

Table A-1 contains a description of each port mapper procedure.

Table A-1 Port Mapper Procedures

Procedure	Description
PMAPPROC_NULL	This procedure does not do any 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 port mapper program on the same machine. The program passes its program number *prog*, version number *vers*, transport protocol number *prot*, and the port *port* on which it awaits a service request. The procedure returns a boolean response whose value is TRUE if the procedure successfully established the mapping, and FALSE otherwise. The procedure refuses to establish a mapping if one already exists for the tuple "(*prog*, *vers*, *prot*)."
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.
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 zero means the program has not been registered. The *port* field of the argument is ignored.

Table A-1 (continued) Port Mapper Procedures

Procedure	Description
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.
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. This procedure sends a response only if the procedure was successfully executed and is silent (no response) otherwise. The port mapper communicates with the remote program using UDP/IP only. The procedure returns the remote program's port number, and the bytes of results are the results of the remote procedure.

Appendix B

XDR Protocol Specification

This chapter describes the XDR protocol, a standard for describing and encoding data. The XDR standard assumes that bytes (or octets) are portable, where a byte is defined as 8 bits of data. It also assumes that hardware that encodes bytes onto various media will preserve the bytes' meanings across hardware boundaries. (For example, the Ethernet standard suggests that bytes be encoded in "little-endian" style, or least significant bit first.)

Once XDR data is shared among machines, it shouldn't matter that the data produced on an IRIS is consumed by a VAX (or vice versa). Similarly, the choice of operating systems should have no influence on how the data is represented externally. For programming languages, data produced by a C program should be readable by a FORTRAN or Pascal program.

Topics in this chapter include:

- basic block size
- XDR data types
- discussion of common questions about XDR

Note: For information about the structure and syntax of XDR language, see Chapter 6, "XDR and RPC Language Structure." For details about XDR programming, see Chapter 7, "XDR Programming Notes."

Basic Block Size

Representation of all items requires a multiple of four bytes (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. If the number of bytes needed to contain the data are not a multiple of 4, those *n* bytes are followed by enough (0 to 3) residual zero bytes, *r*, to make the total byte count a multiple of 4.

Include the familiar graphic box notation for illustration and comparison. In most illustrations, each box (delimited by a plus sign at each of the four corners, and vertical bars and dashes) depicts a byte. Ellipses (...) between boxes indicate zero or more additional bytes, where required.

Block

```
+--------+--------+...+--------+--------+...+--------+
| byte 0 | byte 1 |...|byte n-1|    0   |...|    0   |
+--------+--------+...+--------+--------+...+--------+
|<-----------n bytes---------->|<------r bytes------>|
|<-----------n+r (where (n+r) mod 4 = 0)>-----------|
```

XDR Data Types

This section describes the data types defined in the XDR standard, showing how each data type is declared in XDR language, and including a graphic representation of how each type is encoded.

Integers

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. The data description of integers is *integer*.

Integer

```
(MSB)                   (LSB)
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
+-------+-------+-------+-------+
<------------32 bits------------>
```

Unsigned Integers

An XDR unsigned integer is a 32-bit datum that encodes a non-negative integer in the range [0,4294967295]. It is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively. The data description of unsigned integers is *unsigned*.

Unsigned Integer

```
(MSB)                   (LSB)
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
+-------+-------+-------+-------+
<------------32 bits------------>
```

Enumerations

Enumerations have the same representation as integers and are handy for describing subsets of integers. The data description of enumerated data is:

```
enum { name-identifier = constant, ... } identifier;
```

The three colors (red, yellow, and blue) could be described by an enumerated type, as follows:

```
enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
```

It is an error to encode enumerations that have not been given assignments in the *enum* declaration.

Booleans

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard.

Booleans are declared like this:

```
bool identifier;
```

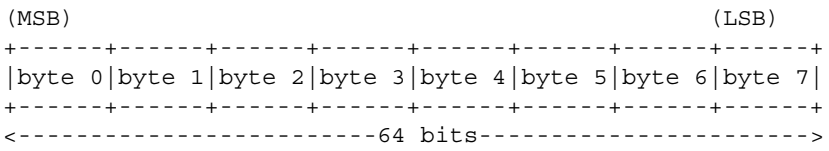
This is equivalent to:

```
enum { FALSE = 0, TRUE = 1 } identifier;
```

Hyper Integers and Hyper Unsigned

The XDR standard also defines 64-bit (8-byte) numbers called hyper integers and unsigned hyper integers. Their representations are the obvious extensions of the integer and unsigned integer defined in the preceding sections in this chapter. They are represented in two's complement notation. The most and least significant bytes are 0 and 7, respectively.

Hyper Integer or Unsigned Hyper Integer



Floating Points

The XDR 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. (See the ANSI/IEEE 754-1985 floating-point standard.)

The following fields describe the single-precision floating-point number:

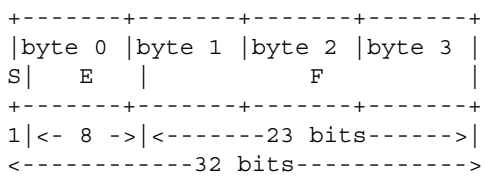
S	The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.
E	The exponent of the number, base 2. Eight bits are devoted to this field. The exponent is biased by 127.
F	The fractional part of the number's mantissa, base 2. Twenty-three bits are devoted to this field.

Therefore, the floating-point number is described by:

$$(-1)^{S} * 2^{\{E - Bias\}} * 1.F$$

It is declared like this:

Single-Precision Floating-Point



Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a single-precision, floating-point number are 0 and 31. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 9, respectively. Note that these numbers refer to the mathematical positions of the bits, and not to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted regarding the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). According to IEEE specifications, the "NaN" (not a number) is system dependent and should not be used externally.

Double-Precision Floating Points

The XDR standard defines the encoding for the double-precision floating-point data type *double* (64 bits or 8 bytes). The encoding used is the ANSI/IEEE 754/1985 standard for normalized double-precision, floating-point numbers.

The following fields describe the double-precision floating-point number:

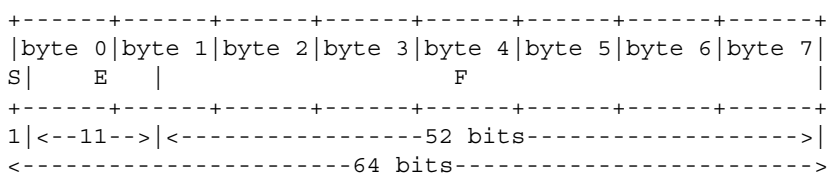
S — The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E — The exponent of the number, base 2. Eleven bits are devoted to this field. The exponent is biased by 1023.

F — The fractional part of the number's mantissa, base 2. Fifty-two bits are devoted to this field.

It is declared as follows:

Double-Precision Floating-Point



Just as the most and least significant bytes of a number 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. Note that these numbers refer to the mathematical positions of the bits, and not to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). According to IEEE specifications, the "NaN" (not a number) is system dependent and should not be used externally.

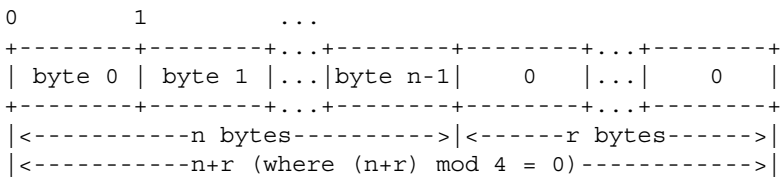
Fixed-Length Opaque Data

At times, fixed-sized, uninterpreted data needs to be passed among machines. This data is called *opaque* and is declared like this:

```
opaque identifier[n];
```

Constant *n* is the (static) number of bytes necessary to contain the opaque data. If *n* is not a multiple of 4, 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 4.

Fixed-Length Opaque



Variable-Length Opaque Data

The XDR 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* and followed by the *n* bytes of the sequence.

Byte *m* of the sequence always precedes byte *m*+1 of the sequence, and byte 0 of the sequence always follows the sequence's length (count). Enough residual zero bytes (0 to 3), *r*, make the total byte count a multiple of 4.

Variable-length opaque data is declared like this:

```
opaque identifier<m>;
```

or

```
opaque identifier<>;
```

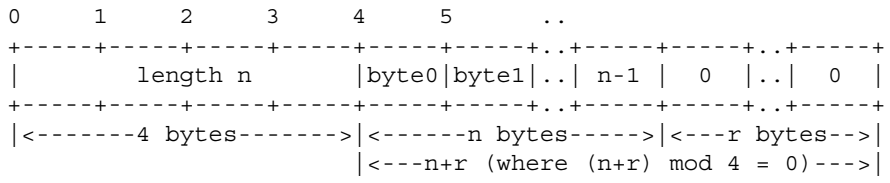
The constant *m* denotes an upper bound of the number of bytes the sequence may contain. If *m* is not specified, as in the second declaration, it is assumed to be the maximum length, $(2^{32}) - 1$.

The constant *m* would normally be found in a protocol specification. For example, a filing protocol may state that the maximum data transfer size is 8192 bytes, as follows:

```
opaque filedata<8192>;
```

It is an error to encode a length greater than the maximum described in the specification.

Variable-Length Opaque



Strings

The XDR standard defines a string of *n* (numbered 0 through *n*–1) ASCII bytes to be the number *n* encoded as an *unsigned integer*, and followed by the *n* bytes of the string. Byte *m* of the string always precedes byte *m*+1 of the string, and byte 0 of the string always follows the string's length. If *n* is not a multiple of 4, the *n* bytes are followed by enough (0 to 3) residual zero bytes, *r*, to make the total byte count a multiple of 4.

Counted byte strings are declared as follows:

```
string object<m>;
```

or

```
string object<>;
```

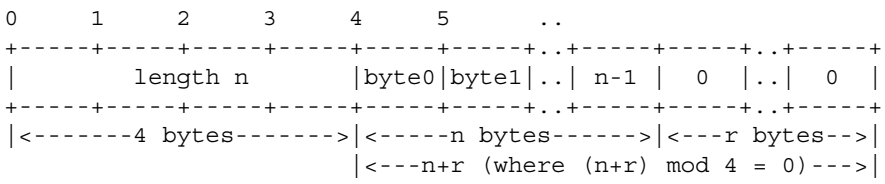
The constant *m* denotes an upper bound of the number of bytes that a string may contain. If *m* is not specified, as in the second declaration, it is assumed to be the maximum length, $(2^{32}) - 1$.

The constant *m* would normally be found in a protocol specification. For example, a filing protocol may state that a filename can be no longer than 255 bytes, as follows:

```
string filename<255>;
```

It is an error to encode a length greater than the maximum described in the specification.

String



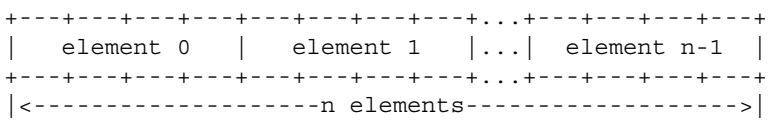
Fixed-Length Arrays

Declarations for fixed-length arrays of homogeneous elements are in this form:

```
type-name identifier[n];
```

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 4 bytes. Although 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 type *string*, yet each element will vary in length.

Fixed-Length Array



Variable-Length Arrays

Counted arrays provide the ability to encode variable-length arrays of homogeneous elements. The array is encoded as the element count *n* (an unsigned integer) followed by the encoding of each of the array's elements, starting with element 0 and progressing through element *n*–1.

Declaration for variable-length arrays follow this form:

```
type-name identifier<m>;
```

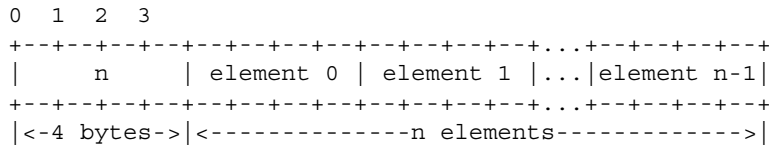
or

```
type-name identifier<>;
```

The constant *m* specifies the maximum acceptable element count of an array; if *m* is not specified, as in the second declaration, it is assumed to be (2^{32}) –1.

It is an error to encode a value of *n* that is greater than the maximum described in the specification.

Counted Array



Structures

The data description for structures in the XDR standard is very similar to data description in standard C:

```
struct {
    component-declaration-A;
    component-declaration-B;
    ...
} identifier;
```

The components of the structure are encoded in the order of their declaration in the structure. Each component's size is a multiple of 4 bytes, although the components may be different sizes.

Discriminated Unions

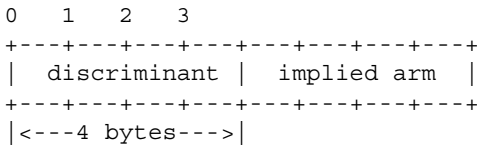
A discriminated union is a type composed of a discriminant followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of discriminant is either *int*, *unsigned int*, or an enumerated type, such as *bool*. The component types are called "arms" of the union, and are preceded by the value of the discriminant, which implies their encoding.

Discriminated unions are declared like this:

```
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 not specified, a valid encoding of the union cannot take on unspecified discriminant values. The size of the implied arm is always a multiple of 4 bytes. The discriminated union is encoded as its discriminant followed by the encoding of the implied arm.

Discriminated Union



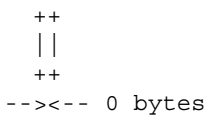
Voids

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, and in unions, where some arms may contain data and others do not.

The declaration is:

```
void;
```

Void



Constants

The data declaration for a constant follows this form:

```
const name-identifier = n;
```

const defines a symbolic name for a constant; it does not declare any data. The symbolic constant may be used anywhere a regular constant can be used. For example, the following defines the symbolic constant DOZEN, equal to 12:

```
const DOZEN = 12;
```

Typedefs

A **typedef** does not declare data either, but it 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**. For example, the following defines a new type called *eggbox* using an existing type called *egg*:

```
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 was considered a variable. For example, the following two declarations are equivalent to 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 this form can be converted to the alternative form by removing the **typedef** part and placing the identifier after the *struct*, *union*, or *enum* keyword, instead of at the end:

```
typedef <<struct, union, or enum definition>> identifier;
```

For example, there are two ways to define the type *bool*:

```
typedef enum {    /* using typedef */
    FALSE = 0,
    TRUE = 1
} bool;

enum bool {       /* preferred alternative */
    FALSE = 0,
    TRUE = 1
};
```

The reason this syntax is preferred is that you do not have to wait until the end of a declaration to figure out the name of the new type.

Optional Data

Optional data is a kind of *union* that occurs so frequently that it is given a special declaration syntax of its own:

```
type-name *identifier;
```

This syntax is equivalent to the following *union* declaration:

```
union switch (bool opted) {
    case TRUE:
        type-name element;
    case FALSE:
        void;
} identifier;
```

This syntax 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 following example defines a type *stringlist*, which encodes lists of arbitrary length strings:

```
struct *stringlist {
    string item<>;
    stringlist next;
};
```

It could be declared equivalently as a *union*:

```
union stringlist switch (bool opted) {
    case TRUE:
        struct {
            string item<>;
            stringlist next;
        } element;
    case FALSE:
        void;
};
```

Or, it could be declared as a variable-length array:

```
struct stringlist<1> {
    string item<>;
    stringlist next;
};
```

Both declarations obscure the intention of the *stringlist* type, however, so the optional-data declaration is preferred over both of them.

The optional-data type is also closely correlated with how recursive data structures are represented in high-level languages, such as Pascal and C. In these cases, recursive data structures use pointers. In fact, the syntax is the same as that of the C language for pointers.

Areas for Future Enhancement

The XDR standard lacks representations for bit fields and bitmaps, since the standard is based on bytes. Also missing are packed (or binary-coded) decimals.

It is not the intent of the XDR standard to describe every kind of data that people have ever sent (or will ever want to send) from machine to machine. It only describes the most commonly used data types of high-level languages, such as Pascal and C so applications written in these languages will be able to communicate easily over some medium.

One could imagine extensions to XDR that would let it describe almost any existing protocol, such as TCP. The minimum requirement for these extensions is support for different block sizes and byte orders. The XDR discussed here could then be considered the 4-byte, big-endian member of a larger XDR family.

Common Questions about XDR

This section attempts to answer questions you may have about XDR.

- Why have a language for describing data?

  There are many advantages to using a data description language, such as XDR, over using diagrams. Languages are more formal than diagrams and lead to less ambiguous descriptions of data. Languages are also easier to understand and allow one to think of other issues instead of the low-level details of bit encoding. Also, there is a close analogy between the types of XDR and a high-level language such as C or Pascal, which makes the implementation of XDR encoding and decoding modules an easier task. Finally, the language specification itself is an ASCII string that can be passed from machine to machine to perform on-the-fly data interpretation.

- Why is there only one byte order for an XDR unit?

  Supporting two byte orderings requires a higher level protocol for determining in which byte order the data is encoded. Since XDR is not a protocol, it cannot support two byte orderings. The advantage, however, is that data in XDR format can be written to a magnetic tape, for example, and any machine will be able to interpret it, since no higher level protocol is necessary for determining the byte order.

- Why does XDR use big-endian byte order?

  Yes, it is unfair that XDR uses big-endian byte order, but having only one byte order means you have to be unfair to somebody. Many architectures, such as the MIPS R2000/3000, Motorola 68000, and IBM 370, support the big-endian byte order.

- Why is the XDR unit four bytes wide?

  There is a trade-off in choosing the XDR unit size. Choosing a small size such as two makes the encoded data small but causes alignment problems for machines that aren't 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 big. Four was chosen as a compromise. The four-byte data unit is big enough to support most architectures efficiently, except for rare machines such as the 8-byte-aligned Cray. Four is also small enough to keep the encoded data restricted to a reasonable size.

- Why must variable-length data be padded with zeros?

  Forcing the padded bytes to be zero ensures that the same data is encoded into the same thing on all machines, enabling the encoded data to be meaningfully compared or checksummed.

- Why is there no explicit data typing?

  Data typing has a relatively high cost for what small advantages it may have. One cost is the expansion of data due to the inserted type fields. Another is the added cost of interpreting these type fields and acting accordingly. Most protocols already know what type they expect, so data typing supplies only redundant information. However, one can still get the benefits of data typing using XDR. One way is to encode two things: first a string, which is the XDR data description of the encoded data, and then the encoded data itself. Another way is to assign a value to all the types in XDR, and then define a universal type that takes this value as its discriminant and for each value describes the corresponding data type.

Appendix C

IRIX Name Service Implementation

This appendix details the functioning of the IRIX name service implementation, called Unified Name Service (UNS), included with IRIX beginning with the 6.5 release. The IRIX name service is made up of a set of C library routines, cache files, a resolver daemon, and protocol libraries. Each of the elements is considered separately in some depth.

Topics in this appendix include:

- "Overview of UNS" on page 296
- "UNS Programming Steps" on page 296
- "UNS Library Routines" on page 297
- "UNS Cache Files" on page 299
- "UNS Name Service Daemon Operation" on page 300
- "How UNS Protocol Libraries Work" on page 307
- "NFS Interface to UNS" on page 313

For specific tasks associated with installing UNS, refer to the chapter on Unified Name Service in *IRIX Admin: Networking and Mail.*

Overview of UNS

The Unified Name Service (UNS) is a name service architecture consisting of a C library API, a cache database, a management daemon, and a number of protocol libraries. It permits new name service protocols to be added very simply by the creation of a shared library with a couple of well-defined entry points, while providing a system-wide cache of all results.

This name service API maintains library-level compatibility with previous releases, and no applications need to be recompiled. The protocol code, which existed in the specific API routines, has been moved out of the C library into separate shared libraries.

UNS Programming Steps

To make full use of the UNS name service, follow these steps:

1. Understand the name service process and determine:
   - The network layout.
   - The configuration required.
   - The details of the resolve order as shown in the */etc/nsswitch.conf* file as described in *IRIX Admin: Networking and Mail.*
   - The relationship between this name service and other name services installed.
2. Configure and test the name service system as described in *IRIX Admin: Networking and Mail*.
3. Add configuration information to the *nsswitch.conf* file as described in *IRIX Admin: Networking and Mail*.
4. Write any special programming using the routines described in "UNS Library Routines."

UNS Library Routines

Starting in IRIX 6.5, the standard name service API routines, which contained protocol code to directly converse with name service daemons, such as **getpwname()**, **gethostent()**, and **getprotobynumber()**, all remain but have been reimplemented as wrappers around two new library routines: **ns_lookup()**, which fetches a single item from a named table, and **ns_list()**, which enumerates an entire name service table.

When a C library routine such as **gethostbyname()** is called in an application, sufficient memory for the returned data structure is allocated, and the routine **ns_lookup()** is called with the key, together with a domain and the name of the table containing this information.

The **ns_lookup()** routine mmaps a global shared cache database corresponding to the table name, and attempts to look up the key in this database. If the lookup fails, then the routine opens a file associated with the key, table, and domain, and parses the data the way it has been done historically with flat configuration files. The file is generated on the fly by a cache miss daemon that acts as a user-level NFS file server.

The daemon determines the resolve order for the request, then calls routines in shared libraries for each of the protocols supported. Once the data is found, it is stored in the global shared cache database and a file is generated in memory using the format of the flat text file.

The **gethostbyname()** routine then parses the result into the appropriate data structure and returns.

getXbyY() Routine

Each **getXbyY()** style routine simply sets up a global memory buffer, calls **ns_lookup()** with a normalized key, the name of a map containing the data, and the domain in which the map lives; it then parses the results into a map-specific data structure. Reentrant routines of the form **getXbyY_r()**, which have been added, behave exactly as the **getXbyY()** routines, except that they use passed-in memory buffers instead of a global space. All of the standard routines are simply wrappers around the reentrant versions to reduce code space in the C library.

getXent() Routine

The **getXent()** style routines are wrappers around the **ns_list()** routine that provide a concatenation of all records in each of the supported back-end databases for a table in what appears to be a flat ASCII file. Reentrant routines of the form **getXent_r(),** which have been added, behave exactly as the **getXent()** routines, except that they use passed-in memory buffers instead of a global space. Again, all of the standard routines are simply wrappers around the reentrant versions in order to reduce space.

ns_lookup() Routine

The **ns_lookup()** routine mmaps the cache file for the given table if it has not already been opened, then attempts to look up the given key in the cache. The cache is a shared, multi-reader, multi-writer, hash database written specifically for this name service implementation, and named MDBM.

If the cache file cannot be opened, or the key does not already exist in the cache, then a separate daemon is contacted to act as the cache miss handler, locating the information within a name service and inserting it in the database. This daemon is contacted through the NFS protocol and the result of the lookup is returned to the client in the format of the flat system configuration file.

ns_list() Routine

The **ns_list()** routine contacts the daemon through the NFS protocol and asks for a concatenation file for a given domain and table, then returns a file pointer to this newly formed concatenation file. The **getXent()** wrapper routines then use stdio to walk through this file, parsing each line into a C data structure, and returning these sequentially. The **getXent_r()** routines are identical, and use the same file pointer, but they use passed-in buffer space to hold the return data instead of dynamically allocated space.

The arguments to **ns_lookup** are a table structure, the domain name for the query, a table name, a key for the query, a buffer to place the results in, and a length for this buffer. The table structure contains a database pointer, a time stamp lock pointer, and a flags field, which determines whether the cache file needs to be closed between calls. It returns an integer result of NS_SUCCESS, NS_NOTFOUND, or NS_FATAL

To see a definition of all return codes and structures, look in the */usr/include/ns_api.h* header file.

The arguments to **ns_list** are the domain name, table name, and an optional protocol name. The routine returns a file pointer.

UNS Cache Files

The UNS cache files are multi-reader, multi-writer, mmapped hash database files. The new database, MDBM, is a simple, extremely fast, single-key, file format.

There is a cache file for each table maintained by the name service daemon in a well-known location. The C library routines always look for the cache files in the */var/ns/cache* directory.

The cache files are writable only by root, and the C library routines always open the cache files as read-only. The cache files can be set to a fixed size, which allows them to be mapped once, then the file descriptor closes, and the mapping remains throughout the life of the process. If the caches are a variable size then they are remapped on each lookup unless the *stayopen* flag is given to the **setXent()** call associated with the table. This is similar behavior to the treatment of files in the historic file-only name service implementations.

Cache file entries are made up of a **time_t**, which can be compared to the current clock for timeouts, a **time_t** which is compared to the time stamp in the map structure, a status character to support negative caching, and the data. Timeouts are handled by all applications. When an application notices that the information is out of date, it requests new information from the daemon. When a cache file is opened with a fixed size, then the cache is split ahead of time to that size. Any time adding an element results in the splitting of the page, a **shake** function is called instead, to free up space for the new data.

The format of keys in the database is:

key\0*domain*\0*protocol*

where *domain* and *protocol* are not given if they are the default, and not specified in the lookup.

UNS Name Service Daemon Operation

The IRIX name service daemon, *nsd*, acts as a cache miss handler for the name service cache files, and lets all protocols speak with remote name servers. The protocol handlers are separated into protocol libraries, which are opened dynamically whenever the protocols are needed, according to the resolution order specified in the daemon configuration file. The *nsd* daemon implements a base set of functionality needed by the protocol libraries.

Name Service Configuration Files and Data Structures

The daemon behavior is completely controlled by the daemon configuration files. A configuration file exists for the client behavior in */etc/nsswitch.conf*, and a similar file exists under */var/ns/domains/DOMAINNAME/nsswitch.conf* for each domain supported by this daemon. If the file */etc/nsswitch.conf* does not exist, a default configuration is used. Server-side domain directories must contain an *nsswitch.conf* file, or the domain is ignored.

The *nsswitch.conf* file is made up of lines in the format:

map: library library library

where each element in the line can have an attribute list associated with it in the format:

(attribute=value, attribute=value, attribute=value)

These attributes may also exist on a line alone, in which case they set the attributes on the domain. And a library may be followed by a control field of the form:

```
[status=action]
```

All of the data from *nsswitch.conf* is maintained in the daemon in four data structure trees:

- A linked list of libraries that have been opened.
- A linked list of cache files, one for each table.
- A btree of file structures. (A **btree()** is a hash indexed binary tree.)
- A set of attribute lists.

The library data is kept in a simple linked list; one structure for each protocol library that has been opened by the daemon. The structure contains the library name as found in the *nsswitch.conf* file, the pathname for the DSO, and an array of function pointers to each of the protocol library entry points.

The map structures are also kept in a simple linked list, and contain information about the cache files the daemon maintains. There is one entry per table that is inserted into the list the first time a request has been made for data from that table. In contains the name of the cache file, a pointer to the database structure, and information about the mapping. Cache files are closed and unmapped when the global **shake** function is called.

The majority of the information in the *nsswitch.conf* files is saved in an in-memory filesystem. Each data item is stored in a file structure and placed into a large global btree. The file structure contains a set of attributes, and possibly a pointer to a map structure containing information on the cache file that is updated when this file is changed, or to a library structure that contains the function pointers for changing this structure. The data field can either be data as read from the back-end databases or a directory list. The hash used for the btree is the file ID, which is simply a 32-bit unsigned value stored in the file structure.

The filesystem tree is rooted with a root file referenced by a global variable. Each *nsswitch.conf* file results in a new file structure (domain), and a reference in the root directory. Each line in the *nsswitch.conf* file results in a new file structure (table), and a reference in the corresponding domain directory. Each library on a line results in a new file structure (callout) and a reference in the table directory. Each directory file structure also contains a reference to the parent. When the reference count on a file goes to zero, it is removed, and the reference count is decremented for each file it points to. Removing the global reference on the root file effectively removes all files in the tree.

Attributes are stored in linked lists attached to file structures. Each attribute list is terminated by an empty structure referencing the attribute list of the parent directory. Searching an attribute list starts with the local attributes then follows the link to the parent list and so on. As a consequence, all attributes are inherited by the children. Attribute structures are separately reference counted, so that removal of a parent directory while a file is in use does not necessarily result in the removal of the attribute list it points to.

Understanding the UNS Runtime Loop

Once the configuration files have been read, the daemon falls into an infinite select loop waiting for input, then dispatching to handler routines. On startup the daemon opens a request socket for reading and sets up a handler for this file descriptor. Whenever the select loop wakes up with data on a file descriptor, the handler for the file descriptor is called. New descriptors can be added or removed at any time by the protocol library code using the utility routines **nsd_callback_new()** and **nsd_callback_remove()**.

Only one callback is set up by default. This callback is the dispatch handler for the NFS protocols. A new packet is parsed as an NFS request, and is answered out of the in-memory filesystem. When a file is referenced that does not already exist in the tree, a new file structure is generated and placed into the tree. A list of callout libraries is inherited from the parent directory, and control is returned to the central loop, which walks the structure through each of the callout library routines until a result is obtained.

The loop through the callout list calls a callout procedure in one of the protocol libraries.

- If the library routine returns the code NSD_OK, the request has been filled, and the input specific return procedure is called to return the results to the calling application.
- If the library returns the NSD_ERROR code, then an error occurred while trying to handle the request and an error should be returned immediately to the client.
- If a code of NSD_NEXT is returned, then the library did not find the result and the next callout procedure is called.
- If the NSD_CONTINUE code is returned, the protocol routine had to send a request to an external daemon or is doing something that will take a long time, so the loop should start working on the next request. The protocol code now owns the request, so there must be a way for the request to start processing again in the future or a leak will occur. The two typical ways for this to continue are that a result comes in on a socket resulting in a handler being called, or a timeout occurs. At any time in the callout list, the default behavior of the return code may be overridden by an entry in the *nsswitch.conf* file. For instance, suppose the following line were in the configuration file:

  ```
  hosts: nis [notfound=return] files
  ```

  Instead of continuing on to the `files` callout when a result is not found in the NIS maps, an error is returned to the client. The `files` callout is called only if NIS is not running, or did not contain the requested record.

Handlers can be set up at any time by protocol code, but typically a socket is set up once during initialization for each library. Timeouts are usually placed on each forwarded request in case the remote agent fails to respond to the request within a reasonable time period. There is a global timeout list for the daemon's central **select()** loop. Each time **select()** is called, the next timeout is first popped off of the stack and used to determine what the **select()** timeout should be. If **select()** wakes up due to a timeout, the handler in the timeout structure is called. Handlers are created using the daemon routine **nsd_callback_new()**, and removed using **nsd_callback_remove()**. Timeouts are created using **nsd_timeout_new()**, and removed using **nsd_timeout_remove()**.

Understanding UNS Utility Functions

The name service daemon contains a number of utility functions that should be used by protocol libraries. These include routines to manipulate return values, set up callbacks handlers for new file descriptors, set up timeouts on the central select loop, and handle errors. All prototypes for these functions are defined in */usr/include/nsdapi.h.*

- **Handling Results**

  The **nsd_set_result()** function provides a convenient way to set the return status and data for a request.The function takes four arguments: a pointer to the file structure; a status code, which should be one of NS_SUCCESS, NS_NOTFOUND, NS_TRYAGAIN, NS_UNAVAIL, NS_BADREQ, and NS_FATAL; a pointer to the result string; the length of the result; and a function pointer to a routine to free this string if needed. There are three routines predefined: DYNAMIC, which is a pointer to the standard **free()** function in the C library; STATIC, which is a null pointer; VOLATILE, which results in **nsd_set_result()** copying the data into a new dynamically allocated buffer. It returns an integer which is either NSD_OK if successful or NSD_ERROR if unsuccessful. If a result already exists, it is freed using the existing free function pointer, and the new result is set.

  ```
  int nsd_set_result(nsd_file_t *, int, char *, int, nsd_free_proc *);
  ```

  The **nsd_append_result()** utility function is similar to the **nsd_set_result()** function, but it appends the given string to the end of an already existing result string if one exists. There is no need to pass a free routine, as this function always copies the data into a new dynamically allocated buffer.

  This function takes three arguments: a pointer to the request structure, a pointer to the result string to be appended, and the length of the string. It returns an integer that is NSD_OK on success, or NSD_ERROR when unsuccessful. On error, the current result string and code is unchanged.

```
int nsd_append_result(nsd_file_t *, int, char *, int);
```

The **nsd_append_element()** function is identical to the **nsd_append_result()** routine except that the result strings are joined by a newline character. This routine assumes that all result strings it is given are null terminated strings.

```
int nsd_append_element(nsd_file_t *, int, char *, int);
```

- **Handling File Descriptor Callbacks**

  The **nsd_callback_new()** function is used to set up a file descriptor callback for the daemon main loop. When **select()** wakes up with data on a file descriptor, the callback handler is looked up in a table, and the corresponding function is called. Protocol libraries can set up callbacks at any time for a file descriptor that they have opened. This routine registers the new handler function and causes select to wake up on new data waiting on the descriptor. If a handler was already registered for the descriptor, it is replaced.

  This function takes three arguments: an integer file descriptor, a pointer to the handler function, and a flag that contains options for what events the callback should be used. It should be made up of NSD_READ, NSD_WRITE, and NSD_EXCEPT. It returns a pointer to the handler function on success, or a null pointer on failure. The only cause for failure is that the file descriptor is out of range.

  ```
  nsd_callback_proc *nsd_callback_new(int, nsd_callback_proc
  *,unsigned);
  ```

  The **nsd_callback_remove()** function clears a handler from the list of file descriptors. This function takes one argument, which is the integer file descriptor, and returns an integer, which is NSD_OK or NSD_ERROR.

  ```
  int nsd_callback_remove(int);
  ```

  The **nsd_callback_get()** function returns the callback handler function pointer, given the integer file descriptor.

  ```
  nsd_callback_proc *nsd_callback_get(int);
  ```

- **Handling Timeouts**

  The **nsd_timeout_new()** function is used to set up timeout handlers for the central select loop. Any time a protocol routine returns NS_CONTINUE, the routine should set up a timeout handler to continue the request processing.

This function takes four arguments: a pointer to the file structure, an unsigned timeout value in milliseconds, a pointer to a timeout handler routine, and a pointer to any local data needed by the protocol code. It returns a pointer to the timeout structure on success, or a null pointer on failure. The local data pointer can be nil if the calling routine does not need data associated with the timeout.

```
nsd_times_t *nsd_timeout_new(nsd_file_t *,long,
nsd_timeout_proc *, void *);
```

The **nsd_timeout_remove()** function is called to remove a timeout from the timeout list. This is typically called when a protocol function receives a reply from a remote daemon, and no longer needs the select loop to timeout to continue processing.

This function takes one argument, a pointer to the file structure, and returns an integer result, which is NSD_OK for success or NSD_ERROR for failure. Failure usually indicates that there was no matching timeout on the list.

```
int nsd_timeout_remove(nsd_file_t *);
```

- **Handling Attributes**

  The **nsd_attr_store()** routine adds an attribute to an attribute list. Attributes should be used instead of global variables when possible. Attribute lists are tied together from most specific to least specific, walking backwards up the daemon data structure tree.

  This function takes three arguments: a pointer to the pointer to the beginning of this attribute list, a pointer to a string for the key, and a pointer to a string for the data. It returns a pointer to the attribute structure if successful or a null pointer on error.

  ```
  nsd_attr_t *nsd_attr_store(nsd_attr_t **, char *, char *);
  ```

  The **nsd_attr_delete()** routine removes the attribute from the given list. Continuations to other lists are not followed, which means that if **nsd_attr_fetch()** were immediately called with this key, it may find a result.

  This function takes two arguments: a pointer to the pointer to the first attribute in the list and a pointer to the string for the key. It returns an integer, which is NSD_OK on success or NSD_ERROR if the attribute was not found.

  ```
  int nsd_attr_delete(nsd_attr_t **, char *);
  ```

  The **nsd_attr_fetch()** routine searches through an attribute list, following continuations to other lists, searching for a matching attribute. Key comparisons are case-insensitive.

This function takes two arguments: a pointer to the beginning of the attribute list, and a pointer to the string for the key. It returns a pointer to the attribute structure if found or a null pointer on failure.

```
nsd_attr_t *nsd_attr_fetch(nsd_attr_t *, char *);
```

The three routines **nsd_attr_fetch_long()**, **nsd_attr_fetch_string()**, and **nsd_attr_fetch_bool()** are simple wrappers around **nsd_attr_fetch()**. They take a pointer to the attribute list, a string for the key, and a default value. The **nsd_attr_fetch_long()** routine also takes a radix. These routines return the value of the attribute interpreted as a long, string, or boolean, depending on the function called, or the default value if the key was not found.

```
long nsd_attr_fetch_long(nsd_attr_t *, char *, int, long);
char *nsd_attr_fetch_string(nsd_attr_t *, char *, char *);
int nsd_attr_fetch_bool(nsd_attr_t *, char *, int);
```

- **Handling Memory**

  The **nsd_shake()** routine should be called to free up resources when allocating new resources fails. This results in a call to all of the protocol-specific **shake()** routines. This frees memory, close and unmap files, and generally tries to reduce the resources used. The name service daemon and many of the protocol libraries are aggressive about caching results, connections to files or remote daemons, and so on.

  This routine takes an integer level from 0 to 9, and returns no results.

  ```
  void nsd_shake(int);
  ```

  The three routines **nsd_malloc()**, **nsd_calloc()**, and **nsd_strdup()** are wrappers around the standard **malloc()**, **calloc(),** and **free()** routines, which call **nsd_shake()** on failure, then retry the allocation. The standard **free()** routine can be called to give up memory.

  ```
  void *nsd_malloc(size_t);
  void *nsd_calloc (size_t,size_t);
  char *nsd_strdup(char *);
  ```

- **Handling Access and Output**

  The three routines **nsd_open()**, **nsd_mdbm_open()**, and **nsd_mmap()** are wrapper functions around **open()**, **mdbm_open(),** and **mmap**(). On failure they call the **nsd_shake()** function then try again. The standard routines **close()**, **mdbm_close(),** and **mmap()** can be used to close these files.

  ```
  int nsd_open(const char *, int, mode_t);
  MDBM *nsd_mdbm_open(const char *, int, mode_t, int);
  void *nsd_mmap(void *, size_t, int, int, int, off_t);
  ```

The **nsd_logprintf()** routine takes the same arguments as **printf()** plus an integer level from 0 to 6, but results in a message to the log or to the console, depending on arguments to the daemon. If *nsd* receives a SIGUSR2 signal, it will cycle through the logging levels, thus allowing a developer, debugger to select the logging level without having to restart *nsd*. It should be used to print error messages. The log levels are defined in *ns.daemon.h.*

```
void nsd_logprintf(int char *, ...);
```

How UNS Protocol Libraries Work

All of the name service protocol code that existed inside the API routines in the C library is in separate protocol libraries, which are used only by the name service daemon. Each library has a small set of entry points, which are used by the daemon command routines. These routines are **init()**, **lookup()**, **list()**, **verify()**, **dump()**, and **shake()**.

Library Init Routine

The **init()** routine in a library is called when the library is first opened, and again whenever the daemon receives a SIGHUP signal. Typically, the **init()** procedure reads any protocol-specific configuration files, such as */etc/resolv.conf* for DNS, and sets up any global data needed by the library, such as a list of domains or server addresses.

The **init()** procedure takes no arguments, and returns an integer, which is NSD_OK or NSD_ERROR.

```
int init(void);
```

The **init()** procedure may set up handlers for new requests of an alternative protocol-specific form, such as the *nisserv* library, which accepts Sun RPC requests for NIS version 2.

This routine may also set up handlers for results dealing with forwarded requests. Most of the name service protocols reformat the request into a different form and send it to another daemon, then set up a timeout and callback. When the results come back from the remote system, they go through this handler routine, which parses the results into an internal form again, and returns a successful result code to the main loop.

The **init()** routine may also create some false requests to take care of initialization that can happen asynchronously. The *nis* and *nisserv* callouts use this feature to register with portmap. They send off a packet to the portmap daemon, set up a handler and timeout, then give control back to the main loop so as not to hang if there are problems registering.

Library Lookup Routine

The **lookup()** routine is the most called of all routines in the name server and is the one that most people think of as the protocol. This routine converts the internal request format into a protocol-specific format and sends it to a remote daemon. When results come back, they are converted into an internal format again, and a status code is returned. It is up to the initial request handler to set up the reply.

The **lookup()** routine takes one file pointer argument and returns an integer, which is NSD_OK, NSD_ERROR, NSD_NEXT, or NSD_CONTINUE.

```
int lookup(nsd_file_t *);
```

In the simple case the **lookup()** routine simply fetches data out of a file, converts it into the proper format, and returns it immediately.

Library List Routine

The **list()** routine concatenates all records into an internal flat file. This is used by the **getXent()** routines or for administration.

The **list()** function takes one file pointer argument and returns an integer, which is NSD_OK, NSD_ERROR, NSD_NEXT, or NSD_CONTINUE.

```
int list(nsd_file_t *);
```

Library Dump Routine

The **dump()** routine is used for protocol debugging and outputs the current state of the library to the given file descriptor. When the daemon is sent a SIGUSR1 signal, it opens a file */usr/tmp/nsd.dump* and writes its state, then calls each of the library routines to do the same.

```
void dump(int);
```

Library Verify Routine

The **verify()** routine checks that the results previously returned in **lookup()** or **list()** are still valid. The **verify()** function takes one file pointer and returns an integer which is NSD_OK or NSD_error.

```
int verify(nsd_file_t*)
```

Library Shake Routine

The **shake()** function is called when the daemon runs short of resources. This function frees up any resources used by the protocol library that are not needed. For instance, the *files* callout **shake()** function closes and unmaps all of the open files.

Any protocol routine that runs out of resources, such as attempting a **malloc()** that fails or failing to open a new file, should call the daemon utility function **nsd_shake(),** which frees any unneeded global data then calls each of the protocol-specific **shake()** functions. **shake()** is called with an integer priority from 0 through 9, determined by need. After calling **nsd_shake()**, the protocol routine tries again to do whatever failed before returning an error. The utility routines **nsd_malloc()**, **nsd_calloc()**, and **nsd_strdup()** do exactly this.

The **shake()** function takes one integer argument from 0 to 9 and returns an integer, which is NSD_OK or NSD_ERROR.

```
int shake(int);
```

Files Callout Library

The files library **mmap()** flat files into the daemon memory and searches through them for matching lines in the same fashion as the C library API fallback routines. The filename is determined by the map name, and the directory is determined by the domain name. By default this is */var/ns/domain/file* or */etc/file* for the *.local* domain. Either of these can be overridden using attributes “file” or “directory” attached to the files callout in the appropriate *nsswitch.conf* file.

The *passwd.** map is special. For any line of the form [+-]@?[\S]+, it verifies the element by making a recursive call into the daemon and returning the NSD_NEXT code to the main loop. If the directive [notfound=return] is specified after the files callout in

nsswitch.conf, the behavior is identical to the historic behavior of forcing calls into NIS, except that any library may follow files, not only NIS.

The list routine simply copies the entire mapped file into the result instead of attempting to do any parsing.

NIS Callout Library(Optional)

The *nis* library implements the client side of the Sun YP RPC protocol and the YPBIND protocol. Internal requests are reformatted into RPC requests and sent to a remote host, a callback and timeout are set up, and control is returned to the main daemon loop. When a response comes back to the socket owned by the *nis* library, a handler is called that parses the YP RPC result packet into the internal format and returns it to the client. Responses are mapped back to the original request structure using the XID field in the RPC header of the response packet.

The library also maintains a socket for incoming YPBIND RPC requests which are answered using data maintained by the *nis* library.

If any request comes in and the daemon has not already bound to a server, or if a request to a server times out, then a bind broadcast/multicast is sent out, and the request is held until the daemon is able to bind to a new server. If the daemon is unable to bind within a couple of seconds, an NS_TRYAGAIN status is returned to the client so that it will resend the request instead of falling back to local files. If the file*/var/yp/domain/binding/ypservers* exists, then the hosts listed in this file is sent unicast bind requests instead of a broadcast sent out.

The *nis* library fakes for maps that exist in the *nsswitch.conf* file but not in the NIS version 2 standard. These include *services.bynumber*, *group.bymember*, and *rpc.byname*. The library first attempts to look up data using these names, then falls back to stepping through the reverse map file if that fails.

The **list()** routine connects to the *ypserv* daemon using TCP, appending the entire results to the file data.

Nisserv Callout Library (Optional)

The *nisserv* callout library implements the server side of the Sun YP RPC protocol. It opens a socket on *init* on which it accepts new requests. It looks up these requests using the standard callout list, and replies to the requestor using the YP protocol.

When the YP_ALL request is received, it enumerates only the maps for which the boolean *enumerate* attribute is set. If this attribute is not set for any callout, it enumerates the MDBM database instead, provided the *mdbm* library is listed as a callout.

Note: *yp_all* simply enumerates the MDBM database, and is not supported for anything else. The internal data format needs to change before it can support the other databases.

DNS Callout Library

The *dns* library implements the client side of the Domain Name Service Protocol. New requests are converted from the internal format to a DNS packet format and sent to a remote server, then a timeout and callback is set up and control is given back to the main loop. When a response comes back from the server, it comes to a socket owned by the *dns* library and passes through a DNS response handler. The response is mapped back to the original request using the DNS header *xid* field, and the packet is parsed back into the internal format to be returned to the client.

The order for contacting servers is controlled by the *resolv.conf* file, or by the *servers* attribute attached to the *dns* callout in *nsswitch.conf*. The domain is the same as the request domain except in the case of the *.local* domain. When the *.local* domain is used, the domain in the *dns* request is determined by the domain or search fields in *resolv.conf* or by the *domain* attribute in *nsswitch.conf*.

The map *hosts.byname* is turned into a class IN, type A request to DNS. The map *hosts.byaddr* is turned into a class IN, type PTR request to DNS. Any other map is turned into a class IN, type TXT request to DNS using the DNS domain *table.domain*; where any "." characters in the table are replaced with "_". For instance a call for the key *uucp*; in the *passwd.byname* map for the domain fruit.com results in a lookup of *uucp.passwd_byname.fruit.com* in the IN class, and returns a TXT type.

MDBM Callout Library

The *mdbm* library uses the MDBM database format to store data in local files. A set of parser scripts are provided to parse flat files into the databases. This supports a faster lookup method than the files library. The files default to */var/ns/domain/table.m* for each table, or */etc/table.m* in the *.local* domain. This can be overridden by setting the file attribute on the table in the appropriate *nsswitch.conf* file.

The **list()** command results in a **mdbm_next()** loop, appending each successive value to the end of the result.

Berkeley DB Callout Library

The *berkeleydb* library is a key-value database. This library supports arbitrary sized values and is faster than NDBM. The files default to */var/ns/domain/table.db* or */etc/table.db* for the local domain. This can be overridden by setting the file attribute on the table in the appropriate *nsswitch.conf* file.

The **list()** command results in a **(DB*)db->seq()** loop appending each successive value to the end of the result.

NDBM Callout Library

The *ndbm* library is the standard IRIX database mechanism, which is a key/value like hash files, and similar in functionality to the original DBM as well as Berkeley DB.

NFS Interface to UNS

The primary interface to the *nsd* daemon from the API routines is through the Network File System. The name service daemon acts as a user level NFS file server for an in-memory stacked filesystem. The daemon is mounted onto the local system at startup, and all the API routines simply open files in the filesystem tree managed by the name service daemon.

The name service daemon has a special mount command called *nsmount*. This command determines the port that the name service is running on, and the initial file handle for the requested domain directory then passes this to the kernel. Future versions of the NFS protocol will hopefully allow the name service daemon to be treated just like any other NFS server so that the regular mount command, *automount*, and *autofs* can be used.

It is possible to mount the name service daemon from another system, and this technique is expected to support large networks of systems and trees of domains. The administrator can explicitly restrict a portion of the namespace to the local host by setting the *local* attribute on the top element of the subtree. By default the *.local* domain sets the *local* attribute to true so other systems cannot read local passwords, and so on. The default location of the mount point is */ns/domain*, where *domain* is the requested domain in the **ns_lookup()** or **ns_list()** routine. There is a special domain labeled *.local* that always exists that provides a system local domain to override any parent domain information. All of the API **getXbyY()** routines currently use the *.local* domain.

The daemon filesystem tree is organized as: */ns/domain/table/key*, and there is a special domain, *.local*, to represent the local view of the namespace, and "dot" directories under each table to represent the callout libraries. To look up the login name *uucp* using the local namespace view, open the file */ns/.local/passwd.byname/uucp*. If you want only the NIS entry for *uucp*, open */ns/.local/passwd.byname/.nis/uucp*. The special key *.all* in a map returns a concatenation of all the records in a table, so opening the file */ns/.local/passwd.byname/.all* gives you a giant *passwd* file containing all users in the local domain. Executing *cat /ns/.local/passwd.byname/.nis/.all* is equivalent to running *ypcat passwd.byname*. Or *cat /ns/.local/passwd.byname/.files/.all* is identical to *cat /etc/passwd* on most systems.

Removing a file in the filesystem maintained by the name service daemon results in the cached file structure being removed in the daemon. The directory entries cannot be removed. Instead this is done by editing the *nsswitch.conf* files and sending the daemon a SIGHUP signal. Attempting to remove a directory results in the timeout routine being run on that subdirectory so that all dynamic elements under that directory are removed.

In the IRIX operating system, extended attributes are supported on each name service file. The attributes on the file depend on the library that looked them up, but always include *domain*, *table*, *key*, *timeout*, *library*, *version*, and *server*. The **timeout** is the time in seconds since epoch that the cache entry disappears from the daemon. The **library** is the name of the library as given in *nsswitch.conf*, that provided the data in the file, and **server** is the address of the system that provided the data. The server may not be the actual authorized owner of the information, but is instead simply the system from which you got the information. These can be read using the *attr* command. For example, to get the source of a key, run:

```
attr -g library /ns/.local/passwd.byname/uucp
```

Only the **get** and **list** functions work with the name service daemon. All information in the name server tree is read-only.

Index

A

accept, 14
address
  binding, 46
address, manipulation, 27
array, fixed length, 287
array, fixed-size, 171
array, variable length, 287
array, XDR, 168
asynchronous processing, 229
authentication, RPC, 71, 265

B

batch, RPC, 266
big endian, 27
binding
  address, 46
binding, RPC, 71
block size, XDR, 280
boolean, XDR, 282
booleans, XDR, 155
broadcast, RPC, 266
byte, arrays, XDR, 168
byte ordering, 27

C

cache file mapping, 299
client/server model, 29
connection establishment
  transport interface, 204
connectionless
  socket, 17
connectionless-mode service (transport interface), 219
connectionless-mode transport service, 196
connection-mode transport service, 192
connection release, 216
constants, 290
constants, XDR, 152
constructed data type filters, 166

D

daemon, Internet, 51
data, optional, 291
data, transfer, 15
datagram
  receive, 62
  send, 60
data transfer, 212
data-type filters, 166
descriptors,adding to protocol library, 300
discriminated unions, 172

discriminated unions, XDR, 288
double precision, XDR, 284

E

entry points, 307
enumeration filters, XDR, 166
enumerations, XDR, 151, 281
event handling, 207
examples, XDR, 169
exceptions, rpcgen, 155

F

fcntls
  F_SETOWN, 41
  FASYNC, 41
  FNDELAY, 40
FD_CLR, 19
FD_ISSET, 19
FD_SET, 19
FD_SETSIZE, 19
FD_ZERO, 19
filters, number, 165
fixed-length array, 287
fixed-size array, 171
floating point, XDR, 165, 282

G

gethostbyaddr(3), 23
gethostbyname(3), 23
getnetbyname(3), 24
getnetbynumber(3), 24
getprotobyname(3), 24
getprotobynumber(3), 24
getservbyname(3), 25
getservbynumber(3), 25
getsockopt(2), 49
groups
  signal process, 42

H

host, name, 23
htonl(3), 27
htons(3), 27
hyper integer, unsigned, 282

I

inetd, 51
  tcpmux, 51
integer, unsigned, 281
integer, XDR, 280
interrupt-driven socket I/O, 41
ioctls
  SIOCADDMULTI, 64
  SIOCATMARK, 39
  SIOCDELMULTI, 64
  SIOCGIFBRDADDR, 58
  SIOCGIFCONF, 55
  SIOCGIFFLAGS, 57
  TIOCNOTTY, 31
ioctls, SIOCGIFDSTADDR, 58
I/O multiplexing, 19
I/O streams, XDR, 177
IP
  broadcasting, 54
  multicast datagram, receive, 62
  multicast datagram, send, 60

multicasting, 59

L

language, RPC, 143
library, UNS, 297
library, XDR, 162
linked lists, XDR, 181
listen, 14
little endian, 27
local management (transport interface), 197

M

mapping, name to address, 186
memory, streams, XDR, 177
message, authentication, RPC, 71
message, protocol, RPC, 267
message authentication, RPC, 265
model, RPC, 68
modes of service (transport interface), 191
multicasting, IP, 59
multiplexing
  input, 19
  output, 19

N

names, host, 23
names, network, 24
names, protocol, 24
names, service, 25
name service
  cache files, 300
  functions, 303
  implementation, 296
  library entry points, 307
Name-to-Address Mapping, 186
network
  library routines, 22
  names, 24
Network Selection, 186
Network Services Library, 191
no data, 166
non-blocking sockets, 40
non-filter primitives, 175
nsd_set_result, 303
ntohl(3), 27
ntohs(3), 27
null authentication, 271
number filters, XDR, 165

O

object, XDR, 179
opaque data, variable length, 285
opaque data, XDR, 171, 285
Open Systems Interconnection, 186
operation directions, XDR, 176
optional data, 291
options, socket, 49
OSI (Open Systems Interconnection), 186

P

pointers, XDR, 174
port mapper, RPC, 274
primitives, non-filter, 175
primitives, XDR library, 165
procedures, RPC, 75

process groups, 42
protocol
  select, 46
protocol independence, 243
protocol library, adding descriptors, 300
protocol library utility functions, 303
protocol names, 24
protocol requirements, RPC, 264
pseudo terminals
  terminal
    pseudo, 44
pty creation, 44

R

read(2), 15
receive IP multicast datagram, 62
record marking, RPC, 274
record streams, 177
recv(2), 15
recvfrom(2), 17
reference model (transport interface), 187
remote, programs, RPC, 75
rendezvous independence, RPC, 71
routines, library, 22
RPC
  authentication, 265
  authentication, UNIX, 271
  authentication protocols, 271
  batch, 266
  binding, 71
  broadcast, 266
  generating XDR routines, 89
  language, 143
  message authentication, 71, 265
  message protocol, 267
  model, 68
  null authentication, 271
  parameter authentication, 271
  port mapper, 274
  procedures, 75
  programs, 152
  protocol requirements, 264
  record marking, 274
  remote programs, 75
  rendezvous independence, 71
  transports, 70
  UNIX authentication, 271
rpcbind(1M), 78
rpcgen
  C preprocessor, 95
  debugging, 94
  declarations, 153
  local to remote procedure, 81
  server broadcasting, 97
  server procedures, 97
  special cases, 155
  timeout changes, 96
rwho server, 35

S

select(2), 19
select protocol, 46
send(2), 15
send IP multicast datagram, 60
sendto(2), 17
server/client model, 29
service names, 25
setsockopt(2), 49
signal handling, 42
signals
  SIGCHLD, 43
  SIGIO, 41
  SIGURG, 41

size, block, 280
sockets
  connectionless, 17
  interrupt-driven I/O, 41
  I/O, 15
  non-blocking, 40
  options, 49
special cases, rpcgen, 155
state transition rules, 197
state transition tables (transport interface), 237
stream access, XDR, 176
STREAMS, 186, 189, 229
streams, record, 177
strings, 286
strings, XDR, 155, 167
structures, XDR, 149, 288

T

TCP/IP
  record streams, 177
transfer, data, 15
transmit datagram, 60
transport interface
  connection establishment, 204
  connectionless-mode service, 219
  local management, 197
  reference model, 187
  state transition tables, 237
transport service
  connectionless-mode, 196
  connection mode, 192
transport service data units (TDSU), 212
TSDU (Transport Service Data Units), 212
typedefs, 290
typedefs, XDR, 152

U

unions, discriminated, 172
unions, XDR, 150
UNIX authentication, RPC, 271
UNS, 296
unsigned integer, 281

V

variable-length array, 287
variable-length opaque data, 285
void, XDR, 155

W

write(2), 15

X

XDR
  basic block size, 280
  booleans, 155, 282
  byte arrays, 168
  constants, 152
  discriminated union, 172
  discriminated unions, 288
  double precision, 284
  enumeration filters, 166
  enumerations, 151, 281
  examples, 169
  fixed-size array, 171
  floating point, 282
  floating point filters, 165
  future directions, 292
  hyper integer, 282
  hyper unsigned, 282

integer, 280
I/O streams, 177
language, 143
language syntax, 147
library, 162
library primitives, 165
linked lists, 181
memory streams, 177
non-filter primitives, 175
number filters, 165
object, 179
opaque data, 171, 285
operation directions, 176
pointers, 174
record streams, 177
routine generation, 89
specification, 279
stream access, 176
stream implementation, 179
strings, 155, 167
structures, 149, 288
typedefs, 152
unions, 150
void, 155

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-0810-080.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389