CONVEX Network File System Reference Set

Document No. 710-001730-202

Second Edition, Rev. 1
April 1988

CONVEX Computer Corporation
Richardson, Texas

CONVEX Network File System
User's Guide
Order No. DSW-111
Second Edition, Rev. 1

Printed in the United States of America

Revision Information for *CONVEX Network File System Reference Set*

Edition	Document No.	Description
Second Rev. 1	710-001730-202	Released with CONVEX UNIX V6.2, April 1988. Includes the following changes: *RPC Protocol Specification* Chapter 2, corrected formatting Appendix A, corrected protocols *XDR Protocol Specification* Chapter 2 corrected *xdr* library primitives corrected "Fixed-Size Arrays" section Chapter 4, corrected formatting Chapter 5, corrected Table 5-1 Appendix A, added *xdr* routines *RPC Programming Guide* Chapter 4 corrected *svc_run()* code corrected formatting errors Appendix A added *rpc* routines
2.0	710-001730-201	Released with CONVEX UNIX V6.1, October 1987. Includes the following changes: Chapter 2 added description of advisory record locking added description of how to use remote execution utilities
1.0	710-000630-000	Initial release with CONVEX UNIX V6.0, April 1987.

NFS Overview

List of Figures

NFS Protocol Spec.

YP Protocol Spec.

RPC Protocol Spec.

XDR Protocol Spec.

RPC Programming Guide

Appendices

List of Tables

List of Figures

Preface

Intended Audience, Prerequisites

This set of manuals documents the Network File System (*nfs*), the Yellow Pages (*yp*), the Remote Procedure Call (*rpc*) package, and the External Data Representation (*xdr*) software. Included are an introductory product overview (the *Network File System Overview*), highly technical product specifications (the various *Protocol Specifications*), and a guide to *rpc* programming (the *RPC Programming Guide*).

Although some introductory material is included here, these manuals are not intended for casual or novice users. Instead, they are intended for application programmers and system managers who need to know specifics of a product. Prerequisites for this set of manuals include an advanced understanding of the C programming language and a detailed knowledge of the CONVEX UNIX operating system. Novice users are referred to the *CONVEX Network File System User's Guide*, which is included with the *nfs* documentation package.

Notational and Typographical Conventions Used

The following conventions are used in this document:

- Mnemonics enclosed in "less than" and "greater than" signs designate ASCII nonprintable characters. For example, **<CR>** stands for "carriage return."

- Within command sequences set off from regular text, **boldface** type indicates literals. Words appearing in **boldface** must be typed just as they appear. *Italics* within command sequences indicate generic commands or filenames. Substitute actual commands or filenames for the *italicized* words. For example, the command sequence

  ld [*switches*] [*object files*] [*libraries*]

  instructs you to type the command *ld*, followed by your choice of switches, object files and libraries.

  Italics within text indicate commands, filenames, or programs.

- Brackets [ ] designate optional entries.

- A horizontal ellipsis ... shows repetition of the preceding item(s).

- A vertical ellipsis shows continuation of a sequence where not all of the statements in an example are shown.

- Commands, utilities, and files that are documented in the *CONVEX UNIX Programmer's Manual* are italicized; occurrences that include a number enclosed in parentheses refer to the appropriate section of the manual (for example, *vmstat*(1) means that the command *vmstat* is located in Section 1 of the *Programmer's Manual*).

- The | symbol is used to denote command sequences in which you must pick no more than one alternative from a list of command options. In the following command sequence, for example:

`(fp)>` **s[et] s[pu-selftest]** = [*d[isable* |*e[nable]*

you must choose either *d[isable]* or *e[nable]*, but you cannot choose both.

- The pound sign (#) signifies the superuser prompt. The percent symbol (%) signifies the standard C shell user prompt.

Associated Documents

The following documents provide information that you will find useful as you learn about the Network File System and its related utilities.

- *CONVEX System Manager's Guide.* This manual provides all of the information needed to manage and maintain a previously installed and configured CONVEX system. Of particular relevance to succesful operation of *nfs* are the chapters on system start-up and shutdown and network (LAN) management.

- *CONVEX Network File System User's Guide.* This manual describes how to use *nfs*, *rpc*, *xdr*, and the *yp*.

- *CONVEX Network File System System Manager's Guide.* This volume describes how to install and maintain *nfs* and *yp*.

- *CONVEX Networking Utilities System Manager's Guide* describes installing, configuring, testing, and debugging the software used by CONVEX local area networks (LANs). Also included in this guide are instructions for changing system files and troubleshooting problems in the network.

- *CONVEX Networking Utilities User's Guide* shows users how to use the CONVEX networking utilities.

- *CONVEX Interprocess Communication (IPC) Programming Guide* provides application programmers with the necessary information to develop network applications.

A

Reporting Problems

A.1 Introduction

The *contact* utility is the recommended way to report software and documentation problems to the Technical Assistance Center (TAC). It is an interactive tool that prompts you for the information necessary to report a problem to the TAC.

You must have a UNIX-to-UNIX Communications Protocol (UUCP) connection to the TAC to use *contact*. A UUCP system allows communication between UNIX systems by either dial-up or hard-wired communication lines. See *uucp*(1) or the entry in *info*(1) (online information system) for more information.

You must know the name and version number of the product involved. If you do not know the version number of the program or utility you are having trouble with, use the *vers* command. The syntax for the command is

vers *filename*

where *filename* is the the full pathname of the program. If you don't know the full pathname of the program, type

which *program*

For more information on these commands, see *vers*(1) and *which*(1) in the *CONVEX UNIX Programmer's Manual*, Part I.

A.2 Information Required to Report a Problem

contact requires the following information:

1. Your name, title, phone number, and corporate name.
2. The name and version of the product involved. Use the *vers* command if you don't know the version number of the program or utility.
3. A short (1 line) summary of the problem.
4. A detailed description of the problem. Include source code and a stack backtrace whenever possible. (See *adb*(1) or *csd*(1) for information on obtaining stack backtraces.) The more information provided, the quicker your problem can be isolated and solved.
5. The priority of the problem. You are shown a list of six levels from which to select.
6. Instructions on how to reproduce the problem, including the command syntax used, any flags invoked, or anything else you attempted to make your program run.

7. Any other comments about the problem or files you wish to submit.

You will have a chance to review your report before you submit it. You can edit the report if you find an error in what you have typed. If you change your mind and don't want to submit the report, you can abort the *contact* session; the file is saved in your home directory in a file named *dead.report*.

The following figure is a sample *contact* session. User input is in bold lettering, and the system response is in constant-width lettering.

Figure A–1: Sample *contact* Session

```
%contact (RETURN)
Welcome to contact version 0.11 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
> Margaret Atwood, systems programmer, 814-4444, University r
> of Chicago (RETURN)
> (CTRL-D)

Enter the name of the product involved
> CONVEX UNIX Programmer's Manual, Part I (RETURN)

Enter the version number (in the form X.X or X.X.X.X) of the product
> Revision 4.0 (RETURN)

Enter a short (1 line) summary of the problem
> The finger command manual page lists nonexistent bug (RETURN)

Enter a detailed description of the problem (^D to terminate)
> The finger(1) man page says, under the BUGS section, that "Only the first
line of the .project file is printed."  Happily, this is not true! (RETURN)
> (CTRL-D)

Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem is resolved.
2) Serious     - work can proceed around the problem, with difficulty.
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
> 4 (RETURN)

Enter the instructions by which the problem may be reproduced (^D to terminate)
> a) put more than one line in .project (RETURN)
> b) read the man page for finger(1) (RETURN)
> (CTRL-D)

Enter any comments that are applicable (^D to terminate) (RETURN)
> (CTRL-D)

Do you have any suggestions or comments on the documentation that you
referenced when you were trying to resolve your problem (for example,
additions, corrections organization, accessibility)? (^D to terminate)
> The man page should be updated. (RETURN)
> (CTRL-D)

Are there any files that should be included in this report (yes | no)?
> no (RETURN)

Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
> 3 (RETURN)

Problem report submitted.
%
```

CONVEX Network File System Overview

November 1, 1987

CONVEX Computer Corporation

Table of Contents

List of Figures

1

Basics

Introduction

This chapter gives an overview of the CONVEX implementation of the Network File System (*nfs*), developed by Sun Microsystems. *nfs* allows users to mount directories across the network, and then to treat remote files as if they were local. Advanced users may want to skip the first few sections and go straight to examples of how it works. Beginning users may not be interested in the next chapter, which discusses network file system architecture.

The Network File System (*nfs*) is a facility for sharing files in a heterogeneous environment of machines, operating systems, and networks. Sharing is accomplished by mounting a remote filesystem, then reading or writing files in place. *nfs* is open-ended and can be easily interfaced with other systems.

A distributed network often provides more aggregate computing power than a mainframe computer, with far less variation in response time over the course of the day. Thus, a networked computing environment is generally more cost-effective than a central mainframe computer, particularly when considering the value of people's time. For large programming projects and database applications, however, a mainframe has often been preferred, because all files can be stored on a single machine.

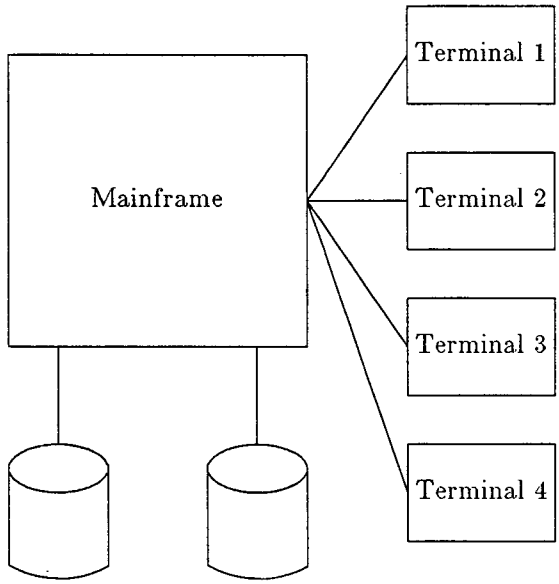
Even in a network environment, sharing programs and data is sometimes difficult. Files either have to be copied to each machine where they were needed, or users have to log in to the remote machine with the required files. Network logins are time-consuming, and having multiple copies of a file gets confusing as incompatible changes are made to separate copies.

To solve this problem, Sun designed a distributed filesystem that permits client systems to access shared files on a remote system. Client machines request resources provided by other machines, called servers. A server machine makes particular filesystems available, which client machines can mount as local filesystems. Thus, users can access remote files as if they were on the local machine.

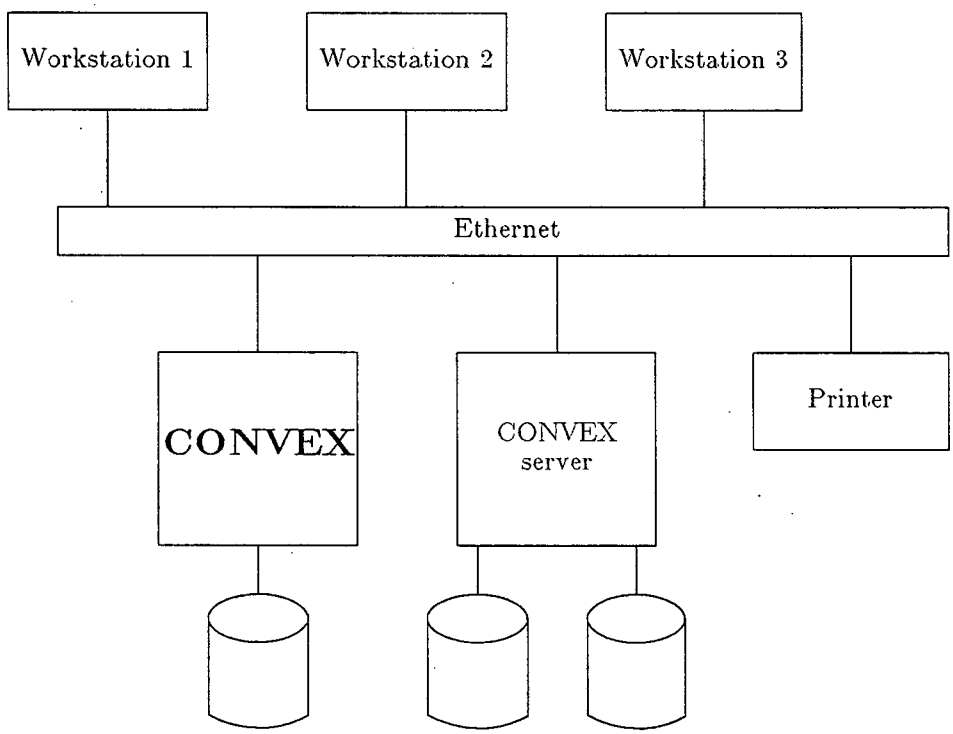
nfs was not designed by extending the operating system onto the network. Instead, *nfs* was designed to fit into Sun's network services architecture. Thus, *nfs* is not a distributed operating system, but rather, an interface to allow a variety of machines and operating systems to play the role of client or server. Sun has opened the *nfs* interface to customers and other vendors, in order to encourage the development of a rich set of applications working together on a single network.

Computing Environments

The traditional computing environment looks like:



The major problem with this environment is competition for CPU cycles. The workstation environment solves that problem, but requires more disk drives. A network environment looks like:



Sun's goal with *nfs* was to make all disks available as needed. Individual workstations have access to all information residing anywhere on the network. Printers and supercomputers may also be available on the network.

Terms and Concepts

A machine that provides resources to the network is a *server*, and a machine that employs these resources is a *client*. A machine may be both a server and a client. A person logged in on a client machine is a *user*; a program or set of programs that run on a client is an *application*. There is a distinction between the code implementing the operations of a filesystem (called *filesystem operations*) and the data making up the filesystem's structure and contents (called *filesystem data*).

A traditional filesystem is composed of directories and files, each of which has a corresponding *inode* (index node) containing administrative information about the file, such as location, size, ownership, permissions, and access times. *inodes* are assigned unique numbers within a filesystem, but a file on one filesystem could have the same number as a file on another filesystem. This is a problem in a network environment, because remote filesystems need to be mounted dynamically, and numbering conflicts would cause havoc. To solve this problem, the virtual file system (*vfs*) was designed. *vfs* is based on the *vnode*, a generalized implementation of *inodes* that are unique across filesystems.

The Remote Procedure Call (*rpc*) facility provides a mechanism whereby one process (the *caller* process) can have another process (the *server* process) execute a procedure call, as if the caller process had executed the procedure call in its own address space (as in the local model of a procedure call). Because the caller and the server are now two separate processes, they no longer have to live on the same physical machine.

The *rpc* mechanism is implemented as a library of procedures, plus a specification for portable data transmission, known as the eXternal Data Representation (*xdr*). Both *rpc* and *xdr* are portable, providing a kind of standard I/O library for interprocess communication. Thus programmers now have a standardized access to sockets without having to be concerned about the low-level details of the *accept()*, *bind()*, and *select()* procedures.

The Yellow Pages (*yp*) is a network service to ease the job of administering networked machines. The *yp* is a centralized read-only database. For a client on the network file system, this means that an application's access to data served by the *yp* is independent of the relative locations of the client and the server. The *yp* database on the server provides password, group, network, and host information to client machines.

Comparison With Predecessors

The Network File System (*nfs*) is composed of a modified kernel, a set of library routines, and a collection of utility commands. *nfs* presents a network client with a complete remote filesystem. Since *nfs* is largely transparent to the user, this document tells you about things you might not otherwise notice. *nfs* is an open system that can accommodate other machines on the net, even non-systems, without compromising security.

Users may be familiar with two previous networking schemes, *rcp* and *ftp*. The first is a remote copy utility program that uses the networking facilities of 4.2BSD to copy files from one machine to another. *ftp* is the user interface to the standard File Transfer Protocol. *ftp* enables users to transfer files to and from a remote site.

nfs, *rcp*, and *ftp*

The remote copy utility (*rcp*) allows data transfer only in units of files. The client of *rcp* supplies the path name of a file on a remote machine, and receives a stream of bytes in return. Access control is based on the client's login name and host name.

The major problem is that *rcp* is not transparent to the user, who winds up with a redundant copy of the desired file. *nfs*, by contrast, is transparent—only one copy of the file is necessary.

Another problem is that *rcp* does nothing but copy files. In a sense, there needs to be one remote command for every regular command: for example, *rdiff* to perform differential file comparisons across machines. By providing entire filesystems, *nfs* makes this unnecessary.

ftp, on the other hand, is an interactive program that provides many more features than *rcp*. *ftp* enables users to transfer files to and from a remote machine, delete files remotely, perform remote shell operations, and so forth. Despite its advantages, however, *ftp* presents many of the same problems encountered with *rcp*. *ftp* does not provide the transparent file access provided by *nfs*, for example. Neither does it eliminate the need for additional commands—*ftp* users must manipulate a large set of commands to perform remote operations. *nfs*, of course, enables users to access remote file systems using essentially the same command set they use for local operations.

Examples of How It Works

This section gives three examples of how to use *nfs*.

Mounting a Remote Filesystem

Suppose that you want to read some online manual pages. These pages are not available on the server machine, called *server*, but are available on a machine called *docserv*. Mount the directory containing the manuals as follows:

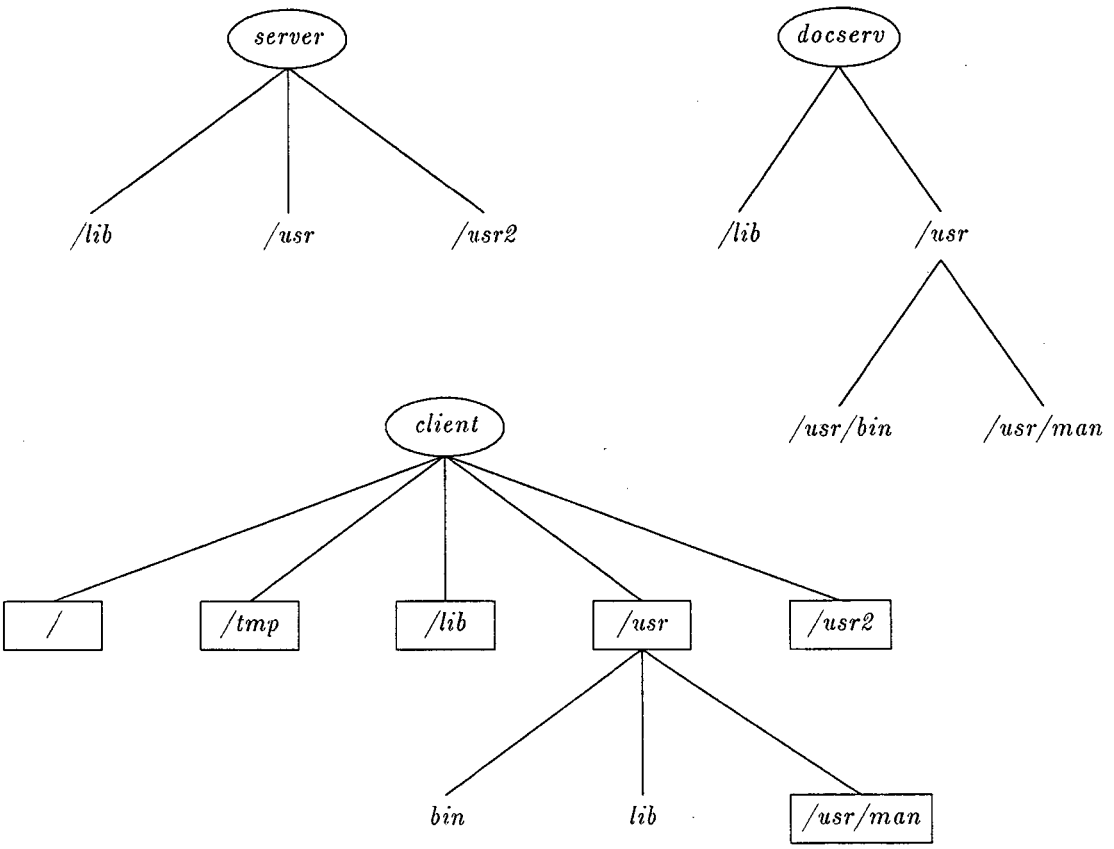
```
client# /etc/mount doc:/usr/man /usr/man
```

Note that you have to be superuser in order to do this. Now you can use the *man* command whenever you want. Try running the *df* command after you've mounted the remote filesystem. Its output will look something like:

```
Filesystem          kbytes    used   avail capacity  Mounted on
/dev/da0a             4775    2765    1532    64%    /
/dev/da0e             5695    3666    1459    72%    /tmp
server:/lib           7295    4137    2428    63%    /lib
server:/usr          39315   31451    3932    89%    /usr
server:/server      326215  245993   47600    84%    /usr2
doc:/usr/man        346111  216894   94605    70%    /usr/man
```

You can remotely mount not only filesystems, but also directory hierarchies inside filesystems. In this example, */usr/man* is not a real mount point—it is a subdirectory of the */usr* filesystem. Figure 1-1 is a diagram of the three machines involved here. Ellipses represent machines and boxes represent remote filesystems.

Figure 1–1: Mounting Directories



Exporting a Filesystem

Suppose that you and a colleague need to work together on a programming project. The source code is on your machine, in the directory */usr/proj*. It does not matter whether your workstation is a diskless node, or has local disk. Suppose that after creating the proper directory, your colleague tried to remotely mount your directory. Unless you have explicitly exported the directory, your colleague's remote mount will fail with a *permission denied* message.

To export a directory, become superuser, and edit the file */etc/exports*. If your colleague is on a machine named *cohort*, then you need to put this one line in */etc/exports*:

```
/usr/proj    cohort
```

Without the keyword *cohort*, anybody on the network could remotely mount your directory */usr/proj*. *nfs* mount request server *mountd*(8c) reads the */etc/exports* file if necessary whenever it receives a request for a remote mount. Now your colleague can remotely mount the source directory by issuing this command:

```
cohort# /etc/mount client:/usr/proj /usr/proj
```

Because both you and your colleague can change files on */usr/proj*, it is best to use the *rcs*(1) source code control system for concurrency control.

Note that you can enable or disable over-the-net superuser privileges for client machines on either a machine-by-machine basis, or a filesystem-by-filesystem basis. For details, see "Superuser Access to Remote Files" in the *CONVEX Network File System System Manager's Guide*.

Administering a Server Machine

System administrators must know how to set up the *nfs* server machine so that client workstations can mount all the necessary filesystems. You export filesystems (that is, make them available) by placing appropriate lines in the */etc/exports* file. Here is a sample */etc/exports* file for a typical server machine:

```
/
/usr
/usr2
/usr/src      staff
```

The pathnames specified in */etc/exports* must be real filesystems—that is, directory mount points for disk devices. A netgroup, such as *staff*, may be specified after the filesystem, in which case remote mounts are limited to machines that are a member of this netgroup. At any one time, the system administrator can see which filesystems have been remotely mounted, by executing the *showmount*(8) command.

2

Architecture of *nfs*

Design Goals

This chapter discusses the design and implementation of *nfs*.

Transparent Information Access

Users are able to get directly to the files they want without knowing the network address of the data. To the user, all universes look alike: there seems to be no difference between reading or writing a file contained on a private disk, and reading or writing a file on a disk in the next building. Information on the network is truly distributed.

Different Machines and Operating Systems

No single vendor can supply tools for all the work that needs to get done, so appropriate services must be integrated on a network. In keeping with its policy of supplying open systems, Sun is promoting *nfs* as a standard for the exchange of data between different machines and operating systems.

Easy Extensibility

A distributed system must have an architecture that allows integration of new software technologies without disturbing the extant software environment. To allow this, *nfs* provides network services, rather than a new network operating system. That is, *nfs* does not depend on extending the underlying operating system onto the network, but instead offers a set of protocols for data exchange. These protocols can be easily extended.

Easy Network Administration

The administration of large networks can be complicated and time-consuming. With the network services provided, however, the administration of network services should be no more difficult to administer than a set of local filesystems on a timesharing system. UNIX has a convenient set of maintenance commands developed over the years. Some new utilities are provided for network administration, but most of the old utilities have been retained.

The Yellow Pages (*yp*) facility is the first example of a network service made possible with *nfs*. By storing password information and host addresses in a centralized database, the yellow pages ease the task of network administration. More information about the *yp* facility is presented in Chapter 3 of the *CONVEX Network File System System Manager's Guide* and in the *CONVEX Yellow Pages Protocol Specification.*

The most obvious use of *yp* is for administration of */etc/passwd.* Since *nfs* uses a protection scheme across the network, it is advantageous to have a common */etc/passwd* database for all machines on the network. *yp* allows a single point of administration, and gives all machines access to a recent version of the data, whether or not it is held locally. To install the *yp* version of */etc/passwd,* existing applications were not changed; they were simply relinked with library

routines that know about the *yp* service. Conventions have been added to library routines that access */etc/passwd* to allow each client to administer its own local subset of */etc/passwd;* the local subset modifies the client's view of the system version. Thus, a client is not forced to completely bypass the system administrator in order to accomplish a small amount of personalization.

The *yp* interface is implemented using *rpc* and *xdr*, so the service is available to non-UNIX operating systems and non-Sun machines. *yp* servers do not interpret data, so it is possible for new databases to take advantage of the *yp* service without modifying the servers.

Reliability

Reliability of the UNIX-based filesystem derives primarily from the robustness of the 4.2BSD filesystem. The file server protocol is also designed so that client workstations can continue to operate even when the server crashes and reboots. Continuation after reboot is achieved without making assumptions about the fail-safe nature of the underlying server hardware.

The major advantage of a stateless server is robustness in the face of client, server, or network failures. Should a client fail, it is not necessary for a server (or human administrator) to take any action to continue normal operation. Should a server or the network fail, it is only necessary that clients continue to attempt to complete *nfs* operations until the server or network gets fixed. This robustness is especially important in a complex network of heterogeneous systems, many of which are not under the control of a disciplined operations staff, and which may be running untested systems often rebooted without warning.

Performance

The flexibility of *nfs* allows configuration for a variety of cost and performance trade-offs. For example, configuring servers with large, high-performance disks may yield better performance at lower cost than having many machines with small, inexpensive disks. Furthermore, it is possible to distribute the filesystem data across many servers and get the added benefit of multiprocessing without losing transparency. In the case of read-only files, copies can be kept on several servers to avoid bottlenecks.

Sun has also added several performance enhancements to *nfs*, such as "fast paths" to eliminate the work done for high-runner operations, asynchronous service of multiple requests, caching of disk blocks, and asynchronous read-ahead and write-behind. The fact that caching and read-ahead occur on both client and server effectively increases the cache size and read-ahead distance. Caching and read-ahead do not add state to the server; nothing (except performance) is lost if cached information is thrown away. In the case of write-behind, both the client and server attempt to flush critical information to disk whenever necessary, to reduce the impact of an unanticipated failure; clients do not free write-behind blocks until the server verifies that the data is written.

The CONVEX performance goal was to achieve the same throughput as a previous release of the system that did not support remote access via *nfs*. This goal has been achieved.

nfs Implementation

In the CONVEX implementation of *nfs*, three entities must be considered: the operating system interface, the virtual file system (*vfs*) interface, and the network file system (*nfs*) interface. The operating system interface has been preserved in the CONVEX implementation of *nfs*, thereby ensuring compatibility for existing applications.

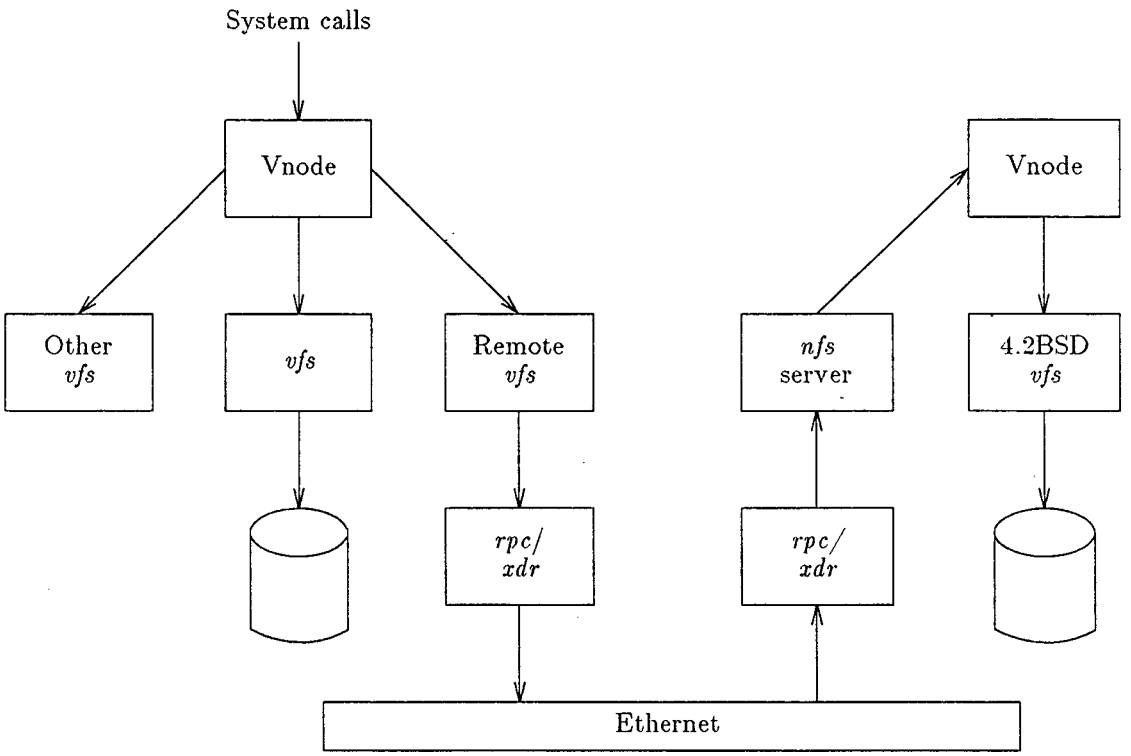
vnodes are a reimplementation of *inodes* that cleanly separate filesystem operations from the semantics of their implementation. Above the *vfs* interface, the operating system deals in *vnodes;* below this interface, the filesystem may or may not implement *inodes*. The *vfs* interface can

connect the operating system to a variety of filesystems (for example, 4.2BSD or MS-DOS). A local *vfs* connects to filesystem data on a local device.

The remote *vfs* defines and implements the *nfs* interface, using the remote procedure call (*rpc*) mechanism. *rpc* allows communication with remote services in a manner similar to the procedure calling mechanism available in many programming languages. The *rpc* protocols are described using the external data representation (*xdr*) package. *xdr* permits a machine-independent representation and definition of high-level protocols on the network.

Figure 2-1 shows the flow of a request from a client (at the top left) to a collection of filesystems.

Figure 2–1: System Call Request Flow



In the case of access through a local *vfs*, requests are directed to filesystem data on devices connected to the client machine. In the case of access through a remote *vfs*, the request is passed through the *rpc* and *xdr* layers onto the net. In the current implementation, Sun uses the UDP/IP protocols and the Ethernet. On the server side, requests are passed through the *rpc* and *xdr* layers to an *nfs* server; the server uses *vnodes* to access one of its local *vfs*s and service the request. This path is retraced to return results.

The CONVEX implementation of *nfs* provides five types of transparency:

- **Filesystem Type**: The *vnode*, in conjunction with one or more local *vfs*s (and possibly remote *vfs*s) permits an operating system (hence client and application) to interface transparently to a variety of filesystem types.

- **Filesystem Location**: Since there is no differentiation between a local and a remote *vfs*, the location of filesystem data is transparent.

- **Operating System Type**: The *rpc* mechanism allows interconnection of a variety of operating systems on the network and makes the operating system type of a remote server transparent.

- **Machine Type**: The *xdr* definition facility allows a variety of machines to communicate on the network and makes the machine type of a remote server transparent.

- **Network Type**: *rpc* and *xdr* can be implemented for a variety of network and internet protocols, thereby making the network type-transparent.

Simpler *nfs* implementations are possible. In particular, a client (or server) may be added to the network by implementing one side of the *nfs* interface. An advantage of the CONVEX implementation is that the client and server sides are identical; thus, it is possible for any machine to be client, server, or both. Users at client machines with disks can arrange to share over *nfs* without having to appeal to a system administrator, or configure a different system on their workstation.

nfs Interface

As mentioned in the preceding section, a major advantage of *nfs* is the ability to mix filesystems. In keeping with this, CONVEX encourages other vendors to develop products to interface with these network services. *rpc* and *xdr* have been placed in the public domain and serve as a standard for anyone wishing to develop applications for the network. Furthermore, the *nfs* interface itself is open and can be used by anyone wishing to implement an *nfs* client or server for the network.

The *nfs* interface defines traditional filesystem operations for reading directories, creating and destroying files, reading and writing files, and reading and setting file attributes. The interface is designed so that file operations address files with an uninterpreted identifier, starting byte address, and length in bytes. Commands are provided for *nfs* servers to initiate service (*mountd*) and to serve a portion of their filesystem to the network (*/etc/exports*). Many commands are provided for constructing the *yp* database facility. A client builds its view of the filesystems available on the network with the *mount* command.

The *nfs* interface is defined so that a server can be *stateless*. This means that a server does not have to remember from one transaction to the next anything about its clients, transactions completed or files operated on. For example, there is no *open* operation, as this would imply state in the server; of course, the interface uses an *open* operation, but the information in the operation is remembered by the client for use in later *nfs* operations.

An interesting problem occurs when a UNIX application *unlink*s an open file. This is done to achieve the effect of a temporary file that is automatically removed when the application terminates. If the file in question is served by *nfs*, the *unlink* removes the file, since the server does not remember that the file is open. Thus, subsequent operations on the file will fail. In order to avoid state on the server, the client operating system detects the situation, renames the file rather than unlinking it, and *unlink*s the file when the application terminates. In certain failure cases, this leaves unwanted "temporary" files on the server; these files are removed as a part of periodic filesystem maintenance.

Another example of how *nfs* provides a friendly interface to CONVEX UNIX without introducing state is the *mount* command. A client of *nfs* "builds" its view of the filesystem on its local devices using the *mount* command; thus, it is natural for the client to initiate its contact with *nfs* and build its view of the filesystem on the network with an extended *mount* command. This *mount* command does not imply state in the server, since it only acquires information for the

client to establish contact with a server. The *mount* command may be issued at any time, but is typically executed as a part of client initialization. The corresponding *umount* command is only an informative message to the server, but it does change state in the client by modifying its view of the filesystem on the network.

The major advantage of a stateless server is robustness in the face of client, server or network failures. Should a client fail, it is not necessary for a server (or human administrator) to take any action to continue normal operation. Should a server or the network fail, it is only necessary that clients continue to attempt to complete *nfs* operations until the server or network is fixed. This robustness is especially important in a complex network of heterogeneous systems, many of which are not under the control of a disciplined operations staff and may be running untested systems and/or may be rebooted without warning.

An *nfs* server can be a client of another *nfs* server. A server will not, however, act as an intermediary between a client and another server. Instead, a client may ask what remote mounts the server has and then attempt to make similar remote mounts. The decision to disallow intermediary servers is based on several factors. First, the existence of an intermediary impacts the performance characteristics of the system; the potential performance implications are so complex that it seems best to require direct communication between a client and server. Second, the existence of an intermediary complicates access control; it is much simpler to require a client and server to establish direct agreements for service. Finally, disallowing intermediaries prevents cycles in the service arrangements; this is preferable to detection or avoidance schemes.

nfs currently implements file protection by making use of the authentication mechanisms built into *rpc*. This retains transparency for clients and applications that make use of file protection. Although the *rpc* definition allows other authentication schemes, their use may have adverse effects on transparency.

Although *nfs* is "UNIX-friendly," it does not support all UNIX filesystem operations. For example, the "special file" abstraction of devices is not supported for remote filesystems because it is felt that the interface to devices would greatly complicate the *nfs* interface; instead, devices are implemented in a local */dev vfs*. Other incompatibilities are due to the fact that *nfs* servers are stateless. For example, file locking and guaranteed APPEND_MODE are not supported in the remote case.

The CONVEX decision to omit certain features from *nfs* is motivated by a desire to preserve the stateless implementation of servers and to define a simple, general interface to be implemented and used by a wide variety of customers. The availability of open *rpc* and *nfs* interfaces means that customers and users who need stateful or complex features can implement them "beside" or "within" *nfs*. Implementation of a set of tools for use by applications that need file or record locking, replicated data, or other features implying state and/or distributed synchronization is being considered; however, these will not be made part of the base *nfs* definition.

3

Yellow Pages Database

Introduction

This chapter explains the network database mechanism, called the yellow pages. Although this material is not intended for system administrators, it is heavily slanted in that direction. The yellow pages are another network service offered for the first time with CONVEX UNIX V6.0. They permit password information and host addresses for an entire network to be held in a single database. This greatly eases the task of system and network administration.

What Are the Yellow Pages?

The yellow pages (*yp*) constitute a distributed network lookup service:

- *yp* is a lookup service: it maintains a set of databases for querying. Programs can ask for the value associated with a particular key, or all the keys, in a database.
- *yp* is a network service: programs need not know the location of data, or how it is stored. Instead, they use a network protocol to communicate with a database server that knows those details.
- *yp* is distributed: databases are fully replicated on several machines known as *yp* servers. Servers propagate updated databases among themselves, thus ensuring consistency. At steady state, it doesn't matter which server answers a request; the answer is the same everywhere.

yp Map

The yellow pages serve information stored in *yp maps*. Each map contains a set of keys and associated values. For example, the *hosts* map contains (as keys) all host names on a network, and (as values) the corresponding Internet addresses. Each *yp* map has a *mapname*, used by programs to access data in the map. Programs must know the format of the data in the map. Currently, most maps are derived from ASCII files formerly found in */etc: passwd, group, hosts. networks*, and others. The format of data in the *yp* map is usually identical to the format of the ASCII file. Maps are implemented by *dbm*(3) files located in */etc/yp* subdirectories on *yp* servers.

yp Domain

A *yp domain* is a named set of *yp* maps. You can determine your *yp* domain by executing the *domainname*(1) command. Note that *yp* domains are different from both Internet domains and *sendmail* domains. A *yp* domain is simply a directory in */etc/yp* containing a set of maps.

A domain name is required for retrieving data from a *yp* database. For instance, if your *yp* domain is *convex* and you want to find the Internet address of host *dbserver*, you must ask *yp* for the value associated with the key *dbserver* in the map *hosts.byname* within the *yp* domain *convex*. Each machine on the network belongs to a default domain, set in */etc/rc.local* at boot time with the *domainname*(1) command.

A *yp* server holds all the maps of a *yp* domain in a subdirectory of */etc/yp*, named after the domain. In the example above, maps for the *convex* domain would be held in */etc/yp/convex*. This information is used internally by the *yp*.

Servers and Clients

Servers provide resources, and clients consume them. A server or a client is not necessarily the same thing as a machine. To illustrate, let's consider two different services: the *nfs* (network file system) and the *yp*.

- *nfs* allows client machines to mount remote filesystems and access files in place, provided a server machine has exported the filesystem. However, a server that exports filesystems may also mount remote filesystems exported by other machines, thus becoming a client. So a given machine may be both server and client, or client only, or server only.

- The *yp* server, by contrast, is a process (rather than a machine) running on a machine that may be an *nfs* server. A process can request information out of the *yp* database, obviating the need to have such information on every machine. All processes that make use of *yp* services are *yp* clients. Sometimes clients are served by *yp* servers on the same machine, but other times by *yp* servers running on another machine. If a remote machine running a *yp* server process crashes, client processes can obtain *yp* services from another machine. This is so that *yp* services are almost always available.

Masters and Slaves

yp servers are either master or slave. For any map, one *yp* server is designated the master, and all changes to the map should be made on that machine. The changes propagate from master to slaves. A newly built map is timestamped internally when *makedbm* creates it. If you build a *yp* map on a slave server, you break the *yp* update algorithm (temporarily), and must get all versions in synch manually. Moral: after you decide which server is the master, do all database updates and builds there, not on slaves.

It is possible for different maps to have different servers as master. A given server may even be master with regard to one map, and slave with regard to another. This can get confusing quickly. It is recommended that a single server be master for all maps created by *ypinit* in a single domain. This document assumes the simple case, in which one server is the master for all maps in a database.

4

Overview of the Yellow Pages

Introduction

In releases prior to CONVEX UNIX V6.0, *nfs* was not offered, and each machine on the network had its own copy of */etc/hosts*, a file containing the Internet address of each machine on the network. Every time a machine was added to the network, each */etc/hosts* file had to be updated.

yp is a network service containing network-wide databases such as */etc/hosts*. Servers spread throughout the network contain copies of the databases. When an arbitrary machine on the network wants to look up something in */etc/hosts*, it makes an *rpc* call to one of the servers to get the information. One server is the master — the only one whose database may be modified. The other servers are slaves, and they are periodically updated so that their information is in synch with that of the master.

yp can serve any number of databases. Normally that includes files that previously lived in */etc*, such as */etc/hosts* and */etc/networks*. However, users can add their own databases to the *yp*.

yp itself simply serves information, and has no idea what it means. Thus, two parts of *yp* must be considered: how it operates, and what files formerly in */etc* now live in the *yp*. This has serious ramifications for users.

yp Network Service

The following sections describe how names and data are handled by *yp*. *yp* servers and clients are also described. The chapter concludes with a discussion of default *yp* files.

Naming

Imagine a company with two different networks, each of which has its own separate list of hosts and passwords. Within each network, user names, numerical user IDs, and host names are unique. There is, however, duplication between the two networks. If these two networks are ever connected, chaos could result. The host name, returned by the *hostname*(1) command and the *gethostname*(2) system call, may no longer uniquely identify a machine. Thus a new command and system call, *domainname*(1) and *getdomainname*(2), have been added. In the example above, each of the two networks could be given a different domain name; it is, however, always simpler to use a single domain whenever possible.

The relevance of domains to *yp* is that data is stored in /etc/yp/*domainname*. In particular, a machine can contain data for several different domains.

Data Storage

The data is stored in *dbm*(3) format. Thus the database *hosts.byname* for the domain *convex* is stored as */etc/yp/convex/hosts.byname.pag* and */etc/yp/convex/hosts.byname.dir*. The command *makedbm*(8) takes an ASCII file such as */etc/hosts* and converts it into a *dbm* file suitable for use by the *yp*. However, system administrators normally use the makefile in */etc/yp* to create new *dbm* files. This makefile in turn calls *makedbm*.

Servers

To become a server, a machine must contain the *yp* databases, and must also be running the *yp* daemon *ypserv*. The *ypinit*(8) command invokes this daemon automatically. It also takes a flag saying whether you are creating a master or a slave. When updating the master copy of a database, you can force the change to be propagated to all the slaves with the *yppush*(8) command. This pushes the information out to all the slaves. Conversely, from a slave, the *ypxfr*(8) command gets the latest information from the master. The makefile in */etc/yp* first executes *makedbm* to make a new database, and then calls *yppush* to propagate the change throughout the network.

Clients

Remember that a client machine (which is not a server) does not access local copies of */etc* files, but rather makes an *rpc* call to a *yp* server each time it needs information from a *yp* database. The *ypbind*(8) daemon remembers the name of a server. When a client boots, *ypbind* broadcasts asking for the name of the *yp* server. Similarly, *ypbind* broadcasts asking for the name of a new *yp* server if the old server crashes. The *ypwhich*(1) command gives the name of the server that *ypbind* currently points at.

Since client machines no longer have entire copies of files in the *yp*, two new commands *ypcat*(1) and *ypmatch*(1) have been provided. The command *ypcat hosts* is equivalent to *cat /etc/hosts* in a pre-6.0 system; as you might guess, *ypcat passwd* is equivalent to *cat /etc/passwd* . To look for someone's password entry, searching through the password file no longer suffices; you have to issue one of the following commands

```
% ypcat passwd | grep username
% ypmatch username passwd
```

where you replace *username* with the login name you're searching for.

Default *yp* Files

By default, C1s have seven files from */etc* in the *yp*: */etc/passwd, /etc/pwrestrict, /etc/group, /etc/hosts, /etc/networks, /etc/services, /etc/protocols;* there is also a new file *netgroup,* which many sites should create and put in the *yp* database.

Library routines such as *getpwent*(3), *getgrent*(3), *gethostent*(3), and *getpwrestent*(3) have been rewritten to take advantage of the *yp*. Thus, C programs that call these library routines must be relinked in order to function correctly.

hosts

The *hosts* file is stored as two different files in the *yp*. The first, *hosts.byname*, is indexed by hostname. The second, *hosts.byaddr*, is indexed by Internet address. Remember that this actually expands into four files, with suffixes *.pag*, and *.dir*. When a user program calls the library routine *gethostbyname*(3), a single *rpc* call to a server retrieves the entry from the *hosts.byname* file. Similarly, *gethostbyaddr*(3) retrieves the entry from the *hosts.byaddr* file. Of course if the *yp* is not running (which is caused by commenting *ypbind* out of the */etc/rc* file), then *gethostbyname* reads the */etc/hosts* files, just as it always has.

Maps sometimes have nicknames. Although the *ypcat* command is a general *yp* database print program, it knows about the standard files in the *yp*. Thus *ypcat hosts* is translated into *ypcat hosts.byaddr*, since there is no file called *hosts* in the *yp*. The command *ypcat -x* furnishes a list of expanded nicknames.

Normally, the hosts file for the *yp* is the same as the */etc/hosts* file on the machine serving as a *yp* master. In this case, the makefile in */etc/yp* checks to see if */etc/hosts* is newer than the *dbm* file. If it is, it uses a simple *sed* script to recreate *hosts.byname* and *hosts.byaddr*, run them through *makedbm*, and then call *yppush*. See *ypmake*(8) for details.

passwd

The *passwd* file is similar to the *hosts* file. It exists as two separate files, *passwd.byname* and *passwd.byuid*. The *ypcat* program prints it, and *ypmake* updates it. If *getpwent* always went directly to the *yp* as does *gethostent*, however, then everyone would be forced to have an identical password file. Consequently, *getpwent* reads the local */etc/passwd* file, just as it always did. But now it interprets "+" entries in the password file to mean: interpolate entries from the *yp* database. If you wrote a simple program using *getpwent* to print all the entries from your password file, it would print a virtual password file; rather than printing out + signs, it would print whatever entries the local password file included from the *yp* database.

Others

Of the other five files in */etc*, */etc/group* and */etc/pwrestrict* are treated like */etc/passwd*, in that *getgrent* and *getpwrestent* only consult the *yp* if explicitly told to do so by the */etc/group* and */etc/pwrestrict* files. The files */etc/networks*, */etc/services*, */etc/protocols*, */etc/ethers*, and */etc/netgroup* are treated like */etc/hosts;* for these files, the library routines go directly to the *yp* without consulting the local files.

Changing Your Password

To change data in the *yp*, the system administrator must log in to the master machine and edit databases there; *ypwhich* tells where the master server is. Since changing a password is so commonly done, however, the *yppasswd*(1) command has been provided to change your *yp* password. It has the same user interface as the *passwd*(1) command. This command only works if the *yppasswdd*(8C) server has been started up on the *yp* master server machine.

Index

CONVEX Network File System Protocol Specification

November 1, 1987

CONVEX Computer Corporation

Table of Contents

1

Introduction

The Network Filesystem (*nfs*) protocol provides transparent remote access to shared filesystems over local area networks. The *nfs* protocol is designed to be machine, operating system, network architecture, and transport protocol independent. This independence is achieved through the use of Remote Procedure Call (*rpc*) primitives built on top of an eXternal Data Representation (*xdr*).

The supporting mount protocol allows the server to hand out remote access privileges to a restricted set of clients. Thus, it allows clients to attach a remote directory tree at any point on some local filesystem.

Remote Procedure Call

The remote procedure call specification, described in the *RPC Programming Guide*, provides a clean, procedure-oriented interface to remote services. Each server supplies a program that is a set of procedures. The combination of host address, program number, and procedure number specifies one remote service procedure.

rpc is a high-level protocol built on top of low-level transport protocols. It does not depend on services provided by specific protocols, so it can be used easily with any underlying transport protocol. Currently the only supported transport protocol is UDP/IP.

The *rpc* protocol includes a slot for authentication parameters on every call. The contents of the authentication parameters are determined by the "flavor" (type) of authentication used by the server and client. A server may support several different flavors of authentication at once: *AUTH_NONE* passes no authentication information (this is called null authentication); *AUTH_UNIX* passes the UNIX uid, gid, and groups with each call.

Servers have been known to change over time, and so can the protocol that they use. So *rpc* provides a version number with each *rpc* request. Thus, one server can service requests for several different versions of the protocol at the same time.

External Data Representation

The external data representation specification, described in the *XDR Protocol Specification*, provides a common way of representing a set of data types over a network. This takes care of problems such as different byte ordering on different communicating machines. It also defines the size of each data type so that machines with different structure alignment algorithms can share a common format over the network.

In this document we use the *xdr* data definition language to specify the parameters and results of each *rpc* service procedure that a *nfs* server provides. The *xdr* data definition language reads a lot like C, although a few new constructs have been added. The notation

```
stringname[SIZE];
string data<DSIZE>;
```

defines *name,* which is a fixed size block of *SIZE* bytes, and *data,* which is a variable size block of up to *DSIZE* bytes. This same notation is used to indicate fixed length arrays, and arrays with a variable number of elements up to some maximum. The discriminated union definition

```
union switch (enum status) {
   NFS_OK:
      struct {
            filename     file1;
            filename     file2;
            integer            count;
      }
   NFS_ERROR:
      struct {
            errstat            error;
            integer            errno;
      }
   default:
      struct {}
}
```

means the first thing over the network is an enumeration type called *status;* if its value is *NFS_OK,* the next thing on the network is the structure containing *file1, file2,* and *count.* If the value of *status* is neither *NFS_OK* nor *NFS_ERROR,* then there is no more data to look at.

Stateless Servers

The *nfs* protocol is stateless. That is, a server does not need to maintain state about any of its clients in order to function correctly. Stateless servers have a distinct advantage over stateful servers in the event of a crash. With stateless servers, a client need only retry a request until the server responds; it does not even need to know that the server has crashed. The client of a stateful server, on the other hand, needs to detect a server crash and rebuild the server's state when it comes back up.

This may not sound like an important issue, but it affects the protocol in some strange ways. We feel that it is worth a bit of extra complexity in the protocol to be able to write very simple servers that don't need fancy crash recovery.

2

nfs Protocol Definition

Introduction

Although the *nfs* protocol is designed to be operating system independent, it was designed in a UNIX environment. As such, it has some typically UNIX features. Keep that in mind when you are in doubt about how something should work.

The protocol definition is given as a set of procedures with arguments and results defined using *xdr*. A brief description of the function of each procedure should provide enough information to allow implementation on most machines. A different section is provided for each supported version of the protocol. Most of the procedures, and their parameters and results, are self-explanatory. A few do not fit into the normal mold, however.

The *LOOKUP* procedure looks up one component of a pathname at a time. It is not obvious at first why it does not simply evaluate the whole pathname, track down the directories, and return a file handle when it is done. There are two good reasons not to do this. First, pathnames need separators between the directory components, and different operating systems use different separators. If a Network Standard Pathname Representation were defined, then every pathname would have to be parsed and converted at each end. Second, if pathnames were passed, the server would have to keep track of the mounted filesystems for all of its clients, so that it could break the pathname at the right point and pass the remainder on to the correct server.

Another procedure requiring explanation is the *READDIR* procedure. *READDIR* provides a network standard format for representing directories. The same argument as above could have been used to justify a *READDIR* procedure that returns only one directory entry per call. The problem is efficiency. Directories can contain many entries, and a remote call to return each would just be too slow.

Version 2

The released version of the *nfs* protocol is actually the second. This section describes various obsolete procedures and parameters in the second version, which will probably be removed in later versions.

Server/Client Relationship

The *nfs* protocol is designed to allow servers to be as simple and general as possible. Sometimes the simplicity of the server can be a problem, if the client wants to implement complicated filesystem semantics.

For example, UNIX allows removal of open files. A process can open a file and, while it is open, remove it from the directory. The file can be read and written as long as the process keeps it open, even though the file has no name in the filesystem. It is impossible for a stateless server to implement these semantics. The client can do some tricks like renaming the file on remove, and only removing it on close. The server provides enough functionality to implement most filesystem semantics on the client.

Every *nfs* client can also be a server, and remote and local mounted filesystems can be freely

intermixed. This leads to some interesting problems when a client travels down the directory tree of a remote filesystem and reaches the mount point on the server for another remote filesystem. Allowing the server to following the second remote mount means it must do loop detection, server lookup, and user revalidation. Instead, this implementation does not let clients cross a server's mount point. When a client does a *LOOKUP* on a directory that the server has mounted a filesystem on, the client sees the underlying directory instead of the mounted directory. A client can do remote mounts that match the server's mount points to maintain the server's view.

Permission Issues

The *nfs* protocol, strictly speaking, does not define the permission checking used by servers. However, it is expected that a server does normal UNIX permission checking using *AUTH_UNIX* style authentication as the basis of its protection mechanism. The server gets the client's effective *uid,* effective *gid,* and groups on each call, and uses them to check permission. Various problems with this method can be resolved in interesting ways.

Using *uid* and *gid* implies that the client and server share the same *uid* list. Every server and client pair must have the same mapping from user to *uid* and from group to *gid.* Since every client can also be a server, this tends to imply that the whole network shares the same *uid/gid* space. This is acceptable for the short term, but a more workable network authentication method will be necessary before long.

Since for now clients and servers must share the same *uid* list, make sure that the password files on client and server are identical. Otherwise, *uid* and *gid* lists will not match. The Yellow Pages, of course, solve the problem neatly by providing a network-wide database that can be used to ensure consistent password-to-uid mapping.

Another problem arises due to the semantics of open. UNIX does its permission checking at open time and then that the file is open, and has been checked on later read and write requests. With stateless servers this breaks down, because the server has no idea that the file is open and it must do permission checking on each read and write call. On a local filesystem, a user can open a file, then change the permissions so that no one is allowed to touch it, then still be able to write to the file because it is open. On a remote filesystem, by contrast, the write would fail. To get around this problem, the server's permission checking algorithm should allow the owner of a file to access it no matter what the permissions are set to.

A similar problem has to do with paging in from a file over the network. The kernel checks for execute permission before opening a file for demand paging, then reads blocks from the open file. The file may not have read permission but after it is opened it doesn't matter. An *nfs* server can't tell the difference between a normal file read and a demand page-in read. To make this work, the server allows reading of files if the *uid* given in the call has execute or read permission on the file.

In UNIX, the user ID zero has access to all files no matter what permission and ownership they have. This superuser permission is not allowed on the server since anyone who can become superuser on their workstation could gain access to all remote files. Instead, the server maps *uid* 0 to –2 before doing its access checking. This works as long as the *nfs* is not used to supply root filesystems, where superuser access cannot be avoided. Eventually servers will have to allow some kind of limited superuser access.

rpc Information

- The *nfs* service uses *AUTH_UNIX* style authentication except in the *NULL* procedure where *AUTH_NONE* is also allowed.
- *nfs* currently is supported on UDP/IP only.

- These are the *rpc* constants needed to call the *nfs* service. They are given in decimal.

```
PROGRAM          100003
VERSION          2
```

- The *nfs* protocol currently uses the UDP port number 2049. This is a bug in the protocol and will be changed shortly.

Sizes

These are the sizes, given in decimal bytes, of various *xdr* structures used in the protocol.

- MAXDATA 8192—The maximum number of bytes of data in a *READ* or *WRITE* request. (Note: This constant varies from file system to file system according to the file system's maximum block size. The maximum block size supported by CONVEX UNIX is 64K.)
- MAXPATHLEN 1024—The maximum number of bytes in a pathname argument.
- MAXNAMLEN 255—The maximum number of bytes in a file name argument.
- COOKIESIZE 4—The size in bytes of the opaque "cookie" passed by *READDIR*.
- FHSIZE 32—The size in bytes of the opaque file handle.

Basic Data Types

The following *xdr* definitions are basic structures and types used in other structures later on.

stat

```
typedef enum {
        NFS_OK = 0,
        NFSERR_PERM=1,
        NFSERR_NOENT=2,
        NFSERR_IO=5,
        NFSERR_NXIO=6,
        NFSERR_ACCES=13,
        NFSERR_EXIST=17,
        NFSERR_NODEV=19,
        NFSERR_NOTDIR=20,
        NFSERR_ISDIR=21,
        NFSERR_FBIG=27,
        NFSERR_NOSPC=28,
        NFSERR_ROFS=30,
        NFSERR_NAMETOOLONG=63,
        NFSERR_NOTEMPTY=66,
        NFSERR_DQUOT=69,
        NFSERR_STALE=70,
        NFSERR_WFLUSH=99
} stat;
```

The *stat* type is returned with every procedure's results. A value of *NFS_OK* indicates that the call completed successfully and the results are valid. The other values indicate some kind of error occurred on the server side during the servicing of the procedure. The error values are derived from error numbers.

- Not owner. Caller does not have correct ownership to perform requested operation.
- No such file or directory. The file or directory specified does not exist.
- I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.
- No such device or address.
- Permission denied. The caller does not have the correct permission to perform the requested operation.
- File exists. The file specified already exists.
- No such device.
- Not a directory. The caller specified a non-directory in a directory operation.
- Is a directory. The caller specified a directory in a non-directory operation.
- File too large. The operation caused a file to grow beyond the server's limit.
- No space left on device. The operation caused the server's filesystem to reach its limit.
- Read-only filesystem. Write attempted on a read-only filesystem.
- File name too long. The file name in an operation was too long.
- Directory not empty. Attempted to remove a directory that was not empty.
- Disk quota exceeded. The client's disk quota on the server has been exceeded.
- The *fhandle* given in the arguments was invalid. That is, the file referred to by that file handle no longer exists, or access to it has been revoked.
- The server's write cache used in the *WRITECACHE* call got flushed to disk.

ftype

```
typedef enum {
        NFNON = 0,
        NFREG = 1,
        NFDIR = 2,
        NFBLK = 3,
        NFCHR = 4,
        NFLNK = 5,
       NFSOCK = 6
} ftype;
```

The enumeration *ftype* gives the type of a file. The type *NFNON* indicates a non-file, *NFREG* is a regular file, *NFDIR* is a directory, *NFBLK* is a block-special device, *NFCHR* is a character-special device, and *NFLNK* is a symbolic link. *NFSOCK* is a socket.

fhandle

```
typedef opaque      fhandle[FHSIZE];
```

The *fhandle* is the file handle that the server passes to the client. All file operations are done using file handles to refer to a file or directory. The file handle can contain whatever information the server needs to distinguish an individual file.

timeval

```
typedef struct {
        unsigned seconds;
        unsigned useconds;
} timeval;
```

The *timeval* structure is the number of seconds and microseconds since midnight January 1, 1970 Greenwich Mean Time. It is used to pass time and date information.

fattr

```
typedef struct {
        ftype  type;
        unsigned mode;
        unsigned nlink;
        unsigned uid;
        unsigned gid;
        unsigned size;
        unsigned blocksize;
        unsigned rdev;
        unsigned blocks;
        unsigned fsid;
        unsigned fileid;
        timeval       atime;
        timeval       mtime;
        timeval       ctime;
} fattr;
```

The *fattr* structure contains the attributes of a file; *type* is the type of the file; *nlink* is the number of hard links to the file (the number of different names for the same file); *uid* is the user identification number of the owner of the file; *gid* is the group identification number of the group of the file; *size* is the size in bytes of the file; *blocksize* is the size in bytes of a block of the file; *rdev* is the device number of the file if it is type *NFCHR* or *NFBLK; blocks* is the number of blocks the file takes up on disk; *fsid* is the filesystem identifier for the filesystem containing the file; *fileid* is a number that uniquely identifies the file within its filesystem; *atime* is the time when the file was last accessed for either read or write; *mtime* is the time when the file data was last modified (written); and *ctime* is the time when the status of the file was last changed. Writing to the file also changes *ctime* if the size of the file changes.

mode is the access mode encoded as a set of bits. The bits are the same as the mode bits returned by the *stat*(2) system call in UNIX. Notice that the file type is specified both in the mode bits and in the file type. This is really a bug in the protocol and should be fixed in future versions. The descriptions given below specify the *mode* bit positions using octal numbers.

- 0040000. This is a directory. The *type* field should be *NFDIR*.
- 0020000. This is a character special file. The *type* field should be *NFCHR*.
- 0060000. This is a block special file. The *type* field should be *NFBLK*.
- 0100000. This is a regular file. The *type* field should be *NFREG*.
- 0120000. This is a symbolic link file. The *type* field should be *NFLNK*.
- 0140000. This is a named socket. The *type* field should be *NFNON*.
- 0004000. Set user ID on execution.
- 0002000. Set group ID on execution.
- 0001000. Save swapped text even after use.
- 0000400. Read permission for owner.
- 0000200. Write permission for owner.
- 0000100. Execute and search permission for owner.
- 0000040. Read permission for group.
- 0000020. Write permission for group.
- 0000010. Execute and search permission for group.
- 0000004. Read permission for others.
- 0000002. Write permission for others.
- 0000001. Execute and search permission for others.

sattr

```
typedef struct {
   unsigned  mode;
   unsigned  uid;
   unsigned  gid;
   unsigned  size;
   timeval   atime;
   timeval   mtime;
} sattr;
```

The *sattr* structure contains the file attributes which can be set from the client. The fields are the same as for *fattr* above. A *size* of zero means the file should be truncated. A value of -1 indicates a field that should be ignored.

filename

```
typedef string      filename<MAXNAMLEN>;
```

The type *filename* is used for passing file names or pathname components.

path

```
typedef string      path<MAXPATHLEN>;
```

The type *path* is a pathname. The server considers it as a string with no internal structure, but to the client it is the name of a node in a filesystem tree.

attrstat

```
typedef union switch (stat status) {
   NFS_OK:
       fattr  attributes;
   default:
       struct {}
} attrstat;
```

The *attrstat* structure is a common procedure result. It contains a *status* and, if the call succeeded, it also contains the attributes of the file on which the operation was done.

diropargs

```
typedef struct {
   fhandle   dir;
   filename  name;
} diropargs;
```

The *diropargs* structure is used in directory operations. The *fhandle dir* is the directory in which to find the file *name*. A directory operation is one in which the directory is affected.

diropres

```
typedef union switch (stat status) {
   NFS_OK:
       struct {
              fhandle      file;
              fattr  attributes;
       }
   default:
       struct {}
} diropres;
```

The results of a directory operation are returned in a *diropres* structure. If the call succeeded a new file handle *file* and the *attributes* associated with that file are returned along with the *status*.

Server Procedures

The following sections define the *rpc* procedures supplied by a *nfs* server. The *rpc* procedure number and version are given in the header, along with the name of the procedure. The synopsis of procedures has this format:

```
<proc #>. <proc name> ( <arguments> ) returns ( <results> )
        <argument declarations>
        <results declarations>
```

In the first line, *proc name* is the name of the procedure, *arguments* is a list of the names of the arguments, and *results* is a list of the names of the results. The second and third lines give the *xdr argument declarations* and *results declarations*. Afterward, there is a description of what the procedure is expected to do, and how its arguments and results are used. If there are bugs or problems with the procedure, they are listed at the end.

All of the procedures in the *nfs* protocol are assumed to be synchronous. When a procedure returns to the client, the client can assume that the operation has completed, and any data associated with the request is now on stable storage. For example, a client WRITE request may cause the server to update data blocks, filesystem information blocks (such as indirect blocks in UNIX) and file attribute information (size and modify times). When the WRITE returns to the client, it can assume that the write is safe, even in case of a server crash, and it can discard the data written. This is a very important part of the statelessness of the server. If the server waited to flush data from remote requests the client would have to save those requests so that it could resend them in case of a server crash.

Do Nothing

Procedure 0, Version 2.

```
0. NFSPROC_NULL ( ) returns ( )
```

This procedure does no work. It is made available in all *rpc* services to allow server response testing and timing.

Get File Attributes

Procedure 1, Version 2.

```
1. NFSPROC_GETATTR (file) returns (reply)
        fhandle file;
        attrstat reply;
```

If *reply.status* is *NFS_OK*, then *reply.attributes* contains the attributes for the file given by *file*.

Bugs: the *rdev* field in the attributes structure is a device specifier. It should be removed or generalized.

Set File Attributes

Procedure 2, Version 2.

```
2. NFSPROC_SETATTR (file, attributes) returns (reply)
        fhandle       file;
        sattr  attributes;
        attrstat reply;
```

The *attributes* argument contains fields that are either –1 or are the new value for the attributes of *file*. If *reply.status* is *NFS_OK*, then *reply.attributes* has the attributes of the file after the *setattr* operation has completed.

Bugs: the use of –1 to indicate an unused field in *attributes* is wrong.

Get Filesystem Root

Procedure 3, Version 2.

```
3. NFSPROC_ROOT ( ) returns ( )
```

Obsolete. This procedure is no longer used because finding the root file handle of a filesystem requires moving pathnames between client and server. To do this right, we would have to define a network standard representation of pathnames. Instead, the function of looking up the root file handle is done by the *MNTPROC_MNT* procedure (see section entitled *Mount Protocol Definition* for details).

Look Up File Name

Procedure 4, Version 2.

```
4. NFSPROC_LOOKUP (which) returns (reply)
        diropargs which;
        dirоpres  reply;
```

If *reply.status* is *NFS_OK*, then *reply.file* and *reply.attributes* are the file handle and attributes for the file *which.name* in the directory given by *which.dir*.

Bugs: there is some question as to what is the correct reply to a LOOKUP request when *which.name* is a mount point on the server for a remotely mounted filesystem. Currently, we return the *fhandle* of the underlying directory. This is not completely acceptable, as the clients see a different view of the filesystem than the server does.

Read From Symbolic Link

Procedure 5, Version 2.

```
5. NFSPROC_READLINK (file) returns (reply)
        fhandle      file;
        union switch (stat status) {
           NFS_OK:
               path  data;
           default:
               struct {}
        } reply;
```

If *status* has the value NFS_OK, then *reply.data* is the data in the symbolic link given by *file*.

Read From File

Procedure 6, Version 2.

```
6. NFSPROC_READ (file, offset, count, totalcount)
   returns (reply)
        fhandle      file;
        unsigned offset;
        unsigned count;
        unsigned totalcount;
        union switch (stat status) {
           NFS_OK:
               fattr attributes;
               stringdata<MAXDATA>;
           default:
               struct {}
        } reply;
```

This procedure returns up to *count* bytes of *data* from the file given by *file*, starting at *offset* bytes from the beginning of the file. The first byte of the file is at offset zero. The file attributes after the read takes place are returned in *attributes*.

Bugs: the argument *totalcount* is unused and should be removed.

Write to Cache

Procedure 7, Version 2.

```
7. NFSPROC_WRITECACHE ( ) returns ( )
```

Obsolete.

Write to File

Procedure 8, Version 2.

```
8. NFSPROC_WRITE (file,beginoffset,offset,totalcount,data)
  returns (reply)
       fhandle       file;
       unsigned beginoffset;
       unsigned offset;
       unsigned totalcount;
       string data<MAXDATA>;
       attrstat reply;
```

This procedure writes *data* beginning *offset* bytes from the beginning of *file*. The first byte of the file is at offset zero. If *reply.status* is *NFS_OK*, then *reply.attributes* contains the attributes of the file after the write has completed. The write operation is atomic. Data from this *WRITE* will not be mixed with data from another client's *WRITE*.

Bugs: the arguments *beginoffset* and *totalcount* are ignored and should be removed.

Create File

Procedure 9, Version 2.

```
9. NFSPROC_CREATE (where, attributes) returns (dir)
       diropargs where;
       sattr    attributes;
       dirupres  dir;
```

The file *where.name* is created in the directory given by *where.dir*. The initial attributes of the new file are given by *attributes*. A *reply.status* of *NFS_OK* indicates that the file was created, and *reply.file* and *reply.attributes* are its file handle and attributes. Any other *reply.status* means that the operation failed and no file was created.

Bugs: this routine should pass an exclusive create flag meaning: create the file only if it is not already there.

Remove File

Procedure 10, Version 2.

```
10. NFSPROC_REMOTE (which) returns (status)
       diropargs which;
       stat     status;
```

The file *which.name* is removed from the directory given by *which.dir*. A *status* of *NFS_OK* means the directory entry was removed.

Rename File

Procedure 11, Version 2.

```
11. NFSPROC_RENAME (from, to) returns (status)
        diropargs from;
        diropargs to;
        stat     status;
```

The existing file *from.name* in the directory given by *from.dir* is renamed to *to.name* in the directory given by *to.dir*. If *status* is *NFS_OK*, the file was renamed. The *RENAME* operation is atomic on the server; it cannot be interrupted in the middle.

Create Link to File

Procedure 12, Version 2.

```
12. NFSPROC_LINK (from, to) returns (status)
        fhandle  from;
        diropargs to;
        stat     status;
```

This procedure creates the file *to.name* in the directory given by *to.dir*, which is a hard link to the existing file given by *from*. If the return value of *status* is *NFS_OK*, a link was created. Any other return value indicates an error, and the link is not created.

A hard link should have the property that changes to either of the linked files are reflected in both files. When a hard link is made to a file, the attributes for the file should have a value for *nlink* which is one greater than the value before the link.

Create Symbolic Link

Procedure 13, Version 2.

```
13. NFSPROC_SYMLINK (from, to, attributes) returns (status)
        diropargs from;
        path     to;
        sattr    attributes;
        stat     status;
```

This procedure creates the file *from.name* with ftype *NFLNK* in the directory given by *from.dir*. The new file contains the pathname *to* and has initial attributes given by *attributes*. If the return value of *status* is *NFS_OK*, a link was created. Any other return value indicates an error, and the link is not created.

A symbolic link is a pointer to another file. *to* is not interpreted by the server, just stored in the newly created file. A *READLINK* operation returns the data to the client for interpretation.

Bugs: on servers the attributes are never used, since symbolic links always have mode 0777.

Create Directory

Procedure 14, Version 2.

```
14. NFSPROC_MKDIR (where, attributes) returns (reply)
        diropargs where;
        sattr   attributes;
        dirupres  reply;
```

The new directory *where.name* is created in the directory given by *where.dir*. The initial attributes of the new directory are given by *attributes*. A *reply.status* of *NFS_OK* indicates that the new directory was created, and *reply.file* and *reply.attributes* are its file handle and attributes. Any other *reply.status* means that the operation failed and no directory was created.

Remove Directory

Procedure 15, Version 2.

```
15. NFSPROC_RMDIR (which) returns (status)
        diropargs    which;
        stat         status;
```

The existing, empty directory *which.name* in the directory given by *which.dir* is removed. If *status* is *NFS_OK*, the directory was removed.

Read From Directory

Procedure 16, Version 2.

```
16. NFSPROC_READDIR (dir, cookie, count) returns (entries)
        fhandle       dir;
        opaque cookie[COOKIESIZE];
        unsigned count;
        union switch (stat status) {
           NFS_OK:
               typedef union switch (boolean valid) {
                  TRUE:
                      struct {
                         unsigned  fileid;
                         filename  name;
                         opaque    cookie[COOKIESIZE];
                         entry     nextentry;
                      }
                  FALSE:
                      struct {}
               } entry;
               boolean      eof;
           default:
        } entries;
```

This procedure returns a variable number of directory entries, with a total size of up to *count* bytes, from the directory given by *dir*. Each *entry* contains a *fileid* that is a unique number to identify the file within a filesystem, the *name* of the file, and a *cookie* that is an opaque pointer to

the next entry in the directory. The cookie is used in the next READDIR call to get more entries starting at a given point in the directory. The special cookie zero (all bits zero) can be used to get the entries starting at the beginning of the directory. The *fileid* field should be the same number as the *fileid* in the the attributes of the file (see the section entitled *fattr* under *Basic Data Types.*) The *eof* flag has a value of *TRUE* if no more entries are in the directory; *valid* is used to mark the end of the entries. If the returned value of *status* is *NFS_OK*, then it is followed by a variable number of *entries.*

Get Filesystem Attributes

Procedure 17, Version 2.

```
17. NFSPROC_STATFS (file) returns (reply)
        fhandle       file;
        union switch (stat status) {
           NFS_OK:
               struct {
                  unsigned  tsize;
                  unsigned  bsize;
                  unsigned  blocks;
                  unsigned  bfree;
                  unsigned  bavail;
               } fsattr;
           default:
               struct {}
        } reply;
```

If *reply.status* is *NFS_OK*, then *reply.fsattr* gives the attributes for the filesystem that contains *file.* The attribute fields contain the following values:

- The optimum transfer size of the server in bytes. This is the number of bytes the server would like to have in the data part of *READ* and *WRITE* requests.
- The block size in bytes of the filesystem.
- The total number of *bsize* blocks on the filesystem.
- The number of free *bsize* blocks on the filesystem.
- The number of *bsize* blocks available to non-privileged users.

Bugs: this call does not work well if a filesystem has variable-size blocks.

3

Mount Protocol Definition

Introduction

The mount protocol is separate from, but related to, the *nfs* protocol. It provides all of the operating system specific services to get the *nfs* off the ground—looking up path names, validating user identity, and checking access permissions. Clients use the mount protocol to get the first file handle, which allows them entry into a remote filesystem.

The mount protocol is kept separate from the *nfs* protocol to make it easy to plug in new access checking and validation methods without changing the *nfs* server protocol.

Notice that the protocol definition implies stateful servers because the server maintains a list of client's mount requests. The mount list information is not critical for the correct functioning of either the client or the server. It is intended for advisory use only, for example, to warn possible clients when a server is going down.

Version 1

Version one of the mount protocol communicates with the version two of the *nfs* protocol. The only connecting point is the structure, which is the same for both protocols.

rpc Information

- The mount service uses style authentication only.
- The mount service is currently supported on UDP/IP only.
- These are the *rpc* constants needed to call the MOUNT service. They are given in decimal.

```
PROGRAM      100005
VERSION      1
```

- Consult the server's portmapper, described in the *RPC Protocol Specification*, to find which port number the mount service is registered on.

Sizes

These are the sizes given in decimal bytes of various *xdr* structures used in the protocol.

- The maximum number of bytes in a pathname argument.
- The maximum number of bytes in a name argument.
- The size in bytes of the opaque file handle.

Basic Data Types

This section presents the data types used by the *nfs*.

fhandle

```
typedef opaque fhandle[FHSIZE];
```

The is the file handle that the server passes to the client. All file operations are done using file handles to refer to a file or directory. The file handle can contain whatever information the server needs to distinguish an individual file.

This is the same as the *xdr* definition in version 2 of the *nfs* protocol; see the section on *fhandle* under *Basic Data Types*.

fhstatus

```
typedef union switch (unsigned status) {
  0:
      fhandle      directory;
  default:
      struct {}
}
```

If a *status* of zero is returned, the call completed successfully, and a file handle for the directory follows. A non-zero status indicates some sort of error. In this case the status is a UNIX error number.

dirpath

```
typedef string dirpath<MNTPATHLEN>;
```

The type is a normal pathname of a directory.

name

```
typedef string name<MNTNAMLEN>;
```

The type is an arbitrary string used for various names.

Server Procedures

The following sections define the *rpc* procedures supplied by a mount server. The *rpc* procedure number and version are given in the header, along with the name of the procedure. The synopsis of procedures has this format:

```
<proc #>. <proc name> ( <arguments> ) returns ( <results> )
        <argument declarations>
        <results declarations>
```

In the first line, *proc name* is the name of the procedure, *arguments* is a list of the names of the arguments, and *results* is a list of the names of the results. The second and third lines give the *xdr argument declarations* and *results declarations*. Afterward, there is a description of what the procedure is expected to do, and how its arguments and results are used. Any bugs or problems with the procedure are listed at the end.

Do Nothing

Procedure 0, Version 1.

```
0. MNTPROC_NULL ( ) returns ( )
```

This procedure does no work. It is made available in all *rpc* services to allow server response testing and timing.

Add Mount Entry

Procedure 1, Version 1.

```
1. MNTPROC_MNT (directory) returns (reply)
        dirpath  dirname;
        fhstatus reply;
```

If *reply.status* is 0, *reply.directory* contains the file handle for the directory *dirname*. This file handle may be used in the *nfs* protocol. This procedure also adds a new entry to the mount list for this client mounting *dirname*.

Return Mount Entries

Procedure 2, Version 1.

```
2. MNTPROC_DUMP ( ) returns (mountlist)
        union switch (boolean more_entries) {
           TRUE:
               struct {
                      name     hostname;
                      dirpath        directory;
                      mountlist nextentry;
               }
           FALSE:
               struct {}
        } mountlist;
```

Returns the list of remote mounted filesystems. The *mountlist* contains one entry for each *hostname* and *directory* pair.

Remove Mount Entry

Procedure 3, Version 1.

```
3. MNTPROC_UMNT (directory) returns ( )
       dirpath      directory;
```

Removes the mount list entry for *directory.*

Remove All Mount Entries

Procedure 4, Version 1.

```
4. MNTPROC_UMNTALL ( ) returns ( )
```

Removes all of the mount list entries for this client.

Return Export List

Procedure 5, Version 1.

```
5. MNTPROC_EXPORT ( ) returns (exportlist)
       union switch (boolean more_entries) {
          TRUE:
              struct {
                 dirpath   filesys;
                 typedef union switch (boolean more_groups) {
                    TRUE:
                       struct {
                          name   grname;
                          groups nextgroup;
                       }
                    FALSE:
                       struct {}
                 } groups;
                 mountlist nextentry;
              }
          FALSE:
              struct {}
       } exportlist;
```

Returns in *exportlist* a variable number of export list entries. Each entry contains a filesystem name and a list of groups that are allowed to import it. The filesystem name is in *exportlist.filesys,* and the group name is in *exportlist.groups.grname.*

Bugs: the exportlist should contain more information about the status of the filesystem, such as a read-only flag.

Index

S

T

U

V

W

X

CONVEX Yellow Pages Protocol Specification

November 1, 1987

CONVEX Computer Corporation

Table of Contents

1

Introduction and Terminology

Introduction

The Yellow Pages (*yp*) is a network lookup service providing read access to a replicated database. The lookup service is provided by a set of *yp* database servers. The client interface to this service uses the Remote Procedure Call (*rpc*) mechanism.

Translating or mapping a name to its value is one of the most common operations performed in computer systems. Common examples are the translation of a variable name to a virtual memory address, the translation of a user name to a system ID or list of capabilities, and the translation of a network node name to an Internet address. There are two fundamental read-only operations that can be performed on a map: match and enumerate. Match means to look up a name (which we call a **key**) and return its current value. Enumerate means to return each key-value pair in turn.

The *yp* supplies match and enumerate operations in a network environment, where high availability and reliability are required. It provides that availability and reliability by replicating both databases and database servers on multiple nodes within a single local net, and within the Internet. The database is replicated: all changes are made at a single server and eventually propagate to the remaining servers without locking. The *yp* is appropriate for an environment in which changes to the mapping databases occur on the order of tens per day.

The *yp* operates on an arbitrary number of map databases. Map names provide the lower of two levels of a naming hierarchy. Maps are themselves grouped into named sets, called **domains**. Domain names provides a second, higher level of naming. Map names must be unique within a domain, but may be duplicated in different domains. The *yp* client interface requires that both a map name and a domain name be supplied to perform match and enumeration operations.

The *yp* achieves high availability by replication. One area not addressed by the protocol which has to be addressed by the implementors is global consistency among the replicated copies of the database. Every implementation should be designed so that at steady state a request yields the same result when it is made of any *yp* database server. Update and update-propagation mechanisms must be implemented to supply the required degree of consistency.

rpc—Remote Procedure Call

The Remote Procedure Call (*rpc*) mechanism defines a paradigm for interprocess communication modeled on function calls. Clients call functions that optionally return values. All inputs and outputs to the functions are in the client's address space. The function is executed by a server program.

Using *rpc*, clients address servers by a program number (this identifies the application level protocol that the server speaks) and a version number. Additionally, each server procedure has a procedure number assigned to it.

In an Internet environment, clients must also know the server's host Internet address, and the server's rendezvous port. The server listens for service requests at ports associated with a particular transport protocol—TCP/IP or UDP/IP.

The format of the data structures used as inputs to and outputs from the remotely executed procedures are typically defined by header files that are included when the client interface functions are compiled. Levels above the client interface package need not know any particulars of the *rpc* interface to the server.

xdr—External Data Representation

The External Data Representation (*xdr*) specification establishes standard representations for basic data types (such as strings, signed and unsigned integers, and structures and unions) in a way that allows them to be transferred among machines with varying architectures. *xdr* provides primitives to encode (that is, translate from the local host's representation to the standard representation) and decode (translate from the standard representation to the local host's representation) basic data types. Constructor primitives allow arbitrarily complex data types to be made from the basic types.

The *yp*'s *rpc* input and output data structures are described using *xdr*'s data definition language. In general, the data definition language looks like the C language, with a few extra constructs. One such extra construct is the *discriminated union*. This is like a C language union, in that it can hold various objects, but differs from it in that a discriminant indicates which object it currently holds. The discriminant is the first thing across the wire. Consider a simple example:

```
union switch (long int) {
        1:
                string exmpl_name<16>
        0:
                unsigned int exmpl_error_code
        default:
                struct {}
}
```

The example should be interpreted as follows: the first object to be encoded/decoded (that is, the discriminant) is a long integer. If it has the value one, the next object is a string. If the discriminant has the value zero, the next object is an unsigned integer. If the discriminant takes any other value, don't encode or decode any more data.

A *string* data type in the *xdr* data definition language adds the ability to specify the maximum number of elements in an byte array or string of potentially variable size. For instance:

```
string domain<YPMAXDOMAIN>;
```

states that the byte sequence *domain* may be less than or equal to *YPMAXDOMAIN* bytes long.

An additional primitive data type is a *boolean*, which takes the value one to mean *TRUE* and zero to mean *FALSE*.

2

yp Database Servers

Maps and Map Operations

Map Structure

Maps are named sets of key-value pairs. Keys and their values are counted binary objects, and may be ASCII information, but need not be. The data comprising a map is determined by the client applications that are the final customers for the data, not by the *yp*. *yp* has no syntactic nor semantic knowledge of the map contents. Neither does the *yp* determine or know any map's name. Map names are managed by the *yp*'s clients. Conflict in the map namespace must be resolved by human administrators outside the *yp* system.

Typical implementations for *yp* maps are files or database management systems. The design of the *yp*'s map database is an implementation detail, and is unspecified by the protocol.

Match Operation

yp supports an exact match operation in the *YPPROC_MATCH* procedure. That is, if a match string and some key in the map are exactly the same, the value of the key is returned. No pattern matching, case conversion, or wildcarding is supported.

Map Entry Enumeration

It is possible to get the first key-value pair in a map with *YPPROC_FIRST*, and the next key-value pair with *YPPROC_NEXT*. Calling "get first" once and "get next" until the return value indicates there are no more entries in the map retrieves each entry once. Making the same calls on the same map at the same *yp* database server enumerates all entries in the same order. The actual order, however, is unspecified. Enumerating a map at a different *yp* database server does not necessarily return entries in the same order.

Entire Map Retrieval

The *YPPROC_ALL* operation retrieves all key-value pairs in a map, with a single *rpc* request. This is faster than map entry enumeration, and more reliable, since it uses TCP. Ordering is the same as when enumeration is applied.

Map Update

The update of *yp* maps is an implementation detail which is outside the specification of the *yp* service.

Master and Slave *yp* Database Servers

For each map, there is one *yp* database server, called the map's *master*. Map updates take place only on the master. An updated map should be transferred from the master to the rest of the *yp* database servers, which are *slave* servers for this map.

It is possible for each map to have a different *yp* database server as its master, or for all maps to have the same master, or any other combination. The choice of how to set up map masters is one of implementation and administrative policy.

Map Propagation and Consistency

Getting map updates from the master to the slaves is called map propagation. Neither technology nor algorithms for map propagation are specified by the protocol. Map propagation may be entirely manual: for instance, a person could copy the maps from the master to the slaves at a regular interval, or when a change is made on the master. This is unnecessarily labor intensive.

In order to escape from the idiosyncrasies of any particular implementation, all maps should be uniformly timestamped.

Functions to Aid in Map Propagation

The way a map is transferred from one server to another is not specified by the *yp* protocol. One possibility would be for the system administrator to do it manually. Another would be for the *yp* database server to activate another process to perform the map transfer. A third would be for a server to enumerate a recent version of the map, using the normal client map enumeration functions.

The *YPPROC_XFR* procedure requests the *yp* server to update a map, and permits the actual transfer agent (some server process) to call back the requester with a summary status.

Domains

Domains provide a second level for naming within the *yp* subsystem. They are names for sets of maps; they therefore create separate map name spaces. Domains provide an opportunity to break large organizations up into administerable chunks, and the ability to create parallel, non-interfering test and production environments.

Ideally, the domain of interest to a client ought to be associated with the invoking user, but in practice it is useful for client machines to be in a default domain. Implementations of the *yp* client interface should supply some mechanism for telling processes the domain name they should use. This is needed not only because the concept of domain is a useless one as far as most programs are concerned, but, more importantly, so that programs can be written that are insensitive to both location and the invoking user.

Restrictions

The following capabilities are not included in the current *yp* protocols.

Map Update Within the *yp*

All write (and delete) access to the *yp*'s map database is assumed to be outside of the *yp* subsystem. It is probable that write access to the map database will be included in later versions of the *yp* protocols.

Version Commitment Across Multiple Requests

The *yp* protocol was designed to keep the *yp* database server stateless with regard to its clients. Therefore, there is no facility for contracting with a server to preallocate any resource beyond that required to service any single request. In particular, there is no way to get a server to commit to use a single version of a map while trying to enumerate that map's entries. Use *YPPROC_ALL* to avoid these problems.

Guaranteed Global Consistency

There is no facility for locking maps during the update or propagation phases, therefore it is virtually guaranteed that the map database be globally inconsistent during those phases. The set of client applications for which the *yp* is an appropriate lookup service is one that (by definition) must be tolerant of transient inconsistencies.

Access Control

The *yp* database servers make no attempt to restrict access to the map data by any means. All syntactically correct requests are serviced.

yp Database Server Protocol Definition

This section describes version 2 of the protocol. It is likely that changes will be made to successive versions.

rpc Constants

The following numbers are in decimal:

- *yp* database server protocol program number.
- Current *yp* protocol version.

Other Manifest Constants

The following numbers are in decimal:

- Total maximum size of key and value for any pair. The absolute sizes of the key and value may divide this maximum arbitrarily.
- Maximum number of characters in a domain name.
- Maximum number of characters in a map name.
- Maximum number of characters in a *yp* host name.

Remote Procedure Return Values

This section presents the return status values returned by several of the *yp* remote procedures. All numbers are in decimal.

ypstat

```
typedef enum {
        YP_TRUE    = 1,  /* General purpose success code. */
        YP_NOMORE  = 2,  /* No more entries in map. */
        YP_FALSE   = 0,  /* General purpose failure code.*/
        YP_NOMAP   = -1,  /* No such map in domain. */
        YP_NODOM   = -2,  /* Domain not supported. */
        YP_NOKEY   = -3,  /* No such key in map. */
        YP_BADOP   = -4,  /* Invalid operation. */
        YP_BADDB   = -5,  /* Server database is bad. */
        YP_YPERR   = -6,  /* YP server error. */
        YP_BADARGS = -7,  /* Request arguments bad. */
        YP_VERS    = -8        /* YP server version mismatch. */
} ypstat
```

ypxfrstat

```
typedef enum {
        YPXFR_SUCC   =  1,  /* Success */
        YPXFR_AGE    =  2,  /* Master's version not newer */
        YPXFR_NOMAP  = -1,  /* Can't find server for map */
        YPXFR_NODOM  = -2,  /* Domain not supported */
        YPXFR_RSRC   = -3,  /* Local resource alloc failure */
        YPXFR_RPC    = -4,  /* RPC failure talking to server */
        YPXFR_MADDR  = -5,  /* Can't get master address */
        YPXFR_YPERR  = -6,  /* YP server/map db error */
        YPXFR_BADARGS= -7,  /* Request arguments bad */
        YPXFR_DBM    = -8,  /* Local database failure */
        YPXFR_FILE   = -9,  /* Local file I/O failure */
        YPXFR_SKEW   = -10,  /* Map version skew in transfer */
        YPXFR_CLEAR  = -11,  /* Can't clear local ypserv */
        YPXFR_FORCE  = -12,  /* Must override defaults */
        YPXFR_XFRERR = -13,  /* ypxfr error */
        YPXFR_REFUSED= -14   /* ypserv refused transfer */
} ypxfrstat
```

Basic Data Structures

This section defines the data structures used as inputs to and outputs from the *yp* remote procedures.

domainname

```
typedef string domainname<YPMAXDOMAIN>
```

mapname

```
typedef string mapname<YPMAXMAP>
```

peername

```
typedef string peername<YPMAXPEER>
```

keydat

```
typedef string keydat<YPMAXRECORD>
```

valdat

```
typedef string valdat<YPMAXRECORD>
```

ypmap_parms

```
typedef struct {
        domainname
        mapname
        unsigned long ordernum
        peername
} ypmap_parms
```

This contains parameters giving information about map *mapname* within domain *domainname*; *peername* is the name of the map's master *yp* database server. If any of the three strings is null, it indicates information is unknown or unavailable. The *ordernum* element contains a binary value representing the value of the map's order number; if unavailable, this is 0.

ypreq_xfr

```
typedef struct {
        struct ypmap_parms map_parms
        unsigned long transid
        unsigned long prog
        unsigned short port
} ypreq_xfr
```

ypresp_val

```
typedef struct {
        ypstat
        valdat
} ypresp_val
```

ypresp_key_val

```
typedef struct {
        ypstat
        keydat
        valdat
} ypresp_key_val
```

ypresp_master

```
typedef struct {
        ypstat
        peername
} ypresp_master
```

ypresp_order

```
typedef struct {
        ypstat
        unsigned long ordernum
} ypresp_order
```

ypresp_all

```
typedef union switch (boolean more) {
        TRUE:
                ypresp_key_val
        FALSE:
                struct { }
} ypresp_all
```

ypresp_xfr

```
typedef struct {
        unsigned long transid
        ypxfrstat xfrstat
} ypresp_xfr
```

ypmaplist

```
typedef struct {
        mapname
        ypmaplist *
} ypmaplist
```

ypresp_maplist

```
typedef struct {
        ypstat
        ypmaplist *
} ypresp_maplist
```

yp Database Server Remote Procedures

This section contains a specification for each function that can be called as a remote procedure. The input and output parameters are described using the *xdr* data definition language.

Do Nothing

Procedure 0, Version 2.

```
0. YPPROC_NULL ( ) returns ( )
```

This takes no arguments, does no work, and returns nothing. It is made available in all *rpc* services to allow server response testing and timing.

Do You Serve This Domain?

Procedure 1, Version 2.

```
1. YPPROC_DOMAIN (domain) returns (serves)
        domainname domain;
        boolean serves;
```

This returns *TRUE* if the server serves *domain*, and *FALSE* otherwise. This procedure allows a potential client to determine if a given server supports a certain domain.

Answer Only If You Serve This Domain

Procedure 2, Version 2.

```
2. YPPROC_DOMAIN_NONACK (domain) returns (serves)
        domainname domain;
        boolean serves;
```

This procedure returns TRUE if the server serves *domain*; otherwise it does not return. The intent of the function is that it be called in a broadcast environment, in which it is useful to restrict the number of useless messages. If this function is called, the client interface implementation must be written so as to regain control in the negative case, for instance by means of a time-out on the response.

The current implementation currently does *return* in the FALSE case by forcing an *rpc* decode error.

Return Value of a Key

Procedure 3, Version 2.

```
3. YPPROC_MATCH (req) returns (resp)
        ypreq_key req;
        ypresp_val resp;
```

This returns the value associated with the datum *keydat* in *req*. If the *status* element in *resp* has the value *YP_TRUE*, the value data are returned in the datum *valdat*.

Get First Key-Value Pair in Map

Procedure 4, Version 2.

```
4. YPPROC_FIRST (req) returns (resp)
        ypreq_key req;
        ypresp_key_val resp;
```

If *status* has the value *YP_TRUE*, this returns the first key-value pair from the map named in *req* to the *keydat* and *valdat* elements within *resp*. When *status* contains the value *YP_NOMORE*, the map is empty.

Get Next Key-Value Pair in Map

Procedure 5, Version 2.

```
5. YPPROC_NEXT (req) returns (resp)
        ypreq_key req;
        ypresp_key_val resp;
```

If *status* has the value *YP_TRUE*, this returns the key-value pair following the key-value named *req* to the *keydat* and *valdat* elements within *resp*. If the passed key is the last key in the map, the value of *status* is *YP_NOMORE*.

Transfer Map

Procedure 6, Version 2.

```
6. YPPROC_XFR (req) returns (resp)
        ypreq_xfr req;
        ypresp_xfr resp;
```

The action taken in response to this request is unspecified, and is implementation dependent. The intention is to indicate to the server that a map should be updated, and to allow the actual transfer agent (whether it be the *yp* server process, or some other process) to call back the requester with a summary status.

The transfer agent should call back the program running on the requesting host with program number *req.prog*, program version 1, and listening at port *req.port*. The procedure number is 1, and the callback data is of type *ypresp_xfr*. The *transid* field should turn around *req.transid*, and the *xfrstat* field should be set appropriately.

Reinitialize Internal State

Procedure 7, Version 2.

```
7. YPPROC_CLEAR ( ) returns ( )
```

The action taken in response to this request is unspecified, and is implementation dependent. Different server implementations may have different amounts of internal state (open files, or the current map, for example). This request signals that all such state should be expunged.

Get All Key-Value Pairs in Map

Procedure 8, Version 2.

```
8. YPPROC_ALL (req) returns (resp)
        ypreq_nokey req;
        ypresp_all resp;
```

This allows all key-value pairs from a map to be transferred with a single *rpc* request. When the union's discriminant is FALSE, no more key-value pairs are returned. The status field of the last *rpresp_key_val* structure should be consulted to determine why the flow of returned key-value pairs has stopped.

Get Map Master Name

Procedure 9, Version 2.

```
9. YPPROC_MASTER (req) returns (resp)
        ypreq_nokey req;
        ypresp_master resp;
```

This returns the map's master *yp* server inside the *resp* structure.

Get Map Order Number

Procedure 10, Version 2.

```
10. YPPROC_ORDER (req) returns (resp)
        ypreq_nokey req;
        ypresp_order resp;
```

This returns a map's order number as an unsigned long integer, which indicates when the map was built. This quantity represents the number of seconds since the midnight before 1 January 1970 Greewich Mean Time.

Get All Maps in Domain

Procedure 11, Version 2.

```
11. YPPROC_MAPLIST (req) returns (resp)
        domainname req;
        ypresp_maplist resp;
```

This returns a list of all the maps in a domain.

3

yp Binders

Introduction

In order that any network service be usable, there must be some way for potential clients to find the servers. This chapter describes the *yp* binder, an optional element in the *yp* subsystem that supplies *yp* database server addressing information to potential *yp* clients.

In order to address a *yp* server in the Internet environment, a client must know the server's Internet address, and the port at which the server is listening for service requests. No contract is negotiated between a *yp* server and a potential client; therefore the addressing information is sufficient to bind the client to the server.

Of the many possible ways for a client to get the addressing information, one alternative is to supply an entity to cache the bindings, and to serve that binding database to potential *yp* clients. The theory is that if finding the service takes a lot of work, allocate a specialist to do it, rather than burden every client with a job that is irrelevant to its real function. A *yp* binder only makes sense if it is easier for a client to find the *yp* binder than to find a *yp* database server, and if the *yp* binder can itself find a *yp* database server.

We make the assumption that a *yp* binder is present at every network node, and because of this, addressing the *yp* binder is easier than addressing a *yp* database server. The scheme for finding a local resource is implementation-specific, but given that a resource is guaranteed to be local, there may be some efficient way of finding it. We further assume that the *yp* binder can find a *yp* database server somehow, but that the way is either complicated, time-consuming, or resource-consuming. If either of these assumptions is untrue, then probably your implementation is not a good bet for a *yp* binder.

If a *yp* binder is implemented, it can provide added value beyond the binding: it can verify that the binding is correct and that the *yp* database server is alive and well, for instance. The degree of sureness in a binding that the *yp* binder gives to a client is a parameter that can be tuned appropriately in the implementation.

yp Binder Protocol Definition

This section describes version 2 of the protocol. It is likely that changes will be made to successive versions..

rpc Constants

All numbers are decimal.

- The *yp* binder protocol program number.
- The current *yp* binder protocol version.

Other Manifest Constants

All numbers are decimal.

- The maximum number of characters in a domain name. This is identical to the constant defined above within the *yp* database server protocol section.

ypbind_resptype

```
enum ypbind_resptype {
        YPBIND_SUCC_VAL = 1,
        YPBIND_FAIL_VAL = 2
}
```

This discriminates between success responses and failure responses to a *YPBINDPROC_DOMAIN* request.

ypbinderr

```
typedef enum {
        YPBIND_ERR_ERR 1    /* Internal error */
        YPBIND_ERR_NOSERV 2/* No bound server for domain */
        YPBIND_ERR_RESC 3   /* Can't allocate system resource */
} ypbinderr
```

The error case of most interest to a *yp* binder client is *YPBIND_ERR_NOSERV*; it means that the binding request cannot be satisfied because the *yp* binder doesn't know how to address any *yp* database server in the named domain.

Basic Data Structures

This section defines the data structures used as inputs to and outputs from the *yp* binder remote procedures.

domainname

```
typedef string domainname<YPMAXDOMAIN>
```

This is identical to the domainname string defined above within the *yp* database server protocol section.

ypbind_binding

```
typedef struct {
        unsigned long ypbind_binding_addr
        unsigned short ypbind_binding_port
} ypbind_binding
```

This contains the information necessary to bind a client to a *yp* database server in the Internet

environment: *ypbind_binding_addr* holds the host IP address (4 bytes), and *ypbind_binding_port* holds the port address (2 bytes). Both IP address and port address must be in ARPA network byte order (most significant byte first, or big endian), regardless of the host machine's native architecture.

ypbind_resp

```
typedef struct {
        union switch (enum ypbind_resptype status) {
                YPBIND_SUCC_VAL:
                        ypbind_binding
                YPBIND_FAIL_VAL:
                        ypbinderr
                default:
                        { }
        }
} ypbind_resp
```

This is the response to a *YPBINDPROC_DOMAIN* request.

ypbind_setdom

```
typedef struct {
        domainname
        ypbind_binding
        version
} ypbind_setdom
```

This is the input data structure for the *YPBINDPROC_SETDOM* procedure.

yp Binder Remote Procedures

Like the *yp* procedures earlier, these procedures are described using the *xdr* data definition language.

Do Nothing

Procedure 0, Version 2.

```
0. YPBINDPROC_NULL ( ) returns ( )
```

This does no work. It is made available in all *rpc* services to allow server response testing and timing.

Get Current Binding for a Domain

Procedure 1, Version 2.

```
1. YPBINDPROC_DOMAIN (domain) returns (resp)
        domainname domain;
        ypbind_resp resp;
```

This returns the binding information necessary to address a *yp*database server within the Internet environment.

Set Domain Binding

Procedure 2, Version 2.

```
2. YPBINDPROC_SETDOM (setdom) returns ( )
        ypbind_setdom setdom;
```

This instructs a *yp* binder to use the passed information as its current binding information for the passed domain.

Index

X

Y

CONVEX Remote Procedure Call Protocol Specification

April 1988

CONVEX Computer Corporation
Richardson, Texas

CONVEX Remote Procedure Call
Protocol Specification

Printed in the United States of America

Table of Contents

Appendices

List of Tables

List of Figures

1

Introduction

This document specifies a message protocol used in implementing the Remote Procedure Call (*rpc*) package. The message protocol is specified with the eXternal Data Representation (*xdr*) language.

This document assumes that the reader is familiar with both *rpc* and *xdr*. It does not attempt to justify *rpc* or its uses. Also, the casual user of *rpc* does not need to be familiar with the information in this document.

Terminology

The document discusses servers, services, programs, procedures, clients, and versions. A server is a machine where some 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.

For example, a network file service may be composed of two programs. One program may deal with high-level applications such as file system access control and locking. The other may deal with low-level file I/O, and have procedures like "read" and "write." A client machine of the network file service would call the procedures associated with the two programs of the service on behalf of some user on the client machine.

rpc Model

The remote procedure call model is similar to the local procedure call model. In the local case, the caller places arguments to a procedure in some well-specified location (such as a result register). It then transfers control to the procedure, and eventually gains back control. At that point, the results of the procedure are extracted from the well-specified location, and the caller continues execution.

The remote procedure call is similar, except that one thread of control winds through two processes—one is the caller's process, the other is a server's process. That is, the caller process sends a call message to the server process and waits (blocks) for a reply message. The call message contains the procedure's parameters, among other things. The reply message contains the procedure's results, among other things. Once the reply message is received, the results of the procedure are extracted, and caller's execution is resumed.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message. Note that in this model, only one of the two processes is active at any given time. That is, the *rpc* protocol does not explicitly support multi-threading of caller or server processes.

Transports and Semantics

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 only deals with the specification and interpretation of messages.

Because of transport independence, the *rpc* protocol does not attach specific semantics to the remote procedures or their execution. Some semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, *rpc* message passing using UDP/IP is unreliable. Thus, if the caller retransmits call messages after short time-outs, the only thing he can infer from no reply message is that the remote procedure was executed zero or more times (and from a reply message, one or more times). On the other hand, *rpc* message passing using TCP/IP is reliable. No reply message means that the remote procedure was executed at most once, whereas a reply message means that the remote procedure was exactly once.

Binding and Rendezvous Independence

The act of binding a client to a service is *not* part of the remote procedure call specification. This important and necessary function is left up to some higher-level software.

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 this task.

Message Authentication

The *rpc* protocol provides the fields necessary for a client to identify himself to a service and vice versa. Security and access control mechanisms can be built on top of the message authentication.

2

rpc Protocol Requirements

Introduction

The *rpc* protocol must provide for the following:

- Unique specification of a procedure to be called.
- Provisions for matching response messages to request messages.
- Provisions for authenticating the caller to service and vice versa.

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 misspecification of a procedure's parameters).
- Reasons why remote authentication failed.
- Any other reasons why the desired procedure was not called.

Remote Programs and Procedures

The *rpc* call message has three unsigned fields: remote program number, remote program version number, and remote procedure number. The three fields uniquely identify the procedure to be called. Program numbers are administered by some central authority (like Sun). Once an implementor has a program number, he can implement his remote program; the first implementation would most likely have the version number of 1. 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 speaking old and new protocols through the same server process possible.

The procedure number identifies the procedure to be called. These numbers are documented in the 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).

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 is caused by a disagreement about the protocol between client and service.)

Authentication

Provisions for authentication of caller to service and vice versa are provided as part of the *rpc* protocol. The call message has two authentication fields, the credentials and verifier. The reply message has one authentication field, the response verifier. The *rpc* protocol specification defines all three fields to be the following opaque type:

```
enum auth_flavor {
        AUTH_NULL       = 0,
        AUTH_UNIX       = 1,
        AUTH_SHORT      = 2
        /* and more to be defined */
};

struct opaque_auth {
        union switch (enum auth_flavor) {
                default: string auth_body<400>;
        };
};
```

Any *opaque_auth* structure is an *auth_flavor* enumeration followed by a counted string, whose bytes are opaque to the *rpc* protocol implementation. See Chapter 5 of the *XDR Protocol Specification* for more information about opaque data.

The interpretation and semantics of the data contained within the authentication fields is specified by individual, independent authentication protocol specifications. Appendix A defines three authentication protocols.

If authentication parameters were rejected, the response message contains information stating why they were rejected.

Program Number Assignment

Program numbers are given out in groups of 0x20000000 (536870912) according to the following chart:

```
        0 - 1fffffff defined by Sun
20000000 - 3fffffff defined by user
40000000 - 5fffffff transient
60000000 - 7fffffff reserved
80000000 - 9fffffff reserved
a0000000 - bfffffff reserved
c0000000 - dfffffff reserved
e0000000 - ffffffff reserved
```

The first group is a range of numbers administered by Sun Microsystems, and should be identical for all *nfs* customers. The second range is for applications peculiar to a particular customer. This range is intended primarily for debugging new programs. When a customer develops an application that might be of general interest, that application should be given an assigned number in the first range. The third group is for applications that generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

Other Uses of the *rpc* Protocol

The intended use of this protocol is for calling remote procedures. That is, each call message is matched with a response message. The protocol itself, however, is a message-passing protocol with which other (non-*rpc*) protocols can be implemented. The *rpc* message protocol may also be used for batching and broadcast. These two protocols are discussed but not defined below.

Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server; batching uses reliable byte stream protocols (like TCP/IP) for their transport. In the case of batching, the client never waits for a reply from the server and the server does not send replies to batch requests. A sequence of batch calls is usually terminated by a legitimate *rpc* in order to flush the pipeline (with positive acknowledgment).

Broadcast *rpc*

In broadcast *rpc* based protocols, the client sends an a broadcast packet to the network and waits for numerous replies. Broadcast *rpc* uses unreliable, packet based protocols (like UDP/IP) as their transports. Servers that support broadcast protocols only respond when the request is successfully processed, and are silent in the face of errors.

3

rpc Message Protocol

Protocol Definition

This chapter defines the *rpc* message protocol in the *xdr* data definition language. The message is defined in a top-down style. Note: This is an *xdr* specification, not C code.

Figure 3–1: *rpc* Message Protocol Definition

```
enum msg_type {
        CALL = 0,
        REPLY = 1
};

/*
 * A reply to a call message can take on two forms:
 * the message was either accepted or rejected.
 */
enum reply_stat {
        MSG_ACCEPTED = 0,
        MSG_DENIED = 1
};

/*
 * Given that a call message was accepted, the following is
 * the status of an attempt to call a remote procedure.
 */
enum accept_stat {
        SUCCESS = 0,    /* RPC executed successfully */
        PROG_UNAVAIL = 1, /* remote hasn't exported program */
        PROG_MISMATCH= 2, /* remote can't support version # */
        PROC_UNAVAIL = 3, /* program can't support procedure */
        GARBAGE_ARGS = 4  /* procedure can't decode params */
};

/*
 * Reasons why a call message was rejected:
 */
enum reject_stat {
        RPC_MISMATCH = 0, /* RPC version number != 2 */
        AUTH_ERROR = 1    /* remote can't authenticate caller */
};

/*
 * Why authentication failed:
 */
enum auth_stat {
        AUTH_BADCRED = 1,    /* bad credentials (seal broken) */
```

```
        AUTH_REJECTEDCRED=2, /* client must begin new session */
        AUTH_BADVERF = 3,    /* bad 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; the service side cannot
 * treat this id as any type of sequence number.
 */
struct rpc_msg {
        unsigned        xid;
        union switch (enum msg_type) {
                CALL:  struct call_body;
                REPLY: struct reply_body;
        };
};

/*
 * Body of an RPC request call:
 * In version 2 of the RPC protocol specification, rpcvers
 * must be equal to 2.  The fields prog, vers, and proc
 * specify the remote program, its version number, and the
 * procedure within the remote program to be called.  After
 * these fields are two  authentication parameters: cred
 * (authentication credentials) and verf (authentication
 * verifier).  The two  authentication parameters are
 * followed by the parameters to the remote procedure,
 * which are specified by the specific program protocol.
 */
struct call_body {
        unsigned rpcvers;   /* must be equal to two (2) */
        unsigned prog;
        unsigned vers;
        unsigned proc;
        struct opaque_auth cred;
        struct opaque_auth verf;
        /* procedure specific parameters start here */
};

/*
 * Body of a reply to an RPC request.
 * The call message was either accepted or rejected.
 */
struct reply_body {
        union switch (enum reply_stat) {
                MSG_ACCEPTED:       struct accepted_reply;
```

```
                MSG_DENIED:  struct rejected_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 that the server generates in order to validate
 * itself to the caller.  It is followed by a union whose
 * discriminant is an enum accept_stat.  The SUCCESS arm
 * of the union is protocol specific.  The PROG_UNAVAIL,
 * PROC_UNAVAIL, and GARBAGE_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 {
        struct opaque_auth verf;
        union switch (enum accept_stat) {
                SUCCESS: struct {
                        /*
                         * procedure-specific results start here
                         */
                };
                PROG_MISMATCH: struct {
                        unsigned low;
                        unsigned  high;
                };
                default: struct {
                        /*
                         * void. Cases include PROG_UNAVAIL,
                         * PROC_UNAVAIL, and GARBAGE_ARGS.
                         */
                };
        };
};

/*
 * 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 refuses
 * 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, failure status is returned.
 */
struct rejected_reply {
        union switch (enum reject_stat) {
                RPC_MISMATCH: struct {
                        unsigned low;
                        unsigned high;
                };
```

```
            AUTH_ERROR: enum auth_stat;
        };
};
```

Authentication Parameter Specification

As previously stated, authentication parameters are opaque, but open-ended to the rest of the *rpc* protocol. This section defines some of the "flavors" of authentication that have been implemented.

Null Authentication

Often calls must be made where the caller does not know who he is and the server does not care who the caller is. 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 *auth_body* string are undefined. It is recommended that the string length be zero.

UNIX Authentication

The caller of a remote procedure may wish to identify himself as he is identified on a UNIX system. The value of the *credential's* discriminant of an *rpc* call message is *AUTH_UNIX*(1). The bytes of the *credential's* string encode the the following (*xdr*) structure:

```
struct auth_unix {
        unsigned    stamp;
        string      machinename<255>;
        unsigned    uid;
        unsigned    gid;
        unsigned    gids<10>;
};
```

The *stamp* is an arbitrary ID that the caller machine may generate. The *machinename* is the name of the caller's machine (like "krypton"). The *uid* is the caller's effective user ID. The *gid* is the caller's effective group id. The *gids* is a counted array of groups that contain the caller as a member. The *verifier* accompanying the credentials should be of *AUTH_NULL* (defined above).

The value of the discriminate of the *response verifier* received in the reply message from the server may be *AUTH_NULL* or *AUTH_SHORT*. In the case of *AUTH_SHORT*, the bytes of the *response verifier*'s string encode an *auth_opaque* structure. This new *auth_opaque* structure may now be passed to the server instead of the original *AUTH_UNIX* flavor credentials. The server keeps a cache which maps shorthand *auth_opaque* structures (passed back by way of a *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 *auth_opaque* structure at any time. If this happens, the remote procedure call message is rejected due to an authentication error. The reason for the failure is *AUTH_REJECTEDCRED*. At this point, the caller may wish to try the original *AUTH_UNIX* style of credentials.

Record-Marking Standard

When *rpc* messages are passed on top of a byte stream protocol (like 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). One *rpc* message fits into one RM record.

A record is composed of one or more record fragments. A record fragment is a four-byte header followed by *0* to $2^{31}-1$ bytes of fragment data. The bytes encode an unsigned binary number; as with *xdr* integers, the byte order is from highest to lowest. The number encodes two values — a Boolean that 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 that is the length in bytes of the fragment's data. The Boolean value is the highest-order bit of the header; the length is the 31 low-order bits. (Note that this record specification is *not* in *xdr* standard form!)

A

Port Mapper Program Protocol

Introduction

The port mapper program maps *rpc* program and version numbers to UDP/IP or TCP/IP port numbers. This program makes dynamic binding of remote programs possible.

This 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.

rpc Protocol

The protocol is specified by the *xdr* description language.

```
Port Mapper RPC Program Number: 100000
        Version Number: 1
        Supported Transports:
                UDP/IP on port 111
                TCP/IP on port 111
```

Transport Protocol Numbers

```
#define IPPROTO_TCP 6      /* protocol number for TCP/IP */
#define IPPROTO_UDP 17     /* protocol number for UDP/IP */
```

rpc Procedures

Here is a list of *rpc* procedures:

Do Nothing

Procedure 0, Version 2.

```
0. PMAPPROC_NULL () returns ()
```

This procedure does no work. By convention, procedure zero of any protocol takes no parameters and returns no results.

Set a Mapping

Procedure 1, Version 2.

```
1. PMAPPROC_SET (prog,vers,prot,port) returns (resp)
        unsigned prog;
        unsigned vers;
        unsigned prot;
        unsigned port;
        boolean resp;
```

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 version number *vers,* transport protocol number *prot,* and the port *port* on which it awaits service request. The procedure returns *resp,* 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 set *[prog,vers,prot].*

Unset a Mapping

Procedure 2, Version 2.

```
2. PMAPPROC_UNSET (prog,vers,dummy1,dummy2) returns (resp)
        unsigned prog;
        unsigned vers;
        unsigned dummy1;  /* value always ignored */
        unsigned dummy2;  /* value always ignored */
        boolean resp;
```

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.*

Look Up a Mapping

Procedure 3, Version 2.

```
3. PMAPPROC_GETPORT (prog,vers,prot,dummy) returns (port)
        unsigned prog;
        unsigned vers;
        unsigned prot;
        unsigned dummy;      /* this value always ignored */
        unsigned port;       /* zero means program not registered */
```

Given a program number *prog,* version number *vers,* and transport protocol number *prot,* this procedure returns the port number on which the program is awaiting call requests. A port value of zeros means the program has not been registered.

Dumping the Mappings

Procedure 4, Version 2.

```
4. PMAPPROC_DUMP () returns (maplist)
        struct maplist {
                union switch (boolean) {
                        FALSE: struct { /* void, end of list */ };
                        TRUE: struct {
                                unsigned prog;
                                unsigned vers;
                                unsigned prot;
                                unsigned port;
                                struct maplist the_rest;
                        };
                };
        } maplist;
```

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.

Indirect Call Routine

Procedure 5, Version 2.

```
5. PMAPPROC_CALLIT (prog,vers,proc,args) returns (port,res)
        unsigned prog;
        unsigned vers;
        unsigned proc;
        string args<>;
        unsigned port;
        string res<>;
```

This procedure allows a caller to call another remote procedure on the same machine without knowing the remote procedure's port number. Its intended use is 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. Note:

- This procedure only sends a response 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.

Index

CONVEX External Data Representation Protocol Specification

April 1988

CONVEX Computer Corporation
Richardson, Texas

CONVEX External Data Representation
Protocol Specification

Printed in the United States of America

Table of Contents

Appendices

List of Tables

1

Basics

Introduction

This manual describes library routines that allow a C programmer to describe arbitrary data structures in a machine-independent fashion. The eXternal Data Representation (*xdr*) standard is the backbone of the Remote Procedure Call package, in the sense that data for remote procedure calls is transmitted using the standard. *xdr* library routines should be used to transmit data that is accessed (read or written) by more than one type of machine.

This manual contains a description of *xdr* library routines, a guide to accessing currently available *xdr* streams, information on defining new streams and data types, and a formal definition of the *xdr* standard. *xdr* was designed to work across different languages, operating systems, and machine architectures. Most users (particularly *rpc* users) only need the information in Chapters 1 and 2 of this document. Programmers wishing to implement *rpc* and *xdr* on new machines need the information in Chapters 3 through 5. Advanced topics, not necessary for all implementations, are covered in Chapter 6.

C programs that want to use *xdr* routines must include the file *<rpc/rpc.h>*, which contains all the necessary interfaces to the *xdr* system. Since the C library *libc.a* contains all the *xdr* routines, compile as normal:

```
% cc program.c
```

Justification

Consider the following two programs, *writer:*

```
#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);
                }
        }
}
```

and *reader:*

```
#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");
}
```

The two programs appear to be portable, because (a) they pass *lint* checking, and (b) they exhibit the same behavior when executed on two different hardware architectures, a CONVEX supercomputer and a VAX.

Piping the output of the *writer* program to the *reader* program gives identical results on a CONVEX supercomputer or a VAX:

```
convex% writer | reader
0 1 2 3 4 5 6 7
convex%


vax% writer | reader
0 1 2 3 4 5 6 7
vax%
```

With the advent of local area networks and Berkeley's 4.2 BSD came the concept of "network pipes"—a process produces data on one machine, and a second process consumes data on another machine. A network pipe can be constructed with *writer* and *reader*. Here are the results if the first produces data on a CONVEX supercomputer, and the second consumes data on a VAX.

```
convex% writer | rsh vax reader
0 16777216 33554432 50331648 67108864 83886080 100663296
117440512
convex%
```

Identical results can be obtained by executing *writer* on the VAX and *reader* on the CONVEX supercomputer. These results occur because the byte ordering of long integers differs between the VAX and the CONVEX supercomputer, even though word size is the same. Note that 16777216 is $2 sup 24$—when four bytes are reversed, the 1 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()* calls with calls to an *xdr* library routine *xdr_long()*, a filter that knows the standard representation of a long integer in its external form. Here are the revised versions of *writer:*

```
#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);
                }
        }
}
```

and *reader:*

```
#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");
}
```

The new programs were executed on a CONVEX supercomputer, on a VAX, and from a CONVEX supercomputer to a VAX; the results are shown below.

```
convex% writer | reader
0 1 2 3 4 5 6 7
convex%


vax% writer | reader
0 1 2 3 4 5 6 7
vax%


convex% writer | rsh vax reader
0 1 2 3 4 5 6 7
convex%
```

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 have no meaning outside the machine where they are defined.

xdr Library

The *xdr* library solves data portability problems. It allows you to write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it makes sense to use the library even when the data is not shared among machines on a network. (Note that these library routines may be less efficient than hand-coded C routines.)

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.

Let's examine the two programs more closely. There is a family of *xdr* stream creation routines in which each member treats the stream of bits differently. In our example, data is manipulated using standard I/O routines, so we 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* clients never need to create *xdr* streams; the *rpc* system itself creates these streams, which are then passed to the clients.

The *xdr_long()* primitive is characteristic of most *xdr* library primitives and all client *xdr* routines. First, the routine returns *FALSE* (0) if it fails, and *TRUE* (1) if it succeeds. Second, for each data type, *xxx*, there is an associated *xdr* routine of the form:

```
xdr_xxx(xdrs, fp)
        XDR *xdrs;
        xxx *fp;
{
}
```

In our case, *xxx* is long, and the corresponding *xdr* routine is a primitive, *xdr_long*. The client could also define an arbitrary structure, *xxx*, in which case the client would also supply the routine *xdr_xxx*, describing each field by calling *xdr* routines of the appropriate type. In all cases the first parameter, *xdrs*, can be treated as an opaque handle, and passed to the primitive routines.

xdr routines are direction independent; that is, the same routines are called to serialize or deserialize data. This feature is critical to software engineering of portable data. The idea is to call the same routine for either operation—this almost guarantees that serialized data can also be deserialized. One routine is used by both producer and consumer of networked data. This is implemented by always passing the address of an object rather than the object itself — only in the case of deserialization is the object modified. This feature is not shown in our trivial example, but its value becomes obvious when nontrivial data structures are passed among machines. If needed, you can obtain the direction of the *xdr* operation.

Let's look at a slightly more complicated example. Assume that a person's gross assets and liabilities are to be exchanged among processes. Also assume that these values are important enough to warrant their own data type:

```
struct gnumbers {
        long g_assets;
        long g_liabilities;
};
```

The corresponding *xdr* routine describing this structure would be:

```
bool_t              /* TRUE is success, FALSE is failure */
xdr_gnumbers(xdrs, gp)
        XDR *xdrs;
        struct gnumbers *gp;
{
        if (xdr_long(xdrs, &gp->g_assets) &&
           xdr_long(xdrs, &gp->g_liabilities))
               return(TRUE);
        return(FALSE);
}
```

Note that the parameter *xdrs* is never inspected or modified; it is only passed on to the subcomponent routines. It is imperative to inspect the return value of each *xdr* routine call, and to give up immediately and return *FALSE* if the subroutine fails.

This example also shows that the type *bool_t* is declared as an integer whose only values are *TRUE* (1) and *FALSE* (0). This document uses the following definitions:

```
#define bool_t      int
#define TRUE 1
#define FALSE       0

#define enum_t int  /* enum_t used for generic enums */
```

Keeping these conventions in mind, *xdr_gnumbers()* can be rewritten as follows:

```
xdr_gnumbers(xdrs, gp)
        XDR *xdrs;
        struct gnumbers *gp;
{
        return(xdr_long(xdrs, &gp->g_assets) &&
               xdr_long(xdrs, &gp->g_liabilities));
}
```

This document uses both coding styles.

2

xdr Library Primitives

Introduction

This chapter gives a synopsis of each *xdr* primitive. It starts with basic data types and moves on to constructed data types. Finally, *xdr* utilities are discussed. The interface to these primitives and utilities is defined in the include file *<rpc/xdr.h>*, automatically included by *<rpc/rpc.h>*.

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] * [char, short, int, long, hyper]*

Specifically, the primitives are

```
bool_t xdr_char(xdrs, cp)
        XDR *xdrs;
        char *cp;

bool_t xdr_u_char(xdrs, ucp)
        XDR *xdrs;
        u_char *ucp;

bool_t xdr_short(xdrs, sip)
        XDR *xdrs;
        short *sip;

bool_t xdr_u_short(xdrs, sup)
        XDR *xdrs;
        u_short *sup;

bool_t xdr_int(xdrs, ip)
        XDR *xdrs;
        int *ip;

bool_t xdr_u_int(xdrs, up)
        XDR *xdrs;
        unsigned *up;

bool_t xdr_long(xdrs, lip)
        XDR *xdrs;
        long *lip;

bool_t xdr_u_long(xdrs, lup)
        XDR *xdrs;
        u_long *lup;

bool_t xdr_hyper(xdrs, hp)
```

```
        XDR *xdrs;
        hyper *hp;

bool_t xdr_u_hyper(xdrs, hp)
        XDR *xdrs;
        u_hyper *hp;
```

Note that *hyper* type is equivalent to the 64-bit CONVEX long long integer type and is an extension to the XDR Protocol Specification.

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(xdrs, fp)
        XDR *xdrs;
        float *fp;

bool_t xdr_double(xdrs, dp)
        XDR *xdrs;
        double *dp;
```

The first parameter, *xdrs, is an xdr* stream handle. The second parameter is the address of the floating-point number that provides data to the stream or receives data from it. All routines return *TRUE* if they complete successfully, and *FALSE* otherwise.

Note: Since the numbers are represented in IEEE floating-point, routines may fail when decoding a valid IEEE representation into a machine-specific representation, or vice versa.

Enumeration Filters

The *xdr* library provides a primitive for generic enumerations. The primitive assumes that a C *enum* has the same representation inside the machine as a C integer. The Boolean type is an important instance of the *enum*. The external representation of a Boolean is always one (*TRUE*) or zero (*FALSE*).

```
#define bool_t      int
#define FALSE       0
#define TRUE 1

#define enum_t int

bool_t xdr_enum(xdrs, ep)
        XDR *xdrs;
        enum_t *ep;

bool_t xdr_bool(xdrs, bp)
        XDR *xdrs;
        bool_t *bp;
```

The second parameters *ep* and *bp* are addresses of the associated type that provides data to, or

receives data from, the stream *xdrs*. The routines return *TRUE* if they complete successfully, and *FALSE* otherwise.

No Data Routines

Occasionally, an *xdr* routine must be supplied to the *rpc* system, even when no data is passed or required. The library provides such a routine:

```
bool_t xdr_void();  /* always returns TRUE */
```

Constructed Data Type Filters

Constructed or compound data type primitives require more parameters and perform more complicated functions than the primitives discussed above. 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. This is done by an *xdr* operation, *XDR_FREE*. To review, the three *xdr* directional operations are *XDR_ENCODE, XDR_DECODE,* and *XDR_FREE*.

Strings

In C, a string is defined as a sequence of bytes terminated by a null byte, which is not considered when calculating string length. However, when a string is passed or manipulated, a pointer to it is employed. Therefore, the *xdr* library defines a string to be a *char ** , and not a sequence of characters. The external representation of a string is drastically different from its internal representation. Externally, strings are represented as sequences of ASCII characters, while internally, they are represented with character pointers. Conversion between the two representations is accomplished with the routine *xdr_string():*

```
bool_t xdr_string(xdrs, sp, maxlength)
        XDR *xdrs;
        char **sp;
        u_int maxlength;
```

The first parameter *xdrs* is the *xdr* stream handle. The second parameter *sp* is a pointer to a string (type *char ***). The third parameter *maxlength* specifies the maximum number of bytes allowed during encoding or decoding; its value is usually specified by a protocol. For example, a protocol specification may say that a file name may be no longer than 255 characters. The routine returns *FALSE* if the number of characters exceeds *maxlength,* and *TRUE* if it doesn't.

The behavior of *xdr_string()* is similar to the behavior of other routines discussed in this section. The direction *XDR_ENCODE* is easiest to understand. The parameter *sp* points to a string of a certain length; if it 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 the value is *NULL,* then a string of the appropriate length is allocated and **sp* is set to this string. If the original value of **sp* is non-null, then the *xdr* package assumes that a target area has been allocated, which can hold strings no longer than *maxlength.* In either case, the string is decoded into the target area. The routine then appends a null character to the string.

In the *XDR_FREE* operation, the string is obtained by dereferencing *sp*. If the string is not

NULL, it is freed and **sp* is set to *NULL*. In this operation, *xdr_string* ignores the *maxlength* parameter.

Byte Arrays

Often variable-length arrays of bytes are preferable to strings. Byte arrays differ from strings in the following three ways: 1) the length of the array (the byte count) is explicitly located in an unsigned integer, 2) the byte sequence is not terminated by a null character, and 3) the external representation of the bytes is the same as their internal representation. The primitive *xdr_bytes()* converts between the internal and external representations of byte arrays:

```
bool_t xdr_bytes(xdrs, bpp, lp, maxlength)
        XDR *xdrs;
        char **bpp;
        u_int *lp;
        u_int maxlength;
```

The usage of the first, second, and fourth parameters is identical to the first, second and third parameters of *xdr_string()*, respectively. The length of the byte area is obtained by dereferencing *lp* when serializing; **lp* is set to the byte length when deserializing.

Arrays

The *xdr* library package provides a primitive for handling arrays of arbitrary elements. The *xdr_bytes()* routine treats a subset of generic arrays, in which the size of array elements is known to be 1, and the external description of each element is builtin. The generic array primitive, *xdr_array()* requires parameters identical to those of *xdr_bytes()* plus two more: the size of array elements, and an *xdr* routine to handle each of the elements. This routine is called to encode or decode each element of the array:

```
bool_t
xdr_array(xdrs, ap, lp, maxlength, elementsiz, xdr_element)
        XDR *xdrs;
        char **ap;
        u_int *lp;
        u_int maxlength;
        u_int elementsiz;
        bool_t (*xdr_element)();
```

The parameter *ap* is the address of the pointer to the array. If **ap* is *NULL* when the array is being deserialized, *xdr* allocates an array of the appropriate size and sets **ap* to that array. The element count of the array is obtained from **lp* when the array is serialized; **lp* is set to the array length when the array is deserialized. The parameter *maxlength* is the maximum number of elements that the array is allowed to have; *elementsiz* is the byte size of each element of the array (the C function *sizeof()* can be used to obtain this value). The routine *xdr_element* is called to serialize, deserialize, or free each element of the array.

Examples

Before defining more constructed data types, it is appropriate to present three examples.

Example A

A user on a networked machine can be identified by (a) the machine name, such as *krypton*: see

gethostname(3); (b) the user's UID: see *geteuid*(2); and (c) 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 < 256 chars */
#define NGRPS 20    /* user can't be in > 20 groups */

bool_t
xdr_netuser(xdrs, nup)
        XDR *xdrs;
        struct netuser *nup;
{
        return(xdr_string(xdrs, &nup->nu_machinename, NLEN) &&
           xdr_int(xdrs, &nup->nu_uid) &&
           xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen, NGRPS,
               sizeof (int), xdr_int));
}
```

Example B

A party of network users could be implemented as an array of *netuser* structure. The declaration and its associated *xdr* routines are as follows:

```
struct party {
        u_int p_len;
        struct netuser *p_nusers;
};
#define PLEN 500 /* max number of users in a party */

bool_t
xdr_party(xdrs, pp)
        XDR *xdrs;
        struct party *pp;
{
        return(xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
           sizeof (struct netuser), xdr_netuser));
}
```

Example C

The well-known parameters to *main(), argc,* and *argv* can be combined into a structure. An array of these structures can make up a history of commands. The declarations and *xdr* routines might look like:

```
struct cmd {
        u_int c_argc;
        char **c_argv;
```

```
};
#define ALEN 1000   /* args cannot be > 1000 chars */
#define NARGC 100   /* commands cannot have > 100 args */

struct history {
        u_int h_len;
        struct cmd *h_cmds;
};
#define NCMDS 75  /* history is no more than 75 commands */

bool_t
xdr_cmd(xdrs, cp)
        XDR *xdrs;
        struct cmd *cp;
{
        return(xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
            sizeof (char *), xdr_wrapstring));
}

bool_t
xdr_history(xdrs, hp)
        XDR *xdrs;
        struct history *hp;
{
        return(xdr_array(xdrs, &hp->h_cmds, &hp->h_len, NCMDS,
            sizeof (struct cmd), xdr_cmd));
}
```

The most confusing part of this example is that the routine *xdr_wrapstring()* is needed to package the *xdr_string()* routine, because the implementation of *xdr_array()* only passes two parameters to the array element description routine; *xdr_wrapstring()* supplies the third parameter to *xdr_string()*.

By now the recursive nature of the *xdr* library should be obvious. Let's continue with more constructed data types.

Opaque Handles

In some protocols, handles are passed from a server to client. The client passes the handle back to the server at some later time. Handles are never inspected by clients; they are obtained and submitted. That is to say, handles are opaque. The primitive *xdr_opaque()* is used for describing fixed sized, opaque bytes.

```
bool_t xdr_opaque(xdrs, p, len)
        XDR *xdrs;
        char *p;
        u_int len;
```

The parameter *p* is the location of the bytes; *len* is the number of bytes in the opaque object. By definition, the actual data contained in the opaque object are not machine portable. The opaque data type is described in more detail in Chapter 5.

Fixed-Sized Arrays

The *xdr* library provides a primitive, *xdr_vector*, for fixed-length arrays.

```
#define NLEN 255    /* machine names must be < 256 chars */
#define NGRPS 20    /* user belongs to exactly 20 groups */

struct netuser {
     char *nu_machinename;
     int nu_uid;
     int nu_gids[NGRPS];
};

bool_t
xdr_netuser(xdrs, nup)
     XDR *xdrs;
     struct netuser *nup;
{
     int i;

     if (!xdr_string(xdrs, &nup->nu_machinename, NLEN))
        return(FALSE);
     if (!xdr_int(xdrs, &nup->nu_uid))
        return(FALSE);
     if (!xdr_vector(xdrs, nup->nu_gids, NGRPS, sizeof(int),
        xdr_int)) {
           return(FALSE);
     }
     return(TRUE);
}
```

Exercise: Rewrite example A so that it uses varying-length arrays and so that the *netuser* structure contains the actual *nu_gids* array body as in the example above.

Discriminated Unions

The *xdr* library supports discriminated unions. A discriminated union is a C union and an *enum_t* value that selects an "arm" of the union.

```
struct xdr_discrim {
       enum_t value;
       bool_t (*proc)();
};

bool_t xdr_union(xdrs, dscmp, unp, arms, defaultarm)
       XDR *xdrs;
       enum_t *dscmp;
       char *unp;
       struct xdr_discrim *arms;
       bool_t (*defaultarm)();  /* may equal NULL */
```

First the routine translates the discriminant of the union located at **dscmp*. The discriminant is always an *enum_t*. Next the union located at **unp* is translated. The parameter *arms* is a pointer to an array of *xdr_discrim* structures. Each structure contains an order pair of *[value,proc]*. If the

union's discriminant is equal to the associated *value*, then the *proc* is called to translate the union. The end of the *xdr_discrim* structure array is denoted by a routine of value *NULL* (0). If the discriminant is not found in the *arms* array, then the *defaultarm* procedure is called if it is non-null; otherwise the routine returns *FALSE*.

Example D

Suppose the type of a union may be integer, character pointer (a string), or a *gnumbers* structure. Also, assume the union and its current type are declared in a structure. The declaration is:

```
enum utype { INTEGER=1, STRING=2, GNUMBERS=3 };

struct u_tag {
        enum utype utype;   /* the union's discriminant */
        union {
                int ival;
                char *pval;
                struct gnumbers gn;
        } uval;
};
```

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_wrapstring },
        { __dontcare__, NULL }
        /* always terminate arms with a NULL xdr_proc */
}

bool_t
xdr_u_tag(xdrs, utp)
        XDR *xdrs;
        struct u_tag *utp;
{
        return(xdr_union(xdrs, &utp->utype, &utp->uval,
                u_tag_arms, NULL));
}
```

The routine *xdr_gnumbers()* was presented in Chapter 1; *xdr_wrapstring()* was presented in example C. The default arm parameter to *xdr_union()* (the last parameter) is *NULL* in this example. Therefore the value of the union's discriminant may legally take on only values listed in the *u_tag_arms* array. This example also demonstrates that the elements of the arm's array do not need to be sorted.

It is worth pointing out that the values of the discriminant may be sparse, though in this example they are not. It is always good practice to assign explicitly integer values to each element of the discriminant's type. This practice both documents the external representation of the discriminant and guarantees that different C compilers emit identical discriminant values.

Exercise: Implement *xdr_union()* using the other primitives in this section.

Pointers

In C it is often convenient to put pointers to another structure within a structure. The primitive *xdr_reference()* makes it easy to serialize, deserialize, and free these referenced structures:

```
bool_t xdr_reference(xdrs, pp, size, proc)
        XDR **xdrs;
        char **pp;
        u_int ssize;
        bool_t (*proc)();
```

Parameter *pp* is the address of the pointer to the structure; parameter *ssize* is the size in bytes of the structure (use the C function *sizeof()* to obtain this value); and *proc* is the *xdr* routine that describes the structure. When decoding data, storage is allocated if **pp* is *NULL*.

There is no need for a primitive *xdr_struct()* to describe structures within structures, because pointers are always sufficient.

Exercise: Implement *xdr_reference()* using *xdr_array()*. Warning: *xdr_reference()* and *xdr_array()* are NOT interchangeable external representations of data.

Example E

Suppose there is a structure containing a person's name and a pointer to a *gnumbers* structure containing the person's gross assets and liabilities. The construct is:

```
struct pgn {
        char *name;
        struct gnumbers *gnp;
};
```

The corresponding *xdr* routine for this structure is:

```
bool_t
xdr_pgn(xdrs, pp)
        XDR *xdrs;
        struct pgn *pp;
{
        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 serializing data will most likely cause a memory fault and, on UNIX, a core dump for debugging.

It is the explicit responsibility of the programmer to expand non-dereferenceable pointers into their specific semantics. This usually involves describing data with a two-armed discriminated union. One arm is used when the pointer is valid; the other is used when the pointer is invalid (*NULL*). Chapter 6 has an example (linked list encoding) that deals with invalid pointer interpretation.

Exercise: Using the *xdr_union()*, *xdr_reference()* and *xdr_void()* primitives, implement a generic pointer handling primitive that implicitly deals with *NULL* pointers. The *xdr* library does not provide such a primitive because it does not want to give the illusion that pointers have meaning in the external world.

Non-Filter Primitives

xdr streams can be manipulated with the primitives discussed in this section.

```
u_int xdr_getpos(xdrs)
        XDR *xdrs;

bool_t xdr_setpos(xdrs, pos)
        XDR *xdrs;
        u_int pos;

xdr_destroy(xdrs)
        XDR *xdrs;
```

The routine *xdr_getpos()* returns an unsigned integer that describes the current position in the data stream. Warning: In some *xdr* streams, the returned value of *xdr_getpos()* is meaningless; the routine returns a –1 in this case (though –1 should be a legitimate value).

The routine *xdr_setpos()* sets a stream position to *pos*. Warning: In some *xdr* streams, setting a position is impossible; in such cases, *xdr_setpos()* returns *FALSE*. This routine also fails 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

At times you may wish to optimize *xdr* routines by taking advantage of the direction of the operation — *XDR_ENCODE*, *XDR_DECODE*, or *XDR_FREE*. The value *xdrs->x_op* always contains the direction of the *xdr* operation. Programmers are not encouraged to take advantage of this information. Therefore, no example is presented here. However, an example in Appendix A demonstrates the usefulness of the *xdrs->x_op* field.

3

xdr Stream Access

Introduction

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 files, and memory. Chapter 4 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 as follows:

```
#include <stdio.h>
#include <rpc/rpc.h>      /* xdr streams part of rpc */

void
xdrstdio_create(xdrs, fp, x_op)
        XDR *xdrs;
        FILE *fp;
        enum xdr_op x_op;
```

The routine *xdrstdio_create()* initializes an *xdr* stream pointed to by *xdrs*. The *xdr* stream interfaces to the standard I/O library. Parameter *fp* is an open file, and *x_op* is an *xdr* direction.

Memory Streams

Memory streams allow the streaming of data into or out of a specified area of memory:

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

void
xdrmem_create(xdrs, addr, len, x_op)
        XDR *xdrs;
        char *addr;
        u_int len;
        enum xdr_op x_op;
```

The routine *xdrmem_create()* initializes an *xdr* stream in local memory. The memory is pointed to by parameter *addr;* parameter *len* is the length in bytes of the memory. The parameters *xdrs* and *x_op* are identical to the corresponding parameters of *xdrstdio_create()*. Currently, the UDP/IP implementation of *rpc* uses *xdrmem_create()*. Complete call or result messages are built in memory before calling the *sendto()* system routine.

Record (TCP/IP) Streams

A record stream is an *xdr* stream built on top of a record-marking standard that is built on top of the file or 4.2BSD connection interface:

```
#include <rpc/rpc.h>        /* xdr streams part of rpc */

xdrrec_create(xdrs,
  sendsize, recvsize, iohandle, readproc, writeproc)
        XDR *xdrs;
        u_int sendsize, recvsize;
        char *iohandle;
        int (*readproc)(), (*writeproc)();
```

The routine *xdrrec_create()* provides an *xdr* stream interface that allows for a bidirectional, arbitrarily long sequence of records. The contents of the records are meant to be data in *xdr* form. The stream's primary use is for interfacing *rpc* to TCP connections. However, it can be used to stream data into or out of normal files.

The parameter *xdrs* is similar to the corresponding parameter described above. The stream does its own data buffering similar to that of standard I/O. The parameters *sendsize* and *recvsize* determine the size in bytes of the output and input buffers, respectively; if their values are zero (0), then predetermined defaults are used. When a buffer needs to be filled or flushed, the routine *readproc* or *writeproc* is called, respectively. The usage and behavior of these routines are similar to the system calls *read()* and *write()*. However, the first parameter to each of these routines 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*, then it has the following form:

```
/*
 * returns the actual number of bytes transferred.
 * -1 is an error
 */
int
xxx(iohandle, buf, len)
        char *iohandle;
        char *buf;
        int nbytes;
```

The *xdr* stream provides means for delimiting records in the byte stream. The implementation details of delimiting records in a stream are discussed in Appendix A. The primitives that are specific to record streams are as follows:

```
bool_t
xdrrec_endofrecord(xdrs, flushnow)
        XDR *xdrs;
        bool_t flushnow;

bool_t
xdrrec_skiprecord(xdrs)
        XDR *xdrs;

bool_t
xdrrec_eof(xdrs)
        XDR *xdrs;
```

The routine *xdrrec_endofrecord()* causes the current outgoing data to be marked as a record. If the parameter *flushnow* is *TRUE*, then the stream's *writeproc()* is called; otherwise, *writeproc()* is called when the output buffer has been filled.

The routine *xdrrec_skiprecord()* causes an input stream's position to be moved past the current record boundary and onto the beginning of the next record in the stream.

If there is no more data in the stream's input buffer, then the routine *xdrrec_eof()* returns *TRUE*. That is not to say that there is no more data in the underlying file descriptor.

4

xdr Stream Implementation

Introduction

This chapter provides the abstract data types needed to implement new instances of *xdr* streams.

The *xdr* Object

The following 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.

Macros for accessing operations *x_getpostn()*, *x_setpostn()*, and *x_destroy()* are defined in Appendix A. 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 or put. The routine may return *NULL* if it cannot return a buffer segment of the requested size. (The *x_inline* routine is for cycle squeezers. Use of the resulting buffer is not data-portable. Users are encouraged not to use this feature.)

The operations *x_getbytes()* and *x_putbytes()* blindly get and put sequences of bytes from or to the underlying stream; they return *TRUE* if they are successful, and *FALSE* otherwise. The routines have identical parameters (replace *xxx*):

```
bool_t
xxxbytes(xdrs, buf, bytecount)
        XDR *xdrs;
        char *buf;
        u_int bytecount;
```

The operations *x_getlong()* and *x_putlong()* receive and put long numbers from and to the data stream. It is the responsibility of these routines to translate the numbers between the machine representation and the (standard) external representation. The primitives *htonl()* and *ntohl()* can be helpful in accomplishing this. Chapter 5 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. The routines return *TRUE* if they succeed, and *FALSE* otherwise. They have identical parameters:

```
bool_t
xxxlong(xdrs, lp)
        XDR *xdrs;
        long *lp;
```

Implementors of new *xdr* streams must make an *xdr* structure (with new operation routines) available to clients, using some kind of create routine.

5

xdr Standard

Introduction

This chapter defines the external data representation standard. The standard is independent of languages, operating systems and hardware architectures. Once data is shared among machines, it should not matter that the data was produced on a C1, but 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.

The external data representation standard depends on the assumption that bytes (or octets) are portable. A byte is defined to be eight bits of data. It is assumed 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 "little endian" style. Both CONVEX and VAX hardware implementations adhere to the standard.

The *xdr* standard also suggests a language used to describe data. The language is a bastardized C; it is a data definition language, not a programming language. (The Xerox Courier Standard uses bastardized Mesa as its data definition language.)

Basic Block Size

The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered 0 through $n-1$, where $(n ~ mod ~ 4) == 0$. The bytes are read or written to some byte stream such that byte m always precedes byte $m+1$.

Integer

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 definition of integers is *integer*.

Unsigned Integer

An *xdr* unsigned integer is a 32-bit datum that encodes a nonnegative 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 definition of unsigned integers is *unsigned*.

Enumerations

Enumerations have the same representation as integers. Enumerations are handy for describing subsets of the integers.

The data definition of enumerated data is as follows:

```
typedef enum { name = value, .... } type-name;
```

For example the three colors red, yellow and blue could be described by an enumerated type:

```
typedef enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
```

Booleans

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. A Boolean is an enumeration with the following form:

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

Hyper Integer and Hyper Unsigned

The standard also defines 64-bit (8-byte) numbers called *hyper integer* and *hyper unsigned*. Their representations are the obvious extensions of the integer and unsigned defined above. The most and least significant bytes are 0 and 7, respectively. (In CONVEX C, 64-bit integers are referred to as "long longs.")

Floating Point and Double Precision

The standard defines the encoding for the floating-point data types *float* (32 bits or 4 bytes) and *double* (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized single- and double-precision floating-point numbers. See the IEEE floating-point standard for more information. The standard encodes the following three fields, which describe the floating-point number:

- The sign of the number. Values 0 and 1 represent positive and negative, respectively.
- The exponent of the number, base 2. Floats devote 8 bits to this field, while doubles devote 11 bits. The exponents for float and double are biased by 127 and 1023, respectively.
- The fractional part of the number's mantissa, base 2. Floats devote 23 bits to this field, while doubles devote 52 bits.

Therefore, the floating-point number is described by:

$$(-1)^{S} * 2^{E-Bias} * 1.F$$

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.

Doubles have the analogous extensions. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 12, respectively.

The IEEE specification should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). Under IEEE specifications, the

"NaN" (not a number) is system dependent and should not be used.

Opaque Data

At times fixed-sized uninterpreted data needs to be passed among machines. This data is called *opaque* and is described as:

```
typedef opaque type-name[n];
opaque name[n];
```

where n is the (static) number of bytes necessary to contain the opaque data. If n is not a multiple of four, then the n bytes are followed by enough (up to 3) zero-valued bytes to make the total byte count of the opaque object a multiple of four.

Counted Byte Strings

The standard defines a string of n (numbered 0 through $n-1$) bytes to be the number n encoded as *unsigned*, and followed by the n bytes of the string. If n is not a multiple of four, then the n bytes are followed by enough (up to 3) zero-valued bytes to make the total byte count a multiple of four. The data definition of strings is as follows:

```
typedef string type-name<N>;
typedef string type-name<>;
string name<N>;
string name<>;
```

Note that the data definition language uses angle brackets ($<$ and $>$) to denote anything that is varying-length (as opposed to square brackets to denote fixed-length sequences of data).

The constant N denotes an upper bound of the number of bytes that a string may contain. If N is not specified, it is assumed to be $2 sup 32 - 1$, the maximum length. The constant N would normally be found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than 255 bytes, such as:

```
string filename<255>;
```

The *xdr* specification does not say what the individual bytes of a string represent; this important information is left to higher-level specifications. A reasonable default is to assume that the bytes encode ASCII characters.

Fixed Arrays

The data definition for fixed-size arrays of homogeneous elements is as follows:

```
typedef elementtype type-name[n];
elementtype name[n];
```

Fixed-size arrays of elements numbered 0 through $n-1$ are encoded by individually encoding the elements of the array in their natural order, 0 through $n-1$.

Counted 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 $n-1$. The data definition for counted arrays is similar to that of counted strings:

```
typedef elementtype type-name<N>;
typedef elementtype type-name<>;
elementtype name<N>;
elementtype name<>;
```

Again, the constant N specifies the maximum acceptable element count of an array; if N is not specified, it is assumed to be $2 sup 32 - 1$.

Structures

The data definition for structures is very similar to that of standard C:

```
typedef struct {
        component-type component-name;
        ...
} type-name;
```

The components of the structure are encoded in the order of their declaration in the structure.

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 the discriminant is always an enumeration. The component types are called "arms" of the union. The discriminated union is encoded as its discriminant followed by the encoding of the implied arm. The data definition for discriminated unions is as follows:

```
typedef union switch (discriminant-type) {
        discriminant-value: arm-type;
        ...
        default: default-arm-type;
} type-name;
```

The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. Most specifications neither need nor use default arms.

Missing Specifications

The standard lacks representations for bit fields and bitmaps, since the standard is based on bytes. This is not to say that no specification should be attempted.

Library Primitive / *xdr* Standard Cross-Reference

Table 5-1 describes the association between the C library primitives discussed in Chapter 2 and the standard data types defined in this chapter.

Table 5-1: Primitives and Data Types

C Primitive	XDR Type
xdr_int xdr_long xdr_short	integer
xdr_u_int xdr_u_long xdr_u_short	unsigned
	hyper integer hyper unsigned
xdr_float	float
xdr_double	double
xdr_enum	enum_t
xdr_bool	bool_t
xdr_string xdr_bytes	string
xdr_array	(varying arrays)
	(fixed arrays)
xdr_opaque	opaque
xdr_union	union
xdr_reference	-
	struct

6

Advanced Topics

This chapter describes techniques for passing data structures that are not covered in the preceding chapters. Such structures include linked lists (of arbitrary lengths). Unlike the simpler examples covered in the earlier chapters, the following examples are written using both the *xdr* C library routines and the *xdr* data description language. Chapter 5 describes the *xdr* data definition language used below.

Linked Lists

The last example in Chapter 2 presented a C data structure and its associated *xdr* routines for a person's gross assets and liabilities. The example is duplicated below:

```
struct gnumbers {
        long g_assets;
        long g_liabilities;
};

bool_t
xdr_gnumbers(xdrs, gp)
        XDR *xdrs;
        struct gnumbers *gp;
{
        if (xdr_long(xdrs, &(gp->g_assets)))
                return(xdr_long(xdrs, &(gp->g_liabilities)));
        return(FALSE);
}
```

Now assume that we wish to implement a linked list of such information. A data structure could be constructed as follows:

```
typedef struct gnnode {
        struct gnumbers gn_numbers;
        struct gnnode *nxt;
};

typedef struct gnnode *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 *nxt* field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the *nxt* field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

The *xdr* data description of this linked list is described by the recursive type declaration of *gnumbers_list*:

```
struct gnumbers {
        unsigned g_assets;
        unsigned g_liabilities;
};


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 it is *TRUE,* then 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 in it (though the *nxt* field implicitly carries the information), while the *xdr* data description has no pointer explicitly declared in it.

Hints for writing a set of *xdr* routines to successfully (de)serialize a linked list of entries can be taken from the *xdr* description of the pointer-less data. The set consists of the mutually recursive routines *xdr_gnumbers_list, xdr_wrap_list,* and *xdr_gnnode*:

```
bool_t
xdr_gnnode(xdrs, gp)
        XDR *xdrs;
        struct gnnode *gp;
{
        return(xdr_gnumbers(xdrs, &(gp->gn_numbers)) &&
                xdr_gnumbers_list(xdrs, &(gp->nxt)) );
}


bool_t
xdr_wrap_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        return(xdr_reference(xdrs, glp, sizeof(struct gnnode),
           xdr_gnnode));
}


struct xdr_discrim choices[2] = {
        /*
         * called if another node needs (de)serializing
         */
        { TRUE, xdr_wrap_list },
        /*
         * called when no more nodes need (de)serializing
         */
        { FALSE, xdr_void }
}
```

```
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;

        more_data = (*glp != (gnumbers_list)NULL);
        return(xdr_union(xdrs, &more_data, glp, choices, NULL);
}
```

The entry routine is *xdr_gnumbers_list();* its job is to translate between the Boolean value *more_data* and the list pointer values. If there is no more data, the *xdr_union()* primitive calls *xdr_void()* and the recursion is terminated. Otherwise, *xdr_union()* calls *xdr_wrap_list(),* whose job is to dereference the list pointers. The *xdr_gnnode()* routine actually (de)serializes data of the current node of the linked list, and recursively calls *xdr_gnumbers_list()* to handle the remainder of the list.

You should convince yourself that these routines function correctly in all three directions *(XDR_ENCODE, XDR_DECODE,* and *XDR_FREE)* for linked lists of any length (including zero). Note that the Boolean *more_data* is always initialized, but in the *XDR_DECODE* case it is overwritten by an externally generated value. Also note that the value of the *bool_t* is lost in the stack. The essence of the value is reflected in the list's pointers.

The unfortunate side effect of (de)serializing a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list. This is due to the recursion. The routines are also hard to code (and understand) due to the number and nature of primitives involved (such as *xdr_reference, xdr_union,* and *xdr_void*).

The following routine collapses the recursive routines (it also has other optimizations that are discussed below):

```
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;

        while (TRUE) {
                more_data = (*glp != (gnumbers_list)NULL);
                if (!xdr_bool(xdrs, &more_data))
                        return(FALSE);
                if (!more_data)
                        return(TRUE);  /* we are done */
                if (!xdr_reference(xdrs, glp, sizeof(struct gnnode),
                    xdr_gnumbers))
                        return(FALSE);
                glp< = &((*glp)->nxt);
        }
}
```

The claim is that this one routine is easier to code and understand than the three recursive routines above. (It is also buggy, as discussed below.) The parameter *glp* is treated as the address of the pointer to the head of the remainder of the list to be (de)serialized. Thus, *glp* is set to the address of the current node's *nxt* field at the end of the while loop. The discriminated

union is implemented in-line; the variable *more_data* has the same use in this routine as in the routines above. Its value is recomputed and re-(de)serialized each iteration of the loop. Since **glp* is a pointer to a node, the pointer is dereferenced using *xdr_reference()*. Note that the third parameter is truly the size of a node (data values plus *nxt* pointer), while *xdr_gnumbers()* only (de)serializes the data values. This tricky optimization works only because the *nxt* data comes after all legitimate external data.

The routine is buggy in the *XDR_FREE* case. The bug is that *xdr_reference()* frees the node **glp*. On return, the assignment:

*glp = &((*glp)->nxt)*

cannot be guaranteed to work since **glp* is no longer a legitimate node. The following is a rewrite that works in all cases; the hard part is to avoid dereferencing a pointer that has not been initialized or that has been freed:

```
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;
        bool_t freeing;
        gnumbers_list *next;  /* the next value of glp */

        freeing = (xdrs->x_op == XDR_FREE);
        while (TRUE) {
                more_data = (*glp != (gnumbers_list)NULL);
                if (!xdr_bool(xdrs, &more_data))
                        return(FALSE);
                if (!more_data)
                        return(TRUE);  /* we are done */
                if (freeing)
                        next = &((*glp)->nxt);
                if (!xdr_reference(xdrs, glp, sizeof(struct gnnode),
                   xdr_gnumbers))
                        return(FALSE);
                glp = (freeing) ? next : &((*glp)->nxt);
        }
}
```

Note that this is the first example in this document that actually inspects the direction of the operation *xdrs->x_op*. The claim is that the correct iterative implementation is still easier to understand or code than the recursive implementation. It is certainly more efficient with respect to C stack requirements.

Record-Marking Standard

A record is composed of one or more record fragments. A record fragment is a four-byte header followed by 0 to $2^{31}-1$ bytes of fragment data. The bytes encode an unsigned binary number; as with *xdr* integers, the byte order is from highest to lowest. The number encodes two values — a Boolean that 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 high-order bit of the header; the length is the 31 low-order bits.

(Note that this record specification is *not* in *xdr* standard form and cannot be implemented using *xdr* primitives!)

A

Synopsis of *xdr* Routines

xdr_array()

```
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
        XDR *xdrs;
        char **arrp;
        u_int *sizep, maxsize, elsize;
        xdrproc_t elproc;
```

A filter primitive that translates between arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, and *sizep* is the address of the element count of the array. This element count cannot exceed *maxsize*. The parameter *elsize* is the *sizeof()* each of the array's elements, and *elproc* is an *xdr* filter that translates between the array elements' C form and their external representation. This routine returns one if it succeeds, zero otherwise.

xdr_bool()

```
xdr_bool(xdrs, bp)
        XDR *xdrs;
        bool_t *bp;
```

A filter primitive that translates between Booleans (C integers) and their external representations. When encoding data, this filter produces values of either one or zero. This routine returns one if it succeeds, zero otherwise.

xdr_bytes()

```
xdr_bytes(xdrs, sp, sizep, maxsize)
        XDR *xdrs;
        char **sp;
        u_int *sizep, maxsize;
```

A filter primitive that translates between counted byte strings and their external representations. The parameter *sp* is the address of the string pointer. The length of the string is located at address *sizep;* strings cannot be longer than *maxsize*. This routine returns one if it succeeds, zero otherwise.

xdr_char()

```
xdr_char(xdrs, cp)
        XDR *xdrs;
        char *cp;
```

A filter primitive that translates between C characters and their external representations. This routine returns one if it succeeds, zero otherwise. Note: encoded characters are not packed, and

occupy 4 bytes each. For arrays of characters, it is worthwhile to consider *xdr_bytes*, *xdr_opaque*, or *xdr_string*.

xdr_destroy()

```
void
xdr_destroy(xdrs)
        XDR *xdrs;
```

A macro that invokes the destroy routine associated with the *xdr*stream, *xdrs*. Destruction usually involves freeing private data structures associated with the stream. Using *xdrs* after invoking *xdr_destroy()* is undefined.

xdr_double()

```
xdr_double(xdrs, dp)
        XDR *xdrs;
        double *dp;
```

A filter primitive that translates between C *double* precision numbers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_enum()

```
xdr_enum(xdrs, ep)
        XDR *xdrs;
        enum_t *ep;
```

A filter primitive that translates between C *enum s* (actually integers) and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_float()

```
xdr_float(xdrs, fp)
        XDR *xdrs;
        float *fp;
```

A filter primitive that translates between C *float s* and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_free()

```
void
xdr_free(xdrs, proc, objp)
        xdrproc_t proc;
        char *objp;
```

Generic freeing routine. The first argument is the *xdr* routine for the object being freed. The second argument is a pointer to the object itself. Note: the pointer passed to this routine is *not* freed, but what it points to *is* recursively freed.

xdr_getpos()

```
u_int
xdr_getpos(xdrs)
        XDR *xdrs;
```

A macro that invokes the get-position routine associated with the *xdr* stream, *xdrs*. The routine returns an unsigned integer, which indicates the position of the *xdr* byte stream. A desirable feature of *xdr* streams is that simple arithmetic works with this number, although the *xdr* stream instances need not guarantee this.

xdr_hyper()

```
xdr_hyper(xdrs, hp)
        XDR *xdrs;
        hyper *hp;
```

A filter primitive that translates between C *hyper* (really CONVEX long long) integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_inline()

```
long *
xdr_inline(xdrs, len)
        XDR *xdrs;
        int len;
```

A macro that invokes the in-line routine associated with the *xdr* stream, *xdrs*. The routine returns a pointer to a contiguous piece of the stream's buffer; *len* is the byte length of the desired buffer. Note that the pointer is cast to "*long* *". Warning: *xdr_inline()* may return *NULL* if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

xdr_int()

```
xdr_int(xdrs, ip)
        XDR *xdrs;
        int *ip;
```

A filter primitive that translates between C integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_long()

```
xdr_long(xdrs, lp)
        XDR *xdrs;
        long *lp;
```

A filter primitive that translates between C *long* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_opaque()

```
xdr_opaque(xdrs, cp, cnt)
        XDR *xdrs;
        char *cp;
        u_int cnt;
```

A filter primitive that translates between fixed size opaque data and its external representation. The parameter *cp* is the address of the opaque object, and *cnt* is its size in bytes. This routine returns one if it succeeds, zero otherwise.

xdr_pointer()

```
xdr_pointer(xdrs, objpp, objsize, xdrobj)
        XDR *xdrs;
        char **objpp;
        u_int objsize;
        xdrproc_t xdrobj;
```

Like *xdr_reference* except that *xdr_pointer* serializes NULL pointers, whereas *xdr_reference* does not. Thus *xdr_pointer* can *xdr* recursive data structures, such as binary trees or linked lists, correctly, whereas *xdr_reference* fails.

xdr_reference()

```
xdr_reference(xdrs, pp, size, proc)
        XDR *xdrs;
        char **pp;
        u_int size;
        xdrproc_t proc;
```

A primitive that provides pointer chasing within structures. The parameter *pp* is the address of the pointer; *size* is the *sizeof()* the structure that **pp* points to; and *proc* is an *xdr* procedure that filters the structure between its C form and its external representation. This routine returns one if it succeeds, zero otherwise.

xdr_setpos()

```
xdr_setpos(xdrs, pos)
        XDR *xdrs;
        u_int pos;
```

A macro that invokes the set position routine associated with the *xdr* stream *xdrs*. The parameter *pos* is a position value obtained from *xdr_getpos()*. This routine returns one if the *xdr* stream could be repositioned, zero otherwise. Warning: it is difficult to reposition some types of *xdr* streams, so this routine may fail with one type of stream and succeed with another.

xdr_short()

```
xdr_short(xdrs, sp)
        XDR *xdrs;
        short *sp;
```

A filter primitive that translates between C *short* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_string()

```
xdr_string(xdrs, sp, maxsize)
        XDR *xdrs;
        char **sp;
        u_int maxsize;
```

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot cannot be longer than *maxsize*. Note that *sp* is the address of the string's pointer. This routine returns one if it succeeds, zero otherwise.

xdr_u_char()

```
xdr_u_char(xdrs, ucp)
        XDR *xdrs;
        unsigned char *ucp;
```

A filter primitive that translates between *unsigned* C characters and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_hyper()

```
xdr_u_hyper(xdrs, hp)
        XDR *xdrs;
        u_hyper *hp;
```

A filter primitive that translates between C *unsigned hyper* (really CONVEX unsigned long long) integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_int()

```
xdr_u_int(xdrs, up)
        XDR *xdrs;
        unsigned *up;
```

A filter primitive that translates between C *unsigned* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_long()

```
xdr_u_long(xdrs, ulp)
        XDR *xdrs;
        unsigned long *ulp;
```

A filter primitive that translates between C *unsigned long* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_short()

```
xdr_u_short(xdrs, usp)
        XDR *xdrs;
        unsigned short *usp;
```

A filter primitive that translates between C *unsigned short* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_union()

```
xdr_union(xdrs, dscmp, unp, choices, dfault)
        XDR *xdrs;
        int *dscmp;
        char *unp;
        struct xdr_discrim *choices;
        xdrproc_t dfault;
```

A filter primitive that translates between a discriminated C *union* and its corresponding external representation. The parameter *dscmp* is the address of the union's discriminant, while *unp* in the address of the union. This routine returns one if it succeeds, zero otherwise.

xdr_vector()

```
xdr_vector(xdrs, arrp, size, elsize, elproc)
        XDR *xdrs;
        char *arrp
        u_int size, elsize;
        xdrproc_t elproc;
```

A filter primitive that translates between fixed-length arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *size* is is the element count of the array. The parameter *elsize* is the *sizeof* each of the array's elements, and *elproc* is an *xdr* filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

xdr_void()

```
xdr_void()
```

This routine always returns one. It may be passed to *rpc* routines that require a function parameter, where nothing is to be done.

xdr_wrapstring()

```
xdr_wrapstring(xdrs, sp)
        XDR *xdrs;
        char **sp;
```

A primitive that calls *xdr_string(xdrs,sp,MAXUNSIGNED),* where *MAXUNSIGNED* is the maximum value of an unsigned integer. This is handy because the *rpc* package passes only two parameters to *xdr* routines, whereas *xdr_string(),* one of the most frequently used primitives, requires three parameters. This routine returns one if it succeeds, zero otherwise.

xdrmem_create()

```
void
xdrmem_create(xdrs, addr, size, op)
        XDR *xdrs;
        char *addr;
        u_int size;
        enum xdr_op op;
```

This routine initializes the *xdr* stream object pointed to by *xdrs.* The stream's data is written to, or read from, a chunk of memory at location *addr* whose length is no more than *size* bytes long. The *op* determines the direction of the *xdr* stream (either *XDR_ENCODE, XDR_DECODE,* or *XDR_FREE*).

xdrrec_create()

```
void
xdrrec_create(xdrs,
  sendsize, recvsize, handle, readit, writeit)
        XDR *xdrs;
        u_int sendsize, recvsize;
        char *handle;
        int (*readit)(), (*writeit)();
```

This routine initializes the *xdr* stream object pointed to by *xdrs.* The stream's data is written to a buffer of size *sendsize;* a value of zero indicates the system should use a suitable default. The stream's data is read from a buffer of size *recvsize;* it too can be set to a suitable default by passing a zero value. When a stream's output buffer is full, *writeit()* is called. Similarly, when a stream's input buffer is empty, *readit()* is called. The behavior of these two routines is similar to the system calls *read* and *write,* except that *handle* is passed to the former routines as the first parameter. Note that the *xdr* stream's *op* field must be set by the caller. Warning: this *xdr* stream implements an intermediate record stream. Therefore, there are additional bytes in the stream to provide record boundary information.

xdrrec_endofrecord()

```
xdrrec_endofrecord(xdrs, sendnow)
        XDR *xdrs;
        int sendnow;
```

This routine can be invoked only on streams created by *xdrrec_create().* The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if *sendnow* is non-zero. This routine returns one if it succeeds, zero otherwise.

xdrrec_eof()

```
xdrrec_eof(xdrs)
        XDR *xdrs;
        int empty;
```

This routine can be invoked only on streams created by *xdrrec_create()*. After consuming the rest of the current record in the stream, this routine returns one if the stream has no more input, zero otherwise.

xdrrec_skiprecord()

```
xdrrec_skiprecord(xdrs)
        XDR *xdrs;
```

This routine can be invoked only on streams created by *xdrrec_create()*. It tells the *xdr* implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns one if it succeeds, zero otherwise.

xdrstdio_create()

```
void
xdrstdio_create(xdrs, file, op)
        XDR *xdrs;
        FILE *file;
        enum xdr_op op;
```

This routine initializes the *xdr* stream object pointed to by *xdrs*. The *xdr* stream data is written to, or read from, the Standard I/O stream *file*. The parameter *op* determines the direction of the *xdr* stream (either *XDR_ENCODE*, *XDR_DECODE*, or *XDR_FREE*). Warning: the destroy routine associated with such *xdr* streams calls *fflush()* on the *file* stream, but never *fclose()*.

Index

CONVEX Remote Procedure Call Programming Guide

April 1988

CONVEX Computer Corporation
Richardson, Texas

CONVEX Remote Procedure Call
Programming Guide

Printed in the United States of America

Table of Contents

Appendices

List of Tables

List of Figures

1

Introduction to *rpc*

Overview

This document is intended for programmers who wish to write network applications using remote procedure calls (explained below), thus avoiding low-level system primitives based on sockets. The reader must be familiar with the C programming language, and should have a working knowledge of network theory. Detailed information regarding *rpc* entry points and arguments is located in the *RPC Protocol Specification.*

Programs that communicate over a network need a paradigm for communication. A low-level mechanism might send a signal on the arrival of incoming packets, causing a network signal handler to execute. A high-level mechanism would be the Ada *rendezvous.* The method commonly used is the Remote Procedure Call (*rpc*) paradigm, in which a client communicates with a server. In this process, the client first calls a procedure 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.

Layers of *rpc*

The *rpc* interface is divided into three layers. The highest layer is totally transparent to the programmer. To illustrate, at this level a program can contain a call to *rnusers()*, which returns the number of users on a remote machine. You don't have to be aware that *rpc* is being used, since you simply make the call in a program, just as you would call *malloc().*

At the middle layer, the routines *registerrpc()* and *callrpc()* are used to make *rpc* calls: *registerrpc()* obtains a unique system-wide number, while *callrpc()* executes a remote procedure call. The *rnusers()* call is implemented using these two routines. The middle-layer routines are designed for most common applications, and shield the user from knowing about sockets.

The lowest layer is for more sophisticated applications, such as altering the defaults of the routines. At this layer, you can explicitly manipulate sockets that transmit *rpc* messages. This level should be avoided if possible.

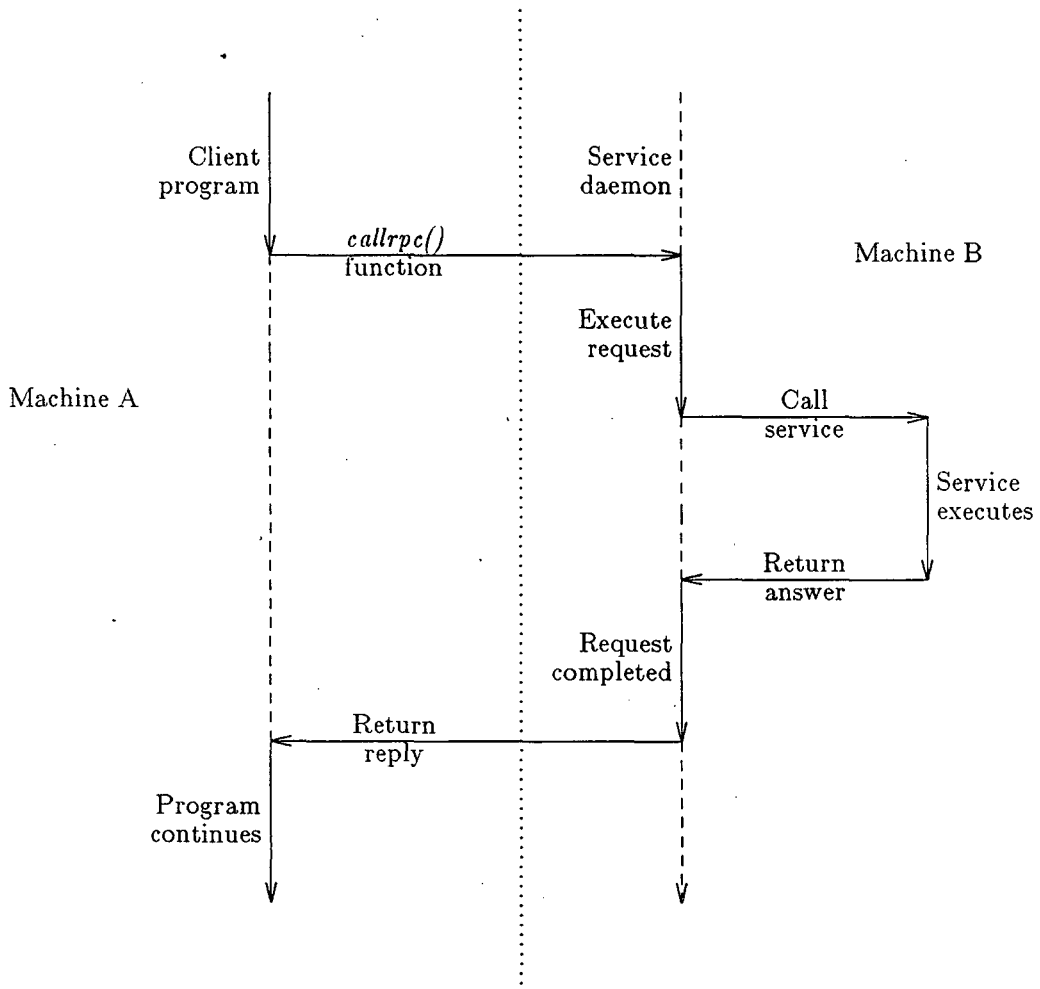
Chapter 2 of this manual illustrates use of the highest two layers, while Chapter 3 presents the low-level interface. Chapter, 4 of the manual discusses miscellaneous topics. The final chapter summarizes all the entry points into the *rpc* system.

Although this document only discusses the interface to C, remote procedure calls can be made from any language. Even though this document discusses *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.

rpc Paradigm

Figure 1-1 is a diagram of the *rpc* paradigm.

Figure 1–1: *rpc* Paradigm



2

Higher Layers of *rpc*

Highest Layer

Imagine you're writing a program that needs to know how many users are logged in to a remote machine. You can do this by calling the library routine *rnusers()*, as illustrated below:

```
#include <stdio.h>

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

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

rpc library routines such as *rnusers()* are in the *rpc* services library *librpcsvc.a*. Thus, the program above should be compiled with:

```
% cc program.c -lrpcsvc
```

This routine, and other *rpc* library routines, are documented in Section 3 of the *CONVEX UNIX Programmer's Manual*. Table 2-1 shows the *rpc* service library routines available to the C programmer.

Table 2-1: *rpc* Service Library Routines

Routine	*Description*
rnusers()	Return number of users on remote machine
rusers()	Return information about users on remote machine
havedisk()	Determine if remote machine has disk
rstat()	Get performance data from remote kernel
rwall()	Write to specified remote machines
getrpcport()	Get *rpc* port number
yppasswd()	Update user password in yellow pages

The other *rpc* services—*ether, mount, rquota,* and *spray*— are not available to the C programmer as library routines. They do, however, have *rpc* program numbers so they can be invoked with *callrpc(),* discussed in the next section.

Intermediate Layer

The simplest interface that explicitly makes *rpc* calls, uses the functions *callrpc()* and *registerrpc().* Using this method, another way to get the number of remote users is:

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

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

        if (argc < 2) {
        fprintf(stderr, "usage: nusers hostname\n");
        exit(-1);
        }
        if (callrpc(argv[1],
          RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
          xdr_void, 0, xdr_u_long, &nusers) != 0) {
        fprintf(stderr, "error: callrpc\n");
        exit(1);
        }
        printf("%d users on %s\n", nusers, argv[1]);
        exit(0);
}
```

A program number, version number, and procedure number defines each *rpc* procedure. The program number defines 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 (adding a new procedure, for example), a new program number doesn't have to be assigned. When you want to call a procedure to find the number of remote users, you look up the appropriate program, version, and procedure numbers in a manual, similar to when you look up the name of memory allocator when you want to allocate memory. The simplest routine in the *rpc* library used to make remote procedure calls is *callrpc().* It has eight parameters. The first is the name of the remote machine. The next three parameters are the program, version, and procedure numbers. The following two parameters define the argument of the *rpc* call, and the final two parameters are for the return value of the call. If it completes successfully, *callrpc()* returns zero, but nonzero otherwise. The exact meaning of the return codes is found in *<rpc/clnt.h>*, and is in fact an *enum clnt_stat* cast into an integer.

Since data types may be represented differently on different machines, *callrpc()* needs both the type of the *rpc* argument, as well as 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 is 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, which stands for User Datagram Protocol. Methods for adjusting the number of retries or for using a different protocol require you to use the lower layer of the *rpc* library, discussed later in this document. The remote server procedure corresponding to the above might look like this:

```
char *
nuser(indata)
        char *indata;
{
        static int nusers;

        /*
         * code here to compute the number of users
         * and place result in variable nusers
         */
        return((char *)&nusers);
}
```

It takes one argument, which is a pointer to the input of the remote procedure call (ignored in our example), and it returns a pointer to the result. In the current version of C, character pointers are the generic pointers, so both the input argument and the return value are cast to *char **.

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 would look like this:

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

char *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, *RUSERPROG, 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 types of the input to and output from the procedure.

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

Warning: the UDP transport mechanism can only deal with arguments and results fewer than 8 Kbytes in length.

Assigning Program Numbers

Program numbers are assigned in groups of 0x20000000 (536870912) according to the following chart:

```
        0 - 1fffffff          defined by sun
 20000000 - 3fffffff          defined by user
 40000000 - 5fffffff          transient
 60000000 - 7fffffff          reserved
 80000000 - 9fffffff          reserved
 a0000000 - bfffffff          reserved
 c0000000 - dfffffff          reserved
 e0000000 - ffffffff          reserved
```

Sun Microsystems administers the first group of numbers, which should be identical for all customers. If a customer develops an application that might be of general interest, that application should be given an assigned number in the first range. The second group of numbers is reserved for specific customer applications. This range is intended primarily for debugging new programs. The third group is reserved for applications that generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

To register a protocol specification, send a request by network mail to *sun!rpc,* or write to:

RPC Administrator
Sun Microsystems
2550 Garcia Ave.
Mountain View, CA 94043

Please include a complete protocol specification similar to those in this manual for *nfs* and *yp.* You will be given a unique program number in return.

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 orders or structure layout conventions, by always converting them to a network standard called *eXternal Data Representation* (*xdr*) before sending them over the wire. The process of converting from a particular machine representation to *xdr* format is called *serializing,* and the reverse process is called *deserializing.* The type field parameters of *callrpc()* and *registerrpc()* can be a built-in procedure like *xdr_u_long()* in the previous example, or a user-supplied one. *xdr* has these built-in type routines:

```
xdr_int()      xdr_u_int()      xdr_enum()
xdr_long()     xdr_u_long()     xdr_bool()
xdr_short()    xdr_u_short()    xdr_string()
```

As an example of a user-defined type routine, to send the structure:

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

call *callrpc()* as:

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

where *xdr_simple()* is written as:

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

xdr_simple(xdrsp, simplep)
        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 nonzero (true in the sense of C) if it completes successfully, and zero otherwise. A complete description of *xdr* is in the *XDR Protocol Specification*, so this section only gives a few examples of *xdr* implementation.

In addition to the built-in primitives, there are also the prefabricated building blocks:

```
xdr_array()      xdr_bytes()
xdr_reference()  xdr_union()
```

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

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

and make an *rpc* call such as:

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

with *xdr_varintarr()* defined as:

```
xdr_varintarr(xdrsp, arrp)
        XDR *xdrsp;
        struct varintarr *arrp;
{
        xdr_array(xdrsp, &arrp->data, &arrp->arrlnth, 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, then the following could also be used to send an array of length *SIZE:*

```
int intarr[SIZE];

xdr_intarr(xdrsp, intarr)
        XDR *xdrsp;
        int intarr[];
{
        int i;

        for (i = 0; i < SIZE; i++) {
        if (!xdr_int(xdrsp, &intarr[i]))
                                                return (0);
        }
        return (1);
}
```

xdr always converts quantities to 4-byte multiples when deserializing. Thus, if either of the examples above involved characters instead of integers, each character would occupy 32 bits. That is the reason for the *xdr* routine *xdr_bytes(),* which is like *xdr_array()* except that 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, which is the same as *xdr_bytes()* without the length parameter. On serializing it gets the string length from *strlen(),* and on deserializing it creates a null-terminated string.

Here is a final example that calls the previously written *xdr_simple()* as well as the built-in functions *xdr_string()* and *xdr_reference()*, which chase pointers:

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

xdr_finalexample(xdrsp, finalp)
        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);
}
```

3

Lowest Layer of *rpc*

Introduction

In the examples given so far, *rpc* takes care of many details automatically for you. In this chapter, we'll show you how you can change the defaults by using lower layers of the *rpc* library. It is assumed that you are familiar with sockets and the system calls for dealing with them. If not, consult the *CONVEX Interprocess Communication Programming Guide* located in the CONVEX Networking binder.

There are several occasions when you may need to use lower layers of *rpc*. First, you may need to use TCP. The higher layer uses UDP, which restricts *rpc* calls to 8 Kbytes of data. Using TCP permits calls to send long streams of data. For an example, see the section *Memory Allocation With xdr* in this chapter. Second, you may want to allocate and free memory while serializing or deserializing with *xdr* routines. There is no call at the higher level to let you free memory explicitly. For more explanation, see Chapter 4. Third, you may need to perform authentication on either the client or server side, by supplying credentials or verifying them. See the explanation in Chapter 4.

More on the Server Side

The server for the *nusers* program shown in Figure 3-1 does the same thing as the one using *registerrpc()* above, but is written using a lower layer of the *rpc* package.

Figure 3–1: *nusers* **Program**

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

main()
{
        SVCXPRT *transp;
        int 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");
```

```
}

nuser(rqstp, tranp)
        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 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;
        }
}
```

First, 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 *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.

When the user specifies *RPC_ANYSOCK* for a socket or gives an unbound socket, the system determines port numbers in the following way: when a server starts, it advertises to a port mapper daemon on its local machine, which picks a port number for the *rpc* procedure if the socket specified to *svcudp_create()* isn't already bound. When the *clntudp_create()* call is made with an unbound socket, the system queries the port mapper on the machine to which the call is being made, and gets the appropriate port number. If the port mapper is not running or has no port corresponding to the *rpc* call, the *rpc* call fails. Users can make *rpc* calls to the port mapper themselves. The appropriate procedure numbers are in the include file *<rpc/pmap_prot.h>*.

After creating an *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 *RUSERSPROG* from the port mapper's tables. Finally, 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. Also, registration is done on the program, rather than 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. The first is that procedure *NULLPROC* (currently zero) returns with no arguments. This can be used as a simple test for detecting if a remote program is running. Second, there is 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 is the *xdr* routine, and the third is a pointer to the data to be returned. Not illustrated above is how a server handles an *rpc* program that passes data. As an example, we can add a procedure *RUSERSPROC_BOOL*, which has an argument *nusers*, and returns *TRUE* or *FALSE* depending on whether there are nusers logged on. It would look like this:

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

        if (!svc_getargs(transp, xdr_u_int, &nuserquery) {
                svcerr_decode(transp);
                return;
        }
        /*
         * code 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 an *SVCXPRT* handle, the *xdr* routine, and a pointer to where the input is to be placed as arguments.

Memory Allocation With *xdr*

xdr routines not only do input and output, they also do memory allocation. This is why the second parameter of *xdr_array()* is a pointer to an array, rather than the array itself. If it is *NULL*, then *xdr_array()* allocates space for the array and returns a pointer to it, putting the size of the array in the third argument. As an example, consider the following *xdr* routine *xdr_chararr1()*, which deals with a fixed array of bytes with length *SIZE:*

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

```
        char *p;
        int len;

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

It might be called from a server like this:

```
char chararr[SIZE];

svc_getargs(transp, xdr_chararr1, chararr);
```

where *chararr* has already allocated space. If you want *xdr* to do the allocation, you would have to rewrite this routine in the following way:

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

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

Then the *rpc* call might look like this:

```
char *arrptr;

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

After using the character array, it can be freed with *svc_freeargs()*. In the routine *xdr_finalexample()* given earlier, if *finalp->string* was *NULL* in the call:

```
svc_getargs(transp, xdr_finalexample, &finalp);
```

then:

```
svc_freeargs(xdrsp, xdr_finalexample, &finalp);
```

frees the array allocated to hold *finalp->string;* otherwise, it frees nothing. The same is true for *finalp->simplep*.

To summarize, each *xdr* routine is responsible for serializing, deserializing, and allocating memory. When an *xdr* routine is called from *callrpc()*, the serializing part is used. When called from *svc_getargs()*, the deserializer is used. And when called from *svc_freeargs()*, the memory deallocator is used. When building simple examples like those in this section, a user doesn't have to worry about the three modes. The *xdr* reference manual has examples of more sophisticated *xdr* routines that determine which of the three modes they are in to function correctly.

Calling Side

When you use *callrpc()*; you have no control over the *rpc* delivery mechanism or the socket used to transport the data. To illustrate the layer of *rpc* that lets you adjust these parameters, consider the code shown in Figure 3-2 (this code enables you to call the *nusers* service).

Figure 3–2: Calling *nusers* Service

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

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

        if (argc < 2) {
                fprintf(stderr, "usage: nusers hostname\n");
                exit(-1);
        }
        if ((hp = gethostbyname(argv[1])) == NULL) {
                fprintf(stderr, "can't get addr for %s\n",argv[1]);
                exit(-1);
        }
        pertry_timeout.tv_sec = 3;
        pertry_timeout.tv_usec = 0;
        addrlen = sizeof(struct sockaddr_in);
        bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
                hp->h_length);
        server_addr.sin_family = AF_INET;
        server_addr.sin_port =  0;
        if ((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);
```

```
        }
        clnt_destroy(client);
}
```

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 get TCP (Transport Control Protocol), you would use *clnttcp_create()*.

The parameters to *clntudp_create()* are the server address, the length of the server address, the program number, the version number, a time-out value (between tries), 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()* time-out divided by the *clntudp_create()* time-out.

There is one thing to note when using the *clnt_destroy()* call. It deallocates any space associated with the *CLIENT* handle, but it does not close the socket associated with it, which was passed as an argument to *clntudp_create()*. The reason is that if multiple client handles are using the same socket, then it is possible to close one handle without destroying 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 time-out 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 would use this connection. The server side of an *rpc* call using TCP has *svcudp_create()* replaced by *svctcp_create()*.

4

Other *rpc* Features

This chapter discusses some other aspects of *rpc* that are occasionally useful.

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 a file descriptor, the *svc_run()* call won't work. The code for *svc_run()* is as follows:

```
void
svc_run()
{
        fd_set readfds;

        for (;;) {
                readfds = svc_fdset;
                switch (select(_rpc_dtablesize(), &readfds, NULL, NULL, NULL)) {

                case -1:
                        if (errno == EINTR)
                                continue;
                        perror("rstat: 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 socket(s) 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.

Broadcast *rpc*

The *portmapper* is a daemon that converts *rpc* program numbers into DARPA protocol port numbers; see *portmap*(8). You can't do broadcast *rpc* without the portmapper, *pmap*, in conjunction with standard *rpc* protocols. Here are the main differences between broadcast *rpc* and normal *rpc* calls:

- Normal *rpc* expects one answer, whereas broadcast *rpc* expects many answers (one or more answer from each responding machine).
- Broadcast *rpc* can only be supported by packet-oriented (connectionless) transport protocols like 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 portmap port. Thus, only services that register themselves with their portmapper are accessible via the broadcast *rpc* mechanism.

Broadcast *rpc* Synopsis

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

enum clnt_stat      clnt_stat;

clnt_stat =
clnt_broadcast(prog, vers, proc, xargs, argsp, xresults,
        resultsp, eachresult)
        u_long      prog;               /* program number */
        u_long      vers;               /* version number */
        u_long      proc;               /* procedure number */
        xdrproc_t   xargs;              /* xdr routine for args */
        caddr_t     argsp;              /* pointer to args */
        xdrproc_t   xresults;           /* xdr routine for results */
        caddr_t     resultsp;           /* pointer to results */
        bool_t      (*eachresult)();    /* call with each result gotten */
```

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)
        caddr_t resultsp;
        struct sockaddr_in *raddr;  /* addr of responding machine */
```

If *done* is *TRUE*, then 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*. To interpret *clnt_stat* errors, feed the error code to *clnt_perrno()*.

Batching

The *rpc* architecture is designed so that clients send a call message, and wait for servers to reply that the call succeeded. This implies that clients do not compute while servers are processing a call. This is inefficient if the client does not want or need an acknowledgment for every message sent. It is possible for clients to continue computing while waiting for a response, using *rpc* batch facilities.

rpc messages can be placed in a "pipeline" of calls to a desired server; this is called batching.

Batching assumes that: 1) each *rpc* call in the pipeline requires no response from the server, and the server does not send a response message; and 2) 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. Furthermore, the TCP/IP implementation can buffer many call messages, and send them to the server in one *write()* system call. This overlapped execution greatly decreases the interprocess 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 legitimate call in order to flush the pipeline.

A contrived example of batching follows. Assume a string rendering service (like a window system) has two similar calls: one renders a string and returns void results, and the other renders a string and remains silent. The service (using the TCP/IP transport) may look like Figure 4-1.

Figure 4–1: String Rendering Service

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

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");
}

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 he screwed up
                 */
                svcerr_decode(transp);
                break;
            }
            /*
             * call 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 the face of protocol errors
                 */
                break;
            }
            /*
             * call 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 the following attributes: 1) the result's *xdr* routine must be zero (*NULL*) and 2) the *rpc* call's time-out must be zero.

Figure 4-2 shows an example of a client that uses batching to render a bunch of strings; the batching is flushed when the client gets a null string.

Figure 4–2: Rendering Strings via Batching

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

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

        /* initial as in example 3.3
         */
        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);
}
```

Since the server sends no message, the clients cannot be notified of any of the failures that may occur. Therefore, clients are on their own when it comes to handling errors.

The above example was completed to render all of the (2000) lines in the file */etc/termcap*. The rendering service did nothing but throw the lines away. Timing results are as follows:

- Batched *rpc*—14 seconds
- Regular *rpc*—50 seconds

Running *fscanf()* on */etc/termcap* only requires six seconds. These timings show the advantage of protocols that allow for overlapped execution, although these protocols are often hard to design.

Authentication

In the examples presented so far, the caller never identified itself to the server, and the server never required an ID from the caller. Clearly, some network services, such as a network filesystem, require stronger security than what has been presented so far.

In reality, every *rpc* call is authenticated by the *rpc* package on the server, and similarly, the *rpc* client package generates and sends authentication parameters. Just as different transports (TCP/IP or UDP/IP) can be used when creating *rpc* clients and servers, different forms of authentication can be associated with *rpc* clients; the default authentication type used as a default is type *none*.

The authentication subsystem of the *rpc* package is open ended. That is, numerous types of authentication are easy to support. However, this section deals only with *unix* type authentication, which besides *none* is the only supported type.

Client Side

When a caller creates a new *rpc* client handle as in:

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

the appropriate transport instance defaults the associate authentication handle to be

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

The *rpc* client can choose to use *unix* style authentication by setting *clnt->cl_auth* after creating the *rpc* client handle:

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

This causes each *rpc* call associated with *clnt* to carry with it the following authentication credentials structure:

```
/*
 * Unix style 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 is in */
};
```

These fields are set by *authunix_create_default()* by invoking the appropriate system calls.

Since the *rpc* user created this new style of authentication, the user is responsible for destroying it with:

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

This should be done in all cases, to conserve memory.

Server Side

Service implementors have a harder time dealing with authentication issues since the *rpc* package passes the service dispatch routine a request that has an arbitrary authentication style associated with it. 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 number */
        struct opaque_auth
                        rq_cred;          /* raw credentials from wire */
        caddr_t rq_clntcred;        /* credentials (read only) */
};
```

The *rq_cred* is mostly opaque, except for one field of interest: the style 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:

- That the request's *rq_cred* is well-formed. Thus the service implementor may inspect the request's *rq_cred.oa_flavor* to determine which style of authentication the caller used. The service implementor may also wish to inspect the other fields of *rq_cred* if the style is not one of the styles supported by the *rpc* package.

- That the request's *rq_clntcred* field is either *NULL* or points to a well-formed structure that corresponds to a supported style of authentication credentials. Remember that

only *unix* style is currently supported, so (currently) *rq_clntcred* could be cast to a pointer to an *authunix_parms* structure. If *rq_clntcred* is *NULL*, the service implementor may wish to inspect the other (opaque) fields of *rq_cred* in case the service knows about a new type of authentication that the *rpc* package does not know about.

Our remote users service example can be extended so that it computes results for all users except UID 16 (Figure 4-3).

Figure 4–3: Extended Remote Users Service Example

```
nuser(rqstp, tranp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
        struct authunix_parms *unix_cred;
        int uid;
        unsigned long nusers;

        /*
         * we don't care about authentication for null proc
         */
        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:
                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 caller is allowed to call this proc
                 */
                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;
        }
}
```

A few things should be noted here. First, it is customary not to check the authentication parameters associated with the *NULLPROC* (procedure number zero). Second, if the authentication parameter's type is not suitable for your service, you should call *svcerr_weakauth()*. And finally, 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 relation between the *rpc* authentication package and the services; *rpc* deals only with authentication and not with individual service's access control. The services themselves must implement their own access control policies and reflect these policies as return statuses in their protocols.

Using *inetd*

An *rpc* server can be started from *inetd*. The only difference from the usual code is that *svcudp_create()* should be called as

```
transp = svcudp_create(0);
```

since *inet* passes a socket as file descriptor 0. Also, *svc_register()* should be called as

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

with the final flag as 0, since the program would already be registered by *inetd*. Remember that if you want to exit from the server process and return control to *inet*, you need to explicitly exit, since *svc_run()* never returns.

/etc/inetd.conf is the *inetd* configuration file. Figure 4-4 shows a sample */etc/inetd.conf* file:

Figure 4–4: Sample *letc/inetd.conf* File

```
#
# Internet server configuration database
#
ftp         stream   tcp    nowait  root    /usr/etc/in.ftpd           ftpd
telnet      stream   tcp    nowait  root    /usr/etc/in.telnetd        telnetd
shell       stream   tcp    nowait  root    /etc/in.rshd               rshd
login       stream   tcp    nowait  root    /etc/in.rlogind            rlogind
exec        stream   tcp    nowait  root    /usr/etc/in.rexecd         rexecd
syslog      dgram    udp    wait    root    /usr/etc/in.syslog         syslog
comsat      dgram    udp    wait    root    /usr/etc/in.comsat         comsat
talk        dgram    udp    wait    root    /usr/etc/in.talkd          talkd
rstatd      dgram    udp    wait    root    1-3 /usr/etc/rpc.rstatd    rstatd
rusersd     dgram    udp    wait    root    1-2 /usr/etc/rpc.rusersd   rusersd
walld       dgram    udp    wait    root    1 /usr/etc/rpc.rwalld      rwalld
mountd      dgram    udp    wait    root    1 /usr/etc/rpc.mountd      mountd
rquotad     dgram    udp    wait    root    1 /usr/etc/rpc.rquotad     rquotad
sprayd      dgram    udp    wait    root    1 /usr/etc/rpc.sprayd      sprayd
rexd        stream   tcp    wait    root    1 /usr/etc/rpc.rexd        rexd
echo        stream   tcp    nowait  root    internal
discard     stream   tcp    nowait  root    internal
chargen     stream   tcp    nowait  root    internal
daytime     stream   tcp    nowait  root    internal
time        stream   tcp    nowait  root    internal
echo        dgram    udp    wait    root    internal
discard     dgram    udp    wait    root    internal
chargen     dgram    udp    wait    root    internal
daytime     dgram    udp    wait    root    internal
time        dgram    udp    wait    root    internal
```

Each of the seven columns in the file is described below:

Column 1 Lists service name as listed in */etc/services*; used to index port numbers. If the service is an *rpc* program, it is listed in */etc/rpc*, not */etc/services*. In these cases, *inetd* searches */etc/rpc* to find the *rpc* program number. (If the service is an *rpc* program, a version number and pathname are listed in Column 6.)

Column 2 Used to select stream or datagram format.

Column 3 Used to select transport (stream format is almost always used with TCP sockets; datagrams almost always use UDP sockets).

Column 4 Determines whether service or *inetd* handles new data after port is initialized: *wait* signifies that *inetd* starts up the requested daemon and does not accept new data over the port (the running daemon handles new requests); *nowait* signifies that *inetd* handles new data. Typically, *tcp* connections use *nowait*; *udp*

connections use *wait* and eventually time themselves out to return control to *inetd*.

Column 5 Used to select program "owner"; root is the default, but any valid user account can be listed here.

Column 6 Lists deamon pathname. "Standard" listings contain a full pathname; "rpc" listings contain a version number (or range of version numbers) and a pathname; "internal" listings reference code built into *inetd*.

Column 7 Lists daemon name.

Note that you can add or delete listings as you add or delete functionality. For best results, do not modify "internal" listings.

If you add new *nfs* or *yp* servers or clients, you should reconfigure *inetd*. To do so, send *inetd* a *SIGHUP* signal as follows:

```
# ps ax | grep inetd
# kill -1 [process number found using previous instruction]
```

Instructions for adding new clients and servers are included in the *CONVEX Network File System System Manager's Guide*.

5

More Examples

Versions

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*, then a server that wants to support both versions would do 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(rqstp, tranp)
        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 put in variable nusers
                 */
                nusers2 = nusers;
                if (rqstp->rq_vers != RUSERSVERS_ORIG)
                        return;
                if (!svc_sendreply(transp, xdr_u_long, &nusers) {
                        fprintf(stderr, "can't reply to RPC call\n");
                        exit(1);
```

```
                } else
                if (!svc_sendreply(transp, xdr_u_short, &nusers2) {
                        fprintf(stderr, "can't reply to RPC call\n");
                        exit(1);
                }
                return;
        default:
                svcerr_noproc(transp);
                return;
        }
}
```

TCP

Figure 5-1 is an example that is essentially *rcp*. The initiator of the *rpc snd()* call takes its standard input and sends it to the server *rcv()*, which prints it on standard output. The *rpc* call uses TCP. This also illustrates an *xdr* procedure that behaves differently on serialization than on deserialization.

Figure 5–1: TCP Example

```
/*
 * 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(xdrs, fp)
        XDR *xdrs;
        FILE *fp;
{
        unsigned long size;
        char buf[BUFSIZ], *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), BUFSIZ,
                          fp)) == 0 && ferror(fp)) {
                                fprintf(stderr, "can't fread\n");
                                exit(1);
                        }
                }
                p = buf;
                if (!xdr_bytes(xdrs, &p, &size, BUFSIZ))
                        return 0;
                if (size == 0)
                        return 1;
                if (xdrs->x_op == XDR_DECODE) {
```

```
                    if (fwrite(buf, sizeof(char), size,
                      fp) != size) {
                            fprintf(stderr, "can't fwrite\n");
                            exit(1);
                    }
                }
        }
}

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

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

        if (argc < 2) {
                fprintf(stderr, "usage: %s servername\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);
        }
}

callrpctcp(host, prognum, procnum, versnum,
          inproc, in, outproc, out)
        char *host, *in, *out;
        xdrproc_t inproc, outproc;
{
        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) {
                fprintf(stderr, "can't get addr for '%s'\n", host);
                exit(-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");
                exit(-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;

        if ((transp = svctcp_create(RPC_ANYSOCK,
          BUFSIZ, BUFSIZ)) == NULL) {
                fprintf("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");
}

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");
                        exit(1);
                }
                return;
        case RCPPROC_FP:
                if (!svc_getargs(transp, xdr_rcp, stdout)) {
                        svcerr_decode(transp);
                        return;
                }
```

```
                if (!svc_sendreply(transp, xdr_void, 0)) {
                        fprintf(stderr, "can't reply\n");
                        return;
                }
                exit(0);
        default:
                svcerr_noproc(transp);
                return;
        }
}
```

Callback Procedures

Occasionally, it is useful to have a server become a client, and make an *rpc* call back the process which is its client. An example is remote debugging, where the client is a window system program, and the server is a debugger running on the remote machine. Most of the time, the user clicks a mouse button at the debugging window, which converts this to a debugger command, and then makes an *rpc* call 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, and the debugger wants to make an *rpc* call to the window program, so that it can inform the user that a breakpoint has been reached.

In order to do an *rpc* callback, you need a program number to make the *rpc* call on. Since this will be a dynamically generated program number, it should be in the transient range, 0x40000000 - 0x5fffffff. The routine *gettransient()* returns a valid program number in the transient range, and registers it with the portmapper. It only talks to the portmapper running on the same machine as the *gettransient()* routine itself. The call to *pmap_set()* is a test and set operation, in that it indivisibly tests whether a program number has already been registered, and if it has not, then reserves it. On return, the *sockp* argument contains a socket that can be used as the argument to an *svcudp_create()* or *svctcp_create()* call. Figure 5-2 is an *rpc* callback example.

Figure 5–2: *rpc* Callback Example

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.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, addr.sin_port))
                continue;
        return (prognum-1);
}
```

The pair of programs shown in Figures 5-3 and 5-4 illustrate how to use the *gettransient()* routine. The client makes an *rpc* call to the server, passing it a transient program number. Then the client waits around to receive a callback from the server at that program number. The server registers the program *EXAMPLEPROG*, so that it can receive the *rpc* call informing it of the callback program number. Then at some random time (on receiving an *ALRM* signal in this example), it sends a callback *rpc* call, using the program number it received earlier.

Figure 5–3: Using *gettransient* Routine, Example 1

```
/*
 * client
 */
#include <stdio.h>
#include <rpc/rpc.h>

int callback();
char hostname[256];

main(argc, argv)
        char **argv;
{
        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 (ans != RPC_SUCCESS) {
                fprintf(stderr, "call: ");
                clnt_perrno(ans);
                fprintf(stderr, "\n");
        }
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn't return\n");
}

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: rusersd\n");
                                exit(1);
                        }
                        exit(0);
                case 1:
                        if (!svc_getargs(transp, xdr_void, 0)) {
                                svcerr_decode(transp);
                                exit(1);
                        }
                        fprintf(stderr, "client got callback\n");
                        if (!svc_sendreply(transp, xdr_void, 0)) {
                                fprintf(stderr, "err: rusersd");
                                exit(1);
                        }
        }
}
```

Figure 5–4: Using *gettransient* Routine, Example 2

```
/*
 * server
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>

char *getnewprog();
char hostname[256];
int docallback();
int pnum;           /* program number for callback routine */

main(argc, argv)
        char **argv;
{
        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 return\n");
}

char *
getnewprog(pnump)
        char *pnump;
{
        pnum = *(int *)pnump;
        return NULL;
}

docallback()
{
        int ans;

        ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0,
                xdr_void, 0);
        if (ans != 0) {
                fprintf(stderr, "server: ");
                clnt_perrno(ans);
                fprintf(stderr, "\n");
        }
}
```

A

Synopsis of *rpc* Routines

auth_destroy()

```
void
auth_destroy(auth)
        AUTH *auth;
```

A macro that destroys the authentication information associated with *auth*. Destruction usually involves deallocation of private data structures. The use of *auth* is undefined after calling *auth_destroy()*.

authnone_create()

```
AUTH *
authnone_create()
```

Creates and returns an *rpc* authentication handle that passes no usable authentication information with each remote procedure call.

authunix_create()

```
AUTH *
authunix_create(host, uid, gid, len, aup_gids)
        char *host;
        int uid, gid, len, *aup_gids;
```

Creates and returns an *rpc* authentication handle that contains authentication information. The parameter *host* is the name of the machine on which the information was created; *uid* is the user's user ID; *gid* is the user's current group ID; *len* and *aup_gids* refer to a counted array of groups to which the user belongs. It is easy to impersonate a user.

authunix_create_default()

```
AUTH *
authunix_create_default()
```

Calls *authunix_create()* with the appropriate parameters.

callrpc()

```
callrpc(host,prognum,versnum,procnum,inproc,in,outproc,out)
        char *host;
        u_long prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
```

Calls the remote procedure associated with *prognum, versnum,* and *procnum* on the machine, *host.* The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results. This routine returns zero if it succeeds, or the value of *enum clnt_stat* cast to an integer if it fails. The routine *clnt_perrno()* is handy for translating failure statuses into messages. Warning: calling remote procedures with this routine uses UDP/IP as a transport; see *clntudp_create()* for restrictions.

clnt_broadcast()

```
enum clnt_stat
clnt_broadcast(prognum, versnum, procnum,
  inproc, in, outproc, out, eachresult)
        u_long prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
        resultproc_t eachresult;
```

Like *callrpc(),* except the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls *eachresult(),* whose form is

```
eachresult(out, addr)
        char *out;
        struct sockaddr_in *addr;
```

where *out* is the same as *out* passed to *clnt_broadcast(),* except that the remote procedure's output is decoded there; *addr* points to the address of the machine that sent the results. If *eachresult()* returns zero, *clnt_broadcast()* waits for more replies; otherwise it returns with appropriate status.

clnt_call()

```
enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
        CLIENT *clnt; u_long procnum;
        xdrproc_t inproc, outproc;
        char *in, *out;
        struct timeval tout;
```

A macro that calls the remote procedure *procnum* associated with the client handle, *clnt,* which is obtained with an *rpc* client creation routine such as *clntudp_create().* The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results; *tout* is the time allowed for results to come back.

clnt_control()

```
bool_t
clnt_control(cl, req, info)
        CLIENT *cl;
        char *info;
```

A macro used to change or retrieve information about a client object. *req* indicates the type of operation, and *info* is a pointer to the information.

Table A-1 shows the supported values of *req* for UDP and TCP. The argument types and the value function are also included.

Table A-1: *req* Values for UDP and TCP

Value	Argument Type	Function
CLSET_TIMEOUT	struct timeval	set total timeout
CLGET_TIMEOUT	struct timeval	get total timeout If you set timeout using *clnt_control*, the timeout parameter passed to *clnt_call* is ignored in all future calls.
CLGET_SERVER_ADDR	struct sockaddr	get server's address

Table A-2 shows the values for *req* that are only valid for UDP.

Table A-2: *req* Values for UDP

Value	Argument Type	Function
CLSET_RETRY_TIMEOUT	struct timevalset	the retry timeout
CLGET_RETRY_TIMEOUT	struct timevalget	the retry timeout Retry timeout is the time that *udp rpc* waits for the server to reply before retransmitting the request.

clnt_create()

```
CLIENT *
clnt_create(host, prog, vers, proto)
        char *host;
        u_long prog, vers;
        char *proto;
```

Generic client creation routine. *host* identifies the name of the remote host where the server is located. *proto* indicates which kind of transport protocol to use. The currently supported values for this field are udp and tcp. Default timeouts are set, but can be modified using *clnt_control*.

Because *udp* based *rpc* messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

clnt_destroy()

```
clnt_destroy(clnt)
        CLIENT *clnt;
```

A macro that destroys the client's *rpc* handle. Destruction usually involves deallocation of private data structures, including *clnt* itself. Use of *clnt* is undefined after calling *clnt_destroy()*. If the *rpc* library opened the associated socket, it will close it also. Otherwise, the socket remains open.

clnt_freeres()

```
clnt_freeres(clnt, outproc, out)
        CLIENT *clnt;
        xdrproc_t outproc;
        char *out;
```

A macro that frees any data allocated by the *rpc/xdr* system when it decoded the results of an *rpc* call. The parameter *out* is the address of the results, and *outproc* is the *xdr* routine describing the results in simple primitives. This routine returns one if the results were successfully freed, zero otherwise.

clnt_geterr()

```
void
clnt_geterr(clnt, errp)
        CLIENT *clnt;
        struct rpc_err *errp;
```

A macro that copies the error structure out of the client handle to the structure at address *errp*.

clnt_pcreateerror()

```
void
clnt_pcreateerror(s)
        char *s;
```

Prints a message to standard error indicating why a client *rpc* handle could not be created. The message is prepended with string *s* and a colon. Used when a *clntraw_create()*, *clntraw_create()*, *clnttcp_create()*, or *clntudp_create()* call fails.

clnt_perrno()

```
void
clnt_perrno(stat)
        enum clnt_stat stat;
```

Prints a message to standard error corresponding to the condition indicated by *stat*. Used after *callrpc()*.

clnt_perror()

```
clnt_perror(clnt, s)
        CLIENT *clnt;
        char *s;
```

Prints a message to standard error indicating why an *rpc* call failed; *clnt* is the handle used to do the call. The message is prepended with string *s* and a colon. Used after *clnt_call()*.

clnt_spcreateerror()

```
char *
clnt_spcreateerror(s);
        char *s;
```

Like *clnt_pcreateerror*, except that it returns a string instead of printing to the standard error. The return value points to static data that is overwritten on each call.

clnt_sperrno()

```
char *
        clnt_sperrno(stat)
        enum clnt_stat stat;
```

Takes the same arguments as *clnt_perrno*, but instead of sending a message to the standard error indicating why an RPC call failed, it returns a pointer to a string that contains the message. The string ends with a newline.

clnt_sperrno is used instead of *clnt_perrno* if the program doesn't have a standard error (as a program running as a server quite likely doesn't), or if the programmer doesn't want the message to be output with *printf*, or if a message format different from that supported by *clnt_perrno* is to be used.

Note: unlike *clnt_sperror* and *clnt_spcreaterror*, *clnt_sperrno* does not return pointer to static data so the result will not get overwritten on each call.

clnt_sperror()

```
char *
clnt_sperror(rpch, s)
        CLIENT *rpch;
        char *s;
```

Like *clnt_perror*, except that (like *clnt_sperrno*) it returns a string instead of printing to standard error. The return value points to static data that is overwritten on each call.

clnt_syslog()

```
clnt_syslog(rpch, s)
        CLIENT *rpch;
        char *s;
```

Like *clnt_perror*, except that it prints the error string to the standard *syslog (8)* file instead of printing to the standard error.

clntraw_create()

```
CLIENT *
clntraw_create(prognum, versnum)
        u_long prognum, versnum;
```

This routine creates a toy *rpc* client for the remote program *prognums*, version *versnum*. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding *rpc* server should live in the same address space; see *svcraw_create()*. This allows simulation of *rpc* and acquisition of *rpc* overheads, such as round-trip times, without any kernel interference. This routine returns *NULL* if it fails.

clnttcp_create()

```
CLIENT *
clnttcp_create(addr,prognum,versnum,sockp,sendsz,recvsz)
        struct sockaddr_in *addr;
        u_long prognum, versnum;
        int *sockp;
        u_int sendsz, recvsz;
```

This routine creates an *rpc* client for the remote program *prognum*, version *versnum;* the client uses TCP/IP as a transport. The remote program is located at Internet address **addr*. If *addr->sin_port* is zero, then it is set to the actual port that the remote program is listening on (the remote *portmap* service is consulted for this information). The parameter **sockp* is a socket; if it is *RPC_ANYSOCK*, then this routine opens a new one and sets **sockp*. Since TCP-based *rpc* uses buffered I/O, the user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz;* values of zero choose suitable defaults. This routine returns *NULL* if it fails.

clntudp_create()

```
CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
        struct sockaddr_in *addr;
        u_long prognum, versnum;
        struct timeval wait;
        int *sockp;
```

This routine creates an *rpc* client for the remote program *prognum*, version *versnum;* the client uses UDP/IP as a transport. The remote program is located at Internet address **addr*. If *addr->sin_port* is zero, then it is set to actual port that the remote program is listening on (the remote *portmap* service is consulted for this information). The parameter **sockp* is a socket; if it is *RPC_ANYSOCK*, then this routine opens a new one and sets **sockp*. The UDP transport resends the call message in intervals of *wait* time until a response is received or until the call times out. The total time for the call to time out is specified by *clnt_call()*. Warning: since UDP-based *rpc* messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

get_myaddress()

```
void
get_myaddress(addr)
        struct sockaddr_in *addr;
```

Stuffs the machine's IP address into **addr*, without consulting the library routines that deal with */etc/hosts*. The port number is always set to *htons(PMAPPORT)*.

pmap_getmaps()

```
struct pmaplist *
pmap_getmaps(addr)
        struct sockaddr_in *addr;
```

A user interface to the *portmap* service, which returns a list of the current *rpc* program-to-port mappings on the host located at IP address **addr*. This routine can return *NULL*. The command *rpcinfo -p* uses this routine.

pmap_getport()

```
u_short
pmap_getport(addr, prognum, versnum, protocol)
        struct sockaddr_in *addr;
        u_long prognum, versnum, protocol;
```

A user interface to the *portmap* service, which returns the port number where a program resides that supports program number *prognum*, version *versnum*, and speaks the transport protocol associated with *protocol*. A return value of zero means that the mapping does not exist or that the *rpc* system failured to contact the remote *portmap* service. In the latter case, the global variable *rpc_createerr* contains the *rpc* status.

pmap_rmtcall()

```
enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum,
  inproc, in, outproc, out, tout, portp)
        struct sockaddr_in *addr;
        u_long prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
        struct timeval tout;
        u_long *portp;
```

A user interface to the *portmap* service, which instructs *portmap* on the host at IP address **addr* to make an *rpc* call on your behalf to a procedure on that host. The parameter **portp* will be modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in *callrpc()* and *clnt_call()*. This procedure should be used for a "ping" and nothing else. See also *clnt_broadcast()*.

pmap_set()

```
pmap_set(prognum, versnum, protocol, port)
        u_long prognum, versnum, protocol;
        u_short port;
```

A user interface to the *portmap* service, which establishes a mapping between the triple *[prognum,versnum,protocol]* and *port* on the machine's *portmap* service. The value of protocol is most likely *IPPROTO_UDP* or *IPPROTO_TCP*. This routine returns one if it succeeds, zero otherwise. Automatically done by *svc_register()*.

pmap_unset()

```
pmap_unset(prognum, versnum)
        u_long prognum, versnum;
```

A user interface to the *portmap* service, which destroys all mappings between the triple *[prognum,versnum,*]* and *ports* on the machine's *portmap* service. This routine returns one if it succeeds, zero otherwise.

registerrpc()

```
registerrpc(prognum,versnum,procnum,procname,inproc,outproc)
        u_long prognum, versnum, procnum;
        char *(*procname)();
        xdrproc_t inproc, outproc;
```

Registers procedure *procname* with the *rpc* service package. If a request arrives for program *prognum,* version *versnum,* and procedure *procnum, procname* is called with a pointer to its parameter(s); *progname* should return a pointer to its static result(s); *inproc* is used to decode the parameters while *outproc* is used to encode the results. This routine returns zero if the registration succeeded, –1 otherwise. Warning: remote procedures registered in this form are accessed using the UDP/IP transport; see *svcudp_create()* for restrictions.

rpc_createerr

```
struct rpc_createerrrpc_createerr;
```

A global variable whose value is set by any *rpc* client creation routine that does not succeed. Use the routine *clnt_pcreateerror()* to print the reason why.

svc_destroy()

```
svc_destroy(xprt)
        SVCXPRT *xprt;
```

A macro that destroys the *rpc* service transport handle, *xprt*. Destruction usually involves deallocation of private data structures, including *xprt* itself. Use of *xprt* is undefined after calling this routine.

svc_fds

```
int svc_fds;
```

Similar to *svc_fdset*, but limited to 32 descriptors. This interface is obsoleted by *svc_fdset*.

svc_fdset

```
fd_set svc_fdset;
```

A global variable reflecting the *rpc* service side's read file descriptor bit mask; a copy of it is suitable as a parameter to the *select* system call. This is only of interest if a service implementor does not call *svc_run*, but rather does his own asynchronous event processing. This variable is read-only (do **not** pass its address to *select*), yet it may change after calls to *svc_getreqset* or any creation routines.

svc_freeargs()

```
svc_freeargs(xprt, inproc, in)
        SVCXPRT *xprt;
        xdrproc_t inproc;
        char *in;
```

A macro that frees any data allocated by the *rpc/xdr* system when it decoded the arguments to a service procedure using *svc_getargs()*.

This routine returns one if the results were successfully freed, zero otherwise.

svc_getargs()

```
svc_getargs(xprt, inproc, in)
        SVCXPRT *xprt;
        xdrproc_t inproc;
        char *in;
```

A macro that decodes the arguments of an *rpc* request associated with the *rpc* service transport handle, *xprt*. The parameter *in* is the address where the arguments will be placed; *inproc* is the *xdr* routine used to decode the arguments. This routine returns one if decoding succeeds, zero otherwise.

svc_getcaller()

```
struct sockaddr_in
svc_getcaller(xprt)
        SVCXPRT *xprt;
```

The approved way of getting the network address of the caller of a procedure associated with the *rpc* service transport handle, *xprt*.

svc_getreq()

```
svc_getreq(rdfds)
        int rdfds;
```

Similar to *svc_getreqset*, but limited to 32 descriptors. This interface is obsoleted by *svc_getreqset*

svc_getreqset()

```
svc_getreqset(rdfds)
        fd_set *rdfds;
```

This routine is only of interest if a service implementor does not call *svc_run*, but instead implements custom asynchronous event processing. It is called when the *select* system call has determined that an *rpc* request has arrived on some *rpc* socket(s); *rdfds* is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of *rdfds* have been serviced.

svc_register()

```
svc_register(xprt, prognum, versnum, dispatch, protocol)
        SVCXPRT *xprt;
        u_long prognum, versnum;
        void (*dispatch)();
        u_long protocol;
```

Associates *prognum* and *versnum* with the service dispatch procedure, *dispatch()*. If *protocol* is zero, the service is not registered with the *portmap* service. If *protocol* is non-zero, then a mapping of the triple *[prognum,versnum,protocol]* to *xprt->xp_port* is established with the local *portmap* service (generally *protocol* is zero, *IPPROTO_UDP* or *IPPROTO_TCP*). The procedure *dispatch()* has the following form:

```
        dispatch(request, xprt)
        struct svc_req *request;
        SVCXPRT *xprt;
```

The *svc_register()* routine returns one if it succeeds, zero otherwise.

svc_run()

This routine never returns. It waits for *rpc* requests to arrive, and calls the appropriate service procedure using *svc_getreq()* when one arrives. This procedure is usually waiting for a *select()* system call to return.

svc_sendreply()

```
svc_sendreply(xprt, outproc, out)
        SVCXPRT *xprt;
        xdrproc_t outproc;
        char *out;
```

Called by an *rpc* service's dispatch routine to send the results of a remote procedure call. The parameter *xprt* is the caller's associated transport handle; *outproc* is the *xdr* routine used to encode the results; and *out* is the address of the results. This routine returns one if it succeeds, zero otherwise.

svc_unregister()

```
void
svc_unregister(prognum, versnum)
        u_long prognum, versnum;
```

Removes all mapping of the double */prognum,versnum/* to dispatch routines, and of the triple */prognum,versnum,*/* to port number.

svcerr_auth()

```
void
svcerr_auth(xprt, why)
        SVCXPRT *xprt;
        enum auth_stat why;
```

Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

svcerr_decode()

```
void
svcerr_decode(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine that can't successfully decode its parameters. See also *svc_getargs()*.

svcerr_noproc()

```
void
svcerr_noproc(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine that doesn't implement the desired procedure number the caller requested.

svcerr_noprog()

```
void
svcerr_noprog(xprt)
        SVCXPRT *xprt;
```

Called when the desired program is not registered with the *rpc* package. Service implementors usually don't need this routine.

svcerr_progvers()

```
void
svcerr_progvers(xprt)
        SVCXPRT *xprt;
```

Called when the desired version of a program is not registered with the *rpc* package. Service implementors usually don't need this routine.

svcerr_systemerr()

```
void
svcerr_systemerr(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

svcerr_weakauth()

```
void
svcerr_weakauth(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient (but correct) authentication parameters. The routine calls *svcerr_auth(xprt,AUTH_TOOWEAK).*

svcraw_create()

```
SVCXPRT *
svcraw_create()
```

This routine creates a toy *rpc* service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding *rpc* client should live in the same address space; see *clntraw_create().* This routine allows simulation of *rpc* and acquisition of *rpc* overheads (such as round trip times), without any kernel interference. This routine returns *NULL* if it fails.

svctcp_create()

```
SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
        int sock;
        u_int send_buf_size, recv_buf_size;
```

This routine creates a TCP/IP-based *rpc* service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be *RPC_ANYSOCK*, in which case a new socket is created. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port. Upon completion, *xprt->xp_sock* is the transport's socket number, and *xprt->xp_port* is the transport's port number. This routine returns *NULL* if it fails. Since TCP-based *rpc* uses buffered I/O, users may specify the size of the *send* and *receive* buffers; values of zero choose suitable defaults.

svcfd_create()

```
SVCXPRT *
svcfd_create(fd, sensize, recvsize)
        int fd;
        u_int sendsize;
        u_int recvsize;
```

Creates a service on top of any open descriptor. Typically, this descriptor is a connected socket for a stream protocol such as TCP. *sendsize* and *recvsize* indicate sizes for the send and receive buffers. If they are zero, a reasonable default is chosen.

svcudp_create()

```
SVCXPRT *
svcudp_create(sock)
        int sock;
```

This routine creates a UDP/IP-based *rpc* service transport, to which it returns a pointer. The transport is associated with the socket *sock,* which may be *RPC_ANYSOCK,* in which case a new socket is created. If the socket is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, *xprt->xp_sock* is the transport's socket number, and *xprt->xp_port* is the transport's port number. This routine returns *NULL* if it fails. Warning: since UDP-based *rpc* messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

xdr_accepted_reply()

```
xdr_accepted_reply(xdrs, ar)
        XDR *xdrs;
        struct accepted_reply *ar;
```

Used for describing *rpc* messages, externally. This routine is useful for users who wish to generate *rpc*-style messages without using the *rpc* package.

xdr_array()

```
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
        XDR *xdrs;
        char **arrp;
        u_int *sizep, maxsize, elsize;
        xdrproc_t elproc;
```

A filter primitive that translates between arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *sizep* is the address of the element count of the array; this element count cannot exceed *maxsize.* The parameter *elsize* is the *sizeof()* each of the array's elements, and *elproc* is an *xdr* filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

xdr_authunix_parms()

```
xdr_authunix_parms(xdrs, aupp)
        XDR *xdrs;
        struct authunix_parms *aupp;
```

Used for describing UNIX credentials, externally. This routine is useful for users who wish to generate these credentials without using the *rpc* authentication package.

xdr_bool()

```
xdr_bool(xdrs, bp)
        XDR *xdrs;
        bool_t *bp;
```

A filter primitive that translates between Booleans (C integers) and their external representations. When encoding data, this filter produces values of either one or zero. This routine returns one if it succeeds, zero otherwise.

xdr_bytes()

```
xdr_bytes(xdrs, sp, sizep, maxsize)
        XDR *xdrs;
        char **sp;
        u_int *sizep, maxsize;
```

A filter primitive that translates between counted byte strings and their external representations. The parameter *sp* is the address of the string pointer. The length of the string is located at address *sizep;* strings cannot be longer than *maxsize.* This routine returns one if it succeeds, zero otherwise.

xdr_callhdr()

```
void
xdr_callhdr(xdrs, chdr)
        XDR *xdrs;
        struct rpc_msg *chdr;
```

Used for describing *rpc* messages, externally. This routine is useful for users who wish to generate *rpc*-style messages without using the *rpc* package.

xdr_callmsg()

```
xdr_callmsg(xdrs, cmsg)
        XDR *xdrs;
        struct rpc_msg *cmsg;
```

Used for describing *rpc* messages, externally. This routine is useful for users who wish to generate *rpc*-style messages without using the *rpc* package.

xdr_char()

```
xdr_char(xdrs, cp)
        XDR *xdrs;
        char *cp;
```

A filter primitive that translates between C characters and their external representations. This routine returns one if it succeeds, zero otherwise. Note: encoded characters are not packed, and occupy 4 bytes each. For arrays of characters, it is worthwhile to consider *xdr_bytes*, *xdr_opaque*, or *xdr_string*.

xdr_double()

```
xdr_double(xdrs, dp)
        XDR *xdrs;
        double *dp;
```

A filter primitive that translates between C *double* precision numbers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_enum()

```
xdr_enum(xdrs, ep)
        XDR *xdrs;
        enum_t *ep;
```

A filter primitive that translates between C *enum*s (actually integers) and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_float()

```
xdr_float(xdrs, fp)
        XDR *xdrs;
        float *fp;
```

A filter primitive that translates between C *float*s and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_hyper()

```
xdr_hyper(xdrs, hp)
        XDR *xdrs;
        hyper *hp;
```

A filter primitive that translates between C *hyper* (really CONVEX long long) integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_inline()

```
long *
xdr_inline(xdrs, len)
        XDR *xdrs;
        int len;
```

A macro that invokes the in-line routine associated with the *xdr* stream, *xdrs*. The routine returns a pointer to a contiguous piece of the stream's buffer; *len* is the byte length of the desired buffer. Note that pointer is cast to *long ** . Warning: *xdr_inline()* may return *NULL* (0) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

xdr_int()

```
xdr_int(xdrs, ip)
        XDR *xdrs;
        int *ip;
```

A filter primitive that translates between C integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_long()

```
xdr_long(xdrs, lp)
        XDR *xdrs;
        long *lp;
```

A filter primitive that translates between C *long* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_opaque()

```
xdr_opaque(xdrs, cp, cnt)
        XDR *xdrs;
        char *cp;
        u_int cnt;
```

A filter primitive that translates between fixed size opaque data and its external representation. The parameter *cp* is the address of the opaque object, and *cnt* is its size in bytes. This routine returns one if it succeeds, zero otherwise.

xdr_opaque_auth()

```
xdr_opaque_auth(xdrs, ap)
        XDR *xdrs;
        struct opaque_auth *ap;
```

Used for describing *rpc* messages, externally. This routine is useful for users who wish to generate *rpc*-style messages without using the *rpc* package.

xdr_pmap()

```
xdr_pmap(xdrs, regs)
        XDR *xdrs;
        struct pmap *regs;
```

Used for describing parameters to various *portmap* procedures, externally. This routine is useful for users who wish to generate these parameters without using the *pmap* interface.

xdr_pointer()

```
xdr_pointer(xdrs, objpp, objsize, xdrobj)
        XDR *xdrs;
        char **objpp;
        u_int objsize;
        xdrproc_t xdrobj;
```

Like *xdr_reference* except that *xdr_pointer* serializes NULL pointers, whereas *xdr_reference* does not. Thus *xdr_pointer* can *xdr* recursive data structures, such as binary trees or linked lists, correctly, whereas *xdr_reference* fails.

xdr_pmaplist()

```
xdr_pmaplist(xdrs, rp)
        XDR *xdrs;
        struct pmaplist **rp;
```

Used for describing a list of port mappings, externally. This routine is useful for users who wish to generate these parameters without using the *pmap* interface.

xdr_reference()

```
xdr_reference(xdrs, pp, size, proc)
        XDR *xdrs;
        char **pp;
        u_int size;
        xdrproc_t proc;
```

A primitive that provides pointer chasing within structures. The parameter *pp* is the address of the pointer; *size* is the *sizeof()* the structure that **pp* points to; and *proc* is an *xdr* procedure that filters the structure between its C form and its external representation. This routine returns one if it succeeds, zero otherwise.

xdr_rejected_reply()

```
xdr_rejected_reply(xdrs, rr)
        XDR *xdrs;
        struct rejected_reply *rr;
```

Used for describing *rpc* messages, externally. This routine is useful for users who wish to generate *rpc*-style messages without using the *rpc* package.

xdr_replymsg()

```
xdr_replymsg(xdrs, rmsg)
        XDR *xdrs;
        struct rpc_msg *rmsg;
```

Used for describing *rpc* messages, externally. This routine is useful for users who wish to generate *rpc* style messages without using the *rpc* package.

xdr_short()

```
xdr_short(xdrs, sp)
        XDR *xdrs;
        short *sp;
```

A filter primitive that translates between C *short* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_string()

```
xdr_string(xdrs, sp, maxsize)
        XDR *xdrs;
        char **sp;
        u_int maxsize;
```

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot be longer than *maxsize*. Note that *sp* is the address of the string's pointer. This routine returns one if it succeeds, zero otherwise.

xdr_u_char()

```
xdr_u_char(xdrs, ucp)
        XDR *xdrs;
        unsigned char *ucp;
```

A filter primitive that translates between *unsigned* C characters and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_hyper()

```
xdr_u_hyper(xdrs, hp)
        XDR *xdrs;
        u_hyper *hp;
```

A filter primitive that translates between C *unsigned hyper* (really CONVEX unsigned long long) integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_int()

```
xdr_u_int(xdrs, up)
        XDR *xdrs;
        unsigned *up;
```

A filter primitive that translates between C *unsigned* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_long()

```
xdr_u_long(xdrs, ulp)
        XDR *xdrs;
        unsigned long *ulp;
```

A filter primitive that translates between C *unsigned long* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_short()

```
xdr_u_short(xdrs, usp)
        XDR *xdrs;
        unsigned short *usp;
```

A filter primitive that translates between C *unsigned short* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_union()

```
xdr_union(xdrs, dscmp, unp, choices, dfault)
        XDR *xdrs;
        int *dscmp;
        char *unp;
        struct xdr_discrim *choices;
        xdrproc_t dfault;
```

A filter primitive that translates between a discriminated C *union* and its corresponding external representation. The parameter *dscmp* is the address of the union's discriminant, while *unp* is the address of the union. This routine returns one if it succeeds, zero otherwise.

xdr_vector()

```
xdr_vector(xdrs, arrp, size, elsize, elproc)
        XDR *xdrs;
        char *arrp;
        u_int size, elsize;
        xdrproc_t elproc;
```

A filter primitive that translates between fixed-length arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *size* is is the element count of the array. The parameter *elsize* is the *sizeof* each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

xdr_void()

```
xdr_void()
```

This routine always returns one.

xdr_wrapstring()

```
xdr_wrapstring(xdrs, sp)
        XDR *xdrs;
        char **sp;
```

A primitive that calls *xdr_string(xdrs,sp,MAXUNSIGNED);* where *MAXUNSIGNED* is the maximum value of an unsigned integer. This is handy because the *rpc* package passes only two parameters to *xdr* routines, whereas *xdr_string()*, one of the most frequently used primitives, requires three parameters. This routine returns one if it succeeds, zero otherwise.

xprt_register()

```
void
xprt_register(xprt)
        SVCXPRT *xprt;
```

After *rpc* service transport handles are created, they should register themselves with the *rpc* service package. This routine modifies the global variable *svc_fds*. Service implementors usually don't need this routine.

xprt_unregister()

```
void
xprt_unregister(xprt)
        SVCXPRT *xprt;
```

Before an *rpc* service transport handle is destroyed, it should unregister itself with the *rpc* service package. This routine modifies the global variable *svc_fds*. Service implementors usually don't need this routine.

Index

Index

D

E

F

N

O

P

R

S

Y