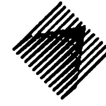
TITAN 

NETWORK FILE SYSTEM (NFS) MANUAL

Release 1.0

Ardent Computer Corporation

Part Number: 340-0012-02

Copyright © 1987, 1988
Ardent Computer Corporation
All Rights Reserved.

This document has been provided pursuant to an agreement with Ardent Computer Corporation containing restrictions on its disclosure, duplication, and use. This document contains confidential and proprietary information constituting valuable trade secrets and is protected by federal copyright law as an unpublished work. This document (or any portion thereof) may not be: (a) disclosed to third parties; (b) copied in any form except as permitted by the agreement; or (c) used for any purpose not authorized by the agreement.

Restricted Rights Legend for Agencies of the U.S. Department of Defense

Use, duplication or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013 of the DoD Supplement to the Federal Acquisition Regulations. Ardent Computer Corporation, 880 West Maude Avenue, Sunnyvale, California 94086.

Restricted Rights Legend for civilian agencies of the U.S. Government

Use, reproduction or disclosure is subject to restrictions set forth in subparagraph (a) through (d) of the Commercial Computer Software—Restricted Rights clause at 52.227-19 of the Federal Acquisitions Regulations and the limitations set forth in Ardent's standard commercial agreement for this software. Unpublished—rights reserved under the copyright laws of the United States.

Doré™ and Titan™ are trademarks of Ardent Computer Corporation. UNIX® is a registered trademark of AT&T. VAX® is a registered trademark of Digital Equipment Corporation. Sun Workstation® is a registered trademark of Sun Microsystems, Inc.

CONTENTS

Preface

Section 1 NFS Programming Guide

1 NFS Overview

Introduction 1-1
Computing Environments 1-2
Terms and Concepts 1-3
Examples of NFS 1-4
Mounting a Remote Filesystem 1-4
Exporting a Filesystem 1-5
Administering a Server Machine 1-5
Administering a Client Machine 1-6
NFS Architecture 1-6
Transparent Information Access 1-6
Different Machines and Operating Systems 1-6
Easily Extensible 1-7
Easy Network Administration 1-7
Reliable 1-8
High Performance 1-8
The TITAN Implementation 1-9
The Operating System Interface 1-9
The FSS Interface 1-9
The NFS Interface 1-11

2 RPC Programming

Introduction 2-1
Introductory Examples 2-2
Highest Layer 2-3
Intermediate Layer 2-3

Assigning Program Numbers 2-6
Passing Arbitrary Data Types 2-6
Lower Layers of RPC 2-9
More on the Server Side 2-9
Memory Allocation with XDR 2-12
The Calling Side 2-13
Other RPC Features 2-15
Select on the Server Side 2-15
Broadcast RPC 2-16
Broadcast RPC Synopsis 2-16
Batching 2-17
Authentication 2-21
The Client Side 2-21
The Server Side 2-22
Using Inetd 2-25
More Examples 2-25
Versions 2-25
TCP 2-27
Callback Procedures 2-29
Synopsis of RPC Routines 2-33
auth_destroy( ) 2-33
authnone_create( ) 2-33
authunix_create( ) 2-34
authunix_create_default( ) 2-34
callrpc( ) 2-34
clnt_broadcast( ) 2-34
clnt_call( ) 2-35
clnt_destroy( ) 2-35
clnt_freeres( ) 2-36
clnt_geterr( ) 2-36
clnt_pcreateerror( ) 2-36
clnt_perrno( ) 2-36
clnt_perror( ) 2-37
clntraw_create( ) 2-37
clnttcp_create( ) 2-37
clntudp_create( ) 2-38
get_myaddress( ) 2-38
pmap_getmaps( ) 2-38
pmap_getport( ) 2-38
pmap_rmtcall( ) 2-39
pmap_set( ) 2-39
pmap_unset( ) 2-39
registerrpc( ) 2-40
rpc_createerr 2-40
svc_destroy( ) 2-40
svc_fds 2-40

svc_freeargs( ) 2-41
svc_getargs( ) 2-41
svc_getcaller( ) 2-41
svc_getreq( ) 2-41
svc_register( ) 2-42
svc_run( ) 2-42
svc_sendreply( ) 2-42
svc_unregister( ) 2-43
svcerr_auth( ) 2-43
svcerr_decode( ) 2-43
svcerr_noproc( ) 2-43
svcerr_noprog( ) 2-43
svcerr_progvers( ) 2-44
svcerr_systemerr( ) 2-44
svcerr_weakauth( ) 2-44
svcraw_create( ) 2-44
svctcp_create( ) 2-45
svcudp_create( ) 2-45
xdr_accepted_reply( ) 2-45
xdr_array( ) 2-46
xdr_authunix_parms( ) 2-46
xdr_bool( ) 2-46
xdr_bytes( ) 2-47
xdr_callhdr( ) 2-47
xdr_callmsg( ) 2-47
xdr_double( ) 2-47
xdr_enum( ) 2-48
xdr_float( ) 2-48
xdr_inline( ) 2-48
xdr_int( ) 2-48
xdr_long( ) 2-49
xdr_opaque( ) 2-49
xdr_opaque_auth( ) 2-49
xdr_pmap( ) 2-49
xdr_pmaplist( ) 2-50
xdr_reference( ) 2-50
xdr_rejected_reply( ) 2-50
xdr_replymsg( ) 2-50
xdr_short( ) 2-51
xdr_string( ) 2-51
xdr_u_int( ) 2-51
xdr_u_long( ) 2-51
xdr_u_short( ) 2-52
xdr_union( ) 2-52
xdr_void( ) 2-52
xdr_wrapstring( ) 2-52

xprt_register( ) 2-53
xprt_unregister( ) 2-53

3 RPC Protocol Specification

Introduction 3-1
The RPC Model 3-1
Transports and Semantics 3-2
Binding and Rendezvous Independence 3-3
Message Authentication 3-3
Requirements 3-3
Remote Programs and Procedures 3-4
Authentication 3-5
Program Number Assignment 3-5
Other Uses and Abuses of the RPC Protocol 3-6
Batching 3-6
Broadcast RPC 3-6
The RPC Message Protocol 3-6
Authentication Parameter Specification 3-10
Null Authentication 3-10
UNIX Authentication 3-10
Record Marking Standard 3-11
Port Mapper Program Protocol 3-11
The Port Mapper RPC Protocol 3-12

4 XDR Protocol Specification

Introduction 4-1
Justification 4-2
XDR Library Primitives 4-7
Number Filters 4-7
Floating Point Filters 4-8
Enumeration Filters 4-8
No Data 4-9
Constructed Data Type Filters 4-9
Strings 4-9
Byte Arrays 4-10
Arrays 4-11
Examples 4-11
Example A 4-11
Example B 4-12
Example C 4-12
Opaque Data 4-13
Fixed Sized Arrays 4-14

Discriminated Unions 4-14
Example D 4-15
Pointers 4-16
Example E 4-17
Pointer Semantics and XDR 4-17
Non-filter Primitives 4-18
XDR Operation Directions 4-18
XDR Stream Access 4-18
Standard I/O Streams 4-19
Memory Streams 4-19
Record (TCP/IP) Streams 4-20
XDR Stream Implementation 4-21
The XDR Object 4-21
XDR Standard 4-23
Basic Block Size 4-23
Integer 4-24
Unsigned Integer 4-24
Enumerations 4-24
Booleans 4-24
Hyper Integer and Hyper Unsigned 4-24
Floating Point and Double Precision 4-25
Opaque Data 4-25
Counted Byte Strings 4-26
Fixed Arrays 4-26
Counted Arrays 4-27
Structures 4-27
Discriminated Unions 4-27
Missing Specifications 4-28
Library Primitive/XDR Standard Cross Reference 4-28
Advanced Topics 4-29
Linked Lists 4-29
The Record Marking Standard 4-33
Synopsis of XDR Routines 4-33
xdr_array( ) 4-33
xdr_bool( ) 4-34
xdr_bytes( ) 4-34
xdr_destroy( ) 4-34
xdr_double( ) 4-34
xdr_enum( ) 4-35
xdr_float( ) 4-35
xdr_getpos( ) 4-35
xdr_inline( ) 4-35
xdr_int( ) 4-36
xdr_long( ) 4-36
xdr_opaque( ) 4-36
xdr_reference( ) 4-36

xdr_setpos( ) 4-37
xdr_short( ) 4-37
xdr_string( ) 4-37
xdr_u_int( ) 4-37
xdr_u_long( ) 4-38
xdr_u_short( ) 4-38
xdr_union( ) 4-38
xdr_void( ) 4-38
xdr_wrapstring( ) 4-39
xdrmem_create( ) 4-39
xdrrec_create( ) 4-39
xdrrec_endofrecord( ) 4-40
xdrrec_eof( ) 4-40
xdrrec_skiprecord( ) 4-40
xdrstdio_create( ) 4-40

5 NFS Protocol Specification

Introduction 5-1
Remote Procedure Call 5-1
External Data Representation 5-2
Stateless Servers 5-3
NFS Protocol Definition 5-3
Server/Client Relationship 5-4
Permission Issues 5-5
RPC Information 5-6
Authentication 5-6
Protocols 5-6
Constants 5-6
Port Number 5-6
Sizes 5-6
MAXDATA 5-7
MAXPATHLEN 5-7
MAXNAMLEN 5-7
COOKIESIZE 5-7
FHSIZE 5-7
Basic Data Types 5-7
Stat 5-7
Ftype 5-10
Fhandle 5-10
Timeval 5-10
Fattr 5-10
Sattr 5-12
Filename 5-12
Path 5-12

Attrstat 5-12
Diropargs 5-13
Diropres 5-13
Server Procedures 5-13
Do Nothing (Procedure 0, Version 2) 5-14
Get File Attributes (Procedure 1, Version 2) 5-14
Set File Attributes (Procedure 2, Version 2) 5-14
Get Filesystem Root (Procedure 3, Version 2) 5-15
Look Up File Name (Procedure 4, Version 2) 5-15
Read From Symbolic Link (Procedure 5, Version 2) 5-15
Read From File (Procedure 6, Version 2) 5-16
Write to Cache (Procedure 7, Version 2) 5-16
Write to File (Procedure 8, Version 2) 5-16
Create File (Procedure 9, Version 2) 5-17
Remove File (Procedure 10, Version 2) 5-17
Rename File (Procedure 11, Version 2) 5-17
Create Link to File (Procedure 12, Version 2) 5-17
Create Symbolic Link (Procedure 13, Version 2) 5-18
Create Directory (Procedure 14, Version 2) 5-18
Remove Directory (Procedure 15, Version 2) 5-18
Read From Directory (Procedure 16, Version 2) 5-19
Get Filesystem Attributes (Procedure 17, Version 2) 5-19
Mount Protocol Definition 5-20
Version 1 5-21
RPC Information 5-21
Authentication 5-21
Protocols 5-21
Constants 5-21
Port Number 5-21
Sizes 5-21
MNTPATHLEN 5-22
MNTNAMLEN 5-22
FHSIZE 5-22
Basic Data Types 5-22
Fhandle 5-22
Fhstatus 5-22
Dirpath 5-23
Name 5-23
Server Procedures 5-23
Do Nothing (Procedure 0, Version 1) 5-23
Add Mount Entry (Procedure 1, Version 1) 5-23
Return Mount Entries (Procedure 2, Version 1) 5-24
Remove Mount Entry (Procedure 3, Version 1) 5-24
Remove All Mount Entries (Procedure 4, Version 1) 5-24
Return Export List (Procedure 5, Version 1) 5-24

6 Yellow Pages Protocol Specification

Introduction 6-1
  RPC — Remote Procedure Call 6-2
  XDR — External Data Representation 6-2
YP Database Servers 6-3
  Map Structure 6-3
  Operations on Maps 6-4
    Match Operation 6-4
    Enumeration Operation 6-4
    Entire Map Retrieval 6-4
    Map Update 6-5
  Master and Slave YP Database Servers 6-5
  Map Propagation and Consistency 6-5
    Functions to Aid in Map Propagation 6-5
  Domains 6-6
  Limitations 6-6
    Map Update Within the YP 6-6
    Version Commitment Across Multiple Requests 6-6
    Guaranteed Global Consistency 6-6
    Access Control 6-7
  YP Database Server Protocol Definition 6-7
    RPC Constants 6-7
    YPMAXRECORD 6-7
    YPMAXDOMAIN 6-7
    YPMAXMAP 6-7
    YPMAXPEER 6-7
  Remote Procedure Return Values 6-8
  Basic Data Structures 6-8
    domainname 6-8
    mapname 6-8
    peername 6-8
    keydat 6-8
    valdat 6-8
    ypmap_parms 6-8
    ypreq_xfr 6-9
    ypresp_val 6-9
    ypresp_key_val 6-9
    ypresp_master 6-9
    ypresp_order 6-9
    ypresp_all 6-10
    ypresp_xfr 6-10
    ypmaplist 6-10
  YP Database Server Remote Procedures 6-10
    Do Nothing (Procedure 0, Version 2) 6-10
    Do You Serve This Domain? (Procedure 1, Version 2) 6-10

Answer Only If You Serve This Domain (Procedure 2, Version 2) 6-11
Return Value of a Key (Procedure 3, Version 2) 6-11
Get First Key-Value Pair in Map (Procedure 4, Version 2) 6-11
Get Next Key-Value Pair in Map (Procedure 5, Version 2) 6-12
Transfer Map (Procedure 6, Version 2) 6-12
Reinitialize Internal State (Procedure 7, Version 2) 6-12
Get All Key-Value Pairs in Map (Procedure 8, Version 2) 6-12
Get Map Master Name (Procedure 9, Version 2) 6-13
Get Map Order Number (Procedure 10, Version 2) 6-13
Get All Maps in Domain (Procedure 11, Version 2) 6-13
YP Binders 6-13
Introduction 6-13
YP Binder Protocol Definition 6-14
RPC Constants 6-14
YPMAXDOMAIN 6-15
ypbind_resptype 6-15
ypbinderr 6-15
Basic Data Structures 6-15
domainname 6-15
ypbind_binding 6-16
ypbind_resp 6-16
ypbind_setdom 6-16
YP Binder Remote Procedures 6-16
Do Nothing (Procedure 0, Version 2) 6-17
Get Current Binding for a Domain (Procedure 1, Version 2) 6-17
Set Domain Binding (Procedure 2, Version 2) 6-17

7 NFS System Administration

Introduction 7-1
TOPS and Network Services 7-2
Debugging TOPS In The Network Environment 7-3
NFS: The Network File System 7-3
How The NFS Works 7-4
Becoming An NFS Server 7-4
Remote Mounting A File System 7-5
Debugging the Network File System 7-5
General Hints 7-6
Remote Mount Failed 7-7
Error Messages 7-9

Programs Hung 7-12
Everything Works Slowly 7-12
Incompatibilities With Earlier UNIX Versions 7-13
No SU Over The Network 7-13
File Operations Not Supported 7-14
Cannot Access Remote Devices 7-14
Clock Skew In User Programs 7-14
YP: The Yellow Pages Service 7-16
The YP Map 7-16
The YP Domain 7-16
Masters And Slaves 7-17
Yellow Pages Overview 7-17
Yellow Pages Installation and Administration 7-19
Setting Up A Master YP Server 7-19
Setting Up A YP Client 7-20
Setting Up A Slave YP Server 7-22
Modifying Existing Maps 7-22
Propagation Of A YP Map 7-24
Making New YP Maps 7-25
Adding A New YP Server 7-26
Changing The Master Server 7-26
Debugging A Yellow Pages Client 7-27
On Client: Commands Hang 7-28
On Client: YP Service Unavailable 7-29
On Client: Ypbind Crashes 7-31
On Client: Ypwhich Inconsistent 7-32
Debugging A Yellow Pages Server 7-32
Different Versions Of A YP Map 7-32
Ypserv Crashes 7-33
Yellow Pages Policies 7-34
Security Under The Yellow Pages 7-34
Global And Local Database Files 7-34
Security Implications 7-35
Special YP Password Change 7-36
Manual Pages Covering Security 7-36
What If The Yellow Pages Is Not Used? 7-36
Adding a New User to a Machine 7-36
Edit the /etc/passwd File 7-36
Make A Home Directory 7-38
The New User's Environment 7-39

Section 2 NFS Reference Manual

Permuted Index

User and System Administration Commands

domainname	1
makedbm	2
mountd	3
nfsd	4
nfsstat	5
portmap	6
rpcinfo	7
showmount	8
ypcat	9
ypinit	10
ypmake	11
ypmatch	12
yppasswd	13
yppasswdd	14
yppoll	15
yppush	16
ypserv	17
ypset	19
ypwhich	20
ypxfr	21

Subroutines

dbm	22
getdomainname	24
gethostent	25
getrpcent	27
getrpcport	28
ypclnt	29
yppasswd	30

File Formats

exports 34
hosts 35
rmtab 36
rpc 37
ypfiles 38

List of Figures

Figure 1-1. Mainframe Computing Environment 1-2
Figure 1-2. Network Computing Environment 1-3
Figure 1-3. The FSS Interface 1-10
Figure 2-1. Network Communications with RPC 2-2

List of Tables

Table 4-1. C/XDR Type Association 4-28

Index

PREFACE

This manual is intended to provide programmers and system administrators with the reference information they need to use the Network File System (NFS) effectively. This book is divided into two parts:

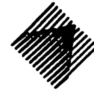
- Section 1—**Programming Guide**
  - Chapter 1—*NFS Overview*
    A general introduction to concepts and architecture.
  - Chapter 2—*RPC Programming*
    A guide for programmers who wish to write network applications using remote procedure calls, thus avoiding low-level system primitives. Readers must be familiar with the C programming language, and should have a working knowledge of network theory.
  - Chapter 3—*RPC Protocol Specification*
    A reference guide for system programmers implementing the NFS on new machines. It is of little interest to programmers writing network applications.
  - Chapter 4—*XDR Protocol Specification*
    A guide for programmers writing complicated applications using remote procedure calls that pass complicated data across the network. It is also a reference guide for system programmers implementing the TITAN NFS on new machines.
  - Chapter 5—*NFS Protocol Specification*
    A reference guide for system programmers implementing the NFS on new machines. It is of little interest to programmers writing network applications.
  - Chapter 6—*Yellow Pages Protocol Specification*
    A reference guide for system programmers implementing a Yellow Pages database facility on new machines. It is of little interest to programmers writing network applications.

- Chapter 7—*Network Services System Administration*
  A reference guide for system administrators implementing and administering the NFS and Yellow Pages database facility on new machines.

- Section 2—**Programmer's Reference Manual**

  Standard reference manual pages for the NFS. These include user and system administration commands, as well as system calls, library subroutines, and file formats.

NFS OVERVIEW



CHAPTER ONE

This chapter gives an overview of the Network File System (NFS), which allows users to mount directories across the network and then to treat remote files as if they were local. Advanced users may wish to skip the first part, which is elementary, and read the examples of how the NFS works. Casual users may not be interested in the third section, which discusses network file system architecture and the TITAN implementation.

Introduction

The Network File System 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.

A distributed network of machines provides more aggregate computing power than a mainframe computer, with far less variation in response time over the course of the day. Thus, a network is generally more cost-effective than a central mainframe. However, a mainframe has often been preferred for large programming projects and database applications because all files can be stored on a single machine.

Those who work with unconnected personal computers know the inconveniences resulting from data fragmentation. Even in a network environment, sharing programs and data has sometimes been difficult. Either files 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 becomes confusing as incompatible changes are made to separate copies.

To solve these problems, the NFS permits client systems to access shared files on a remote system. Client machines request resources provided by other machines, which are called servers. A server machine makes particular filesystems available. Client

machines can mount these as local filesystems. Thus, users access remote files as if they were present on the local machine.

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.

Computing Environments

The current computing environment in many businesses and universities often looks like the diagram in Figure 1-1.

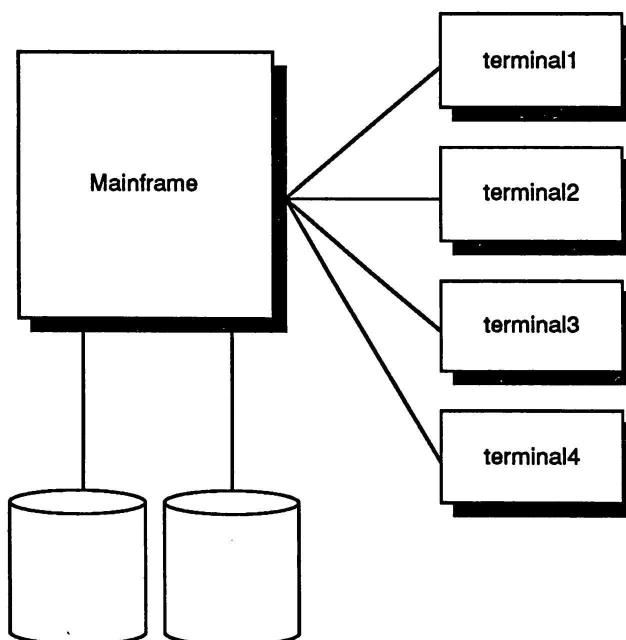


Figure 1-1. Mainframe Computing Environment

The major problem with this environment is competition for CPU cycles. A workstation environment solves that problem, but introduces more disk drives into the picture. A network of workstations looks like the diagram in Figure 1-2. The goal with NFS is to make all disks available as needed. Individual workstations have access to all information residing anywhere on the network. Printers may also be available on the network.

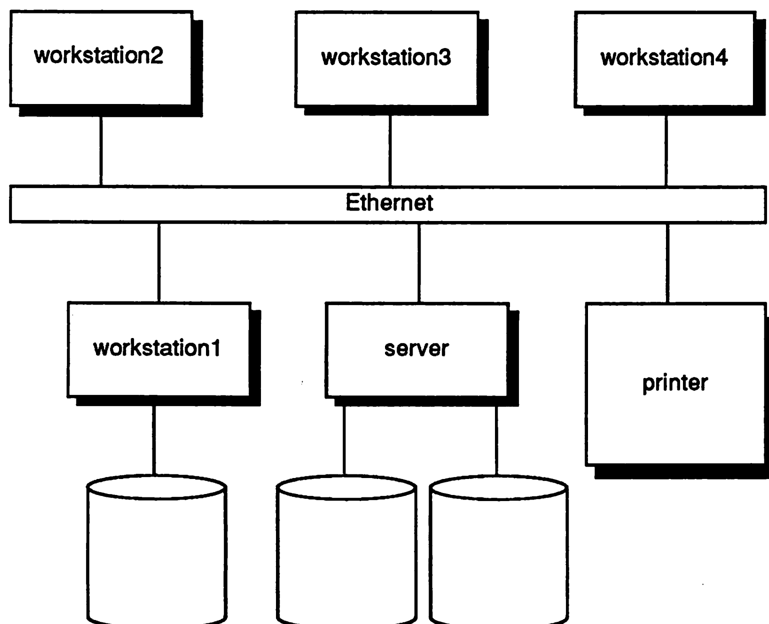


Figure 1-2. Network Computing Environment

Terms and Concepts

A machine which provides resources to the network is a *server*, while a machine which 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*, while 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 structure and contents of the filesystem (called *filesystem data*).

The Remote Procedure Call (RPC) facility provides a mechanism whereby one process (the *caller* process) can cause another process (the *server* process) to 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 need to exist 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 standard I/O library for interprocess

communication, on one machine or across a network. Thus programmers now have standardized access to these facilities without having to be concerned about the low-level details of any particular implementation.

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.

The Network File System (NFS) is composed of a modified UNIX kernel, a set of library routines, and a collection of utility commands. The NFS presents a network client with a complete remote filesystem. Since NFS is largely transparent to the user, this document provides information on subjects of which an NFS user may be unaware. The NFS is an open system that can accommodate other machines on the net, even non-UNIX systems, without compromising security.

Examples of NFS

The following examples illustrate how you mount, export, and administer file systems in a network.

Mounting a Remote Filesystem

Suppose that a user wants to read some on-line manual pages. These pages are not available on the client machine, called *client*, but are available on a machine called *docserv*. The directory containing the manuals can be mounted as follows:

```
client#  mount -f NFS docserv:/usr/man /usr/man
```

Note that the user must be root in order to issue a *mount* command. The *man* command can now be used whenever it is required and as it accesses the manual pages from the */usr/man* directory it will transparently be using the copies located on the machine *docserv*. If the *df* command is run after the remote filesystem has been mounted, its output will look something like this:

```
/           (/dev/dsk/c1d0s0 ):     1636 blocks     1424 i-nodes
/usr        (/dev/dsk/c1d0s2 ):     5368 blocks     4480 i-nodes
/usr/man   (docserv:/usr/man ):    36364 blocks        0 i-nodes
```

Exporting a Filesystem

Suppose that two users on different systems need to work together on a programming project. The source code is on the first user's machine, *leader* in the directory */usr/proj*.

Suppose that after creating the proper directory, the second user tries to remote mount the directory */usr/proj*. Unless the directory has been specifically exported, the remote mount will fail with a "permission denied" message.

To export the directory, the first user must become superuser and edit the file */etc/exports*. Place the following line in */etc/exports*:

```
/usr/proj        junior
```

if the second user is on a machine named *junior*. Without the specification of *junior*, any system on the network is allowed to mount the directory */usr/proj* remotely. The NFS mount request server *mountd*(1M) reads the */etc/exports* file (if one exists) whenever it receives a request for a remote mount. Now the second user can remote mount the source directory by issuing this command:

```
junior# mount -f NFS senior:/usr/proj /usr/proj
```

Now both users are able to access and change files on */usr/proj*. NFS only provides access to the files. Concurrency control must be provided with *sccs*(1) or some other source code control system.

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. Filesystems are exported (that is, made available) by placing appropriate lines in the */etc/exports* file. Here is a sample */etc/exports* file for a typical server machine:

```
/
/usr
/usr/proj        theteam
```

The pathnames specified in */etc/exports* must be real filesystems: that is, directory mount points for disk devices. The root filesystem must be exported if directories directly under root, such as */lib* are to be made available to NFS clients. A "netgroup", such as **theteam,** may be specified after the filesystem, in which case remote mounts are limited to machines that are members of this

netgroup. Netgroups are defined in */etc/netgroups* and are documented in *netgroup(4)*. The *showmount*(1M) command shows which filesystems have been remote mounted.

Administering a Client Machine

The *mount*(1M) command is the tool for administering a client system. System administrators usually set up NFS on a client machine so that users see all the necessary filesystems already mounted by the time they login. The file */etc/fstab* is a convenient place to keep the associations between remote machine file systems and the local mount points. So for example, in */etc/fstab* lines like the following may appear:

```
/dev/dsk/c1d0s2 /usr
docserv:/reference/rel3/usr/man /usr/man - NFS,hard
```

The second line enables the system administrator, or a user to type:

```
client#  mount /usr/man
```

and the remote association to machine *docserv* and its manual pages will be made for the client system.

NFS Architecture

This section describes in a general way the design goals and features of the 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 is no visible 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

The NFS readily exchanges data between different machines and operating systems.

Easily Extensible

A distributed system must have an architecture that allows integration of new software technologies without disturbing the existing software environment. To allow this, the NFS provides network services, rather than a new network operating system. That is, the 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 TITAN operating system (TOPS) supports a convenient set of maintenance commands which have been developed over the years for UNIX™*. 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.

The most obvious use of the YP is for administration of */etc/passwd*. Since the NFS uses a UNIX protection scheme across the network, it is best to have a common */etc/passwd* database for all machines on the network. The YP allows a single point of administration and gives all machines access to a recent version of the data, whether or not it is stored 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.

The YP interface is implemented using RPC and XDR, so the service is available to non-UNIX operating systems. YP servers do not interpret data, so it is possible for new databases to take advantage of the YP service without modifying the servers.

* UNIX is a registered trademark of AT&T.

Reliable

The file server protocol is 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-stop 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 returns to the net. 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.

High Performance

The flexibility of the NFS allows configuration for a variety of cost and performance trade-offs. For example, configuring servers with large, high-performance disks, and clients with no 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 achieve 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.

Several performance enhancements have been made to the NFS, such as "fast paths" for frequent operations, asynchronous service of multiple requests, disk block caches, and asynchronous read-ahead and write-behind. Using caches and read-ahead on both client and server effectively increases the cache size and read-ahead distance. Cache use 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 TITAN Implementation

In the TITAN implementation of the NFS, three entities must be considered: the operating system interface; the logical file system, or File System Switch (FSS) interface; and, the NFS interface.

The Operating System Interface

The UNIX operating system interface has not been modified for the TOPS implementation of the NFS, ensuring compatibility with existing applications.

The FSS Interface

The FSS interface is a set of routines within the System V kernel that separates filesystem operations from the semantics of their implementation. Above the FSS interface, the operating system deals with generic file objects; below this interface, the filesystem specific data structures implement a file object. The FSS interface permits the NFS filesystem support to be integrated into the kernel with minimal modifications. Under the FSS a filesystem implementation provides access to filesystem data on a local device.

The remote filesystem 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 1-3 shows the flow of a request from a client (at the top left) to a server machine where the data requested lives on a local disk. The boxes labeled *filesys* refer to those functions within the operating system that provide access to data on the system's attached disks.

In the case of access through a local filesystem, requests are directed to filesystem data on devices connected to the client machine. In the case of access through a remote filesystem the request is passed through the RPC and XDR layers onto the network. The current implementation uses UDP/IP protocols and Ethernet. On the server side, requests are passed through the RPC and XDR layers to an NFS server. This path is retraced to return results.

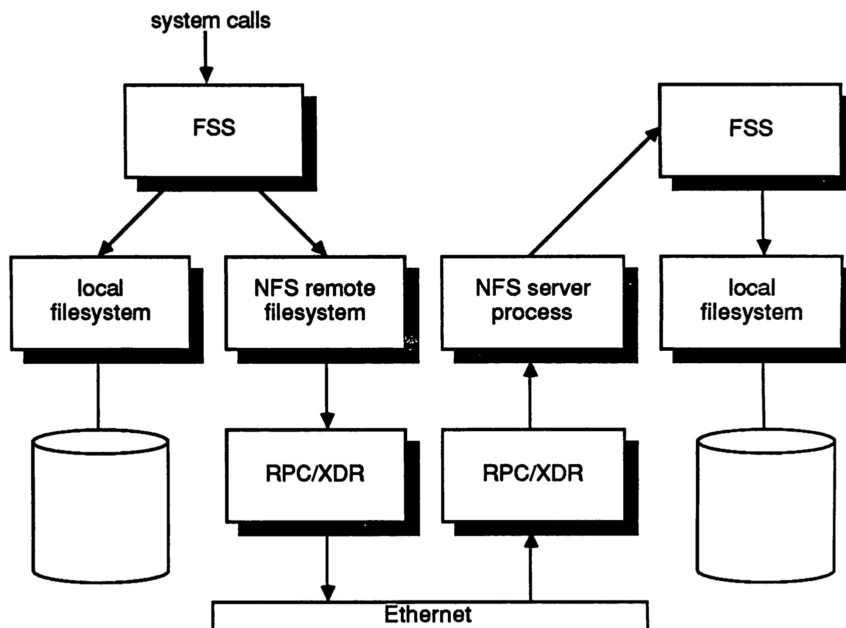


Figure 1-3. The FSS Interface

The TITAN implementation of the NFS provides five types of transparency:

(1) *Filesystem Type:* The FSS, permits an operating system to interface transparently to a variety of filesystem types.

(2) *Filesystem Location:* Since there is no differentiation in the interface to a local or a remote filesystem implementation, the location of filesystem data is transparent.

(3) *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.

(4) *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.

(5) *Network Type:* RPC and XDR can be implemented for a variety of network and internet protocols, thereby making the network type-transparent.

In the TITAN implementation, 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 the NFS without having to appeal to a system administrator, or configure a different system on their workstation.

The NFS Interface

The NFS interface itself is open and can be used by anyone wishing to implement an NFS client or server for the network. The 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*). A client builds its view of the filesystems available on the network with the *mount* command. Many commands are also provided for the YP database facility.

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 TOPS interface uses an *open* operation, but the information is remembered by the client for use in later NFS operations.

The stateless nature of an NFS server causes a problem when an application *unlink*s an open file. This operation achieves the effect of a temporary file that is automatically removed when the application terminates. If the file in question is served by the NFS, the *unlink* removes the file, since the server does not remember that the file is open. Thus, subsequent operations on the file fail. In order to avoid state on the server, the client operating system detects the situation, renames the file rather than unlinking it, and unlinks the file when the application terminates. In certain failure cases, this leaves unwanted "temporary" files on the server; these files should be removed as a part of periodic filesystem maintenance.

Another example of how the NFS provides a friendly interface without introducing state is the *mount* command. A client of the 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 the NFS and build its view of the filesystem on the network via 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. However, a server will not act as an intermediary between a client and another server. Instead, a client may ask what remotely mounted filesystems are available to the server 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 has an impact on 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 circularity in the service arrangements; this is better than detection or avoidance schemes.

The 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 traditional UNIX file protection. Although the RPC definition allows other authentication schemes, their use may have adverse effects on transparency.

Although the NFS is UNIX-friendly, it does not support all filesystem operations. For example, the "special file" abstraction of devices is not supported for remote filesystems because the interface to devices would greatly complicate the NFS interface. Other incompatibilities are due to the fact that NFS servers are stateless. For example, file locking and guaranteed **append_mode** are not

supported for the remote case by the NFS. These services are provided through other, transparent, network services.

The omission of certain features from the NFS preserves the stateless implementation of servers and defines a simple, general interface. 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" the NFS.

RPC PROGRAMMING

CHAPTER TWO

This chapter is intended for programmers who wish to write network applications using remote procedure calls (explained below), thus avoiding low-level system primitives. The reader must be familiar with the C programming language, and should have a working knowledge of network theory.

Introduction

Programs which 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 used by the NFS 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 the service requested, sends back the reply, and the procedure call returns to the client.

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. The user need not be aware that RPC is being used, since the call is simply made in a program, just as *malloc( )* would be called.

NOTE
rnusers is not a supported RPC library routine. It is only used as an example in this chapter.

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 needing to know about low level system primitives.

The lowest layer is used for more sophisticated applications, which may want to alter the defaults of the routines. At this layer,

system primitives used for transmitting RPC messages can be explicitly manipulated. This level should be avoided if possible.

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.

Figure 2-1 presents a diagram of the RPC paradigm.

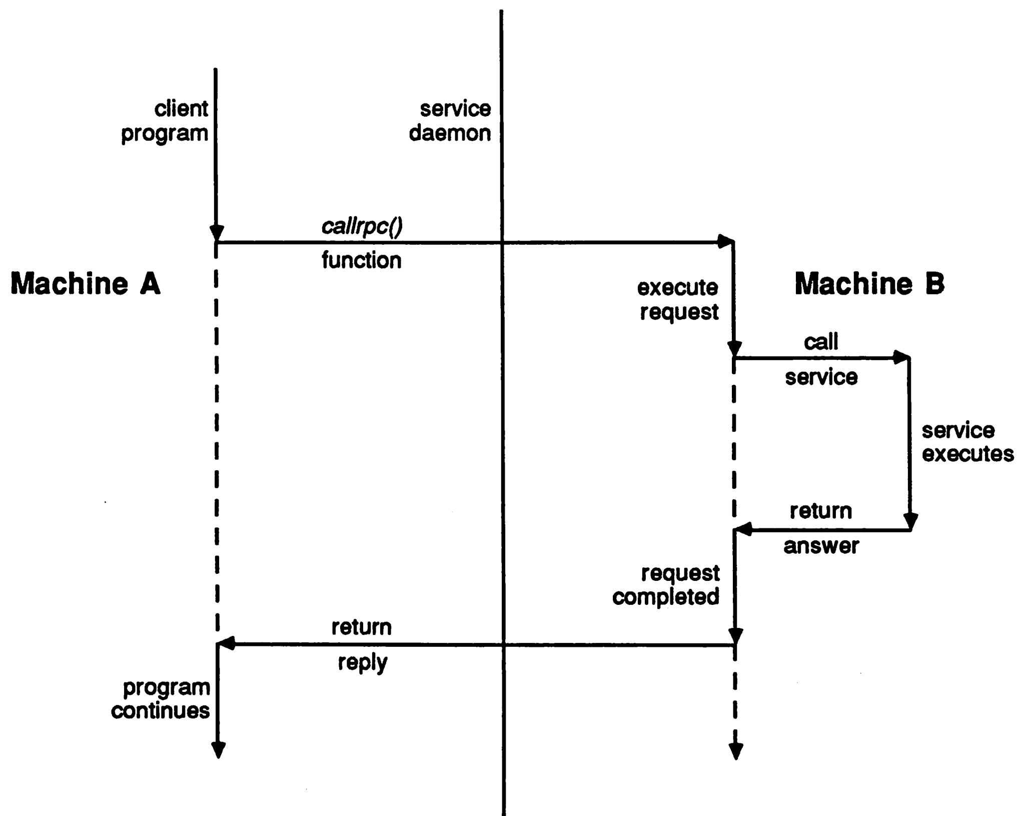


Figure 2-1. Network Communications with RPC

Introductory Examples

This section contains some examples using the RPC library routines.

Highest Layer

Consider a program that needs to know how many users are logged into a remote machine. This can be done by calling the example library routine *rnusers( )*:

```
#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 like *rnusers( )* are included in the C library *librpc.a*. Thus, this program could be compiled with

```
$ cc program.c -lrpc
```

Some library routines are *rstat( )* to gather remote performance statistics, and *ypmatch( )* to glean information from the Yellow Pages (YP). The YP library routines are documented on the manual page **ypclnt**(3N) in Section 2 of this manual.

Intermediate Layer

The simplest interface, which 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("number of users on %s is %d\n", argv[1], nusers);
        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 a procedure is to be called to find the number of remote users, the appropriate program, version, and procedure numbers are looked up in a manual, in a similar manner to looking up the name of a memory allocator when memory is to be allocated.

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( )* includes *xdr_u_long* as its first return parameter, which says that the result is of type **unsigned long**, and **&nusers** as its second return parameter, which is a pointer to where the long result will be placed. Since **RUSERSPROC_NUM** takes no argument, the argument parameter of *callrpc( )* is **xdr_void**.

After trying several times to deliver a message, *callrpc( )* returns with an error code if it gets no answer. 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 the use of the lower layer of the RPC library, discussed later in this chapter. The remote server procedure, *nuser*, corresponding to the earlier example 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 the 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 looks 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 which 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 pro-

NOTE
UDP only deals with arguments and results less than 8K bytes long.

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

Assigning Program Numbers

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

0	-	1FFFFFFF	defined by Sun Microsystems
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, and intends that the numbers be identical across all systems and applications. If a customer develops an application that might be of general interest, that application should be given a number assigned by Sun from 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.

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, if you wish to send the structure

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

then *callrpc* should be called 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 chapter 4, *XDR Protocol Specification*, so this section only gives a few examples of XDR implementation.

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

xdr_array( ) *xdr_bytes( )* *xdr_opaque( )* *xdr_float( )*
xdr_reference( ) *xdr_union( )* *xdr_wrapstring( )* *xdr_double( )*

To send a variable array of integers, they might be packaged up 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 routine could also be used to send out 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 these examples 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. It contains four parameters which are the same as 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 chases 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);
}
```

Lower Layers of RPC

In the examples given so far, RPC takes care of many details automatically. This section shows how to change the defaults by using lower layers of the RPC library.

More on the Server Side

A number of assumptions are built into *registerrpc( )*. One is that the UDP protocol is being used. Another is that the user does not want to do anything unusual while deserializing, since the deserialization process happens automatically before the user's server routine is called. The server for the *nusers* program shown below is written using a lower layer of the RPC package, which does not make these assumptions:

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

int nuser();

main()
{
        SVCXPRT *transp;

        transp = svcudp_create(RPC_ANYSOCK);
        if (transp == NULL){
                fprintf(stderr, "couldn't create an RPC server\n");
                exit(1);
        }
        pmap_unset(RUSERSPROG, RUSERSVERS);
        if (!svc_register(transp, RUSERSPROG, RUSERSVERS, nuser,
            IPPROTO_UDP)) {
                fprintf(stderr, "couldn'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, "couldn'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, "couldn'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 out RPC messages. *registerrpc( )* uses *svcudp_create( )* to get a UDP handle. If a reliable protocol is required, call *svctcp_create( )* instead. If the argument to *svcudp_create( )* is **RPC_ANYSOCK**, the RPC library creates a socket on which to send out RPC calls. Otherwise, *svcudp_create( )* expects its argument to be a valid socket number. If the user specifies a 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 up, 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 **RUSERS** from the portmapper's tables.

Finally, the program number for *nusers* is associated 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( )* which are handled automatically by *registerrpc( )*. 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 is how a server handles an RPC program that passes data. As an example, a procedure, **RUSERSPROC_BOOL,** which has an argument **nusers,** and returns **TRUE** or **FALSE** depending on whether there are **nusers** logged on can be added. It looks 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, "couldn't reply to RPC call\n");
            exit(1);
    }
    return;
}
```

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

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 XDR was wanted to do the allocation, this routine would have to be rewritten in the following way:

```
xdr_chararr2(xdrsp, chararrp)
        XDR *xdrsp;
        char **chararrp;
{
        int len;

        len = SIZE;
```

```
        return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

The RPC call might then look like this:

```
char *arrptr;

arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);
/*
 * use the result here
 */
svc_freeargs(xdrsp, 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. 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. Chapter 4 has examples of more sophisticated XDR routines that determine which of the three modes they are in to function correctly.

The Calling Side

When *callrpc* is used, there is no control over the RPC delivery mechanism or the socket used to transport the data. To illustrate the layer of RPC that allows adjustment of these parameters, consider the following code to call the *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, "cannot 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) {
                perror("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( )*. It 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), *clnttcp_create( )* would be used.

The parameters to *clntudp_create( )* are the server address, the length of the server address, the program number, the version number, a timeout 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( )* timeout divided by the *clntudp_create( )* timeout. To make a stream connection, the call to *clntudp_create( )* is replaced with a call to *clnttcp_create( )*.

NOTE
The **clnt_destroy( )** call 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( )**. If there are multiple client handles using the same socket, it is possible to close one handle without destroying the socket that other handles are using.

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

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

Other RPC Features

This section describes other aspects of RPC, including batching, authentication, and the **inetd** daemon.

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( )*. However, if the other activity involves waiting for a file descriptor, the *svc_run( )* call won't work. The code for *svc_run( )* follows:

```
void
svc_run( )
{
        int readfds;

        for (;;) {
                readfds = svc_fds;
                switch (select(32, &readfds, NULL, NULL, NULL)) {

                case -1:
                        if (errno == EINTR)
                                continue;
```

```
                    perror("rstat: select");
                    return;
            case 0:
                    break;
            default:
                    svc_getreq(readfds);
            }
    }
}
```

Svc_run( ) can be bypassed, and *svc_getreq( )* called directly. To do this, you must know the file descriptors of the socket(s) associated with the programs which are being waited for. Thus, write your own *selects* which wait on both the RPC socket, and their own descriptors.

Broadcast RPC

The **pmap** and RPC protocols implement broadcast RPC. Here are the main differences between broadcast RPC and normal RPC calls:

(1) Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answers from each responding machine).

(2) Broadcast RPC can only be supported by packet-oriented (connectionless) transport protocols like UDP/IP.

(3) 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.

(4) All broadcast messages are sent to the portmap port. Thus, only services that are registered 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 obtained */
```

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; /* address of machine that sent response */
```

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 up 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.

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, while the other renders a string and remains silent. Since the batched calls are buffered, the client eventually does a non-batch call in order to flush the pipeline. The service (using the TCP/IP transport) may look like this:

```
#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, "couldn't create an RPC server\n");
        exit(1);
    }
    pmap_unset(WINDOWPROG, WINDOWVERS);
    if (!svc_register(transp, WINDOWPROG, WINDOWVERS, windowdispatch,
        IPPROTO_TCP)) {
            fprintf(stderr, "couldn'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, "couldn't reply to RPC call\n");
            exit(1);
        }
        return;
    case RENDERSTRING:
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "couldn't decode arguments\n");
            svcerr_decode(transp); /* tell caller he screwed up */
            break;
        }
        /*
```

```
         * put code here to render the string s
         */
        if (!svc_sendreply(transp, xdr_void, NULL)) {
            fprintf(stderr, "couldn't reply to RPC call\n");
            exit(1);
        }
        break;

    case RENDERSTRING_BATCHED:
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "couldn't decode arguments\n");
            /*
             * we are silent in the face of protocol errors
             */
            break;
        }
        /*
         * the code here renders the string,
         * but sends 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 timeout must be zero.

Here is 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:

```
#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];
    char *s = buf;

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

    /*
     * now flush the pipeline
     */
    total_timeout.tv_sec = 20;
    clnt_stat = clnt_call(client, NULLPROC,
        xdr_void, NULL, xdr_void, NULL, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "rpc");
        exit(-1);
    }

    clnt_destroy(client);
}
```

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.

This example was completed to render all of the (2000) lines in the file */etc/termcap*. The rendering service did nothing but to throw the lines away. The example was run in the following four configurations:

(1) machine to itself, regular RPC;

(2) machine to itself, batched RPC;

(3) machine to another, regular RPC, and

(4) machine to another, batched RPC.

The results follow:

(1) 50 seconds;

(2) 16 seconds;

(3) 52 seconds,

(4) 10 seconds.

Running *fscanf( )* on */etc/termcap* only requires six seconds. These timings show the advantage of protocols that allow for overlapped execution, though 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 measures than those which have 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 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.

The 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 sets the default associate authentication handle to be

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

The RPC client can choose to use UNIX type 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 of client machine */
        int     aup_uid;        /* client's UNIX effective uid */
        int     aup_gid;        /* client's current UNIX group id */
        u_int   aup_len;        /* the element length of aup_gids array */
        int     *aup_gids;      /* array of groups to which user belongs */
};
```

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);
```

The Server Side

It is more difficult for service implementors dealing with authentication issues since the RPC package passes the service dispatch routine a request that includes 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 version number */
        u_long          rq_proc;        /* the desired procedure number */
        struct opaque_auth rq_cred; /* raw credentials from the "wire" */
```

```
        caddr_t       rq_clntcred;  /* read only, cooked credentials */
        SVCXPRT       *rq_xptr;     /* associated transport */
};
```

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 makes the following guarantees to the service dispatch routine:

(1) 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.

(2) 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. As only UNIX style is currently supported, **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.

The remote users service example can be extended so that it computes results for all users except UID 16:

```
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 the null procedure
     */
    if (rqstp->rq_proc == NULLPROC) {
        if (!svc_sendreply(transp, xdr_void, 0)) {
```

```
                fprintf(stderr, "couldn'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 the caller is allowed to call this procedure.
             */
            if (uid == 16) {
                svcerr_systemerr(transp);
                return;
            }
            /*
             * code here to compute the number of users
             * and put in variable nusers
             */
            if (!svc_sendreply(transp, xdr_u_long, &nusers) {
                fprintf(stderr, "couldn'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 a particular user's service, call *svcerr_weakauth( )*. Finally, the service protocol itself should return status for access denied; in the example, the protocol does not have such a status, so the service primitive *svcerr_systemerr( )* is called instead.

The last point underscores the relation between the RPC authentication package and the services; RPC deals only with authentication and not with individual services' access control. The services themselves must implement their own access control policies and reflect these policies as return statuses in their protocols.

Using Inetd

TITAN supports a utility daemon named **inetd** that simplifies the administration of RPC servers by controlling them based on an input control file.

If a server is to be started by **inetd**, then the only difference from the usual code is that *svcudp_create( )* should be called as

```
transp = svcudp_create(0);
```

since the program would already be registered by **inetd**. Remember that if you wish to exit from the server process and return control to **inetd**, you must explicitly exit, since *svc_run( )* never returns.

The format of entries in */etc/servers* for RPC services is

```
rpc udp server program version
```

where *server* is the C code implementing the server, and *program* and *version* are the program and version numbers of the service. The key word **udp** can be replaced by **tcp** for TCP-based RPC services.

If the same program handles multiple versions, then the version number can be a range, as in this example:

```
rpc udp /usr/etc/rstatd 100001 1-2
```

More Examples

This sections gives examples of version control, TCP, and callback procedures.

Versions

By convention, the first version number of program **FOO** is **FOOVERS_ORIG** and the most recent version is **FOOVERS**. 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, "couldn't register RUSER service\n");
        exit(1);
}
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_SHORT, nuser,
    IPPROTO_TCP)) {
        fprintf(stderr, "couldn'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, "couldn'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)
            if (!svc_sendreply(transp, xdr_u_long, &nusers) {
                fprintf(stderr, "couldn't reply to RPC call\n");
                exit(1);
        }
        else
            if (!svc_sendreply(transp, xdr_u_short, &nusers2) {
                fprintf(stderr, "couldn't reply to RPC call\n");
                exit(1);
        return;
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

TCP

Here is an example that essentially duplicates the user command *rcp*(1). 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 depending on whether serialization or deserialization is on.

```
/*
 * 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[MAXCHUNK], *p;

        if (xdrs->x_op == XDR_FREE)/* nothing to free */
                return 1;
        while (1) {
                if (xdrs->x_op == XDR_ENCODE) {
                        if ((size = fread (buf, sizeof(char), MAXCHUNK, fp))
                            == 0 && ferror(fp)) {
                                fprintf(stderr, "couldn't fread\n");
                                exit(1);
                        }
                }
                p = buf;
                if (!xdr_bytes(xdrs, &p, &size, MAXCHUNK))
                        return 0;
                if (size == 0)
                        return 1;
                if (xdrs->x_op == XDR_DECODE) {
                        if (fwrite(buf, sizeof(char), size, fp) != size) {
                                fprintf(stderr, "couldn'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 server-name\n", argv[0]);
        exit(-1);
    }
    if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC_FP, RCPVERS,
        xdr_rcp, stdin, xdr_void, 0) != 0)) {
            clnt_perrno(err);
            fprintf(stderr, " couldn'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, "cannot 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, 1024, 1024)) == 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 to 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. However, when the debugger hits a breakpoint, 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, a program number on which to make the RPC call is needed. Since this is 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 that indivisibly tests whether a program number has already been registered and, if not, reserves one. On return, the **sockp** argument will contain a socket that can be used as the argument to an *svcudp_create( )* or *svctcp_create( )* call.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>

gettransient(proto, vers, sockp)
        int *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 err
	 */
	bind(s, &addr, len);
	if (getsockname(s, &addr, &len)< 0) {
		perror("getsockname");
		return (0);
	}
	while (pmap_set(prognum++, vers, proto, addr.sin_port) == 0)
		continue;
	return (prognum-1);
}
```

The following pair of programs 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.

```
/*
 * 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);
	}
	(void)svc_register(xprt, x, 1, callback, 0);

	ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEPROC_CALLBACK,
	    EXAMPLEVERS, xdr_int, &x, xdr_void, 0);
	if (ans != 0) {
```

```
                fprintf(stderr, "call: ");
                clnt_perrno(ans);
                fprintf(stderr, "\n");
        }
        svc_run( );
        fprintf(stderr, "Error: svc_run shouldn't have returned\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)  == FALSE) {
                                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)  == FALSE) {
                                fprintf(stderr, "err: rusersd");
                                exit(1);
                        }
        }
}

/*
 * 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, EXAMPLEPROC_CALLBACK, EXAMPLEVERS,
            getnewprog, xdr_int, xdr_void);
        fprintf(stderr, "server going into svc_run\n");
        alarm(10);
        signal(SIGALRM, docallback);
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn't have returned\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");
        }
}
```

Synopsis of RPC Routines

The rest of this chapter presents a synopsis of the syntax of each RPC routine and a description of its parameters and what it does.

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 UNIX 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( )

```
int
callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
        char *host;
        u_long prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
```

NOTE
Calling remote procedures with this routine uses UDP/IP as a transport; see **clntudp_create( )** for restrictions.

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.

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*, the form of which 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; 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 where the result(s) should be placed; **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_destroy( )

```
void
clnt_destroy(clnt)
        CLIENT *clnt;
```

WARNING
Client destruction routines do not close sockets associated with **clnt**; the user must do so.

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( )**.

clnt_freeres( )

```
bool_t
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, and 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.

clnt_perrno( )

```
void
clnt_perrno(stat)
        enum clnt_stat;
```

Prints a message to standard error corresponding to the condition indicated by **stat**.

clnt_perror( )

```
void
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.

clntraw_create( )

```
CLIENT *
clntraw_create(prognum, versnum)
        u_long prognum, versnum;
```

This routine creates a memory-based loopback (sometimes referred to as a toy) RPC client for the remote program **prognum,** 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 the 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;
```

NOTE
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.

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 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**. The UDP transport resends the call message in intervals of **wait** time until a response is received or until the call times out. This routine returns **NULL** if it fails.

get_myaddress( )

```
void
get_myaddress(addr)
        struct sockaddr_in *addr;
```

Places the machine's IP address in ***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 that returns the port number of a waiting service 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 failed 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 the user's behalf to a procedure on that host. The parameter ***portp** changes to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in *callrpc( )* and *clnt_call( )*; see also *clnt_broadcast( )*.

pmap_set( )

```
bool_t
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.

pmap_unset( )

```
bool_t
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( )

```
int
registerrpc(prognum, versnum, procnum, procname, inproc, outproc)
        u_long prognum, versnum, procnum;
        char *(*procname) ( );
        xdrproc_t inproc, outproc;
```

NOTE
Remote procedures registered in this form are accessed using the UDP/IP transport; see **svcudp_create( )** for restrictions.

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.

rpc_createerr

```
struct rpc_createerr        rpc_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;
```

A global variable reflecting the RPC service side's read file descriptor bit mask; 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 yet it may change after calls to *svc_getreq( )* or any creation routines.

svc_freeargs( )

```
void
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, and 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, and 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)
        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 set. The routine returns when all sockets associated with the value of **rdfds** have been serviced.

svc_register( )

```
bool_t
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 non-zero, then a mapping of the triple **[prognum, versnum, protocol]** to **xprt->xp_port** is also 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, and zero otherwise.

svc_run( )

```
void
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( )

```
bool_t
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 which is 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.

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.

NOTE
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( )

```
bool_t
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( )

```
bool_t
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( )

```
void
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( )

```
bool_t
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( )

```
bool_t
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( )

```
bool_t
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_double( )

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

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

xdr_enum( )

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

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

xdr_float( )

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

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

xdr_inline( )

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

NOTE
xdr_inline( ) may return 0 (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.

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** *.

xdr_int( )

```
bool_t
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( )

```
bool_t
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( )

```
bool_t
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( )

```
bool_t
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( )

```
bool_t
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_pmaplist( )_

```
bool_t
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( )_

```
bool_t
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( )_

```
bool_t
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. It returns one if it succeeds, zero otherwise.

_xdr_replymsg( )_

```
bool_t
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. It returns one if it succeeds, zero

otherwise.

xdr_short( )

```
bool_t
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( )

```
bool_t
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. **sp** is the address of the string's pointer. Strings cannot be longer than **maxsize**. This routine returns one if it succeeds, zero otherwise.

xdr_u_int( )

```
bool_t
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( )

```
bool_t
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( )

```
bool_t
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( )

```
bool_t
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_void( )

```
bool_t
xdr_void( )
```

This routine always returns one.

xdr_wrapstring( )

```
bool_t
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 useful 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 do not need this routine.

xprt_unregister( )

```
void
xprt_unregister(xprt)
        SVCXPRT *xprt;
```

Before an RPC service transport handle is destroyed, it should deregister itself with the RPC service package. This routine modifies the global variable **svc_fds**. Service implementors usually do not need this routine.

RPC PROTOCOL

CHAPTER THREE

This chapter 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.

The information in this chapter assumes that the reader is familiar with both RPC and XDR. The casual user of RPC does not need to be familiar with this information.

Introduction

This chapter discusses servers, services, programs, procedures, clients and versions. A server is a machine where some number of network services are implemented. A service is a collection of one or more remote programs. A remote program implements one or more remote procedures; the procedures, their parameters and results are documented in the specific program's protocol specification (see chapter 2 for an example). 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.

The RPC Model

Consider the local procedure call model. In this 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.

A 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 contents of the call message include the procedure's parameters. The contents of the reply message include the procedure's results. Once the reply message is received, the results of the procedure are extracted, and the caller's execution is resumed.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message. Note that in this 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.

NOTE
RPC is currently implemented on top of TCP/IP and UDP/IP transports.

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 that can be inferred from no reply message is that the remote procedure was executed zero or more times (and from a reply message, one or more times). Conversely, 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 executed 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. (The software may use RPC itself; see chapter 2.)

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 itself to a service and vice versa. Security and access control mechanisms can be built on top of the message authentication.

Requirements

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 (like mis-specification of a procedure's parameters);
- Reasons why remote authentication failed, and
- 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 (currently Sun Microsystems). Once an implementor receives a program number, the remote program can be implemented; the first implementation would most probably 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 possible speaking both old and new protocols through the same server process.

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 not speak protocol version 2. The lowest and highest supported RPC version numbers are returned.
- The remote program is not available on the remote system.
- The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
- The requested procedure number does not exist (this is usually a caller side protocol or programming error).
- The parameters to the remote procedure appear to be garbage from the server's point of view. (Again, this is 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 a part of the RPC protocol. The call message contains two authentication fields; the credentials and verifier. The reply message contains 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.

The interpretation and semantics of the data contained within the authentication fields is specified by individual, independent authentication protocol specifications. Three authentication protocols are described in *Authentication Parameter Specification.*

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 RPC users. The

second range is for applications peculiar to a particular user. This range is intended primarily for debugging new programs. When a user 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 and Abuses of the RPC Protocol

This protocol is intended to be used for calling remote procedures. That is, each call message is matched with a response message. However, the protocol itself is a message passing protocol with which other (non-RPC) protocols can be implemented. The RPC message protocol is currently used for two non-RPC protocols: batching (or pipelining) and broadcast RPC. 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 a broadcast packet to the network and waits for numerous replies. Broadcast RPCs use unreliable, packet based protocols, such as UDP/IP, as their transports. Servers that support broadcast protocols only respond when the request is successfully processed. They are silent in the face of errors.

The RPC Message Protocol

This section defines the RPC message protocol in the XDR data description language. The message is defined in a top down style.

This is an XDR specification, not C code.

```
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,
                /* remote procedure was successfully executed */
        PROG_UNAVAIL = 1,
                /* remote machine exports the program number */
        PROG_MISMATCH = 2,
                /* remote machine can't support version number */
        PROC_UNAVAIL = 3,
                /* remote program doesn't know about procedure */
        GARBAGE_ARGS = 4
                /* remote procedure can't figure out parameters */
        SYSTEM_ERR = 5,
                /* system/implementation error */
};

/*
 * Reasons why a call message was rejected:
 */
enum reject_stat {
        RPC_MISMATCH = 0,
                /* RPC version number was not two (2)*/
        AUTH_ERROR = 1
                /* caller not authenticated on remote machine*/
};

/*
 * Why authentication failed:
 */
enum auth_stat {
        AUTH_BADCRED = 1,
                /* bogus credentials (seal broken) */
        AUTH_REJECTEDCRED = 2,
                /* client should begin new session */
        AUTH_BADVERF = 3,
                /* bogus verifier (seal broken) */
        AUTH_REJECTEDVERF = 4,
                /* verifier expired or was replayed */
```

```
        AUTH_TOOWEAK = 5,
                /* rejected due to security reasons */
        AUTH_INVALIDRESP = 6,
                /* bogus response verifier */
        AUTH_FAILED = 7,
                /* failed for unknown reason */
};

/*
 * 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, and the procedure
 * within the remote program to be called.
 * These fields are followed by two authentication
 * parameters, cred (authentication credentials)
 * and verf (authentication verifier).  The
 * authentication parameters are followed by
 * the parameters to the remote procedure;
 * these parameters 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 which the server generates in order to
 * validate itself to the caller.  It is followed by
 * a union whose discriminant is an enum accept_stat.
 * The SUCCESS arm of the union is protocol specific.
 * The PROG_UNAVAIL, PROC_UNAVAIL, and GARBAGE_ARGS
 * arms of the union are void.  The PROG_MISMATCH
 * arm specifies the lowest and highest version
 * numbers of the remote program that are supported
 * by the server.
 */
struct accepted_reply {
        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
 * refused to authenticate the caller (AUTH_ERROR).
 * In the case of an RPC version mismatch, the server
 * returns the lowest and highest supported RPC version
 * numbers. In the case of refused authentication,
 * the failure status is returned.
 */
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 types of authentication which have been implemented, and are generally supported.

Null Authentication

Often calls must be made where the caller does not know who it is and the server does not care who it 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 itself as it 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 following (XDR) structure:

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

The **stamp** is an arbitrary id which 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 callers effective group id. The **gids** is a counted array of groups which 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(2). 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 short hand **auth_opaque** structures (passed back via 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 short hand **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). This RM/TCP/IP transport is used for passing RPC messages on TCP streams. One RPC message fits into one RM record.

A record is composed of one or more record fragments. A record fragment is a 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 which indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment) and a 31-bit unsigned binary value which is the length in bytes of the fragment's data. The boolean value is the highest-order bit of the header; the length is the 31 low-order bits.

NOTE
This record specification is *not* in XDR standard form.

Port Mapper Program Protocol

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.

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
                RM/TCP/IP on port 111

/*
 * Handy transport protocol numbers
 */
#define IPPROTO_TCP       6
        /* protocol number used for rpc/rm/tcp/ip */
#define IPPROTO_UDP       17
        /* protocol number used for rpc/udp/ip */

/* Procedures */

/*
 * Convention: procedure zero of any protocol takes no parameters
 * and returns no results.
 */
0. PMAPPROC_NULL () returns ()

/*
 * Procedure 1, setting a mapping:
 * When a program first becomes available on a
 * machine, it registers itself with the port mapper program on the
 * same machine.  The program passes its program number (prog),
 * version number (vers), transport protocol number (prot),
 * and the port (port) on which it awaits service request.  The
 * procedure returns success whose value is TRUE if the procedure
 * successfully established the mapping and FALSE otherwise.  The
 * procedure will refuse to establish a mapping if one already exists
 * for the tuple [prog, vers, prot].
 */
1. PMAPPROC_SET (prog, vers, prot, port) returns (success)
        unsigned prog;
        unsigned vers;
        unsigned prot;
        unsigned port;
        boolean success;

/*
 * Procedure 2, Unsetting a mapping:
 * 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.
 */
2. PMAPPROC_UNSET (prog, vers, dummy1, dummy2) returns (success)
        unsigned prog;
        unsigned vers;
        unsigned dummy1;  /* this value is always ignored */
```

```
        unsigned dummy2;  /* this value is always ignored */
        boolean success;

/*
 * Procedure 3, looking-up a mapping:
 * 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 that the program has not been registered.
 */
3. PMAPPROC_GETPORT (prog, vers, prot, dummy) returns (port)
        unsigned prog;
        unsigned vers;
        unsigned prot;
        unsigned dummy;  /* this value is always ignored */
        unsigned port;   /* zero means the program is not registered */

/*
 * Procedure 4, dumping the mappings:
 * This procedure enumerates all entries in the port mapper's database.
 * The procedure takes no parameters and returns a ''list'' of
 * [program, version, prot, port] values.
 */
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;

/*
 * Procedure 5, indirect call routine:
 * The procedures 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.
 *
 * NB:
 * 1. This procedure only sends a response if the procedure was
 * successfully executed and is silent (No response) otherwise.
 * 2. The port mapper communicates with the remote program via
 * UDP/IP only.
 *
 * The procedure returns the port number of the remote program and
 * the bytes of results are the results of the remote procedure.
```

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

XDR PROTOCOL SPECIFICATION

CHAPTER FOUR

This chapter presents library routines which 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 chapter also 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.

Introduction

XDR was designed to work across different languages, operating systems, and machine architectures. Most users (particularly RPC users) only need the information in this and the next section. Programmers wishing to implement RPC and XDR on new machines need the information in the next three sections. Advanced topics, not necessary for all implementations, are covered in the last section.

C programs that want to use XDR routines should include the file *<rpc/rpc.h>*, which contains all the necessary interfaces to the XDR system. If the C library **libc.a** contains the XDR routines then programs can be compiled without additional options, such as:

```
cc program.c
```

Otherwise, the use of the compile flag *-lrpc* requests the inclusion of the RPC library, *librpc.a*.

Justification

Consider the following two programs, **writer** and **reader:**

```
#include <stdio.h>
/*
 * writer.c
 */

main()
{
	long i;

	for (i = 0; i < 8; i++) {
		if (fwrite((char *)&i, sizeof(i), 1, stdout) != 1) {
			fprintf(stderr, "failed!\n");
			exit(1);
		}
	}
}

/*
 * reader.c
 */

#include <stdio.h>

main()
{
	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

(1) they pass **lint** checking, and

(2) they exhibit the same behavior when executed on two different hardware architectures, for example, a Sun Workstation* and a VAX†.

* Sun Workstation is a trademark of Sun Microsystems.
† VAX is a trademark of Digital Equipment Corporation.

Piping the output of the **writer** program to the **reader** program gives identical results on a Sun or a VAX.

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

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

With the advent of local area networks 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 Sun, and the second consumes data on a VAX.

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

Identical results can be obtained by executing **writer** on the VAX and **reader** on the Sun. These results occur because the byte ordering of long integers differs between the VAX and the Sun, even though word size is the same. Note that 16777216 is 2^{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. Shown below are the revised versions of **writer** and **reader**:

```
#include <stdio.h>
#include <rpc/rpc.h>  /* xdr is a sub-library of the rpc library */
/*
 * writer.c
 */

main()
{
        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);
```

```
                }
        }
}

#include <stdio.h>
#include <rpc/rpc.h>  /* xdr is a sub-library of the rpc library */
/*
 * reader.c
 */

main()
{
        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 Sun, on a VAX, and from a Sun to a VAX; the results are shown below.

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

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

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

Dealing with integers is only a small part of portable-data. 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.

The XDR library package solves data portability problems. It allows users 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.

The XDR library has filter routines for many subjects, including strings (null-terminated arrays of bytes), structures, unions, and arrays, to name a few. Using more primitive routines, users can write their 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.

The two programs will now be examined more closely.

A family of XDR stream creation routines exists, in which each member treats the stream of bits differently. In the example given, data is manipulated using standard I/O routines, so *xdrstdio_create( )* is used. The parameters to XDR stream creation routines vary according to their function. In the example, *xdrstdio_create( )* takes a pointer to an XDR structure that it initializes, a pointer to a **FILE** that the input or output is performed on, and the operation. The operation may be **XDR_ENCODE** for serializing in the **writer** program, or **XDR_DECODE** for deserializing in the **reader** program.

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

XDR routines are direction independent; that is, the same routines are called to serialize or deserialize data. This feature is critical to software engineering of portable data. The intention 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 the example, but its value becomes obvious when nontrivial data structures are passed among machines. If needed, the direction of the XDR operation can be obtained. See the next section for details.

Consider 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). In this chapter, the following definitions are used:

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

#define enum_t int  /* enum_t's are used for generic enum's */
```

Using these conventions, *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));
}
```

Both coding styles are used in this chapter.

XDR Library Primitives

This section 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>* which is automatically included by *<rpc/rpc.h>*.

Number Filters

The XDR library provides primitives that translate between C numbers and their corresponding external representations. The primitives cover the set of numbers in:

```
[signed, unsigned] * [short, int, long]
```

Specifically, the six primitives are:

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

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

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

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

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

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

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

Floating Point Filters

The XDR library also provides primitive routines for C's floating point types:

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

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

NOTE
The numbers are represented in IEEE floating point, so routines may fail when decoding a valid IEEE representation into a machine-specific representation, or vice-versa.

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.

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

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 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, which can hold strings no longer than **maxlength**, has been allocated. 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:

- the length of the array (the byte count) is explicitly located in an unsigned integer;
- the byte sequence is not terminated by a null character; and
- 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 are identical to the first, second and third parameters of *xdr_string( )*, respectively. The length of the byte area is obtained by dereferencing **lp** when serializing; ***lp** is set to the byte length when deserializing.

Arrays

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

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

The parameter **ap** is the address of the pointer to the array. If ***ap** is **NULL** when the array is being deserialized, XDR allocates an array of the appropriate size and sets ***ap** to that array. The element count of the array is obtained from ***lp** when the array is serialized; ***lp** is set to the array length when the array is deserialized. The parameter **maxlength** is the maximum number of elements that the array is allowed to have; **elementsize** 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

- the machine name, such as **krypton**: see *gethostname*(3);
- the user's UID: see *geteuid*(2); and
- the group numbers to which the user belongs: see *getgroups*(2).

A structure with this information and its associated XDR routine could be coded as follows:

```
struct netuser {
        char        *nu_machinename;
        int         nu_uid;
        u_int       nu_glen;
        int         *nu_gids;
};
#define NLEN 255 /* machine names must be shorter than 256 chars */
#define NGRPS 20 /* user can't be a member of more than 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 can be no longer than 1000 chars */
#define NARGC 100  /* commands may have no more than 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_wrap_string(xdrs, sp)
        XDR *xdrs;
        char **sp;
{
        return (xdr_string(xdrs, sp, ALEN));
}

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

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));
}
```

Note that the routine *xdr_wrap_string( )* is needed to package the *xdr_string( )* routine because the implementation of *xdr_array( )* only passes two parameters to the array element description routine; *xdr_wrap_string( )* supplies the third parameter to *xdr_string( )*.

By now the recursive nature of the XDR library should be obvious. The following sections describe more constructed data types.

Opaque Data

In some protocols, handles are passed from a server to client. The client passes the handle back to the server at some later time. Handles are never inspected by clients; they are obtained and submitted. That is to say, handles are opaque. The 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.

Fixed Sized Arrays

The XDR library does not provide a primitive for fixed-length arrays (the primitive *xdr_array( )* is for varying-length arrays). Example A could be rewritten to use fixed-sized arrays in the following fashion:

```
#define NLEN 255   /* machine names must be shorter than 256 chars */
#define NGRPS 20   /* user cannot be a member of more than 20 groups */

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

bool_t
xdr_netuser(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);
        for (i = 0; i < NGRPS; i++) {
                if (! xdr_int(xdrs, &nup->nu_gids[i]))
                        return (FALSE);
        }
        return (TRUE);
}
```

Discriminated Unions

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

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

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

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

Assume 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;        /* this is the union's discriminant */
        union {
                int ival;
                char *pval;
                struct gnumbers gn;
        } uval;
};
```

The following constructs and XDR procedure (de)serialize the discriminated union:

```
struct xdr_discrim u_tag_arms[4] = {
        { INTEGER, xdr_int },
        { GNUMBERS, xdr_gnumbers }
        { STRING, xdr_wrap_string },
        { __dontcare__, NULL }
        /* always terminate arms with a NULL xdr_proc */
}
```

```
bool_t
xdr_u_tag(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 earlier; *xdr_wrap_string( )* 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 should be noted that the values of the discriminant may be sparse, though in this example they are not. It is always good practice explicitly to assign integer values to each element of the discriminant's type. This practice both documents the external representation of the discriminant and guarantees that different C compilers emit identical discriminant values.

Pointers

In C 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, ssize, 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 (the C function **sizeof( )** may be used 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.

Note that *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 causes a memory fault and, on UNIX , a core dump for debugging.

It is the explicit responsibility of the programmer to expand non-dereferencable 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). *Advanced Topics,* later is this chapter, has an example (linked lists encoding) that deals with invalid pointer interpretation.

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. In some XDR streams, the returned value (–1) of **xdr_getpos( )** is meaningless, though –1 should be a legitimate value.

The routine *xdr_setpos( )* sets a stream position to **pos**. In some XDR streams, setting a position is impossible; in such cases, *xdr_setpos( )* will return **FALSE**. This routine will also fail if the requested position is out-of-bounds. The definition of bounds varies from stream to stream.

The *xdr_destroy( )* primitive destroys the XDR stream. Use of the stream after calling this routine is undefined.

XDR Operation Directions

At times it may be wished 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 the last section demonstrates the usefulness of the **xdrs->x_op** field.

XDR Stream Access

An XDR stream is obtained by calling the appropriate creation routine. These creation routines take arguments that are tailored to the specific properties of the stream.

Streams currently exist for (de)serialization of data to or from standard I/O **FILE** streams, TCP/IP connections and UNIX files, and memory. The next section 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 are a part of the rpc library */

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 UNIX file or 4.2 BSD connection interface.

```
#include <rpc/rpc.h>  /* xdr streams are a part of the rpc library */

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

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

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

The XDR stream provides means for delimiting records in the byte stream. The implementation details of delimiting records in a stream are discussed in the next section. The primitives that are specific to record streams are as follows:

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

bool_t
xdrrec_skiprecord(xdrs)
        XDR *xdrs;

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

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

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

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

XDR Stream Implementation

This section provides the abstract data types needed to implement new instances of XDR streams.

The XDR Object

The 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 additional param */
        struct  xdr_ops {
                bool_t  (*x_getlong)();
                        /* get a long from underlying stream */
                bool_t  (*x_putlong)();
                        /* put a long to " */
                bool_t  (*x_getbytes)();
                        /* get some bytes from " */
                bool_t  (*x_putbytes)();
                        /* put some bytes to " */
                u_int   (*x_getpostn)();
                        /* return byte offset from beginning */
                bool_t  (*x_setpostn)();
```

```
                    /* repositions position in stream */
        caddr_t (*x_inline)();
                    /* buf quick ptr to buffered data */
        VOID    (*x_destroy)();
                    /* free privates of this xdr_stream */
    }       *x_ops;
    caddr_t x_public;    /* users' data */
    caddr_t x_private;   /* pointer to private data */
    caddr_t x_base;      /* private used 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 the implementation of a stream. That is, the implementation of a stream 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, *Synopsis of RPC Routines*. 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 point of view of the stream, 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 UNIX primitives *htonl( )* and *ntohl( )* can be helpful in

accomplishing this. The next section 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 non-negative 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.

XDR Standard

This section 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 Sun, 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 XDR 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 meanings of bits within bytes across hardware boundaries. For example, the Ethernet standard suggests that bytes be encoded "little endian" style. Both Sun 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 description language, not a programming 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 description 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 description of unsigned integers is **unsigned**.

Enumerations

Enumerations have the same representation as integers. Enumerations are useful for describing subsets of the integers. The data description 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

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.

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:

S The sign of the number. Values 0 and 1 represent positive and negative, respectively.

E 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.

F 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:

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

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 fol-

lowed 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 description of strings is as follows:

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

Note that the data description 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^{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 14 bytes, such as:

```
string filename<14>;
```

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 description 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 description 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 $2^{32}-1$.

Structures

The data description 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 description 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 4-1 describes the association between the C library primitives and the standard data types defined in this section. Look in the *Synopsis* section of the chapter listed in the table to get more details.

Table 4-1. C/XDR Type Association

C Primitive	XDR Type	Chapters
xdr_int xdr_long xdr_short	integer	2, 4
xdr_u_int xdr_u_long xdr_u_short	unsigned	2, 4
–	hyper integer hyper unsigned	4
xdr_float	float	2, 4
xdr_double	double	2, 4
xdr_enum	enum_t	2, 4
xdr_bool	bool_t	2, 4
xdr_string xdr_bytes	string	2, 4 4
xdr_array	(varying arrays)	2, 4
–	(fixed arrays)	2, 4
xdr_opaque	opaque	2, 4
xdr_union	union	2, 4
xdr_reference	–	2
–	struct	4

Advanced Topics

This section describes techniques for passing data structures that are not covered in the preceding sections. Such structures include linked lists (of arbitrary lengths). Unlike the simpler examples covered in the earlier sections, the following examples are written using both the XDR C library routines and the XDR data description language.

Linked Lists

An earlier example 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 you 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 (de)serialize a linked list of entries can be taken from the XDR description of the pointerless 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 there are no more nodes to be (de)serialized */
        { 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.

Readers should convince themselves 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 contains 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);
        }
}
```

This routine is easier to code and understand than the three recursive routines given earlier. 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** is used here in the same way as in the earlier routines. Its value is recomputed and re-(de)serialized for 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 optimization works only because the **nxt** data comes after all legitimate external data.

There is a bug in this routine in the **XDR_FREE** case, in that *xdr_reference( )* will free the node ***glp**. Upon return the assignment **glp = &((*glp)->nxt)** cannot be guaranteed to work since ***glp** is no longer a legitimate node. The following code works in all cases. The hard part is to avoid dereferencing a pointer which has not been initialized or which 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 gnode),
                    xdr_gnumbers))
                        return (FALSE);
                glp = (freeing) ? next : &((*glp)->nxt);
        }
}
```

Note that this is the first example in this chapter which 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.

The 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 proceeds 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.

Synopsis of XDR Routines

This section gives a synopsis of XDR syntax and a brief description of the parameters.

xdr_array( )

```
bool_t
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_bool( )

```
bool_t
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( )

```
bool_t
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_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( )

```
bool_t
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( )

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

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

xdr_float( )

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

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

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_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** *.

NOTE
xdr_inline( ) may return 0 (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()

```
bool_t
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()

```
bool_t
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()

```
bool_t
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_reference()

```
bool_t
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( )

```
bool_t
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, and zero otherwise.

NOTE
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( )

```
bool_t
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( )

```
bool_t
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_int( )

```
bool_t
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( )

```
bool_t
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( )

```
bool_t
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( )

```
bool_t
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_void( )

```
bool_t
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( )

```
bool_t
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 useful 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 that 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 UNIX 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.

NOTE
This XDR stream implements an intermediate record stream. There are therefore additional bytes in the stream to provide record boundary information.

xdrrec_endofrecord( )

```
bool_t
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( )

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

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( )

```
bool_t
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;
```

NOTE
The destroy routine associated with such XDR streams calls *fflush( )* on the **file** stream, but never *fclose( )*.

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**).

NFS PROTOCOL SPECIFICATION

CHAPTER FIVE

This chapter presents the NFS protocol definition. The protocol is defined by arguments to and results returned by eXternal Data Representation (XDR) routines. These procedures, built from Remote Procedure Call (RPC) primitives, make up the NFS server.

RPC primitives are described in chapter 2 and XDR routines are described in chapter 4.

Introduction

The NFS protocol provides transparent remote access to shared filesystems over local area networks. The NFS protocol is designed to be independent of machine, operating system, network architecture, and transport protocol. This independence is achieved through the use of RPC primitives built on top of an 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 RPC specification, described in chapter 2, 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. The only transport protocol currently supported by NFS 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 may change over time, and the protocol which they use may change too, 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 XDR specification, described in chapter 4, 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 chapter, the XDR data definition language is used to specify the parameters and results of each RPC service procedure that an NFS server provides. The XDR data definition language is similar to C, although a few new constructs have been added. The notation

```
string        name[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 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 will be 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.

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

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 issue may not seem important, but it affects the protocol in several ways. It is worth extra complexity in the protocol to be able to write very simple servers with no crash recovery.

NFS Protocol Definition

The NFS protocol is designed to be operating system independent, but it was designed in a UNIX environment. As such, it has some features which are very UNIX-like. When in doubt about how something should work, referring to the way it is done on UNIX should provide assistance.

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. However, a few do not fit into the normal UNIX mold.

The **LOOKUP** procedure looks up one component of a pathname at a time. It is not immediately obvious why it does not take the whole pathname, look 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. A Network Standard Pathname Representation could be defined, but in this case 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 which might seem strange is the **READDIR** procedure. **READDIR** provides a network standard format for representing directories. The argument 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 be too slow.

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 perform some functions, such as 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 follow the second remote mount means it must do loop detection, server lookup and user revalidation.

Instead, the implementation does not let clients cross a server's mount point. When a client does a **LOOKUP** on a directory on

which the server has mounted a filesystem, 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 will do 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.

Another problem arises due to the semantics of *open*. UNIX does its permission checking at open time and then assumes 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, but still be able to write to the file because it is open. On a remote filesystem, by contrast, the write fails. 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 UNIX 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 this does not 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 super-user permission is not allowed on the server since anyone who can become super-user on his own machine 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 super-user access cannot be avoided. Eventually servers will have to allow some kind of limited super-user access.

RPC Information

The following sections provide specific information about the components of the RPC procedures.

Authentication

The NFS service uses **AUTH_UNIX** style authentication except in the **NULL** procedure where **AUTH_NONE** is also allowed.

Protocols

NFS currently is supported on UDP/IP only.

Constants

These are the RPC constants needed to call the NFS service. They are given in decimal.

PROGRAM	100003
VERSION	2

Port Number

The NFS protocol currently uses the UDP port number 2049. This restriction will be waived in a future protocol revision.

Sizes

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

MAXDATA

The maximum number of bytes of data in a **READ** or **WRITE** request is 8192.

MAXPATHLEN

The maximum number of bytes in a pathname argument is 1024.

MAXNAMLEN

The maximum number of bytes in a file name argument is 255.

COOKIESIZE

The size in bytes of the opaque "cookie" passed by **READDIR** is 4.

FHSIZE

The size in bytes of the opaque file handle is 32.

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_OPNOTSUPP=99,
    NFSERR_ENOBUFS=109,
    NFSERR_NAMETOOLONG=117,
    NFSERR_NOTEMPTY=120,
```

```
    NFSERR_DQUOT=123,
    NFSERR_STALE=125,
    NFSERR_REMOTE=126,
    NFSERR_WFLUSH=127
} 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 UNIX error numbers.

NFSERR_PERM
: Not owner. The caller does not have correct ownership to perform the requested operation.

NFSERR_NOENT
: No such file or directory. The file or directory specified does not exist.

NFSERR_IO
: I/O error. A hard error, for example a disk error, occurred when the operation was in progress.

NFSERR_NXIO
: No such device or address.

NFSERR_ACCES
: Permission denied. The caller does not have the correct permission to perform the requested operation.

NFSERR_EXIST
: File exists. The file specified already exists.

NFSERR_NODEV
: No such device.

NFSERR_NOTDIR
: Not a directory. The caller specified a non-directory in a directory operation.

NFSERR_ISDIR
: Is a directory. The caller specified a directory in a non-directory operation.

NFSERR_FBIG
: File too large. The operation caused a file to grow beyond the server's limit.

NFSERR_NOSPC
No space left on device. The operation caused the server's filesystem to reach its limit.

NFSERR_ROFS
Read-only filesystem. Write attempted on a read-only filesystem.

NFSERR_OPNOTSUPP
Operation not supported. The socket cannot support the specified operation.

NFSERR_ENOBUFS
No buffers available. All buffer space for the requested type is exhausted.

NFSERR_NAMETOOLONG
File name too long. The file name in an operation was too long.

NFSERR_NOTEMPTY
Directory not empty. Attempted to remove a directory that was not empty.

NFSERR_DQUOT
Disk quota exceeded. The client's disk quota on the server has been exceeded.

NFSERR_STALE
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.

NFSERR_REMOTE
Too many levels of remoteness. The operation attempted to access one server as the intermediary between a client and another server.

NFSERR_WFLUSH
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
} 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.

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 on 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, that is, 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 that the file takes up on disk; **fsid** is the file system identifier for the filesystem that contains 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 will be fixed in future versions. The descriptions given below specify the 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 an 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**. Afterwards, 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 UNIX 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 which 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 correctly it would be necessary 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;
       diropres  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 remote mounted filesystem. Currently, the **fhandle** of the underlying directory is returned. This is not completely acceptable, as the clients see a different view of the filesystem to that seen by the server.

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** is NFS_OK, this function always returns NFSERR_OPNOTSUPP. Symbolic links are not supported on the TITAN implementation of NFS.

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;
                string  data<MAXDATA>;
            default:
                struct {}
        } reply;
```

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;
```

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;
       diropres  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_REMOVE (which, request) returns (status)
       diropargs        which;
       stat             status;
       svc_req          request;
```

The file **which.name** is removed from the directory given by **which.dir**. The service request **request** checks for duplication. 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, request) returns (status)
       fhandle          from;
       diropargs        to;
       stat             status;
       svc_req          request;
```

Creates the file **to.name** in the directory given by **to.dir**, which is a hard link to the existing file given by **from**. The service request **request** checks for duplication. 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 ensure 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;
```

If **status** is NFS_OK, this function always returns NFSERR_OPNOTSUPP. Symbolic lilnks are not supported in the TITAN implementation of NFS.

Create Directory (Procedure 14, Version 2)

```
14. NFSPROC_MKDIR (where, attributes, request) returns (reply)
        diropargs where;
        sattr     attributes;
        diropres  reply;
        svc_req   request;
```

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 service request **request** checks for duplication. 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, request) returns (status)
        diropargs        which;
        stat             status;
        svc_req          request;
```

The existing, empty directory **which.name** in the directory given by **which.dir** is removed. A service request **request** checks for duplication. 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;
```

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** which is a unique number to identify the file within a filesystem, the **name** of the file, and a **cookie** which 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 . The **eof** flag has a value of **TRUE** if there are no more entries 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:

tsize 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.

bsize The block size in bytes of the filesystem.

blocks The total number of **bsize** blocks on the filesystem.

bfree The number of free **bsize** blocks on the filesystem.

bavail The number of **bsize** blocks available to non-privileged users.

Bugs: This call does not work well if a filesystem accepts variable size blocks.

Mount Protocol Definition

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 **fhandle** structure, which is the same for both protocols.

RPC Information

The following sections provide specific information about the components of the RPC procedures.

Authentication

The NFS service uses **AUTH_UNIX** style authentication except in the **NULL** procedure, where **AUTH_NONE** is also allowed.

Protocols

The mount service is currently supported on UDP/IP only.

Constants

These are the RPC constants needed to call the **MOUNT** service. They are given in decimal.

PROGRAM	100005
VERSION	1

Port Number

The server's portmapper, described in chapter 3, should be consulted to find the port number on which a program is awaiting service.

Sizes

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

MNTPATHLEN

The maximum number of bytes in a pathname argument is 1024.

MNTNAMLEN

The maximum number of bytes in a name argument is 255.

FHSIZE

The size in bytes of the opaque file handle is 32.

Basic Data Types

The following sections describe the basic data types passed by the mount protocol.

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.

This is the same as the **fhandle** 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 **dirpath** is a normal UNIX pathname of a directory.

Name

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

The type **name** 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 the 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**. Afterwards, 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.

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 on the server 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**.

YELLOW PAGES PROTOCOL SPECIFICATION

CHAPTER SIX

The Yellow Pages (YP), a distributed lookup service, is a network service providing 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.

Introduction

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 (called 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, but not distributed: 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 must 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 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 located 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 been assigned a procedure number.

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 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 description language. In general, the data description 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 a 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**.

YP Database Servers

The following sections describe the structure and manipulation of map databases.

Map Structure

Maps are named sets of key-value pairs. Keys and their values are counted binary objects, and may be ASCII information, but they 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. The YP contains no syntactic nor semantic knowledge of the map contents; neither does the YP determine nor 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.

Operations on Maps

The following sections describe the various operations performed on map databases.

Match Operation

The 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.

Enumeration Operation

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 user 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 requestor with a summary status.

Domains

Domains provide a second level for naming within the YP subsystem. They are names for sets of maps and, 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 important, so that programs can be written that are insensitive to both location and the invoking user.

Limitations

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.

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

The following sections describe version 2 of the protocol.

RPC Constants

The YP database server program number and the current YP protocol version number are given in decimal. YPPROG 100004 YPVERS 2

YPMAXRECORD

The total maximum size of key and value for any pair is 1024. The absolute sizes of the key and value may divide this maximum arbitrarily.

YPMAXDOMAIN

The maximum number of characters in a domain name is 64.

YPMAXMAP

The maximum number of characters in a map name is 64.

YPMAXPEER

The maximum number of characters in a YP server host name is 64.

Remote Procedure Return Values

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

```
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;
```

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 int 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** parameter contains a binary value representing the value of the map's order number; if unavailable, this number 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
```

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 procedure 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 procedure returns **TRUE** if the server serves **domain**, and **FALSE** otherwise. It 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 timeout on the response.

The current implementation 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 procedure returns the value associated with the key **keydat** in **req**. If the **status** element in **resp** has the value **YP_TRUE**, the value data are returned in **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 procedure 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 procedure 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 requestor 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 procedure 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 procedure 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 procedure 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 GMT.

Get All Maps in Domain (Procedure 11, Version 2)

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

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

YP Binders

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

Introduction

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.

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 in some way, but that the way is either complicated, time-consuming, or resource-consuming. If either of these assumptions is untrue, then the implementation is probably unsuitable for a YP binder.

If a YP binder is implemented, it can provide added value beyond the binding: for instance, it can verify that the binding is correct and that the YP database server is alive and well. 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.

RPC Constants

The YP binder protocol program number and the current YP binder protocol version are given in decimal.

```
YPBINDPROG    100007
YPBINDVERS         2
```

YPMAXDOMAIN

The maximum number of characters in a domain name is 64. This is identical to the constant defined earlier 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 procedure 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 procedure 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 procedure instructs a YP binder to use the passed information as its current binding information for the passed domain.

NFS SYSTEM ADMINISTRATION

CHAPTER SEVEN

This chapter introduces the network services. It describes the services currently available and defines some terms in the network environment.

Following that, the two types of service now available on the network (NFS service and YP service) are introduced and explained. Within each of these two sections, you will find information about periodic maintenance and trouble-shooting for the service under discussion.

While some of this material tends to be theoretical, its specific implications appear again and again as you become familiar with system administration. For example, a user running the YP must understand that some typical UNIX procedures have changed in the YP environment. That is also true of using the network file system. This chapter covers only those aspects of network services necessary for performing the duties of system administration.

Introduction

Any machine that provides one of the network services is a **server**. A single machine may provide more than one service. A typical configuration would be for one machine to act as both an NFS-server and a YP-server.

In each of the network services, servers are entirely passive. The servers wait for clients to call them; they never call the clients.

A **client** is any entity that accesses a network service. The term entity is used because the thing doing the accessing may be an actual machine or simply a process generated by a piece of software.

The degree to which clients are bound to their servers varies with each of the network services. For example, a YP client binds

randomly to one of the YP servers by broadcasting a request. At any point, the YP client may decide to broadcast for a new server. However, an NFS client selects a specific server from which to mount a given filesystem.

In all cases, the client initiates the binding. The server completes the binding subject to access control rules specific to each service. Since most network administration problems occur at bind time, a system administrator should know how a client binds to a server and what (if any) access control policy each server uses.

TOPS and Network Services

The TITAN operating system, (TOPS), is based on AT&T's System V Release 3 version of the UNIX operating system. Unlike many recently marketed distributed operating systems, the UNIX operating system was designed without knowledge that networks existed. This "networking ignorance" presents three impediments to linking UNIX and TOPS with currently available high performance networks:

(1) The UNIX operating system was never designed to yield to a higher authority (like a network authentication server) for critical information or services. As a result some UNIX semantics are hard to maintain "over the net". For example, it may not always be appropriate to trust user id 0 (root).

(2) Some UNIX execution semantics are difficult. For example, the UNIX operating system allows a user to remove an open file, yet the file does not disappear until closed by everyone. In a network environment a client UNIX machine may not own an open file. Therefore, a server may remove a client's open file.

(3) When a UNIX machine crashes, it takes all its applications down with it. When a network node crashes (whether client or server) it should not drag all of its bound neighbors down. The treatment of node failure on a network raises difficulties in any system and is especially difficult in a UNIX environment. A system of "stateless" protocols has been implemented to circumvent the problem of a crashing server dragging down its bound clients. Stateless here means that a client is independently responsible for completing work, and that a server need not remember anything from one call to the next. In other words, the server keeps no state.

With no state left on the server, there is no state to recover when the server crashes and comes back up. From the client's point of view, a crashed server appears to be only a very slow server.

NFS remains compatible with the UNIX operating system whenever possible. However, there are certain incompatibilities. These are typically of two kinds: first, those issues that would make a networked UNIX environment evolve into a distributed operating system, rather than a collection of network services; and second, those issues that would make crash recovery extremely difficult from both the implementation and administration point of view.

All incompatibilities are documented in the appropriate sections of this chapter.

Debugging TOPS In The Network Environment

Most problems involving NFS network services lie in one of the following four areas, listed here in order of probability.

(1) The network access control policies do not allow the operation or architectural constraints prevent the operation.

(2) The client software or environment is broken.

(3) The server software or environment is broken.

(4) The network is broken.

The following sections present specific instructions on how to check for these causes of failure in the NFS and YP environments.

NFS: The Network File System

The NFS enables users to share file systems over the network. A client may mount or unmount file systems from an NFS server machine. The client always initiates the binding to a server's file system by using the *mount*(1M) command. Typically, a client remembers specific remote file systems and their mount points by placing lines like these in the file */etc/fstab*:

```
titan:/usr2 /usr2 NFS rw,hard
venus:/usr/man /usr/man NFS rw,hard
```

See *fstab*(4) for a full description of the format.

Since clients initiate all remote mounts, NFS servers keep control over who may mount a file system by limiting named file systems to desired clients with an entry in the */etc/exports* file. For example:

```
/usr/local                              # export to any machine
/usr2   bigmo larry curley              # export to only these machines
```

Note that pathnames given in */etc/exports* must be the mount point of a local file system. See *exports*(4) for a full description of the format.

How The NFS Works

Two remote programs implement the NFS service — *mountd*(1M), the mount daemon, and *nfsd*(1M), the NFS daemon. A client's *mount* request talks to **mountd** which checks the access permission of the client and returns a pointer to a filesystem. After the **mount** completes, access to that mount point and below goes through the pointer to the server's **nfsd** daemon using **rpc**(4). Client kernel file access requests (delayed-write and read-ahead) are handled by the *biod*(1M), I/O daemons on the client.

Becoming An NFS Server

An NFS server is simply a machine that exports a file system or systems. The following steps must be taken to enable any machine to export a file system.

(1) The super-user must place the mount-point pathname of the file system to be exported in the file */etc/exports*. See *exports*(4) for file format details. For example, to export */usr/lbin*, the export file would look like:

```
/usr/lbin
```

Of course, an NFS server may only export file systems of its own.

(2) **/etc/mountd** must be running for a remote mount to succeed. This is started from the system startup script, usually */etc/rc*.

(3) Remote mount also needs some number of **nfsd** NFS daemon processes to be running on the NFS server. The actual number depends on the number of client NFS requests that the server should be able to handle concurrently, and thus depends on the speed and capacity of the server machine.

This example shows four **nfsd** daemons. Check the system startup script, such as */etc/rc*, for lines like these:

```
if [ -f /etc/nfsd -a -f /etc/exports ]; then
/etc/nfsd 4                      >/dev/console
```

Add these lines, or ones similar, to the new NFS server's system startup script to enable **nfsds**. You can enable **nfsds** at any time by typing, as super-user:

```
# /etc/nfsd 4
```

After these steps, the NFS server should be able to export the named file system.

Remote Mounting A File System

Any exported file system can be remote mounted onto a machine, as long as its server can be reached over the network and the machine is included in the */etc/exports* list for that file system. On the machine where the file system is to be mounted, type the following, as superuser:

```
# mount -f NFS server_name:/file_system   /mount_point
```

For example, to mount the manual pages from remote machine **elvis** on the directory */usr/elvis.man*:

```
# mount -f NFS elvis:/usr/man /usr/elvis.man
```

To make sure the file system is mounted where it is expected to be, use the *mount*(1M) command without any arguments. This displays the currently mounted file systems.

Frequently used file systems are listed, with any needed options, in the file */etc/fstab*. See *fstab(4)* for the syntax and contents of the file.

Debugging the Network File System

Before trying to debug the NFS, read the section on how the NFS works and also the man pages for *mount*(1M), *nfsd*(1M), *biod*(1M), *showmount*(1M), *rpcinfo*(1M), *mountd*(1M), *fstab*(4), *mtab*(4) and *exports*(4). It is not necessary to understand them fully, but the user should be familiar with the names and functions of the various daemons and database files.

When tracking down an NFS problem keep in mind that, like all network services, there are three main points of failure: the server, the client, or the network itself. The debugging strategy outlined below tries to isolate each individual component to find the one which is not working.

For example, consider a sample mount request as made from an NFS client machine:

```
$ mount -f NFS krypton:/usr/src /krypton.src
```

The example asks the server machine **krypton** to return a file handle (**fhandle**) for the directory */usr/src*. This **fhandle** is then passed to the kernel in the *mount*(2) system call. The kernel looks up the directory */krypton.src* and, if everything is correct, ties the **fhandle** to the directory in a mount record. From now on all file system requests to that directory and below go through the **fhandle** to the server "krypton".

This describes the way in which the system should work. We now look at the things which may go wrong: first, some general pointers and then a list of the possible errors and what might have caused them.

General Hints

When there are network or server problems, programs that access hard mounted remote files fail in ways different from those which access soft mounted remote files. Hard mounted remote file systems cause programs to retry until the server responds again. Soft mounted remote file systems return an error after trying for a while. **Mount** is like any other program; if the server for a remote file system fails to respond it retries the mount request until it succeeds. A soft mount tries once in the foreground then puts itself in the background and keeps trying.

Once a hard mount succeeds, programs that access hard mounted files hang as long as the server fails to respond. In this case, NFS prints a "*server not responding*" message on the console. On a soft mounted file system programs get an error when a file is accessed and the server is dead.

If a client is having NFS trouble, the first check must be to make sure the server is up and running. From a client, type the following command to see if the server is up at all:

```
$ rpcinfo -p server_name
```

It prints out a list of program, version, protocol, and port numbers that resembles:

```
program vers proto   port
 100004    2   udp   1027
 100007    2   tcp   1025
 100007    2   udp   1033
 100007    1   tcp   1025
 100007    1   udp   1033
```

You can use *rpcinfo* to check if the **mountd** server is running:

```
$ rpcinfo -u server_name 100005 1
```

This returns:

```
program 100005 version 1 ready and waiting
```

If these steps fail, try a login on the server's console to see if it is running.

If the server is alive but a client machine can't reach it, check the Ethernet connections between the machines.

If the server and the network are alive, use **ps** to check the client daemons. A **portmap, ypbind,** and several **biod** daemons should be running. For example,

```
$ ps -ef
```

should print lines for **/etc/portmap, /etc/ypbind,** and **biod.**

The four sections below deal with the most common types of failure. The first covers the steps to be taken if a remote mount fails, the next three discuss servers which do not respond when file systems are mounted.

Remote Mount Failed

This section deals with problems related to mounting. If *mount* fails for any reason, check the following sections for specific details about what to do. They are arranged according to where they occur in the mounting sequence and are labelled with the

error message likely to be displayed. It is assumed that YP is in use.

Mount can get its parameters either from the command line or from the file */etc/fstab* (see *mount*(1M)). The next example assumes command line arguments, but the same debugging techniques would apply if */etc/fstab* were the source of the options.

The interaction of the various parts of the **mount** request should be considered. In the example **mount** request given earlier:

```
$ mount -f NFS krypton:/usr/src /krypton.src
```

mount goes through the following steps to mount a remote file system.

(1) *mount* opens */etc/mnttab* and checks that this mount has not already been done.

(2) *mount* parses the first argument into host **krypton** and remote directory */usr/src*.

(3) *mount* may call the YP binder daemon **ypbind** to determine which server machine to find the YP server on. It may then call the **ypserv** daemon on that machine to get the internet protocol (IP) address of krypton.

(4) *mount* calls krypton's port mapper to get the port number of **mountd**.

(5) *mount* calls krypton's **mountd** and passes it */usr/src*.

(6) Krypton's **mountd** reads */etc/exports* and looks for the exported file system that contains */usr/src*.

(7) Krypton's **mountd** may call the Yellow Pages server **ypserv** to expand the host names and netgroups in the export list for */usr/src*.

(8) Krypton's **mountd** does a *getfh*(2) system call on */usr/src* to get the **fhandle**.

(9) Krypton's **mountd** returns the **fhandle**.

(10) Back on the client, *mount* does a *mount*(2) system call with the **fhandle** and */krypton.src*.

(11) *mount* checks if the caller is superuser and if */krypton.src* is a directory.

(12) *mount* does a *statfs*(2) call to krypton's NFS server (**nfsd**).

(13) *mount* opens */etc/mnttab* and adds an entry.

Error Messages

Any one of these steps can fail, some of them in more than one way. The following paragraphs give detailed descriptions of the failures associated with specific error messages.

mount: cannot open /etc/mnttab
The table of mounted file systems is kept in the file */etc/mnttab*(5). This file must exist before mount can succeed. */etc/mnttab* is created when the system is booted, and is maintained automatically after that by the *mount* and *umount* commands.

mount: /dev/nfsd is already mounted, ... is busy, or allowable number of mount points exceeded
This message reveals an attempt to mount a filesystem which is already mounted, or for which there is already an entry in */etc/mnttab*. All NFS *mount* requests that fail with this message display the name **/dev/nfsd** (a byproduct of the implementation) regardless of the actual mount request.

mount: ... or ..., no such file or directory
The **-f** NFS or **krypton:** part of

```
# mount -f NFS krypton:/usr/src /krypton.src
```

was probably omitted. The *mount* command assumes that a local mount is being done unless the **-f** flag is used on the command line, or the requested directory as listed in */etc/fstab* specifies filesystem type NFS.

More simply, this message also appears when for a correct mount request, the specified local mount point is not an existing directory.

mount: cannot open </etc/fstab>
Mount tried to look up the information needed to complete a mount request in */etc/fstab* but there was no such file. This file must be created by the system administrator as part of initial system setup.

... not in hosts database

The system name specified on the mount request suffixed by the ":" is not listed in the file */etc/hosts*. Check the spelling of the hostname and placement of the colon in the mount call.

mount: directory argument < ...> must be a full path name

The second argument to *mount* is the path of the directory to be covered. This must be an absolute path starting at /.

mount: ... server not responding(1): RPC_PMAP_FAILURE - RPC_TIMED_OUT

Either the server to which the mount is being attempted is down, or its portmapper is dead or hung. Make an attempt to log in to that machine: if that attempt succeeds, then the problem may be in the portmapper. Run the following command from your system to test the portmapper on the server system:

```
# rpcinfo -p hostname
```

The result should be a list of registered programs. If this is not the case, kill the remote portmapper and restart it. Restarting the port mapper is a complicated process because all registered services are lost, and their associated daemons must be restarted also. Do this as the super-user:

```
# ps -ef
```

to find the process ids of **portmap** and other service daemons. To kill the found daemons, use the following command:

```
# kill -9 portmap_pid mountd_pid nfsd_pid
```

Now, restart the daemons:

```
# /etc/port map
# /etc/mountd
# /etc/nfsd
```

Another alternative to all this is to simply reboot the server when it is convenient. Because of the stateless nature of the NFS server implementation, there should be no adverse effect on the clients of the system other than the time that they will be suspended awaiting the return of the server.

If the server is up but it is not possible to *rlogin* to it, check the client's Ethernet connection by trying to *rlogin* to some

other machine. Also check the server's Ethernet connection.

mount: ... server not responding: RPC_PROG_NOT_REGISTERED
This means that mount got through to the port mapper but the NFS mount daemon **mountd** was not registered. Check the server to ensure that **/etc/mountd** exists and is running.

mount: /dev/nfsd or . . . , no such file or directory
Either the remote directory does not exist on the server or the local directory doesn't exist. Again note that **/dev/nfsd** is always printed to represent the remote directory.

mount: access denied for . . . : . . .
Your machine on which the mount attempt is being made is not in the server's export list for the file system to be mounted. Get a list of the server's exported file systems by running:

```
# showmount -e hostname
```

If the file system which is wanted is not in the list, or the machine name or netgroup name is not in the user list for the file system, check the */etc/exports* file on the server for the correct file system entry. A file system name that appears in the */etc/exports* file but not in the output from *showmount* indicates a failure in **mountd**. Either it could not parse that line in the file, or it could not find the file system, or the filesystem name was not a local mounted file system. See *exports*(4) for more information.

This message may also indicate that authentication failed on the server. It may be displayed because the machine which is attempting the mount is not in the server's export list, or because the server is not aware of the machine. or because the server does not believe the identity of the machine. Check the server's */etc/exports*.

mount: . . . : no such file or directory
The remote path on the server does not exit or is not a directory.

mount: not super user
You must be logged on as the super-user in order to use the *mount* command because it affects the file system for the whole machine.

Programs Hung

If programs hang doing file related work, the NFS server may be dead. The message

```
NFS server <sysname> not responding, still trying
```

may be displayed on the machine's console. The message includes the name of the NFS server which is down.

This is probably a problem either with one of the NFS servers or with the Ethernet. Programs can also hang if a **yp** server dies (see **yp** below).

If a machine hangs completely, check the server(s) from which filesystems have have been mounted. If one (or more) of them is down, client machines may hang. When the server comes back up, programs continue automatically and are not be affected.

If a soft mounted server dies, other work should not be affected. Programs that time-out trying to access soft mounted remote files will fail, but it is still be possible to get work done on other file systems.

If other clients of the server seem to be functioning correctly, check the Ethernet connection and connection of the server.

Everything Works Slowly

If access to remote files seems unusually slow, check the server by entering (on the server):

```
# ps -ef
```

If the server is functioning and other users are getting good response, block I/O daemons on the client should be checked by typing **ps -ef** (on the client) and looking for **biod**. If the daemons are not running or are hung, find the process ids by typing:

```
# ps -ef | grep biod
```

Now kill the processes:

```
# kill -9 pid1 pid2 pid3 pid4
```

Restart the daemons with

```
# /etc/biod 4
```

To determine whether the daemons are hung, use **ps**, then copy a large file. Another **ps** shows whether the **biods** are accumulating CPU time: if not, they are probably hung.

If **biod** appears to be functioning correctly, check the Ethernet connection. Use *nfsstat*(1), with the **-c** or **-s** option to discover whether a client is doing a lot of retransmitting. A retransmission rate of 5% is considered high. Excessive retransmission usually indicates a bad Ethernet board, a bad Ethernet tap, a mismatch between board and tap, or a mismatch between the client machine's Ethernet board and the server's board.

Incompatibilities With Earlier UNIX Versions

A few things work in different ways, or not at all, on remote NFS file systems. The next section discusses the incompatibilities, and offers suggestions on working around them.

No SU Over The Network

Under the NFS a server exports file systems it owns so that clients may remote mount them. When a client becomes superuser, it is denied permission on remote mounted file systems. Consider the following example:

```
$ cd
$ touch test1 test2
$ chmod 777 test1
$ chmod 700 test2
$ ls -l test*
-rwxrwxrwx  1 jsbach      0 Mar 24 16:12 test1
-rwx------  1 jsbach      0 Mar 24 16:12 test2
```

The example is tried again by the super-user:

```
$ su
Password:
# touch test1
# touch test2
touch: test2: Permission denied
# ls -l test*
-rwxrwxrwx  1 jsbach      0 Mar 21 16:16 test1
-rwx------  1 jsbach      0 Mar 21 16:12 test2
```

The problem usually shows up during the execution of a set-uid root program. Programs that run as root cannot access files or directories unless the permission for "other" allows it.

Another aspect of this problem is that ownership of remote mounted files cannot always be changed; specifically, if files are on a server that does not permit users to execute *chown*. Since root is treated as the "other" user for remote accesses, only root on the server can change the ownership of remote files. For example, consider a user trying to change ownership of a new program, **a.out**, which must be set-uid root. It does not work:

```
$ chmod 4777 a.out
$ su
Password:
# chown root a.out
a.out: Not owner
```

To change the ownership, the user must either login to the server as root and then make the change, or move the file to a file system owned by the user's machine (for example */usr/tmp* will usually be owned by the local machine) and make the change there.

File Operations Not Supported

File locking of directories is not supported on remote file systems.

In addition, append mode and atomic writes are not guaranteed to work on remote files accessed by more than one client simultaneously.

Cannot Access Remote Devices

In the NFS it is not possible to access a remote mounted device or any other character or block special file, or named pipes.

Clock Skew In User Programs

Since the NFS architecture differs in some minor ways from earlier versions of the UNIX operating system, you should be aware of those places where old programs could run up against these incompatibilities. The section *Architectural Incompatibilities* discusses features which do not work over the network.

Because each machine keeps its own time, the clocks may be out of sync between the NFS server and client. This might cause problems. Consider the following example.

Many programs assume that an existing file could not have been created in the future. For example, the command **ls -l** has two basic forms of output, depending upon how old the file is:

```
# date
Sat Apr 12 15:27:48 GMT 1986
# touch file2
# ls -l file*
-rw-r--r--  1 root          0 Dec 27  1984 file
-rw-r--r--  1 root          0 Apr 12 15:27 file2
```

The first type of output from **ls** prints the month, day, and year of last file modification if the file is more than six months old. The second form prints the month, day, hour, and minute of last file modification if the file is less than six months old.

Ls calculates the age of a file by simply subtracting the modification time of the file from the current time. If the result is greater than six months, the file is "old".

Assume that the time on the server is Apr 12 15:30:31, which is three minutes ahead of the local machine's time:

```
# date
Apr 12 15:27:31 GMT 1986
# touch file3
# ls -l file*
-rw-r--r--  1 root          0 Dec 27  1983 file
-rw-r--r--  1 root          0 Apr 12 15:26 file2
-rw-r--r--  1 root          0 Apr 12  1986 file3
```

The difference between the current time and the library's modify time is a huge unsigned number, equal to –180 seconds.

Thus, **ls** believes the new file was created long ago in the past.

Ls has been modified to deal with files which are created a short time in the future.

In general, remember that applications which depend upon local time and/or the file system timestamps have to deal with clock skew problems if remote files are used.

YP: The Yellow Pages Service

The YP is a distributed network lookup service:

- YP is a distributed system; the database is fully replicated at several sites, each of which runs a server process for the database. These sites are known as YP servers. At steady state, it doesn't matter which server process answers a client request; the answer will be the same all over the net. This allows multiple servers per network, and gives YP service a high degree of availability and reliability.

- YP is a lookup service. It maintains a set of databases which may be queried. A client may ask for the value associated with a particular key within a database, and may enumerate every key-value pair within a database.

- YP is a network service. It uses a standard set of access procedures to hide the details of where and how data is stored.

The YP Map

The YP system serves information stored in YP "maps". Each map contains a set of keys and associated values. For example, in a map called **hosts**, all the host names within a network are the keys, and the internet addresses of these host names are the values. Each YP map has a **mapname** used by programs to access it. Programs must know the format of the data in the map. Many of the current maps are derived from ASCII files traditionally found in */etc*: **hosts, group, passwd** and a few others. The format of the data within the YP map is identical, in most cases, to the format within the ASCII file. Maps are implemented as files located in the subdirectories of the directory */etc/yp* on YP server machines.

The YP Domain

A YP domain is a named set of YP maps. Users can determine and set YP domains with the *domainname*(1) command. Note that YP domains differ from both Internet domains and *sendmail* domains. A YP domain is simply a directory in */etc/yp* where a YP server holds all of the YP maps. The name of the subdirectory is the name of the domain. For example, maps for the **literature** domain would be in */etc/yp/literature*.

A domain name is required for retrieving data from a YP database. Each machine on the network belongs to a default domain,

set at boot time in the system startup script, such as */etc/rc,* with the *domainname*(1) command. A domain name must be set on all machines, both servers and clients. Further, use a single name on all machines on a network.

Masters And Slaves

In the YP environment only a few machines have a set of YP databases. The YP service makes the database set available over the network. A YP client machine runs YP processes and requests data from databases on other machines. Two kinds of machines have databases: a YP slave server and a YP master server. The master server updates the databases of the slave servers. Only make changes to databases on the YP master server. The changes propagate from the master server to the YP slave servers. If YP databases are created or changed on slave server machines instead of master server machines, the YP's update algorithm will be broken. Create and modify all databases on the master server machine.

A server may be a master with regard to one map, and a slave with regard to another. Random assignment of maps to server machines could introduce a great deal of confusion. You are strongly urged to make a single server the master for all the maps created by *ypinit*(8) within a single domain. This chapter assumes that one server is the master for all maps in the database.

Yellow Pages Overview

The YP can serve any number of databases. Typically these include some files that used to be found in */etc*; for example, programs used to read the */etc/hosts* file to find an Internet address. When a new machine was added to the network, a new entry had to be added to every machine's */etc/hosts* file. With the YP, programs that need to look at the */etc/hosts* file now do a remote procedure call (RPC) to the servers to get the information.

Most of the information describing the structure of the YP system and the commands available for that system are contained in in Section 2 of this manual and are not repeated here. For quick reference, a list of the reference manual pages and an abstract of their contents are given below.

- **ypserv**(1M) describes the processes which comprise the YP system. These are **ypserv,** the YP database server daemon, and **ypbind,** the YP binder daemon. **Ypserv** must run on each YP server machine. **Ypbind** must run on all machines

which use YP services, both servers and clients.

- **ypfiles**(4) describes the database structure of the YP system.

- **ypinit**(1M) is a database initialization tool. Many maps must be constructed from files located in */etc*, such as */etc/hosts*, */etc/passwd* and others. *Ypinit* does all such construction automatically. In addition, it constructs initial versions of maps required by the system but not built from files in */etc*; an example is the map "ypservers". This tool should be used to set up the master YP server and the slave YP servers for the first time. It should not be used as a general administrative tool for running systems.

- **ypmake**(1M) describes the use of */etc/yp/Makefile*, which builds several commonly-changed components of the YP's database. These are the maps built from several ASCII files normally found in */etc*: *passwd*, *hosts*, *group*, *netgroup*, *networks*, *protocols*, and *services*.

- **makedbm**(1M) describes a low-level tool for building a **dbm** file which is a valid YP map. Databases not built from */etc/yp/Makefile* may be built or rebuilt using *makedbm*. *Makedbm* may also be used to "disassemble" a map so that the key-value pairs which comprise it can be seen. The disassembled form may also be modified with standard tools (such as editors, *awk*, *grep* and *cat*), and is in the form required for input back into *makedbm*.

- **ypxfr**(1M) moves a YP map from one YP server to another, using the YP itself as the transport medium. It can be run interactively, or periodically from **crontab**. In addition, **ypserv** uses **ypxfr** as its transfer agent when it is asked to transfer a map.

- **yppush**(1M) describes a tool to administer a running YP system. It is run on the master YP server. It requests each of the **ypserv** processes within a domain to transfer a particular map, waits for a summary response from the transfer agent, and prints out the results for each server.

- **ypset**(1M) tells a **ypbind** process (by default the local one) to get YP services for a domain from a named YP server.

- **yppoll**(1M) asks any **ypserv** for the information about a single map which it holds internally.

- **ypcat**(1) dumps out the contents of a YP map. It should be used in cases when it does not matter which server's version is seen. If a particular server's map is required, users must *rlogin* to that server (or use *rsh*) and use *makedbm.*
- **ypmatch**(1M) prints the value for one or more specified keys in a YP map. Once again, there is no control over which server's version of the map is seen.
- **ypwhich**(1M) tells which YP server a node is using at the moment for YP services, or which YP server is master of some named map.

Yellow Pages Installation and Administration

The next several sections describe the procedures for creating YP servers, adding clients, mapping, debugging YP clients and servers, and system security.

Setting Up A Master YP Server

To create a new master server, login as the superuser and use the **cd** command to get to the directory */etc/yp.* Run *ypinit*(1M), with the **-m** flag. Set up the default domain name and the hostname; the usual case is that *domainname* and *hostname* have been set up from the system startup script. *Ypinit* prompts for a list of other hosts which also will be YP servers. (Initially, this is the set of YP slave servers, but at some future time any of them might become the YP master server.) It is possible, but not necessary, to add other hosts at this time.

Prior to running *ypinit*, the following files in */etc* must be complete and up-to-date: *passwd, hosts, ethers, group, networks, protocols,* and *services*. In addition, if you know how */etc/netgroup* is going to be set up, do the set up before running *ypinit*; otherwise, *ypinit* makes an empty "netgroup" map.

For security reasons, access to the master YP machine may be restricted to a smaller set of users than that defined by the complete */etc/passwd.* To do this, copy the complete file to somewhere other than */etc/passwd,* and delete the undesired users entries from the remaining */etc/passwd.* For a security-conscious system, do not include in this smaller file the YP escape entry discussed in the next section.

To start providing YP services, invoke */etc/ypserv*. It starts automatically from the system startup script on subsequent reboots.

Setting Up A YP Client

To set up a YP client, edit the local files as described below. Start */etc/ypbind* if it is not running already. With the ASCII databases of */etc* abbreviated and */etc/ypbind* running, the processes on the machine will be clients of the YP services. At this point, a YP server must be available; many processes hang if no YP server is available while **ypbind** is running. The possible alterations to the client's */etc* database (discussed below) should be noted. Because some files may not exist or may be specially altered, the ways in which the ASCII databases are used are not always obvious. The escape conventions used within those files to force inclusion and exclusion of data from the YP databases are found in the following reference pages in Section 2: *passwd*(4), *hosts*(4), *host.equiv*(4) and *group*(4). In particular, note that changing passwords in */etc/passwd* (by editing the file, or by running *passwd*(1)) will only affect the local client's environment. The YP password database should be changed by running *yppasswd*(1).

Once the decision has been made to serve a database with the YP, it is desirable that all nodes in the net access the YP's version of the information, rather than the potentially out-of-date information in their local files. That policy is enforced by running a **ypbind** process on the client node (including nodes which may be running YP servers) and by abbreviating or eliminating the files which traditionally implemented the database. The files in question are */etc/passwd*, */etc/hosts*, */etc/group* and */.rhosts*. The treatment of each file is discussed in this section.

- */etc/hosts.equiv* is not normally served by the YP. However, you can add escape sequences to activate the YP. This reduces problems with *rlogin* or *rsh*, which are sometimes caused by different */etc/hosts.equiv* files on the two machines.

  To allow anyone to log on to a machine, edit */etc/hosts.equiv* to contain a single line, with only the character "+" on it. Such a line means that all further entries will be retrieved from the YP rather than the local file.

  Alternatively, more control may be exercised over logins by using lines of the form:

```
+@trusted_group1
+@trusted_group2
-@distrusted_group
```

Each of the names to the right of the "@" character is assumed to be a net group name, defined in the global netgroup database. The netgroup database is served by the YP.

If none of the escape sequences is used, only the entries in */etc/hosts.equiv* are used; the YP is not used.

- */.rhosts* is not normally served by the YP. Its format is identical to that of */etc/hosts.equiv*. However, since this file controls remote root access to the local machine, unrestricted access is not recommended. The list of trusted hosts should be explicit. Alternately, net group names may be used for the same purpose.

- */etc/hosts* must contain entries for the local host's name and the local loopback name. These are accessed at boot time when the YP service is not yet available. After the system is running, and after the **ypbind** process is up, the */etc/hosts* file is not accessed at all. An example of the hosts file for YP client **zippy** is:

```
127.1           localhost
192.9.1.87      zippy         # John Q. Random
```

- */etc/passwd* should contain entries for root and the primary users of the machine, and an escape entry to force the use of the YP service. A few additional entries are recommended: **daemon,** to allow file-transfer utilities to work; **sync,** to run **sync** on a machine before rebooting, and **operator,** to let a dump operator login. A sample YP client's */etc/passwd* file is shown below:

```
root:wAm0Y41Enf6:0:10:God:/:/bin/csh
jrandom:uHP1gQ2:1429:10:J Random:/usr2/jrandom:/bin/csh
operator:VyZr6V9:333:20:sys op:/usr2/operator:/bin/csh
daemon:*:1:1::/:
sync::1:1::/:/bin/sync
+::0:0:::
```

The last line tells the library routines to use the YP service rather than give up the search. Entries which exist in */etc/passwd* mask analogous entries in the YP maps. In addition, earlier entries in the file mask later ones with the same user name, or the same uid. The order of the entries for **daemon** and for **sync** (which have the same uid) should be

noted and duplicated in users' own files.

- */etc/group* may be reduced to a single line:

  ```
  +:
  ```

  This will force all translation of group names and group ids to be made via the YP service. This is the recommended procedure.

Setting Up A Slave YP Server

The network must be working to set up a slave YP server.

To create a new slave server, login as the super-user and use the **cd** command to get to the directory */etc/yp.* From there, run *ypinit*(1M) with the **-s** switch, and name as master a host already set up as a YP server. Ideally, the named host is the master server, but it can be any host which has its YP database set up. The host must be reachable. The default domain name on the machine intended to be the YP slave server must be set up, and must be set to the same domain name as the default domain name on the machine named as the master.

After running *ypinit,* make copies of */etc/passwd, /etc/hosts,* and */etc/group.* For instance on a machine named **ypslave**:

```
ypslave$ cp /etc/passwd /etc/passwd-
```

In order to ensure that processes on the slave server actually make use of the YP services, rather than the local ASCII files, edit the original files in accordance with the instructions in the section, "Setting Up A YP Client." This ensures that YP slave server is also a YP client. Make backup copies of the edited files. For instance:

```
ypslave$ cp /etc/passwd /etc/passwd+
```

After the YP database is set up by *ypinit,* enter **/etc/ypserv** to start YP services. On subsequent reboots, it starts automatically from */etc/rc.*

Modifying Existing Maps

Databases served by the YP must be changed *on the master server.* The databases expected to change most frequently, such as */etc/passwd,* may be changed by first editing the ASCII file, and then running *make*(1) on */etc/yp/Makefile* — see also *ypmake*(1M).

Databases which are expected to change rarely, or databases for which no ASCII form exists (for example, databases which did not exist before the YP) may be modified "manually". In this case, use *makedbm*(1M) with the **-u** switch to disassemble them into a form which can be modified using standard tools (such as *awk*, *sed* or *vi*). Build a new version in one of two ways using *makedbm*(1M):

- Redirect the output of *makedbm* to a temporary file which can be modified and then fed back into *makedbm*, or
- Pipe the output of *makedbm* into *makedbm*. This is appropriate if the disassembled map can be updated by modifying it with *awk*, *sed* or a *cat* append, for instance.

Suppose you wish to create a non-standard YP map, called **mymap**. The map is to consist of key-value pairs in which the keys are strings such as al, bl, cl, etc. and the values are ar, br, cr etc.

Two possible procedures may be followed when creating new maps. The first is to use an existing ASCII file as input; the second is to use standard input.

For example, consider an existing ASCII file named */etc/yp/mymap.asc*, created with an editor or a shell script on a machine **ypmaster**. The map is located in the subdirectory *home_domain*. The YP map for this file can be created by:

```
ypmaster$  cd /etc/yp
ypmaster$  makedbm mymap.asc home_domain/mymap
```

If you wish to include another 2-tuple, (**dl**, **dr**) to the map, the modification can be made quite simply.

In all situations like this, the map must be modified by first modifying the ASCII file. Modifications which are made to the map, rather than in the ASCII file, will be lost. Make the modification in the following way:

```
ypmaster$  cd /etc/yp
ypmaster$  <make editorial change to mymap.asc>
ypmaster$ makedbm mymap.asc home_domain/mymap
```

If there is no original ASCII file, the YP map can be created from the keyboard as follows: (the machine name is **ypmaster**, and the default domain is **home_domain**):

```
ypmaster$ cd /etc/yp
ypmaster$ makedbm - home_domain/mymap
al ar
bl br
cl cr
<ctl D>
```

To modify the map, use *makedbm* to create a temporary ASCII intermediate file which can be edited using standard tools. For instance:

```
ypmaster$  cd /etc/yp
ypmaster$  makedbm -u home_domain/mymap > mymap.temp
```

At this point *mymap.temp* can be edited to contain the correct information. A new version of the database is created by the commands

```
ypmaster$  makedbm mymap.temp home_domain/mymap
ypmaster$  rm mymap.temp
```

The preceding paragraphs explained how to use some tools, but almost everything can be done automatically by *ypinit*(1M) and */etc/yp/Makefile*, unless non-standard maps have been added to the database or the set of YP servers has been changed after the system is running.

Whether the Makefile in */etc/yp* or some other procedure is used, the goal is the same: a new pair of **dbm** files must end up in the domain directory on the master YP server.

Propagation Of A YP Map

Propagating a map means moving it from place to place — in general, moving it from the master YP server to a slave YP server. Initially, the map is moved by *ypinit*(1M) as described above. After a slave YP server has been initialized, updated maps are transferred from the master server by *ypxfr*(1M). *Ypxfr* may be run in three different ways: periodically by *cron*(1M); by *ypserv*(1M), and interactively by a user.

Maps have differing rates of change; for instance **protocols.byname** may not change for long periods of time, but **passwd.byname** may change several times a day in a large organization. *Crontab*(4) entries can be set up to run *ypxfr* periodically at a rate appropriate for any map in a YP database. *Ypxfr* contacts the master server and transfers the map only if the master's copy is more recent than the local copy.

To avoid a **crontab** entry for each map, you can group the same change characteristics in a shell script, and run the shell script from a single **crontab** script file. Suggested groupings, mnemonically named, can be found in */etc/yp*: *ypxfr_1perhour*, *ypxfr_1perday*, and *ypxfr_2perday*. If the rates of change are inappropriate for a particular environment, these shell scripts are easy to modify or replace.

Run the same shell scripts at each YP slave server in the domain. Alter the time of execution from one server to another to prevent the checking from bogging down the master. To transfer the map from a particular server, specify the server, using *ypxfr*'s **-h** option within the shell script. Finally, check and transfer maps having unique change characteristics by explicit invocations of *ypxfr* within **crontab**.

Ypxfr is also invoked by *ypserv*, responding to a "Transfer Map" request. Such a request is made as an RPC call from *yppush*(1M). *Yppush* is run on the master YP server. It enumerates the YP map **ypserver** to generate a list of YP servers in the domain. To each of the named YP servers, it sends a "Transfer Map" request. *Ypserv* spawns a copy of *ypxfr*, invoking it with the **-C** flag, and passes it the information needed to identify the map and to call back the initiating *yppush* process with a summary status.

In the cases described, *ypxfr*'s transfer attempts and the results may be captured in a log file. If */etc/yp/ypxfr.log* exists, results will be appended to it. No attempt to limit the log file is made; the user is responsible for this. To turn off logging, remove the log file.

In the third case, run *ypxfr* as a command. Typically, this is done only in exceptional situations: for example when setting up a temporary YP server to create a test environment, or when quickly updating a YP server which has been out of service.

Making New YP Maps

Adding a new YP map entails getting copies of the map's **dbm** files into the domain directory on each of the YP servers in the domain. The actual mechanism has been described above. This section only describes the work required to get the proper mechanisms in place so that the propagation works correctly.

After deciding which YP server is the master of the map, modify */etc/yp/Makefile* on the master server so that the map can be conveniently rebuilt. Typically, a human-readable ASCII file is filtered

through *awk*, *sed*, or *grep* to make it suitable for input to *makedbm*(1M). Consult the existing *Makefile* as a source for programming examples. Use other mechanisms already in place in */etc/yp/Makefile* when deciding how to create dependencies that *make*(1) recognizes; specifically, the use of **.time** files allows users to see when the Makefile was last run for the map.

Support on the YP slave servers for propagation of the new maps consists of appropriate entries either in */usr/lib/crontab*, or in one of the **ypxfr** shell scripts mentioned in the previous section. To get an initial copy of the map, run *ypxfr* by hand on each of the slave servers. The map must be globally available before clients begin to access it. If the map is available from some YP servers, but not all, you see unpredictable behavior from client programs.

Adding A New YP Server

To add a new YP slave server, some maps on the master YP server must be modified. If the new server is a host which has not been a YP server before, the host's name must be added to the map **ypservers** in the default domain. The sequence for adding a server named **ypslave** to domain "home_domain" is:

```
ypmaster$ cd /etc/yp
ypmaster$ (makedbm -u home_domain/ypservers;\
        echo ypslave ypslave)|makedbm - tmpmap
ypmaster$ mv tmpmap.dir home_domain/ypservers.dir
ypmaster$ mv tmpmap.pag home_domain/ypservers.pag
ypmaster$ yppush ypservers
```

Set up the new slave YP server's databases by copying the databases from the YP master server **ypmaster**. Following a remote login to the new YP slave, use *ypinit*(1M) as follows:

```
ypslave$ cd /etc/yp
ypslave$ ypinit -s ypmaster
```

Now, complete the steps described in the section *Setting Up A Slave YP Server*.

Changing The Master Server

To change a map's master, first build the map at the new master. Because the old YP master's name occurs as a key-value pair in the existing map, it is not sufficient to use an existing copy at the new master or to send a copy there with *ypxfr*. The key must be reassociated with the new master's name. If the map has an ASCII source file, it must be present in its current version at the new

master. Locally remake the YP map (**example.map**) with the sequence:

```
newmaster$ cd /etc/yp
newmaster$ make example.map
```

/etc/yp/Makefile must be set up correctly for this to work. If the old master is to remain a YP server, edit the */etc/yp/Makefile* so that **example.map** is no longer made there. To this by commenting out the section of *oldmaster:/etc/yp/Makefile* which made **example.map**.

If the map only exists as a **dbm** data base, remake it on the the new master by disassembling an existing copy from any YP server and running the disassembled version back through *makedbm*. For example:

```
newmaster$ cd /etc/yp
newmaster$ ypcat -k example.map | makedbm - mydomain/example.map
```

After making the map on the new master, send a new copy of the map to the other (slave) YP servers. However, do not use *yppush*, as it causes the other slaves to try to get new copies from the old master, rather than the new one.

A typical method is to become superuser on the old master server and type:

```
oldmaster$ /etc/yp/ypxfr -h newmaster example.map
```

This places a copy on the old master. Now you may run *yppush*. The remaining slave servers still believe that the old master is the current master, and attempt to get the current version of the map from the old master. When they do so, they get the new map, which names the new master as the current master.

If this method fails, another must be used. On each YP server machine, the superuser executes the command sequence shown above. This is the worst case solution.

Debugging A Yellow Pages Client

This section is divided into two parts: first, those problems seen on a YP client, and then those problems seen on a YP server.

On Client: Commands Hang

The most common problem at a YP client node is for a command to hang and generate console messages:

```
yp: server not responding for domain <wigwam>. Still trying
```

Occasionally many commands hang, even though the system as a whole appears to be working correctly and new commands can be run.

This message indicates that **ypbind** on the local machine is unable to communicate with **ypserv** in the domain "wigwam". This often happens when machines that run **ypserv** have crashed. It may also occur if the net or the YP server machine is so overloaded that **ypserv** can't get a response back to the local machine's **ypbind** within the timeout period. Under these circumstances, all the other YP client nodes on the net show the same or similar problems. The condition is temporary in most cases; the messages usually go away when the YP server machine reboots and **ypserv** returns or when the load on the YP server nodes and/or the Ethernet decreases.

However, in certain circumstances which are described below, the situation does not improve.

- The YP client has not set, or has incorrectly set, **domainname** on the machine. Clients must use a domain name that the YP servers know. Use *domainname*(1) to see the client domain name, and compare this with the domain name set on the YP servers. Set the domain name in */etc/rc*. When */etc/rc* fails to set, or incorrectly sets, **domainname**, take the following steps: the superuser on the machine in question edits */etc/rc* to fix the **domainname** line with a proper domain name (this assures domain name will be correct every time the machine boots) and set **domainname** "by hand" so it is fixed immediately:

  ```
  # domainname good_domain_name
  ```

- If the domain name is correct, check the local net to make sure that it has at least one YP server machine. Users can only bind to a **ypserv** process on the local net, not on another accessible net. At least one YP server for each machine's domain must be running on the local net. Two or more YP servers improve availability and response characteristics for YP services.

- If the local net has a YP server, check to ensure that it is running. Check other machines on the local net. If several client machines have problems simultaneously, a server problem should be suspected. Try the *ypwhich* command on a client machine which is running. If *ypwhich* never returns an answer, kill its process. On the YP server machine, type the following command:

  ```
  $ ps -ef | grep yp
  ```

  to discover **ypserv** and **ypbind** processes. If the server's **ypbind** daemon is not running, start it by typing:

  ```
  $ /etc/ypbind
  ```

  If a **ypserv** process is running, do a *ypwhich* on the YP server machine. If *ypwhich* returns no answer, **ypserv** has probably hung and should be restarted. The superuser should kill the existing **ypserv** process and start */etc/ypserv*:

  ```
  $ kill -9 [some pid # from ps]
  $ /etc/ypserv
  ```

  If **ps** shows no **ypserv** process running, start one.

On Client: YP Service Unavailable

If YP services are unavailable on one machine although other machines on the network appear to be functioning, many different symptoms may be displayed, including:

- some commands appear to operate correctly while others terminate, printing an error message about the unavailability of YP;

- some commands enter a recovery mode particular to the program involved, and

- some commands or daemons crash with obscure messages or no message at all.

For example, messages like the following may be displayed:

```
my_machine$ ypcat myfile
ypcat: can't bind to yp server for domain <wigwam>.
        Reason: can't communicate with ypbind.

my_machine$ /etc/yp/yppoll myfile
```

```
Sorry, I can't make use of the Yellow Pages. I give up.
```

In such cases, run

```
my_machine$ ls -l
```

on a directory containing files owned by many users, including users not in the local machine's */etc/passwd* file — for example */usr*. If the **ls -l** reports file owners not in the local machine's */etc/passwd* file as numbers, rather than names, this is one more symptom that YP service is not working.

These symptoms usually indicate that the **ypbind** process is not running. The command **ps -ef** may be used to check for one. If it is not found, use

```
my_machine$ /etc/ypbind
```

to start it. YP problems should disappear.

On Client: Ypbind Crashes

If **ypbind** crashes almost immediately each time it is started, problems probably lie in some other part of the system. The presence of the **portmap** daemon should be ascertained by typing:

```
my_machine$  ps -ef | grep portmap
```

If it is not running, reboot the machine.

If **portmap** itself does not stay up or behaves strangely, problems are probably more fundamental. Check the state of the network.

It may be possible to talk to the **portmap** on the local machine from a machine operating normally. From such a machine, type the following command:

```
flipper$ rpcinfo -p your_machine_name
```

If **portmap** is functioning correctly, the output looks like this:

```
   program vers proto   port
    100004    2   udp   1027
    100007    2   tcp   1025
    100007    2   udp   1033
    100007    1   tcp   1025
    100007    1   udp   1033
    100004    2   tcp   1024
    100004    1   udp   1027
```

Note that port numbers will be different on different machines. The four entries which represent the **ypbind** process are:

```
100007   2   tcp   1025
100007   2   udp   1033
100007   1   tcp   1025
100007   1   udp   1033
```

If these entries are not there, **ypbind** has been unable to register its services. Reboot the machine. If they are there, and they change each time an attempt is made to restart */etc/ypbind*, reboot the system even if the **portmap** is up.

On Client: Ypwhich Inconsistent

When **ypwhich** is used several times at the same client node, the answer which is returned varies — the YP server changes. This is normal. The binding of YP client to YP server changes over time on a busy net, or when the YP servers are busy. Whenever possible, the system stabilizes at a point where all clients get acceptable response time from the YP servers. As long as the client machine receives YP service, it does not matter where the service comes from. Often a YP server machine gets its own YP services from another YP server on the net.

Debugging A Yellow Pages Server

This section describes some procedures you can follow to debug a YP server.

Different Versions Of A YP Map

Since YP works by propagating maps among servers, different versions of a map may be found at servers on the network. This version skew is normal only if transient.

Most commonly, normal update is prevented when some YP server or some gateway machine between YP servers is down during a map transfer attempt. When all the YP servers and all the gateways between them are running, **ypxfr** normally succeeds.

If a particular slave server has problems updating, run **ypxfr** interactively on that server. If **ypxfr** fails, a reason will be given. This assists in understanding and fixing the problem. If **ypxfr** fails intermittently, create a log file to enable logging of messages. The following program saves all output from **ypxfr**.

```
ypslave$ cd /etc/yp
ypslave$ touch ypxfr.log
```

The output looks much like that which is generated when **ypxfr** is run interactively, but each line in the log file is timestamped. The timestamp shows when **ypxfr** began its work. If copies of **ypxfr** ran simultaneously but their work took differing amounts of time, they may write their summary status line to the log files in an order different from that in which they were invoked. Any pattern of intermittent failure will show up in the log. When the problem is fixed, turn off logging by removing the log file. If this is not done, the log file grows until it completely fills the file system.

Also on the problem YP slave server, inspect the */usr/lib/crontab* file and the **ypxfr*** shell scripts which it invokes. Typing mistakes in these files cause propagation problems, as do failures to refer to a shell script within **crontab**, or failures to refer to a map within any shell script.

Check to ensure that the YP slave server is in the map **ypservers** within the domain. If it is not, it functions correctly as a server but is not notified by **yppush** when a new copy of a map exists.

Ypserv Crashes

When the **ypserv** process crashes almost immediately and will not stay up even with repeated activations, the method for debugging is virtually identical to that described in the section *On Client: Ypbind Crashes*. Check the **portmap** daemon:

```
ypserver$ ps -ef | grep portmap
```

If it is not found, reboot the server. If it is present, type the following command:

```
ypserver$ rpcinfo -p
```

Output should be similar to this:

```
   program vers proto   port
    100004    2   udp   1027
    100007    2   tcp   1025
    100007    2   udp   1033
    100007    1   tcp   1025
    100007    1   udp   1033
    100004    2   tcp   1024
    100004    1   udp   1027
```

```
100004    1    tcp    1024
```

The four entries representing the **ypserv** process are:

```
100004    2    udp    1027
100004    2    tcp    1024
100004    1    udp    1027
100004    1    tcp    1024
```

If they are not there, **ypserv** has been unable to register its services. Reboot the machine. If they are there and they change each time an attempt is made to restart */etc/ypserv*, reboot the machine.

Yellow Pages Policies

This section describes the policies set by the C library routines when they access the following files on a system running the YP.

/etc/passwd
: Always consulted. If there are + or – entries, the YP password map is consulted, otherwise YP is unused.

/etc/group
: Always consulted. If there are + or – entries, the YP group map is consulted, otherwise YP is unused.

/etc/hosts.equiv
: (Similarly for *.rhosts*.) Always consulted, though neither of these files is in the Yellow Pages database. If there are + or – entries whose arguments are netgroups, the YP netgroup map is consulted, otherwise YP is unused.

/etc/hosts
: Never consulted. The data in the YP database is used instead.

Security Under The Yellow Pages

This section describes ways to ensure the security of a network running YP.

Global And Local Database Files

Of the YP databases, three were formerly in */etc*: */etc/passwd*, */etc/group*, and */etc/hosts*. (Note that a site may add database files of its own.) The YP is divided into local and global file types. A local file is looked for first on the local machine, then in the

Yellow Pages. A global file is checked for only in the Yellow Pages. */etc/passwd* and */etc/group* are the local files in the YP database. The other YP files are global.

For example, a program that calls */etc/passwd* (a local file) first looks in the password file on the local machine; the YP password file is only consulted if the local machine's password file contains + (plus sign) entries. The */etc/passwd* file is local to provide local control of passwords. The only other local file is */etc/group*.

The remaining YP file (*hosts*) is a global file. The information in this file is network wide data, and is accessed only from the YP. However, when booting, each machine needs an entry in */etc/hosts* for itself. In summary, if the YP is running, global files are only checked in the Yellow Pages; a file on a local machine is not consulted.

Security Implications

An */etc/passwd* file and */etc/group* file may also have + entries. A line in an */etc/passwd* file such as

```
+nb::::Napoleon Bonaparte:/usr2/nb:/bin/sh
```

pulls in an entry for **nb** from the YP. It gets the **uid, gid** and **password** from the YP, and gets the gecos, home directory and default shell from the + entry itself. On the other hand, an */etc/passwd* entry such as

```
+nb:
```

gets all information from the YP.

Note that

```
+nb::1189:10:Napoleon Bonaparte:/usr2/nb:/bin/sh
```

is quite different from

```
nb::1189:10:Napoleon Bonaparte:/usr2/nb:/bin/sh
```

In the first of the two examples the password field is obtained from the YP; in the second, the user **nb** has no password. If there is no entry for **nb** in the YP, then the first example works as though you had provided no entry at all for **nb**.

Special YP Password Change

When a password is changed with the *passwd*(1) command, the entry given in the local */etc/passwd* file is changed. If the password is not given explicitly, but rather is pulled in from the YP with a + entry, then the *passwd* command prints the error message

```
Not in passwd file
```

To change a passwd in the YP, use the *yppasswd*(1) command. In order to enable this service, the system administrator must start up the daemon *yppasswdd*(1M) server on the machine serving as the master for the YP password file.

Manual Pages Covering Security

More details may be found on the following reference manual pages in Section 2: *yppasswd*(1), *hosts.equiv*(4), *export*(4), *passwd*(4), *group*(4), *netgroup*(4), *yppasswdd*(1M).

What If The Yellow Pages Is Not Used?

If you decide not to use the Yellow Pages, use the following procedure for bypassing the software implementation. In the system startup script, insert a # at the left of each of these lines to comment them out:

```
#if [ -f /etc/ypbind ]; then
#       /etc/ypbind; echo -n ' ypbind'        >/dev/console
#fi
```

Adding a New User to a Machine

Adding a new user to a machine involves adding an entry to the password file and creating a home directory on the new user's machine as described in the steps below.

Edit the /etc/passwd File

Typically, for a new user, a password file entry needs to be added to every machine on the local network. The superuser must do this, starting on the master YP server machine. The first step is to edit the master YP server's */etc/passwd* file.

Later, the password file entry for the user is copied to the */etc/passwd* file on the new client's partition; without an entry in it, the person administering the new client machine would not be able to login should the YP fail.

On the master YP server, add a new line to the password file. */etc/passwd* is a readable ASCII file with a one line entry for each valid user on the system. Each entry is separated into fields by colons (:); there are seven fields on each line and some fields may be left blank by placing two colons back to back. Avoid the use of certain characters in the password file: these are single and double quotes (` ´ "), backslashes (\ ), and parentheses (( )). *Passwd*(4) gives more information about the file format.

If the new user's name is Mr. Chimp and his account is to be **bonzo**, add a line to the password file similar to the one shown below:

```
bonzo::1947:10:Mr. Chimp:/usr2/bonzo:/bin/sh
```

Note that the second field is blank in the example. This field, when filled, contains an encrypted version of the user's password. When the field is blank, anyone can login simply by typing the user name — no password is required. It is not possible to create a password by making an entry in the */etc/passwd* file: the *passwd*(1) command must be used by someone logged in either as the user in question, or as superuser. Since anyone can login when a user has no password, assign a password and let the new user know what it is. Then, the user can login and change it using *passwd*, or *yppasswd*(1) to change it in the YP database.

After Mr. Chimp has a password, the entry for **bonzo** in the password file will resemble the following:

```
bonzo:3uOmRdrJ4tEVs:1947:10:Mr. Chimp:/usr2/bonzo:/bin/sh
```

Fields in the password file are separated by colons and have the following meanings:

(1) Login name — synonymous with user name.

(2) Encrypted password. Tell all users how to add or change passwords with the *passwd* command and the *yppasswd* command. The system administrator can make this field empty when a user has forgotten a password, thereby enabling login without a password until a new one is set. Note that an asterisk (*) in this field matches no password. The user can only use the *rlogin*(1) command without a password if the

machine name of the machine being logged in from is in the */etc/hosts.equiv* file on the remote machine.

(3) User ID. A number unique to this user. A system knows the user by ID number associated with login name; therefore a login name must have the same user ID number in all password files of machines which are networked in a local domain. Failure to keep IDs unique prevents users from moving files between directories on different machines. The system responds as if the directories are owned by two different users. In addition, file ownership may become confused when an NFS server exports a directory to an NFS client whose password file contains users with **uids** matching those of different users on the NFS server.

(4) Group ID. This field may be used to group together users who are working on similar projects. Normal users should not be placed in this group. Guidance on which group to put a new user in may be obtained from *group*(4) and the file */etc/group*.

(5) Information about user — usually real name, phone number, etc.

(6) The user's home directory — the directory the user logs in to.

(7) Initial shell to use on login. If this field is blank the default */bin/sh* is used.

After the password file has been updated and a password created for the new user, the YP database must be updated by running */etc/yp/make* for */etc/passwd*:

```
# cd /etc/yp
# make passwd
```

Make A Home Directory

After adding a new entry to the password file, create a home directory for the new user to login to. This is the same as the directory given in the sixth field of the password file entry. In the */usr2* directory, make a directory for the new user. Change ownership to the user's login name, and group to the user's group. For example:

```
# cd /usr2
# mkdir bonzo
# chown bonzo bonzo
# chgrp 10 bonzo
```

Note that if the YP databases for the password file have not yet been updated on the machine's YP server, the following error message appears when the *chown* is attempted:

```
unknown user id:  username
```

In that case, use the following set of commands:

```
# cd /usr2
# mkdir bonzo
# chown userid# bonzo
# chgrp 10 bonzo
```

The user's user id number (from the password file entry) is used instead of login name to change the ownership of the home directory.

The New User's Environment

The environment in which the new user is placed on login may be defined in several ways. For example, include a copy of the file *.profile* in a home directory for a user of the Bourne shell. See the *sh*(1) pages in the *Commands Reference Manual* for a discussion of this file.

If the new user is a member of any groups on the site, add the user to */etc/group* as necessary — see *group*(4) and *groups*(1). The changes must be made to the */etc/group* file on the master YP server if the Yellow Pages is used.

PERMUTED INDEX

hosts(4) host name data	base	hosts(4)
rpc(4) rpc program number data	base	rpc(4)
firstkey(3X) nextkey(3X) data	base subroutines /delete(3X)	dbminit(3X)
print values in a YP data	base ypcat(1)	ypcat(1)
yellow pages server and	binder processes /ypbind(1M)	ypserv(1M)
database ypinit(1M)	build and install yellow pages	ypinit(1M)
pages yppasswd(1)	change login password in yellow	yppasswd(1)
yppush(1M) force propagation of a	changed YP map	yppush(1M)
/yellow pages	client interface	ypcln(3N)
/set or display name of	current Yellow Pages domain	domainname(1)
nfsd(1M) NFS	daemon	nfsd(1M)
mapper portmap(1M)	DARPA port to RPC program number	portmap(1M)
hosts(4) host name	data base	hosts(4)
rpc(4) rpc program number	data base	rpc(4)
ypcat(1) print values in a YP	data base	ypcat(1)
firstkey(3X) nextkey(3X)	data base subroutines /delete(3X)	dbminit(3X)
ypmake(1M) rebuild yellow pages	database	ypmake(1M)
ypfiles(4) the yellowpages	database and directory structure	ypfiles(4)
build and install yellow pages	database ypinit(1M)	ypinit(1M)
makedbm(1M) make a yellow pages	dbm file	makedbm(1M)
delete(3X) firstkey(3X)/	dbminit(3X) fetch(3X) store(3X)	dbminit(3X)
dbminit(3X) fetch(3X) store(3X)	delete(3X) firstkey(3X)/	dbminit(3X)
the yellowpages database and	directory structure ypfiles(4)	ypfiles(4)
Pages/ domainname(1) set or	display name of current Yellow	domainname(1)
name of current Yellow Pages	domain /set or display	domainname(1)
of current Yellow Pages domain	domainname(1) set or display name	domainname(1)
/gethostbyname(3N) sethostent(3N)	endhostent(3N) get network host/	gethostent(3N)
getrpcbynumber(3N) get rpc	entry /getrpcbyname(3N)	getrpcent(3N)
endhostent(3N) get network host	entry /sethostent(3N)	gethostent(3N)
exports(4) NFS file systems being	exported	exports(4)
exported	exports(4) NFS file systems being	exports(4)
firstkey(3X)/ dbminit(3X)	fetch(3X) store(3X) delete(3X)	dbminit(3X)
make a yellow pages dbm	file makedbm(1M)	makedbm(1M)
nfsstat(1M) Network	File System statistics	nfsstat(1M)
rmtab(4) remotely mounted	file system table	rmtab(4)
exports(4) NFS	file systems being exported	exports(4)
modifying yellow pages password	file yppasswdd(1M) server for	yppasswdd(1M)
/fetch(3X) store(3X) delete(3X)	firstkey(3X) nextkey(3X) data/	dbminit(3X)
map yppush(1M)	force propagation of a changed YP	yppush(1M)
gethostbyname(3N)/ gethostent(3N)	gethostbyaddr(3N)	gethostent(3N)
gethostent(3N) gethostbyaddr(3N)	gethostbyname(3N) sethostent(3N)/	gethostent(3N)
gethostbyname(3N) sethostent(3N)/	gethostent(3N) gethostbyaddr(3N)	gethostent(3N)
getrpcbynumber(3N)/ getrpcent(3N)	getrpcbyname(3N)	getrpcent(3N)

getrpcent(3N) getrpcbyname(3N)	getrpcbynumber(3N) get rpc entry	getrpcent(3N)
getrpcbynumber(3N) get rpc entry	getrpcent(3N) getrpcbyname(3N)	getrpcent(3N)
number	getrpcport(3N) get RPC port	getrpcport(3N)
endhostent(3N) get network	host entry /sethostent(3N)	gethostent(3N)
master? ypwhich(1M) which	host is the YP server or map	ypwhich(1M)
hosts(4)	host name data base	hosts(4)
of a YP map is at a YP server	host yppoll(1M) what version	yppoll(1M)
	hosts(4) host name data base	hosts(4)
rpcinfo(1M) report RPC	information	rpcinfo(1M)
ypinit(1M) build and	install yellow pages database	ypinit(1M)
/yellow pages client	interface	ypcln(3N)
print the value of one or more	keys from a yp map ypmatch(1)	ypmatch(1)
yppasswd(1) change	login password in yellow pages	yppasswd(1)
dbm file	makedbm(1M) make a yellow pages	makedbm(1M)
ypxfr(1M) transfer a YP	map from some YP server to here	ypxfr(1M)
yppoll(1M) what version of a YP	map is at a YP server host	yppoll(1M)
which host is the YP server or	map master? ypwhich(1M)	ypwhich(1M)
of one or more keys from a yp	map ypmatch(1) print the value	ypmatch(1)
force propagation of a changed YP	map yppush(1M)	yppush(1M)
DARPA port to RPC program number	mapper portmap(1M)	portmap(1M)
host is the YP server or map	master? ypwhich(1M) which	ypwhich(1M)
file yppasswdd(1M) server for	modifying yellow pages password	yppasswdd(1M)
mountd(1M) NFS	mount request server	mountd(1M)
server	mountd(1M) NFS mount request	mountd(1M)
rmtab(4) remotely	mounted file system table	rmtab(4)
showmount(1M) show all remote	mounts	showmount(1M)
hosts(4) host	name data base	hosts(4)
domainname(1) set or display	name of current Yellow Pages/	domainname(1)
nfsstat(1M)	Network File System statistics	nfsstat(1M)
/sethostent(3N) endhostent(3N) get	network host entry	gethostent(3N)
/store(3X) delete(3X) firstkey(3X)	nextkey(3X) data base subroutines	dbminit(3X)
nfsd(1M)	NFS daemon	nfsd(1M)
exports(4)	NFS file systems being exported	exports(4)
mountd(1M)	NFS mount request server	mountd(1M)
	nfsd(1M) NFS daemon	nfsd(1M)
statistics	nfsstat(1M) Network File System	nfsstat(1M)
getrpcport(3N) get RPC port	number	getrpcport(3N)
rpc(4) rpc program	number data base	rpc(4)
DARPA port to RPC program	number mapper portmap(1M)	portmap(1M)
/yellow	pages client interface	ypcln(3N)
ypmake(1M) rebuild yellow	pages database	ypmake(1M)
build and install yellow	pages database ypinit(1M)	ypinit(1M)
makedbm(1M) make a yellow	pages dbm file	makedbm(1M)
or display name of current Yellow	Pages domain domainname(1) set	domainname(1)
/server for modifying yellow	pages password file	yppasswdd(1M)
ypserv(1M) ypbind(1M) yellow	pages server and binder processes	ypserv(1M)
change login password in yellow	pages yppasswd(1)	yppasswd(1)
update user password in yellow	pages yppasswd(3N)	yppasswd(3N)
ypset(1M) point ypbind at a	particular server	ypset(1M)
server for modifying yellow pages	password file yppasswdd(1M)	yppasswdd(1M)
yppasswd(1) change login	password in yellow pages	yppasswd(1)
yppasswd(3N) update user	password in yellow pages	yppasswd(3N)
server ypset(1M)	point ypbind at a particular	ypset(1M)
getrpcport(3N) get RPC	port number	getrpcport(3N)
portmap(1M) DARPA	port to RPC program number mapper	portmap(1M)
program number mapper	portmap(1M) DARPA port to RPC	portmap(1M)

keys from a yp map ypmatch(1)	print the value of one or more	ypmatch(1)
ypcat(1)	print values in a YP data base	ypcat(1)
yellow pages server and binder	processes ypserv(1M) ypbind(1M)	ypserv(1M)
rpc(4) rpc	program number data base	rpc(4)
portmap(1M) DARPA port to RPC	program number mapper	portmap(1M)
yppush(1M) force	propagation of a changed YP map	yppush(1M)
ypmake(1M)	rebuild yellow pages database	ypmake(1M)
showmount(1M) show all	remote mounts	showmount(1M)
table rmtab(4)	remotely mounted file system	rmtab(4)
rpcinfo(1M)	report RPC information	rpcinfo(1M)
mountd(1M) NFS mount	request server	mountd(1M)
system table	rmtab(4) remotely mounted file	rmtab(4)
getrpcbynumber(3N) get	rpc entry /getrpcbyname(3N)	getrpcent(3N)
rpcinfo(1M) report	RPC information	rpcinfo(1M)
getrpcport(3N) get	RPC port number	getrpcport(3N)
rpc(4)	rpc program number data base	rpc(4)
portmap(1M) DARPA port to	RPC program number mapper	portmap(1M)
base	rpc(4) rpc program number data	rpc(4)
information	rpcinfo(1M) report RPC	rpcinfo(1M)
mountd(1M) NFS mount request	server	mountd(1M)
/ypbind(1M) yellow pages	server and binder processes	ypserv(1M)
password file yppasswdd(1M)	server for modifying yellow pages	yppasswdd(1M)
version of a YP map is at a YP	server host yppoll(1M) what	yppoll(1M)
ypwhich(1M) which host is the YP	server or map master?	ypwhich(1M)
transfer a YP map from some YP	server to here ypxfr(1M)	ypxfr(1M)
point ypbind at a particular	server ypset(1M)	ypset(1M)
Yellow Pages/ domainname(1)	set or display name of current	domainname(1)
network host/ /gethostbyname(3N)	sethostent(3N) endhostent(3N) get	gethostent(3N)
showmount(1M)	show all remote mounts	showmount(1M)
mounts	showmount(1M) show all remote	showmount(1M)
nfsstat(1M) Network File System	statistics	nfsstat(1M)
dbminit(3X) fetch(3X)	store(3X) delete(3X) firstkey(3X)/	dbminit(3X)
database and directory	structure /the yellowpages	ypfiles(4)
nextkey(3X) data base	subroutines /firstkey(3X)	dbminit(3X)
nfsstat(1M) Network File	System statistics	nfsstat(1M)
rmtab(4) remotely mounted file	system table	rmtab(4)
exports(4) NFS file	systems being exported	exports(4)
remotely mounted file system	table rmtab(4)	rmtab(4)
server to here ypxfr(1M)	transfer a YP map from some YP	ypxfr(1M)
pages yppasswd(3N)	update user password in yellow	yppasswd(3N)
yppasswd(3N) update	user password in yellow pages	yppasswd(3N)
yp map ypmatch(1) print the	value of one or more keys from a	ypmatch(1)
ypcat(1) print	values in a YP data base	ypcat(1)
server host yppoll(1M) what	version of a YP map is at a YP	yppoll(1M)
/r(3N)yperr_string(3N)ypprot_err(3N)	yellow pages client interface	ypcln(3N)
ypinit(1M) build and install	yellow pages database	ypinit(1M)
ypmake(1M) rebuild	yellow pages database	ypmake(1M)
makedbm(1M) make a	yellow pages dbm file	makedbm(1M)
/set or display name of current	Yellow Pages domain	domainname(1)
/server for modifying	yellow pages password file	yppasswdd(1M)
processes ypserv(1M) ypbind(1M)	yellow pages server and binder	ypserv(1M)
change login password in	yellow pages yppasswd(1)	yppasswd(1)
update user password in	yellow pages yppasswd(3N)	yppasswd(3N)
directory/ ypfiles(4) the	yellowpages database and	ypfiles(4)
ypcat(1) print values in a	YP data base	ypcat(1)
here ypxfr(1M) transfer a	YP map from some YP server to	ypxfr(1M)

yppoll(1M) what version of a	YP map is at a YP server host	yppoll(1M)
value of one or more keys from a	yp map ypmatch(1) print the	ypmatch(1)
force propagation of a changed	YP map yppush(1M)	yppush(1M)
what version of a YP map is at a	YP server host yppoll(1M)	yppoll(1M)
ypwhich(1M) which host is the	YP server or map master?	ypwhich(1M)
transfer a YP map from some	YP server to here ypxfr(1M)	ypxfr(1M)
ypset(1M) point	ypbind at a particular server	ypset(1M)
and binder processes ypserv(1M)	ypbind(1M) yellow pages server	ypserv(1M)
data base	ypcat(1) print values in a YP	ypcat(1)
	ypcln(3N)/	ypcln(3N)
database and directory structure	ypfiles(4) the yellowpages	ypfiles(4)
ypcln(3N)	yp_get_default_domain(3N)yp_bind(/	ypcln(3N)
yellow pages database	ypinit(1M) build and install	ypinit(1M)
database	ypmake(1M) rebuild yellow pages	ypmake(1M)
or more keys from a yp map	ypmatch(1) print the value of one	ypmatch(1)
in yellow pages	yppasswd(1) change login password	yppasswd(1)
in yellow pages	yppasswd(3N) update user password	yppasswd(3N)
modifying yellow pages password/	yppasswdd(1M) server for	yppasswdd(1M)
map is at a YP server host	yppoll(1M) what version of a YP	yppoll(1M)
changed YP map	yppush(1M) force propagation of a	yppush(1M)
pages server and binder/	ypserv(1M) ypbind(1M) yellow	ypserv(1M)
particular server	ypset(1M) point ypbind at a	ypset(1M)
server or map master?	ypwhich(1M) which host is the YP	ypwhich(1M)
some YP server to here	ypxfr(1M) transfer a YP map from	ypxfr(1M)

NAME

domainname – set or display name of current Yellow Pages domain

SYNOPSIS

domainname [*nameofdomain*]

DESCRIPTION

Without an argument, *domainname* displays the name of the current domain. Only the super-user can set the domainname by giving an argument; this is usually done in the startup script */etc/rc*. Currently, domains are only used by the Yellow Pages, to refer collectively to a group of hosts. The YP domain is independent of either the *uucp* or Internet domain.

SEE ALSO

ypinit(1M)

NAME

makedbm – make a yellow pages dbm file

SYNOPSIS

makedbm [**–i** *yp_input_file*] [**–o** *yp_output_name*] [**–d** *yp_domain_name*] [**–m** *yp_master_name*] *infile outfile*
makedbm [**–u** *dbmfilename*]

DESCRIPTION

makedbm takes *infile* and converts it to a pair of files in *dbm*(3X) format, namely *outfile*.pag and *outfile*.dir. Each line of the input file is converted to a single *dbm* record. All characters up to the first tab or space form the key, and the rest of the line is the data. If a line ends with \, then the data for that record is continued on to the next line. It is left for the clients of the Yellow Pages to interpret #; *makedbm* does not itself treat it as a comment character. *infile* can be –, in which case standard input is read.

makedbm is meant to be used in generating *dbm* files for the Yellow Pages, and it generates a special entry with the key *yp_last_modified*, which is the date of *infile* (or the current time, if *infile* is –).

OPTIONS

–i Create a special entry with the key *yp_input_file*.

–o Create a special entry with the key *yp_output_name*.

–d Create a special entry with the key *yp_domain_name*.

–m Create a special entry with the key *yp_master_name*. If no master host name is specified, *yp_master_name* will be set to the local host name.

–u Undo a *dbm* file. That is, print out a *dbm* file one entry per line, with a single space separating keys from values.

EXAMPLE

It is easy to write shell scripts to convert standard files such as */etc/passwd* to the key value form used by *makedbm*. For example, the *awk* program

```
BEGIN { FS = ":"; OFS = "\t"; }
{ print $1, $0 }
```

takes the */etc/passwd* file and converts it to a form that can be read by *makedbm* to make the yellow pages file *passwd.byname*. That is, the key is a username, and the value is the remaining line in the */etc/passwd* file.

SEE ALSO

dbm(3X), yppasswd(1)

NAME

mountd – NFS mount request server

SYNOPSIS

mountd

DESCRIPTION

mountd is an *rpc* server that answers file system mount requests. It reads the file */etc/exports*, described in *exports*(4), to determine which file systems are available to which machines and users. It also provides information as to which clients have file systems mounted. This information can be printed using the *showmount*(1M) command.

SEE ALSO

exports(4), services(4), showmount(1M)

NAME

nfsd – NFS daemon

SYNOPSIS

nfsd [nservers]

DESCRIPTION

nfsd starts the *NFS* server daemons that handle client filesystem requests. *Nservers* is the number of file system request daemons to start. This number should be based on the load expected on this server.

SEE ALSO

mountd(1M), exports(4)

NAME

nfsstat – Network File System statistics

SYNOPSIS

nfsstat [–csnrz]

DESCRIPTION

nfsstat displays statistical information about the Network File System (NFS) and Remote Procedure Call (RPC) interfaces to the kernel. It can also be used to reinitialize this information. If no options are given the default is

nfsstat –csnr

That is, print everything and reinitialize nothing.

OPTIONS

–c Display client information. Only the client side NFS and RPC information will be printed. Can be combined with the **–n** and **–r** options to print client NFS or client RPC information only.

–s Display server information. Works like the **–c** option above.

–n Display NFS information. NFS information for both the client and server side will be printed. Can be combined with the **–c** and **–s** options to print client or server NFS information only.

–r Display RPC information. Works like the **–n** option above.

–z Zero (reinitialize) statistics. Can be combined with any of the above options to zero particular sets of statistics after printing them. The user must have write permission on */dev/kmem* for this option to work.

FILES

/unix	system namelist
/dev/kmem	kernel memory

SEE ALSO

nfs(4)

NAME

portmap – DARPA port to RPC program number mapper

SYNOPSIS

portmap

DESCRIPTION

portmap is a server that converts RPC program numbers into DARPA protocol port numbers. It must be running in order to make RPC calls.

When an RPC server is started, it will tell *portmap* what port number it is listening to, and what RPC program numbers it is prepared to serve. When a client wishes to make an RPC call to a given program number, it will first contact *portmap* on the server machine to determine the port number where RPC packets should be sent.

SEE ALSO

rpcinfo(1M)

BUGS

If *portmap* crashes, all servers must be restarted.

NAME

rpcinfo – report RPC information

SYNOPSIS

rpcinfo –p [host]
rpcinfo –u host program-number [version-number]
rpcinfo –t host program-number [version-number]

DESCRIPTION

rpcinfo makes an RPC call to an RPC server and reports what it finds.

OPTIONS

–p Probe the portmapper on *host*, and print a list of all registered RPC programs. If *host* is not specified, it defaults to the node name returned by *hostname*(1).

–u Make an RPC call to procedure 0 of *program-number* using UDP, and report whether a response was received.

–t Make an RPC call to procedure 0 of *program-number* using TCP, and report whether a response was received.

The *program-number* argument can be either a name or a number. If no version is given, it defaults to 1.

FILES

/etc/rpc names for rpc program numbers

SEE ALSO

rpc(4), portmap(1M)
RPC Programming in the *NFS Manual*.

NAME

showmount – show all remote mounts

SYNOPSIS

/etc/showmount [–a] [–d] [–e] [host]

DESCRIPTION

showmount lists all the clients that have remotely mounted a filesystem from *host*. This information is maintained by the *mountd*(1M) server on *host*, and is saved across crashes in the file */etc/rmtab*. The default value for *host* is the node name returned by *hostname*(1).

OPTIONS

–d List directories that have been remotely mounted by clients.

–a Print all remote mounts in the format

hostname:directory

where *hostname* is the name of the client, and *directory* is the root of the file system that has been mounted.

–e Print the list of exported file systems.

SEE ALSO

rmtab(4), mountd(1M), exports(4)

BUGS

If a client crashes, its entry will not be removed from the list until it reboots and executes *umount –a*.

NAME

ypcat – print values in a YP data base

SYNOPSIS

ypcat [**–k**] [**–t**] [**–d** *domainname*] *mname*
ypcat –x

DESCRIPTION

ypcat prints out values in a Yellow Pages (YP) map specified by *mname*, which may be either a *mapname* or a map *nickname*. Since *ypcat* uses the YP network services, no YP server is specified.

To look at the network-wide password database, *passwd.byname*, (with the nickname *passwd*). type in:

ypcat passwd

Refer to *ypfiles*(4) and *ypserv*(1M) for an overview of the Yellow Pages.

OPTIONS

–k Display the keys for those maps in which the values are null or the key is not part of the value. (None of the maps derived from files that have an ASCII version in */etc* fall into this class.)

–t Inhibit translation of *mname* to *mapname*. For example, *ypcat –t passwd* will fail because there is no map named *passwd*, whereas *ypcat passwd* will be translated to *ypcat passwd.byname*.

–d Specify a domain other that the default domain. The default domain is returned by *domainname*.

–x Display the map nickname table. This lists the nicknames (*mnames*) the command knows of, and indicates the *mapname* associated with each nickname.

SEE ALSO

ypfiles(4), ypserv(1M), ypmatch(1), domainname(1)

NAME

ypinit – build and install yellow pages database

SYNOPSIS

ypinit -m
ypinit -s master_name

DESCRIPTION

ypinit sets up a Yellow Pages database on a YP server. It can be used to set up a master or a slave server. You must be the superuser to run it. It asks a few, self-explanatory questions, and reports success or failure to the terminal.

It sets up a master server using the simple model in which that server is master to all maps in the data base. This is the way to bootstrap the YP system; later if you want you can change the association of maps to masters. All databases are built from scratch, either from information available to the program at runtime, or from the ASCII data base files in */etc*. Some of these files are listed below under **FILES**. Further files may be handled by the Yellow Pages, as required by the local environment. All such files should be in their "traditional" form, rather than the abbreviated form used on client machines.

A YP database on a slave server is set up by copying an existing database from a running server. The *master_name* argument should be the hostname of YP server (either the master server for all the maps, or a server on which the data base is up-to-date and stable).

Refer to *ypfiles*(4) and *ypserv*(1M) for an overview of the Yellow Pages.

OPTIONS

-m Indicates that the local host is to be the YP master.

-s Set up a slave database.

FILES

/etc/passwd
/etc/group

SEE ALSO

makedbm(1M), ypfiles(4), yppush(1M), ypxfr(1M), ypmake(1M), ypserv(1M)

NAME

ypmake – rebuild yellow pages database

SYNOPSIS

cd /etc/yp ; make [*map* **]**

DESCRIPTION

The file called *Makefile* in */etc/yp* is used by *make* to build the Yellow Pages database. With no arguments, *make* creates *dbm* databases for any YP maps that are out-of-date, and then executes *yppush* to notify slave databases that there has been a change.

If you supply a *map* on the command line, *make* will update that map only. Typing *make passwd* will create and *yppush* the password database (assuming it is out of date). Likewise, *make hosts* and *make networks* will create and *yppush* the host and network files, */etc/hosts* and */etc/networks*.

There are three special variables used by *make*: DIR, which gives the directory of the source files; NOPUSH, which when non-null inhibits doing a *yppush* of the new database files; and DOM, used to construct a domain other than the master's default domain. The default for DIR is */etc*, and the default for NOPUSH is the null string.

Refer to *ypfiles*(4) and *ypserv*(1M) for an overview of the Yellow Pages.

SEE ALSO

make(1), makedbm(1M), ypserv(1M)

NAME

ypmatch – print the value of one or more keys from a yp map

SYNOPSIS

ypmatch [**-d** *domain*] [**-k**] [**-t**] *key* ... *mname*
ypmatch **–x**

DESCRIPTION

ypmatch prints the values associated with one or more keys from the Yellow Pages (YP) map (database) specified by a *mname*, which may be either a *mapname* or a map *nickname*.

Multiple keys can be specified; the same map will be searched for all. The keys must be exact values insofar as capitalization and length are concerned. No pattern matching is available. If a key is not matched, a diagnostic message is produced.

OPTIONS

–d Specify a domain other that the default domain.

–k Before printing the value of a key, print the key itself, followed by a colon (':'). This is useful only if the keys are not duplicated in the values, or you've specified so many keys that the output could be confusing.

–t Inhibit translation of nickname to mapname. For example, *ypmatch –t zippy passwd* will fail because there is no map named *passwd*, while *ypmatch zippy passwd* will be translated to *ypmatch zippy passwd.byname* .

–x Display the map nickname table. This lists the nicknames (*mnames*) the command knows of, and indicates the *mapname* associated with each nickname.

SEE ALSO

ypfiles(4), ypcat(1)

NAME

yppasswd – change login password in yellow pages

SYNOPSIS

yppasswd [name]

DESCRIPTION

yppasswd changes (or installs) a password associated with the user *name* (your own name by default) in the Yellow Pages. The Yellow Pages password may be different from the one on your own machine.

yppasswd prompts for the old Yellow Pages password and then for the new one. The caller must supply both. The new password must be typed twice, to forestall mistakes.

New passwords must be at least four characters long if they use a sufficiently rich alphabet and at least six characters long if monocase. These rules are relaxed if you are insistent enough.

Only the owner of the name or the super-user may change a password; in either case you must prove you know the old password.

SEE ALSO

passwd(1), ypfiles(4), yppasswdd(1M)

BUGS

The update protocol passes all the information to the server in one RPC call, without ever looking at it. Thus if you type in your old password incorrectly, you will not be notified until after you have entered your new password.

NAME

yppasswdd – server for modifying yellow pages password file

SYNOPSIS

yppasswdd file [–m arg1 arg2 ...]

DESCRIPTION

yppasswdd is a server that handles password change requests from *yppasswd*(1). It changes a password entry in *file*, which is assumed to be in the format of *passwd*(4). An entry in *file* will only be changed if the password presented by *yppasswd*(1) matches the encrypted password of that entry.

If the –m option is given, then after *file* is modified, a *make*(1) will be performed in */etc/yp*. Any arguments following the flag will be passed to *make*.

This server is not run by default. If it is desired to enable remote password updating for the Yellow Pages, then an entry for *yppasswdd* should be put in the */etc/rc* file of the host serving as the master for the Yellow Pages *passwd* file.

EXAMPLE

If the Yellow Pages password file is stored as */etc/yp/src/passwd*, then to have password changes propagated immediately, the server should be invoked as

/usr/etc/rpc.yppasswdd /etc/yp/src/passwd –m passwd DIR=/etc/yp/src

FILES

/etc/yp/Makefile

SEE ALSO

yppasswd(1), passwd(4), ypfiles(4), ypmake(1M)

CAVEAT

This server will eventually be replaced with a more general service for modifying any map in the Yellow Pages.

NAME

yppoll – what version of a YP map is at a YP server host

SYNOPSIS

yppoll [**-h** *host*] [**-d** *domain*] *mapname*

DESCRIPTION

yppoll asks a *ypserv* process what the order number is, and which host is the master YP server for the named map. If the server is a v.1 YP protocol server, *yppoll* uses the older protocol to communciate with it. In this case, it also uses the older diagnostic messages in case of failure.

OPTIONS

–h *host*
: Ask the *ypserv* process at *host* about the map parameters. If *host* isn't specified, the YP server for the local host is used. That is, the default host is the one returned by *ypwhich*(1M).

–d *domain*
: Use *domain* instead of the default domain.

SEE ALSO

ypserv(1M), ypfiles(4)

NAME

yppush – force propagation of a changed YP map

SYNOPSIS

yppush [-d *domain*] [-v] *mapname*

DESCRIPTION

yppush copies a new version of a Yellow Pages (YP) map from the master YP server to the slave YP servers. It is normally run only on the master YP server by the *Makefile* in */usr/etc/yp/* after the master databases are changed. It first constructs a list of YP server hosts by reading the YP map *ypservers* within the *domain*. Keys within the map *ypservers* are the ASCII names of the machines on which the YP servers run.

A "transfer map" request is sent to the YP server at each host, along with the information needed by the transfer agent (the program which actually moves the map) to call back the *yppush* . When the attempt has completed (successfully or not), and the transfer agent has sent *yppush* a status message, the results may be printed to stdout. Messages are also printed when a transfer is not possible; for instance when the request message is undeliverable, or when the timeout period on responses has expired.

Refer to *ypfiles*(4) and *ypserv*(1M) for an overview of the Yellow Pages.

OPTIONS

-d Specify a *domain*.

-v Verbose. This causes messages to be printed when each server is called, and for each response. If this flag is omitted, only error messages are printed.

FILES

/etc/yp/*domainname*/ypservers.{dir, pag}

SEE ALSO

ypserv(1M), ypxfr(1M), ypfiles(4), YP protocol specification

BUGS

In the current implementation (version 2 YP protocol), the transfer agent is *ypxfr*, which is started by the *ypserv* program. If *yppush* detects that it is speaking to a version 1 YP protocol server, it uses the older protocol, sending a version 1 YPPROC_GET request and issues a message to that effect. Unfortunately, there is no way of knowing if or when the map transfer is performed for version 1 servers. *yppush* prints a message saying that an "old-style" message has been sent. The system administrator should later check to see that the transfer has actually taken place.

NAME

ypserv, ypbind – yellow pages server and binder processes

SYNOPSIS

/etc/ypserv
/etc/ypbind

DESCRIPTION

The Yellow Pages (YP) provides a simple network lookup service consisting of databases and processes. The databases are *dbm*(3) files in a directory tree rooted at */etc/yp*. These files are described in ypfiles(4). The processes are */etc/ypserv*, the YP database lookup server, and */etc/ypbind*, the YP binder. The programmatic interface to YP is described in ypclnt(3N). Administrative tools are described in *yppush*(1M), *ypxfr*(1M), *yppoll*(1), *ypwhich*(1M), and *ypset*(1M). Tools to see the contents of YP maps are described in *ypcat*(1M), and *ypmatch*(1). Database generation and maintenance tools are described in *ypinit*(1M), *ypmake*(1M), and *makedbm*(1M).

Both *ypserv* and *ypbind* are daemon processes typically activated at system startup time from */etc/rc*. *ypserv* runs only on YP server machines with a complete YP database. *ypbind* runs on all machines using YP services, both YP servers and clients.

The *ypserv* daemon's primary function is to look up information in its local database of YP maps. The operations performed by *ypserv* are defined for the implementor by the *YP Protocol Specification*, and for the programmer by the header file *<rpcsvc/yp_prot.h>*. Communication to and from *ypserv* is by means of RPC calls. Lookup functions are described in *ypclnt*(3N), and are supplied as C-callable funtions in */lib/librpc*. There are four lookup functions, all of which are performed on a specified map within some YP domain: *Match*, *Get_first*, *Get_next*, and *Get_all*. The *Match* operation takes a key, and returns the associated value. The *Get_first* operation returns the first key-value pair from the map, and *Get_next* can be used to enumerate the remainder. *Get_all* ships the entire map to the requester as the response to a single RPC request.

Two other functions supply information about the map, rather than map entries: *Get_order_number*, and *Get_master_name*. In fact, both order number and master name exist in the map as key-value pairs, but the server will not return either through the normal lookup functions. (If you examine the map with *makedbm*(1M), however, they will be visible.) Other functions are used within the YP subsystem itself, and are not of general interest to YP clients. They include *Do_you_serve_this_domain?*, *Transfer_map*, and *Reinitialize_internal_state*.

The function of *ypbind* is to remember information that lets client processes on a single node communicate with some *ypserv* process. *ypbind* must run on every machine which has YP client processes; *ypserv* may or may not be running on the same node, but must be running somewhere on the network.

The information *ypbind* remembers is called a *binding* — the association of a domain name with the internet address of the YP server, and the port on that host at which the *ypserv* process is listening for service requests. The process of binding is driven by client requests. As a request for an unbound domain comes in, the *ypbind* process broadcasts on the net trying to find a *ypserv* process that serves maps within that domain. Since the binding is established by broadcasting, there must be at least one *ypserv* process on every net. Once a domain is bound by a particular *ypbind*, that same binding is given to every client process on the node. The *ypbind* process on the local node or a remote node may be queried for the binding of a particular domain by using the *ypwhich*(1) command.

Bindings are verified before they are given out to a client process. If *ypbind* is unable to speak to the *ypserv* process it's bound to, it marks the domain as unbound, tells the client process that the domain is unbound, and tries to bind the domain once again. Requests received for an unbound domain will fail immediately. In general, a bound domain is marked as unbound when the node running *ypserv* crashes or gets overloaded. In such a case, *ypbind* will try to bind any YP server (typically one that is less-heavily loaded) available on the net.

ypbind also accepts requests to set its binding for a particular domain. The request is usually generated by the YP subsystem itself. *ypset*(1M) is a command to access the *Set_domain* facility. It is for unsnarling messes, not for casual use.

FILES

If the file */etc/yp/ypserv.log* exists when *ypserv* starts up, log information will be written to this file when error conditions arise.

SEE ALSO

ypclnt(3N), ypfiles(4), ypcat(1), ypmatch(1), yppush(1M), ypwhich(1M), ypxfr(1M), ypset(1M)
YP Protocol Specification in the *NFS Manual*.

NAME

ypset – point ypbind at a particular server

SYNOPSIS

ypset [**–V1** | **–V2**] [**-h** *host*] [**-d** *domain*] *server*

DESCRIPTION

ypset tells *ypbind* to get YP services for the specified *domain* from the *ypserv* process running on *server*. If *server* is down, or isn't running *ypserv*, this is not discovered until a YP client process tries to get a binding for the domain. At this point, the binding set by *ypset* will be tested by *ypbind*. If the binding is invalid, *ypbind* will attempt to rebind for the same domain.

ypset is useful for binding a client node which is not on a broadcast net, or is on a broadcast net which isn't running a YP server host. It also is useful for debugging YP client applications, for instance where a YP map only exists at a single YP server host.

In cases where several hosts on the local net are supplying YP services, it is possible for *ypbind* to rebind to another host even while you attempt to find out if the *ypset* operation succeeded. That is, you can type "ypset host1", and then "ypwhich", which replies: "host2", which can be confusing. This is a function of the YP subsystem's attempt to load-balance among the available YP servers, and occurs when *host1* does not respond to *ypbind* because it is not running *ypserv* (or is overloaded), and *host2*, running *ypserv*, gets the binding.

server indicates the YP server to bind to, and can be specified as a name or an IP address. If specified as a name, *ypset* attempts to use YP services to resolve the name to an IP address. This works only if the node has a current valid binding for the domain in question. In most cases, *server* should be specified as an IP address.

Refer to *ypfiles*(4) and *ypserv*(1M) for an overview of the Yellow Pages.

OPTIONS

–V1 Bind *server* for the (old) v.1 YP protocol.

–V2 Bind *server* for the (current) v.2 YP protocol.

If no version is supplied, *ypset*, first attempts to set the domain for the (current) v.2 protocol. If this attempt fails, *ypset*, then attempts to set the domain for the (old) v.1 protocol.

–h *host* Set ypbind's binding on *host*, instead of locally. *host* can be specified as a name or as an IP address.

–d *domain* Use *domain* , instead of the default domain.

SEE ALSO

ypwhich(1M), ypserv(1M), ypfiles(4)

NAME

ypwhich – which host is the YP server or map master?

SYNOPSIS

ypwhich [**–d** [*domain*] [**–V1** | **–V2**] [*hostname*]
ypwhich [**–t** *mapname*] [**–d** *domain*] **–m** *mname*
ypwhich –x

DESCRIPTION

ypwhich tells which YP server supplies yellow pages services to a YP client, or which is the master for a map. If invoked without arguments, it gives the YP server for the local machine. If *hostname* is specified, that machine is queried to find out which YP master it is using.

Refer to *ypfiles*(4) and *ypserv*(1M) for an overview of the Yellow Pages.

OPTIONS

–d Use *domain* instead of the default domain.

–V1 Which server is serving v.1 YP protocol-speaking client processes?

–V2 Which server is serving v.2 YP protocol client processes?

If neither version is specified, *ypwhich* attempts attempts to locate the server that supplies the (current) v.2 services. If there is no v.2 server currently bound, *ypwhich* then attempts to locate the server supplying the v.1 services. Since YP servers and YP clients are both backward compatible, the user need seldom be concerned about which version is currently in use.

–t *mapname* Inhibit nickname translation; useful if there is a mapname identical to a nickname.

–m Find the master YP server for a map. No *hostname* can be specified with **–m**. *mname* can be a mapname, or a nickname for a map.

–x Display the map nickname table. This lists the nicknames (*mnames*) the command knows of, and indicates the *mapname* associated with each nickname.

SEE ALSO

ypfiles(4), rpcinfo(1M), ypset(1M), ypserv(1M)

NAME

ypxfr – transfer a YP map from some YP server to here

SYNOPSIS

ypxfr [**–f**] [**-h** *host*] [**-d** *domain*] [**–c**] [**-C** *tid prog ipadd port*] *mapname*

DESCRIPTION

ypxfr moves a YP map to the local host by making use of normal YP services. It creates a temporary map in the directory */etc/yp/domain* (which must already exist), fills it by enumerating the map's entries, fetches the map parameters (master and order number) and loads them. It then deletes any old versions of the map and moves the temporary map to the real mapname.

If *ypxfr* is run interactively, it writes its output to the terminal. However, if it's invoked without a controlling terminal, and if the log file */etc/yp/ypxfr.log* exists, it will append all its output to that file. Since *ypxfr* is most often run from */usr/lib/crontab* , or by *ypserv* , you can use the log file to retain a record of what was attempted, and what the results were.

For consistency between servers, *ypxfr* should be run periodically for every map in the YP data base. Different maps change at different rates: the *services.byname* map may not change for months at a time, for instance, and may therefore be checked only once a day in the wee hours. You may know that *mail.aliases* or *hosts.byname* changes several times per day. In such a case, you may want to check hourly for updates. A *crontab*(4) entry can be used to perform periodic updates automatically. Rather than having a separate *crontab* entry for each map, you can group comands to update several maps in a shell script. Examples (mnemonically named) are in */etc/yp*: *ypxfr_1perday.sh*, *ypxfr_2perday.sh*, and *ypxfr_1perhour.sh*. They can serve as reasonable first cuts.

Refer to *ypfiles*(4) and *ypserv*(1M) for an overview of the Yellow Pages.

OPTIONS

–f Force the transfer to occur even if the version at the master is not more recent than the local version.

–c Don't send a "Clear current map" request to the local *ypserv* process. Use this flag if *ypserv* is not running locally at the time you are running *ypxfr*. Otherwise, *ypxfr* will complain that it can't talk to the local *ypserv*, and the transfer will fail.

–h *host* Get the map from *host* , regardless of what the map says the master is. If *host* is not specified, *ypxfr* will ask the YP service for the name of the master, and try to get the map from there. *host* may be a name or an internet address in the form *a.b.c.d* .

–d *domain* Specify a domain other than the default domain.

–C *tid prog ipadd port*
This option is only for use by *ypserv*. When *ypserv* invokes *ypxfr*, it specifies that *ypxfr* should call back a *yppush* process at the host with IP address *ipaddr* , registered as program number *prog* , listening on port *port* , and waiting for a response to transaction *tid* .

FILES

/etc/yp/ypxfr.log, /etc/yp/ypxfr_1perday.sh, /etc/yp/ypxfr_2perday.sh, /etc/yp/ypxfr_1perhour.sh, /usr/lib/crontab

SEE ALSO

ypfiles(4), ypserv(1M), yppush(1M)
YP Protocol Specification in the *NFS Manual*.

NAME

dbminit, fetch, store, delete, firstkey, nextkey – data base subroutines

SYNOPSIS

```
typedef struct {
        char *dptr;
        int dsize;
} datum;

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;

dbmclose()
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. The functions are obtained with the loader option **-ldbm**.

Keys and *contents* are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has '.dir' as its suffix. The second file contains all data and has '.pag' as its suffix.

Before a database can be accessed, it must be opened by *dbminit*. At the time of this call, the files *file*.dir and *file*.pag must exist. (An empty database is created by creating zero-length '.dir' and '.pag' files.)

Once open, the data stored under a key is accessed by *fetch* and data is placed under a key by *store*. A key (and its associated contents) is deleted by *delete*. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of *firstkey* and *nextkey*. *Firstkey* will return the first key in the database. With any key *nextkey* will return the next key in the database. This code will traverse the data base:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key))
```

A database may be closed by calling *dbmclose*. You must close a database before opening a new one.

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates ok. Routines that return a *datum* indicate errors with a null (0) *dptr*.

BUGS

The '.pag' file will contain holes so that its apparent size is about four times its actual content. Older UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (cp, cat, tp, tar, ar) without filling in the holes.

Dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block. *Store* will return an error in the event that a disk block fills with inseparable data.

Delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *firstkey* and *nextkey* depends on a hashing function, not on anything interesting.

There are no interlocks and no reliable cache flushing; thus concurrent updating and reading is risky.

NAME

getdomainname, setdomainname – get/set name of current domain

SYNOPSIS

```
getdomainname(name, namelen)
char *name;
int namelen;

setdomainname(name, namelen)
char *name;
int namelen;
```

DESCRIPTION

getdomainname returns the name of the domain for the current processor, as previously set by *setdomainname*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

setdomainname sets the domain of the host machine to be *name*, which has length *namelen*. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

The purpose of domains is to enable two distinct networks that may have host names in common to merge. Each network would be distinguished by having a different domain name. At the current time, only the yellow pages service makes use of domains.

RETURN VALUE

If the call succeeds a value of 0 is returned. If the call fails, then a value of –1 is returned and an error code is placed in the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EFAULT] The *name* parameter gave an invalid address.

[EPERM] The caller was not the super-user. This error only applies to **setdomainname.**

BUGS

Domain names are limited to 64 characters.

NAME

gethostent, gethostbyaddr, gethostbyname, sethostent, endhostent – get network host entry

SYNOPSIS

#include <netdb.h>

struct hostent *gethostent()

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr; int len, type;

sethostent(stayopen)
int stayopen

endhostent()

DESCRIPTION

gethostent, *gethostbyname*, and *gethostbyaddr* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network host data base, */etc/hosts* or the Yellow Pages "host" map.

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* address type */
        int     h_length;       /* length of address */
        char    *h_addr;        /* address */
};
```

The members of this structure are:

h_name Official name of the host.

h_aliases A zero terminated array of alternate names for the host.

h_addrtype The type of address being returned; currently always AF_INET.

h_length The length, in bytes, of the address.

h_addr A pointer to the network address for the host. Host addresses are returned in network byte order.

gethostent reads the next line of the file, opening the file if necessary.

sethostent opens and rewinds the file. If the *stayopen* flag is non-zero, the host data base will not be closed after each call to *gethostent* (either directly, or indirectly through one of the other "gethost" calls).

endhostent closes the file.

gethostbyname and *gethostbyaddr* sequentially search from the beginning of the file until a matching host name or host address is found, or until EOF is encountered. Host addresses are supplied in network order.

FILES

/etc/hosts
/etc/yp/*domainname*/hosts.byname
/etc/yp/*domainname*/hosts.byaddr

SEE ALSO

hosts(4), ypserv(1M)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

Use of this routine depends on the local network transport mechanism

NAME

getrpcent, getrpcbyname, getrpcbynumber – get rpc entry

SYNOPSIS

```
#include <netdb.h>

struct rpcent *getrpcent()

struct rpcent *getrpcbyname(name)
char *name;

struct rpcent *getrpcbynumber(number)
int number;

setrpcent(stayopen)
int stayopen

endrpcent( )
```

DESCRIPTION

getrpcent, *getrpcbyname*, and *getrpcbynumber* each return a pointer to an object with the following structure containing the broken-out fields of a line in the rpc program number data base, */etc/rpc* or the Yellow Pages "rpc.bynumber" map.

```
struct  rpcent {
        char    *r_name;        /* name of server for this rpc program */
        char    **r_aliases;    /* alias list */
        long    r_number;       /* rpc program number */
};
```

The members of this structure are:

r_name The name of the server for this rpc program.

r_aliases A zero terminated list of alternate names for the rpc program.

r_number The rpc program number for this service.

getrpcent reads the next line of the file, opening the file if necessary.

setrpcent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getrpcent* (either directly, or indirectly through one of the other "getrpc" calls).

endrpcent closes the file.

getrpcbyname and *getrpcbynumber* sequentially search from the beginning of the file until a matching rpc program name or program number is found, or until EOF is encountered.

FILES

/etc/rpc
/etc/yp/*domainname*/rpc.bynumber

SEE ALSO

rpc(4), rpcinfo(1M)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getrpcport – get RPC port number

SYNOPSIS

int getrpcport(host, prognum, versnum, proto)
char *host;
int prognum, versnum, proto;

DESCRIPTION

getrpcport returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*. It returns 0 if it cannot contact the portmapper, or if *prognum* is not registered. If *prognum* is registered but not with version *versnum*, it will return that port number.

NAME

ypclnt yp_get_default_domain yp_bind yp_unbind yp_match yp_first yp_next yp_all yp_order yp_master yperr_string ypprot_err – yellow pages client interface

SYNOPSIS

```
#include <rpcsvc/ypclnt.h>

yp_bind(indomain)
char *indomain;

void yp_unbind(indomain)
char *indomain;

yp_get_default_domain(outdomain)
char **outdomain;

yp_match(indomain, inmap, inkey, inkeylen, outval, outvallen)
char *indomain;
char *inmap;
char *inkey;
int inkeylen;
char **outval;
int *outvallen;

yp_first(indomain, inmap, outkey, outkeylen, outval, outvallen)
char *indomain;
char *inmap;
char **outkey;
int *outkeylen;
char **outval;
int *outvallen;

yp_next(indomain, inmap, inkey, inkeylen, outkey, outkeylen, outval, outvallen)
char *indomain;
char *inmap;
char *inkey;
int inkeylen;
char **outkey;
int *outkeylen;
char **outval;
int *outvallen;

yp_all(indomain, inmap, incallback)
char *indomain;
char *inmap;
struct ypall_callback incallback;

yp_order(indomain, inmap, outorder)
char *indomain;
char *inmap;
unsigned long *outorder;

yp_master(indomain, inmap, outname)
char *indomain;
char *inmap;
char **outname;

char *yperr_string(incode)
int incode;
```

ypprot_err(incode)
unsigned int incode;

DESCRIPTION

This package of functions provides an interface to the yellow pages (YP) network lookup service. The package can be loaded from the standard library, */usr/librpc.a*. Refer to *ypfiles*(4) and *ypserv*(1M) for an overview of the Yellow Pages, including the definitions of *map* and *domain* , and a description of the various servers, databases, and commands that comprise the YP.

All input parameters names begin with **in**. Output parameters begin with **out**. Output parameters of type *char ** should be addresses of uninitialized character pointers. Memory is allocated by the YP client package using *malloc*(3), and may be freed if the user code has no continuing need for it. For each *outkey* and *outval*, two extra bytes of memory are allocated at the end that contain NEWLINE and NULL, respectively, but these two bytes are not reflected in *outkeylen* or *outvallen* . *indomain* and *inmap* strings must be non-null and null-terminated. String parameters which are accompanied by a count parameter may not be null, but may point to null strings, with the count parameter indicating this. Counted strings need not be null-terminated.

All functions in this package of type int return 0 if they succeed, and a failure code (YPERR_*xxxx*) otherwise. Failure codes are described under **DIAGNOSTICS** below.

The YP lookup calls require a map name and a domain name, at minimum. It is assumed that the client process knows the name of the map of interest. Client processes should fetch the node's default domain by calling *yp_get_default_domain()* , and use the returned *outdomain* as the *indomain* parameter to successive YP calls.

To use the YP services, the client process must be "bound" to a YP server that serves the appropriate domain using *yp_bind*. Binding need not be done explicitly by user code; this is done automatically whenever a YP lookup function is called. *yp_bind* can be called directly for processes that make use of a backup strategy (e.g., a local file) in cases when YP services are not available.

Each binding allocates (uses up) one client process socket descriptor; each bound domain costs one socket descriptor. However, multiple requests to the same domain use that same descriptor. *yp_unbind()* is available at the client interface for processes that explicitly manage their socket descriptors while accessing multiple domains. The call to *yp_unbind()* make the domain *unbound*, and free all per-process and per-node resources used to bind it.

If an RPC failure results upon use of a binding, that domain will be unbound automatically. At that point, the ypclnt layer will retry forever or until the operation succeeds, provided that *ypbind* is running, and either

a) the client process can't bind a server for the proper domain, or

b) RPC requests to the server fail.

If an error is not RPC-related, or if *ypbind* is not running, or if a bound *ypserv* process returns any answer (success or failure), the ypclnt layer will return control to the user code, either with an error code, or a success code and any results.

yp_match returns the value associated with a passed key. This key must be exact; no pattern matching is available.

yp_first returns the first key-value pair from the named map in the named domain.

yp_next() returns the next key-value pair in a named map. The *inkey* parameter should be the *outkey* returned from an initial call to *yp_first()* (to get the second key-value pair) or the one returned from the nth call to *yp_next()* (to get the nth + second

key-value pair).

The concept of first (and, for that matter, of next) is particular to the structure of the YP map being processing; there is no relation in retrieval order to either the lexical order within any original (non-YP) data base, or to any obvious numerical sorting order on the keys, values, or key-value pairs. The only ordering guarantee made is that if the *yp_first()* function is called on a particular map, and then the *yp_next()* function is repeatedly called on the same map at the same server until the call fails with a reason of YPERR_NOMORE, every entry in the data base will be seen exactly once. Further, if the same sequence of operations is performed on the same map at the same server, the entries will be seen in the same order.

Under conditions of heavy server load or server failure, it is possible for the domain to become unbound, then bound once again (perhaps to a different server) while a client is running. This can cause a break in one of the enumeration rules; specific entries may be seen twice by the client, or not at all. This approach protects the client from error messages that would otherwise be returned in the midst of the enumeration. The next paragraph describes a better solution to enumerating all entries in a map.

yp_all provides a way to transfer an entire map from server to client in a single request using TCP (rather than UDP as with other functions in this package). The entire transaction take place as a single RPC request and response. You can use *yp_all* just like any other YP procedure, identify the map in the normal manner, and supply the name of a function which will be called to process each key-value pair within the map. You return from the call to *yp_all* only when the transaction is completed (successfully or unsuccessfully), or your *"foreach"* function decides that it doesn't want to see any more key-value pairs.

The third parameter to *yp_all* is

```
struct ypall_callback *incallback {
        int (*foreach)();
        char *data;
};
```

The function *foreach* is called

```
foreach(instatus, inkey, inkeylen, inval, invallen, indata)
        int instatus;
        char *inkey;
        int inkeylen;
        char *inval;
        int invalllen;
        char *indata;
```

The *instatus* parameter will hold one of the return status values defined in <rpcsvc/yp_prot.h> — either YP_TRUE or an error code. (See *ypprot_err* , below, for a function which converts a YP protocol error code to a ypclnt layer error code.)

The key and value parameters are somewhat different than defined in the synopsis section above. First, the memory pointed to by the *inkey* and *inval* parameters is private to the *yp_all* function, and is overwritten with the arrival of each new key-value pair. It is the responsibility of the *foreach* function to do something useful with the contents of that memory, but it does not own the memory itself. Key and value objects presented to the *foreach* function look exactly as they do in the server's map — if they were not newline-terminated or null-terminated in the map, they won't be here either.

The *indata* parameter is the contents of the *incallback->data* element passed to *yp_all*. The *data* element of the callback structure may be used to share state information between the *foreach* function and the mainline code. Its use is optional, and no part of the YP client package inspects its contents — cast it to something useful, or ignore it as you see fit.

The *foreach* function is a Boolean. It should return zero to indicate that it wants to be called again for further received key-value pairs, or non-zero to stop the flow of key-value pairs. If *foreach* returns a non-zero value, it is not called again; the functional value of *yp_all* is then 0.

yp_order returns the order number for a map.

yp_master returns the machine name of the master YP server for a map.

yperr_string returns a pointer to an error message string that is null-terminated but contains no period or newline.

ypprot_err takes a YP protocol error code as input, and returns a ypclnt layer error code, which may be used in turn as an input to *yperr_string* .

FILES

/usr/include/rpcsvc/ypclnt.h
/usr/include/rpcsvc/yp_prot.h

SEE ALSO

ypfiles(4), ypserv(1M),

DIAGNOSTICS

All integer functions return 0 if the requested operation is successful, or one of the following errors if the operation fails.

```
#define YPERR_BADARGS   1  /* args to function are bad */
#define YPERR_RPC       2  /* RPC failure - domain has been unbound */
#define YPERR_DOMAIN    3  /* can't bind to server on this domain */
#define YPERR_MAP       4  /* no such map in server's domain */
#define YPERR_KEY       5  /* no such key in map */
#define YPERR_YPERR     6  /* internal yp server or client error */
#define YPERR_RESRC     7  /* resource allocation failure */
#define YPERR_NOMORE    8  /* no more records in map database */
#define YPERR_PMAP      9  /* can't communicate with portmapper */
#define YPERR_YPBIND    10 /* can't communicate with ypbind */
#define YPERR_YPSERV    11 /* can't communicate with ypserv */
#define YPERR_NODOM     12 /* local domain name not set */
#define YPERR_BADDB     13 /* YP database is bad */
#define YPERR_VERS      14 /* YP version mismatch */
```

NAME

yppasswd – update user password in yellow pages

SYNPOSIS

```
#include <rpcsvc/yppasswd.h>

yppasswd(oldpass, newpw)
char *oldpass;
struct passwd *newpw;
```

DESCRIPTION

If *oldpass* is indeed the old user password, this routine replaces the password entry with *newpw*. It returns 0 if successful.

RPC INFO

```
program number:
        YPPASSWDPROG

xdr routines:
        xdr_ppasswd(xdrs, yp)
                XDR *xdrs;
                struct yppasswd *yp;
        xdr_yppasswd(xdrs, pw)
                XDR *xdrs;
                struct passwd *pw;
procs:
        YPPASSWDPROC_UPDATE
                Takes struct yppasswd as argument, returns integer.
                Same behavior as yppasswd() wrapper.
                Uses UNIX authentication.
versions:
        YPPASSWDVERS_ORIG

structures:
        struct yppasswd {
                char *oldpass;  /* old (unencrypted) password */
                struct passwd newpw;  /* new pw structure */
        };
```

SEE ALSO

yppasswd(1), yppasswdd(1M)

NAME

exports – NFS file systems being exported

SYNOPSIS

/etc/exports

DESCRIPTION

The file */etc/exports* describes the file systems which are being exported to NFS clients. It is created by the system administrator using a text editor and processed by the *mount* request daemon *mountd*(1M) each time a mount request is received.

The file consists of a list of file systems and the *netgroups*(4) or machine names allowed to remote mount each file system. The file system names are left justified and followed by a list of names separated by white space. The names will be looked up in */etc/netgroups* and then in */etc/hosts*. A file system name with no name list following means export to everyone. A "#" anywhere in the file indicates a comment extending to the end of the line it appears on. Lines beginning with white space are continuation lines.

EXAMPLE

```
/usr       clients               # export to my clients
/usr/local                       # export to the world
/usr2      phoenix sun sundae    # export to only these machines
```

FILES

/etc/exports

BUGS

The identification of the remote system is dependent on the local network transport mechanism employed.

SEE ALSO

mountd(1M)

NAME

hosts – host name data base

SYNOPSIS

/etc/hosts

DESCRIPTION

The *hosts* file contains information regarding the known hosts on the DARPA Internet. For each host a single line should be present with the following information:

Internet address
official host name
aliases

Items are separated by any number of blanks and/or tab characters. A '#' indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official host data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown hosts.

Network addresses are specified in the conventional '.' notation using the *inet_addr*() routine from the Internet address manipulation library, *inet*(3N). Host names may contain any printable character other than a field delimiter, newline, or comment character.

EXAMPLE

Here is a typical line from the */etc/hosts* file:

```
192.9.1.20      gaia            # Alison Shanks
```

FILES

/etc/hosts

BUGS

This file is dependent on the network transport mechanism used.

SEE ALSO

gethostent(3N)

NAME

rmtab – remotely mounted file system table

DESCRIPTION

rmtab resides in directory */etc* and contains a record of all clients that have done remote mounts of file systems from this machine. Whenever a remote *mount* is done, an entry is made in the *rmtab* file of the machine serving up that file system. *umount* removes entries, if of a remotely mounted file system. *umount* –a broadcasts to all servers, and informs them that they should remove all entries from *rmtab* created by the sender of the broadcast message. By placing a *umount* –a command in */etc/rc.boot*, *rmtab* tables can be purged of entries made by a crashed host, which upon rebooting did not remount the same file systems it had before. The table is a series of lines of the form

    hostname:directory

This table is used only to preserve information between crashes, and is read only by *mountd*(1M) when it starts up. *mountd* keeps an in-core table, which it uses to handle requests from programs like *showmount*(1) and *shutdown*(1M).

FILES

/etc/rmtab

SEE ALSO

showmount(1), mountd(1M), mount(1M), umount(1M), shutdown(1M)

BUGS

Although the *rmtab* table is close to the truth, it is not always 100% accurate.

NAME

rpc – rpc program number data base

SYNOPSIS

/etc/rpc

DESCRIPTION

The *rpc* file contains user readable names that can be used in place of rpc program numbers. Each line has the following information:

name of server for the rpc program
rpc program number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

FILES

/etc/rpc

SEE ALSO

getrpcent(3N)

NAME

ypfiles – the yellowpages database and directory structure

DESCRIPTION

The Yellow Pages (YP) network lookup service uses a database of *dbm* files in the directory hierarchy at */etc/yp* . A *dbm* database consists of two files, created by calls to the *dbm*(3X) library package. One has the filename extension *.pag* and the other has the filename extension *.dir*. For instance, the database named *hosts.byname*, is implemented by the pair of files *hosts.byname.pag* and *hosts.byname.dir*. A *dbm* database served by the YP is called a YP *map*. A YP *domain* is a named set of YP maps. Each YP domain is implemented as a subdirectory of */etc/yp* containing the map. Any number of YP domains can exist. Each may contain any number of maps.

No maps are required by the YP lookup service itself, although they may be required for the normal operation of other parts of the system. There is no list of maps which YP serves - if the map exists in a given domain, and a client asks about it, the YP will serve it. For a map to be accessible consistently, it must exist on all YP servers that serve the domain. To provide data consistency between the replicated maps, an entry to run *ypxfr* periodically should be made in */usr/lib/crontab* on each server. More information on this topic is in ypxfr(1M).

YP maps should contain two distinguished key-value pairs. The first is the key YP_LAST_MODIFIED, having as a value a ten-character ASCII order number. The order number should be the operating system time in seconds when the map was built. The second key is YP_MASTER_NAME, with the name of the YP master server as a value. *makedbm* generates both key-value pairs automatically. A map that does not contain both key-value pairs can be served by the YP, but the *ypserv* process will not be able to return values for "Get order number" or "Get master name" requests. In addition, values of these two keys are used by *ypxfr* when it transfers a map from a master YP server to a slave. If *ypxfr* cannot figure out where to get the map, or if it is unable to determine whether the local copy is more recent than the copy at the master, you must set extra command line switches when you run it.

YP maps must be generated and modified only at the master server. They are copied to the slaves using *ypxfr*(1M) to avoid potential byte-ordering problems among YP servers running on machines with different architectures, and to minimize the amount of disk space required for the dbm files. The YP database can be initially set up for both masters and slaves by using *ypinit*(1M).

After the server databases are set up, it is probable that the contents of some maps will change. In general, some ASCII source version of the database exists on the master, and it is changed with a standard text editor. The update is incorporated into the YP map and is propagated from the master to the slaves by running */etc/yp/Makefile* . All supplied maps have entries in */etc/yp/Makefile* ; if you add a YP map, edit this file to support the new map. The makefile uses *makedbm* to generate the YP map on the master, and *yppush* to propagate the changed map to the slaves. *yppush* is a client of the map *ypservers* , which lists all the YP servers. For more information on this topic, see *yppush*(1M).

SEE ALSO

makedbm(1M), ypinit(1M), ypmake(1M), ypxfr(1M), yppush(1M), yppoll(1M), ypserv(1M), rpcinfo(1M),

INDEX

A

administration, network 1:7; 3:3; 7:2
auth_destroy 2:22, 33
authentication 1:12; 2:15, 21–25, 33–34, 43–44, 46; 3:3, 5, 7, 10; 5:2–11, 5–6, 21; 7:2, 11
authentication subsystem 2:21
authentication type 2:21
authunix_create 2:34
authunix_create_default 2:22, 34

B

batching 2:15, 17; 3:6–19
biod 7:4–5, 7, 12–13
broadcast RPC 2:16; 3:6

C

callback 2:25, 29; 6:12–31
callback, RPC 2:30
callrpc 2:1, 3–4, 6–7, 13–15, 31, 34–35, 39
clnt_broadcast 2:16–17, 34–35, 39
clnt_call 2:13–15, 19, 27, 35, 39
clnt_destroy 2:13, 15, 19, 27, 35–36
clnt_freeres 2:36
clnt_geterr 2:36
clnt_pcreateerror 2:36, 40
clnt_perrno 2:17, 27, 31, 34, 36
clnt_perror 2:13, 19, 37
clntraw_create 2:37, 44
clnttcp_create 2:15, 19, 27, 37
clntudp_create 2:10, 13, 15, 22, 34–35, 38

D

daemon, utility 2:25
deserializing 2:6, 8–9, 13; 4:5–14, 9–10
dispatch, service 2:22–23, 43–44
domain 6:1–3, 6–11, 13, 15, 17; 7:16–19, 22–26, 28–29, 33, 38
domainname 6:8–11, 13, 15; 7:16–17, 19, 28
domains 6:1, 6; 7:16

G

get_myaddress 2:38

H

handle, request 2:22
  transport 2:10, 40–42, 53
  UDP 2:10
  XDR 2:8
hard mounted 7:6
hostname 2:7, 31; 5:24; 7:10–11, 19

I

inetd 2:15, 25

L

loopback, memory-based 2:37

M

makedbm 7:18–19, 23–24, 26–27
mapper, port 2:10; 3:11; 7:8–12, 10–11
mappings, program-to-port 2:38
master server 7:17, 19, 22, 24–27
memory-based loopback 2:37
mounted, hard 7:6
  soft 7:6, 12

N

network 1:7, 13; 2:21; 3:1, 3; 7:1–3, 6
network administration 1:7; 3:3; 7:2
network services 1:7, 13; 2:21; 3:1; 7:1–3, 6
nfsd 7:4–5, 9–11
number, procedure 2:4–5, 11, 14, 16, 22, 24, 43; 3:4, 12; 5:1, 13, 24; 6:2, 12
  program 2:4, 11, 15–16, 22, 30–31, 39; 3:4–5, 7, 12; 5:1; 6:2, 7, 12, 14
  version 2:4, 15–16, 22, 25; 3:4, 7, 12; 5:2; 6:2, 7

P

paradigm 2:1; 6:2–2
performance, remote 2:3
pipeline 2:17; 3:6–19
pmap 2:16, 49–50
pmap_getmaps 2:38
pmap_getport 2:38
pmap_rmtcall 2:39
pmap_set 2:30, 39
pmap_unset 2:9, 11, 18, 27, 39
port mapper 2:10; 3:11; 7:8–12, 10–11
procedure number 2:4–5, 11, 14, 16, 22, 24, 43; 3:4, 12; 5:1, 13, 23; 6:2, 12
program number 2:4, 11, 15–16, 22, 30–31, 39; 3:4–5, 7, 12; 5:1; 6:2, 7, 12, 14
program-to-port mappings 2:38
propagation 6:2, 5; 7:24–26, 33
Protocol, User Datagram 2:4

R

registerrpc 2:1, 3, 5–6, 9–11, 31, 40
remote performance 2:3
remote performance statistics 2:3
request handle 2:22
rnusers 2:1, 3
routine, service dispatch 2:22–23, 43–44
RPC 1:3, 7, 9–10, 12; 2:0–7, 9–11, 13, 15–23, 25–27, 29–31, 33–47, 49–50, 52; 3:0–7, 10; 4:1–12, 3, 5, 7, 9, 19–20, 22, 38; 5:1–2, 6, 13, 15, 21, 23; 6:1–4, 7, 10–12, 14, 17; 7:4, 17, 25
  broadcast 2:16; 3:6

RPC callback 2:30
rpc_createerr 2:39–40
rstat 2:3, 15

S

serializing 2:6, 8, 13; 4:5–14, 10, 17, 30–31
server, master 7:17, 19, 22, 24–27
  slave 7:17, 22, 25–26, 32–33
  stateless 1:8, 12; 5:4
service dispatch routine 2:22–23, 43–44
services, network 1:7, 13; 2:21; 3:1; 7:1–3, 6
slave server 7:17, 22, 25–26, 32–33
socket 2:10, 13, 15–16, 19, 27, 30, 37–38, 41, 45; 5:9, 11
soft mounted 7:6, 12
stateless server 1:8, 12; 5:4
statistics, remote performance 2:3
subsystem, authentication 2:21
svc_destroy 2:40
svcerr_auth 2:43–44
svcerr_decode 2:11, 18, 27, 31, 43
svcerr_noproc 2:9, 11, 18, 23, 26–27, 43
svcerr_noprog 2:43
svcerr_progvers 2:44
svcerr_systemerr 2:23–24, 44
svcerr_weakauth 2:23–24, 44
svc_fds 2:15, 40, 53
svc_freeargs 2:13, 18, 41
svc_getargs 2:11–13, 18, 27, 31, 41, 43
svc_getcaller 2:41
svc_getreq 2:15–16, 40–42
svcraw_create 2:37, 44
svc_register 2:9, 11, 18, 26–27, 31, 42
svc_run 2:5, 9, 15–16, 18, 25, 27, 31, 40–42
svc_sendreply 2:9, 11, 18, 23, 26–27, 31, 42
svctcp_create 2:10, 15, 18, 27, 30, 45
svcudp_create 2:9–10, 15, 25, 30–31, 40, 45
svc_unregister 2:43

T

TCP 2:15, 17–19, 21, 25, 27, 37, 45; 3:2, 6, 11; 4:19; 6:2–20, 4; 7:7, 31–34
timestamp 7:33
transport handle 2:10, 21, 40–42, 53

U

UDP 1:9; 2:4–6, 9–10, 15–16, 21, 25, 34, 38, 40, 45; 3:2, 6, 11; 4:19; 5:1–12, 6, 21; 6:2; 7:7, 31–34
UDP handle 2:10
User Datagram Protocol 2:4
utility daemon 2:25

V

version number 2:4, 15–16, 22, 25; 3:4, 7, 12; 5:2; 6:2, 7

X

XDR 1:3, 7, 9; 2:6–8, 11–14, 16, 19, 27, 36, 41–42, 45; 3:1–52, 6, 10; 4:0–1, 3–24, 26, 28; 5:1–3, 6–7, 13, 21; 6:2–3, 10, 16
XDR handle 2:8
xdr_accepted_reply 2:45
xdr_array 2:7–8, 12, 46; 4:11–14, 16, 33
xdr_authunix_parms 2:46
xdr_bool 2:6, 11, 46; 4:8, 31–32, 34
xdr_bytes 2:7–8, 12, 27, 47; 4:10–11, 34
xdr_callhdr 2:47
xdr_callmsg 2:47
xdr_double 2:7, 47; 4:8, 34
xdr_enum 2:6, 48; 4:8, 35
xdr_float 2:7, 48; 4:8, 35
xdr_inline 2:48; 4:35
xdr_int 2:6–8, 31, 48; 4:7, 12, 14–15, 28, 36
xdr_long 2:6, 49; 4:3, 5–7, 29, 36
xdr_opaque 2:7, 49; 4:13–14, 36
xdr_opaque_auth 2:49
xdr_pmap 2:49
xdr_pmaplist 2:50
xdr_reference 2:7–8, 50; 4:16–17, 30–32, 36
xdr_rejected_reply 2:50
xdr_replymsg 2:50
xdr_short 2:7, 51; 4:7, 28, 37
xdr_string 2:7–8, 51; 4:9–10, 12–14, 17, 37, 39
xdr_u_int 2:6, 11, 51; 4:7, 28, 37
xdr_u_long 2:3–6, 9, 13, 23, 26, 51; 4:7, 38
xdr_union 2:7, 52; 4:15–16, 30–31, 38
xdr_u_short 2:7, 26, 52; 4:7, 28, 38
xdr_void 2:3–5, 9, 13, 18–19, 23, 26–27, 31, 52; 4:9, 30–31, 38

xdr_wrapstring 2:7, 18–19, 52; 4:39
xprt_register 2:53
xprt_unregister 2:53

Y

ypbind 7:7–8, 17–18, 20–21, 28–29, 31–33, 36
ypcat 7:19, 27, 29
ypclnt 2:3
ypfiles 7:18
ypinit 7:17–19, 22, 24, 26
ypmake 7:18, 22
ypmatch 2:3; 7:19
yppoll 7:18, 29
yppush 7:18, 25–27, 33
ypserv 7:8, 17–18, 20, 22, 24–25, 28–29, 33–34
ypset 7:18
ypwhich 7:19, 29, 32
ypxfr 7:18, 24–27, 32