# CONVEX NFS Reference Set

Order No. DSW-111

Third Edition
November 1990

CONVEX Computer Corporation
Richardson, Texas USA

**CONVEX**
**NFS Reference Set**

Order No. DSW-111

Copyright 1990 CONVEX Computer Corporation.
All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, electronically stored, or reduced to machine readable form without prior written consent from CONVEX Computer Corporation.

Although the material contained herein has been carefully reviewed, CONVEX Computer Corporation (CONVEX) does not warrant it to be free of errors or omissions. CONVEX reserves the right to make corrections, updates, revisions or changes to the information contained herein. CONVEX does not warrant the material described herein to be free of patent infringement.

Unless provided otherwise in writing with CONVEX Computer Corporation (CONVEX), the program described herein is provided as is without warranty of any kind, either expressed or implied, including, but not limited to the implied warranties of merchantability and fitness for a particular purpose. Some states do not allow the exclusion of implied warranties. The above exclusion may not be applicable to all purchasers because warranty rights can vary from state to state. In no event will CONVEX be liable to anyone for special, collateral, incidental or consequential damages, including any lost profits or lost savings, arising out of the use or inability to use this program. CONVEX will not be liable even if it has been notified of the possibility of such damage by the purchaser or any third party.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.

Ethernet is a trademark of Xerox Corporation.

NFS and Sun Workstation are trademarks of Sun Microsystems, Inc.

UNIX is a trademark of AT&T Bell Laboratories.

Printed in the United States of America.

# Revision Information for

# CONVEX NFS Reference Set

| Edition | Document No. | Description |
|---|---|---|
| Third | 710-001730-205 | Released with ConvexOS V9.0, November 1990. Includes these changes and additions:<br>Added Technical Notes for XDR<br>Added Chapter on rpcgen |
| Second, Rev.2 | 710-001730-203 | Released with CONVEX UNIX V7.0, November 1988. |
| Second, Rev. 1 | 710-001730-202 | Released with CONVEX UNIX V6.2, April 1988. |
| 2.0 | 710-001730-201 | Released with CONVEX UNIX V6.1, October 1987. |
| 1.0 | 710-000630-000 | Initial release with CONVEX UNIX V6.0, April 1987. |

# Contents

## Using This Reference

Purpose and Audience .......... vii
Organization .......... vii
Technical Assistance .......... vii
Associated Documents .......... viii
Ordering Documentation .......... viii
Notational Conventions .......... ix
  Command Syntax .......... ix
  General Conventions .......... ix
Reader Response .......... x

## Network File System: Protocol Specification

Introduction .......... 1-1
  Remote Procedure Call .......... 1-1
  External Data Representation .......... 1-1
  Stateless Servers .......... 1-1
  File System Model .......... 1-3
  RPC Information .......... 1-3
  Basic Data Types .......... 1-4
    stat .......... 1-4
    type .......... 1-6
    fhandle .......... 1-6
    timeval .......... 1-6
    fattr .......... 1-7
    sattr .......... 1-8
    filename .......... 1-8
    path .......... 1-9
    attrstat .......... 1-9
  Server Procedures .......... 1-9
    Get File Attributes .......... 1-10
    Set File Attributes .......... 1-10
    Get file system Root .......... 1-11
    Read From Symbolic Link .......... 1-11
    Read From File .......... 1-12
    Write to File .......... 1-12
    Create File .......... 1-13
    Remove File .......... 1-13

Create Link to File ..... 1-14
Remove Directory ..... 1-15
Get file system Attributes ..... 1-17
Server/Client Relationship ..... 1-18
Pathname Interpretation ..... 1-18
Permission Issues ..... 1-18
Setting RPC Parameters ..... 1-19
RPC Information ..... 1-20
Sizes of XDR Structures ..... 1-20
Basic Data Types ..... 1-21
fhandle ..... 1-21
fhstatus ..... 1-21
dirpath ..... 1-21
Server Procedures ..... 1-22
Do Nothing ..... 1-22
Add Mount Entry ..... 1-22
Remove Mount Entry ..... 1-23
Remove All Mount Entries ..... 1-23
Return Export List ..... 1-23


## Remote Procedure Calls: Protocol Specification


Introduction ..... 2-1
Terminology ..... 2-1
The RPC Model ..... 2-1
Transports and Semantics ..... 2-2
Binding and Rendezvous Independence ..... 2-3
Authentication ..... 2-3
RPC Protocol Requirements ..... 2-4
Programs and Procedures ..... 2-4
Authentication ..... 2-5
Program Number Assignment ..... 2-6
Other Uses of the RPC Protocol ..... 2-6
Batching ..... 2-6
Broadcast RPC ..... 2-6
Null Authentication ..... 2-10
UNIX Authentication ..... 2-10
DES Authentication ..... 2-11
Naming ..... 2-11
DES Authentication Verifiers ..... 2-11
Nicknames and Clock Synchronization ..... 2-12
DES Authentication Protocol (in XDR language) ..... 2-13
Diffie-Hellman Encryption ..... 2-15
Record Marking Standard ..... 2-16
An Example Service Described in the RPC Language ..... 2-17
The RPC Language Specification ..... 2-18
Syntax Notes ..... 2-18

Port Mapper Protocol Specification (in RPC Language) .......... 2-19
Port Mapper Operation .......... 2-21

## Remote Procedure Call: Programming Guide

Layers of RPC .......... 3-2
The Highest Layer .......... 3-2
The Middle Layer .......... 3-2
The Lowest Layer .......... 3-2
The RPC Paradigm .......... 3-3
Higher Layers of RPC .......... 3-4
Highest Layer .......... 3-4
Intermediate Layer .......... 3-5
Assigning Program Numbers .......... 3-8
Passing Arbitrary Data Types .......... 3-10
Lowest Layer of RPC .......... 3-13
More on the Server Side .......... 3-14
Memory Allocation with XDR .......... 3-17
The Calling Side .......... 3-19
Other RPC Features .......... 3-21
Select on the Server Side .......... 3-21
Broadcast RPC .......... 3-21
Broadcast RPC Synopsis .......... 3-22
Batching .......... 3-22
Authentication .......... 3-26
UNIX Authentication .......... 3-27
DES Authentication .......... 3-30
Using inetd(8C) .......... 3-33
More Examples .......... 3-34
Versions .......... 3-34
TCP .......... 3-36
Callback Procedures .......... 3-39

## rpcgen Programming Guide

The rpcgen Protocol Compiler .......... 4-1
Converting Local Procedures into Remote Procedures .......... 4-2
Generating XDR Routines .......... 4-8
The C-preprocessor .......... 4-13
rpcgen Programming Notes .......... 4-14
Timeout Changes .......... 4-14
Handling Broadcast on the Server Side .......... 4-14
Other Information Passed to Server Procedures .......... 4-15
RPC Language .......... 4-16
Definitions .......... 4-16
Structures .......... 4-16
Unions .......... 4-17
Enumerations .......... 4-17

Typedef ........ 4-18
Constants ........ 4-18
Programs ........ 4-19
Declarations ........ 4-20
Special Cases ........ 4-21

## XDR Standard: Protocol Specification

Introduction ........ 5-1
Basic Block Size ........ 5-1
XDR Data Types ........ 5-2
Integer ........ 5-2
Unsigned Integer ........ 5-2
Enumeration ........ 5-2
Boolean ........ 5-3
Hyper Integer and Unsigned Hyper Integer ........ 5-3
Floating-point ........ 5-4
Double-precision Floating-point ........ 5-5
Fixed-length Opaque Data ........ 5-6
Variable-length Opaque Data ........ 5-6
String ........ 5-7
Fixed-length Array ........ 5-8
Variable-length Array ........ 5-8
Structure ........ 5-9
Discriminated Union ........ 5-9
Void ........ 5-10
Constant ........ 5-10
Typedef ........ 5-10
Optional-data ........ 5-11
Areas for Future Enhancements ........ 5-12
Discussion ........ 5-13
Why a Language for Describing Data? ........ 5-13
Why Only One Byte-Order for an XDR Unit? ........ 5-13
Why does XDR use Big-Endian Byte-Order? ........ 5-13
Why is the XDR Unit Four Bytes Wide? ........ 5-13
Why must Variable-Length Data be Padded with Zeros? ........ 5-13
Why is there No Explicit Data-Typing? ........ 5-13
The XDR Language Specification ........ 5-14
Notational Conventions ........ 5-14
Lexical Notes ........ 5-14
Syntax Information ........ 5-15
Syntax Notes ........ 5-16
Example of an XDR Data Description ........ 5-17
References ........ 5-19

## XDR Standard: Technical Notes

Justification ........ 6-2
Number Filters ........ 6-9
Floating Point Filters ........ 6-10
Enumeration Filters ........ 6-10
No Data ........ 6-10
Constructed Data Type Filters ........ 6-11
Strings ........ 6-11
Byte Arrays ........ 6-12
Arrays ........ 6-12
Opaque Data ........ 6-16
Fixed Sized Arrays ........ 6-16
Discriminated Unions ........ 6-17
Pointers ........ 6-19
Pointer Semantics and XDR ........ 6-20
Non-filter Primitives ........ 6-21
XDR Operation Directions ........ 6-21
XDR Stream Access ........ 6-22
Standard I/O Streams ........ 6-22
Memory Streams ........ 6-22
Record (TCP/IP) Streams ........ 6-23
XDR Stream Implementation ........ 6-25
The XDR Object ........ 6-25
Linked Lists ........ 6-27

## Yellow Pages: Protocol Specification

Introduction and Terminology ........ 7-1
RPC Remote Procedure Call ........ 7-2
External Data Representation (XDR) ........ 7-2
YP Database Servers ........ 7-4
Maps and Map Operations ........ 7-4
Map Structure ........ 7-4
Match Operation ........ 7-4
Map Entry Enumeration ........ 7-4
Entire Map Retrieval ........ 7-4
Map Update ........ 7-4
Master and Slave YP Database Servers ........ 7-4
Map Propagation and Consistency ........ 7-5
Functions to Aid in Map Propagation ........ 7-5
Domains ........ 7-5
Restrictions ........ 7-5
Map Update Within the YP ........ 7-5
Version Commitment Across Multiple Requests ........ 7-5
Guaranteed Global Consistency ........ 7-6
Access Control ........ 7-6
YP Database Server Protocol Definition ........ 7-6

RPC Constants .......... 7-6
Other Manifest Constants .......... 7-6
Remote Procedure Return Values .......... 7-6
Basic Data Structures .......... 7-7
YP Database Server Remote Procedures .......... 7-9
YP Binders .......... 7-13
YP Binder Protocol Definition .......... 7-13
RPC Constants .......... 7-13
Other Manifest Constants .......... 7-13
Basic Data Structures .......... 7-14
YP Binder Remote Procedures .......... 7-15

**Appendix A Reporting Problems**

**Index**

# Using This Reference

## Purpose and Audience

The *CONVEX NFS Reference Set* describes:

- NFS protocol specifications
- RPC protocol specifications
- rpcgen programming
- XDR protocol specifications
- YP protocol specifications

This reference addresses system managers.

## Organization

This guide is organized into the following chapters and appendix:

- Chapter 1 descibes NFS protocol.
- Chapter 2 describes RPC protocol.
- Chapter 3 describes RPC programming techniques.
- Chapter 4 describes rpcgen programming techniques.
- Chapter 5 describes XDR protocol.
- Chapter 6 contains XDR technical notes.
- Chapter 7 describes YP protocol.
- Appendix A describes how to use the CONVEX contact utility.

## Technical Assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC).

- Within the continental U.S., call 1(800)952-0379.
- From Canada, call 1(800)345-2384.
- All other locations, contact the local CONVEX office.

## Associated Documents

Using this software may require information not specific to the tasks described in this document.

For more information on CONVEX NFS, you can order these books from CONVEX Computer Corporation:

- *CONVEX NFS System Manager's Guide* (DSW-113). Provides information for the administration of NFS.
- *CONVEX NFS User's Guide* (DSW-112). Provides informaion about various NFS related utilities.
- *ConvexOS Programmer's Reference* (DSW-332). Standard reference for the ConvexOS operating system.

## Ordering Documentation

To order the current edition of this or any other CONVEX document, send requests to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson, TX 75083-3851 USA

Include the order number or the exact title, as listed on the front cover.

# Notational Conventions

This section discusses notational conventions used in this book.

## Command Syntax

Consider this example:

```
COMMAND input_file [...] {a | b} [output_file]
   ①         ②        ③      ④          ⑤
```

1. COMMAND must be typed as it appears.
2. *input_file* indicates a file name that must be supplied by the user.
3. The horizontal ellipsis in brackets indicates that additional input file names may be supplied.
4. Either `a` or `b` must be supplied.
5. *output_file* indicates an optional file name.

## General Conventions

In general, the following conventions are used in this guide:

- **`Bold constant-width font`** identifies user input in examples.
- *Italics*
  - Designate user-supplied variables in a command-line example.
  - Introduce new and important terms.
  - Identify variables in mathematical equations.
  - Indicate titles of documents.
- `Constant-width font` is used to designate input and output, including:
  - Command names and options.
  - System calls.
  - Data structures and types.
  - Directives, program statements, display examples, printout examples, and error messages returned.
- Horizontal ellipsis (...) shows repetition of the preceding item(s).
- Vertical ellipses show that lines of code have been left out of an example.
- Words and abbreviations that indicate keyboard keys you press are identified in a distinctive bold type. For example, **RETURN** refers to the carriage return key. Words separated by a hyphen indicate two keys that you must press simultaneously. For example, **CTRL-X** indicates that you must press and hold down the **CTRL** key and then press the **X** key.
- The word "enter" in a phrase such as "enter `ls`" means that you type the command and then press **RETURN**.
- References to the *ConvexOS Programmer's Reference* appear in the form adb(1), where the name of the man page is followed by its section number enclosed in parentheses.

## Note

**A Note highlights supplemental information.**

## Caution

**A Caution highlights procedures or information necessary to avoid damage to equipment, software, or data.**

## Reader Response

If you have comments or questions about the contents of this book, please notify the CONVEX documentation department by using the Reader's Forum at the end of this document

# Network File System: Protocol Specification

1

## Introduction

The Network File System (NFS) protocol provides transparent remote access to shared file systems over local area networks. The NFS protocol is designed to be machine, operating system, network architecture, and transport protocol independent. This independence is achieved through the use of Remote Procedure Call (RPC) primitives built on top of an External Data Representation (XDR). Implementations exist for a variety of machines, from personal computers to supercomputers.

The supporting mount protocol allows the server to hand out remote access privileges to a restricted set of clients. It performs the operating system-specific functions that allow, for example, to attach remote directory trees to some local file system.

### Remote Procedure Call

The remote procedure call specification provides a procedure- oriented interface to remote services. Each server supplies a program that is a set of procedures. NFS is one such "program." The combination of host address, program number, and procedure number specifies one remote service procedure. RPC does not depend on services provided by specific protocols, so it can be used with any underlying transport protocol. See chapter *Remote Procedure Calls: Protocol Specification.*

### External Data Representation

The External Data Representation (XDR) standard provides a common way of representing a set of data types over a network. The NFS Protocol Specification is written using the RPC data description language. For more information, see chapter *External Data Representation Standard: Protocol Specification.* Implementations of XDR and RPC, are available in the public domain, but NFS does not require their use. Any software that provides equivalent functionality can be used, and if the encoding is exactly the same, it can interoperate with other implementations of NFS.

### Stateless Servers

The NFS protocol is stateless. That is, a server does not need to maintain any extra state information about any of its clients in order to function correctly. Stateless servers have a distinct advantage over stateful servers in the event of a failure. 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, or the network temporarily went down. The client of a stateful server, on the other hand, needs to either detect a server crash and rebuild the server's state when it comes back up, or cause client operations to fail.

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

On the other hand, NFS deals with objects such as files and directories that inherently have state. The goal is to not introduce any extra state in the protocol itself. Another way to simplify recovery is by making operations "idempotent" whenever possible (so that transmissions do not have unpleasant side effects).

# NFS Protocol Definition

Servers have been known to change over time, as well as the protocol that they use; so, RPC provides a version number with each RPC request. This RFC describes Version 2 of the NFS protocol. Even in the Version 2, there are various obsolete procedures and parameters that will be removed in later versions.

## File System Model

NFS assumes a file system that is hierarchical, with directories as all but the bottom-level files. Each entry in a directory (file, directory, device, etc.) has a string name. Different operating systems may have restrictions on the depth of the tree or the names used, as well as using different syntax to represent the "pathname", which is the concatenation of all the "components" (directory and file names) in the name. A "file system" is a tree on a single server (usually a single disk or physical partition) with a specified "root." Some operating systems provide a "mount" operation to make all file systems appear as a single tree, while others maintain a "forest" of file systems. Files are unstructured streams of uninterpreted bytes.

NFS looks up one component of a pathname at a time. It may not be obvious why it does not just take the whole pathname, traipse down the directories, and return a file handle when it is done. There are several good reasons not to do this. First, pathnames need separators between the directory components, and different operating systems use different separators. We could define a Network Standard Pathname Representation, but then every pathname would have to be parsed and converted at each end. Other issues are discussed in *NFS Implementation Issues*.

Although files and directories are similar objects in many ways, different procedures are used to read directories and files. This provides a network standard format for representing directories. The same argument as above could have been used to justify a 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 just too slow.

## RPC Information

Authentication:
The NFS service uses AUTH_UNIX(), AUTH_DES(), or AUTH_SHORT() style authentication, except in the NULL procedure where AUTH_NONE() is also allowed.

Port Protocols:
NFS currently is supported on UDP/IP only.

Port Number:
The NFS protocol currently uses the UDP port number 2049. This is not an officially assigned port, so later versions of the protocol use the portmapping facility of RPC.

## Sizes of XDR Structures

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

```
 /*
  * The maximum number of bytes of data in a READ or WRITE
  * request
  */
const MAXDATA = 8192;

 /* The maximum number of bytes in a pathname argument */
 const MAXPATHLEN = 1024;

 /* The maximum number of bytes in a file name argument */
 const MAXNAMLEN = 255;

 /*
  *  The size in bytes of the opaque "cookie" passed by
  *  READDIR
  */
 const COOKIESIZE  = 4;

 /* The size in bytes of the opaque file handle */
 const FHSIZE = 32;
```

## Basic Data Types

The following XDR definitions are basic structures and types used in other structures described further on.

### stat

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

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. Other values indicate some kind of error occurred on the server side during the servicing of the procedure. The following error values are derived from UNIX error numbers:

| **Error Code** | **Meaning** |
|---|---|
| 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 | Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example. |
| 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 file system to reach its limit. |
| NFSERR_ROFS | Read-only file system. Write attempted on a read-only file system. |
| 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_WFLUSH | The server's write cache used in the WRITECACHE() call got flushed to disk. |

### type

```
enum ftype {            /* enumeration ftype gives the type of a file */
      NFNON = 0,        /* indicates a non-file */
      NFREG = 1,        /* a regular file */
      NFDIR = 2,        /* a directory */
      NFBLK = 3,        /* a block-special device */
      NFCHR = 4,        /*a character-special device */
      NFLNK = 5         /* a symbolic link */
};
```

### fhandle

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

The fhandle is the file handle passed between the server and 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

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

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

### fattr

```
struct fattr {                 /* attributes of a file */

     ftype type;               /* type of the file */

     unsigned int mode;        /* access mode encoded as a set
                                  a set of bits */

     unsigned int nlink;       /* number of hard links to the
                                  file */

     unsigned int uid;         /* user identification number of
                                  the owner of the file */

     unsigned int gid;         /* group identification number of
                                  the group of the file */

     unsigned int size;        /* size in bytes of the file */

     unsigned int blocksize;   /* size in bytes of a block of the
                                  file */

     unsigned int rdev;        /* device number of the file */

     unsigned int blocks;      /* if rdev is type NFCHR or
                                  NFSBLK, blocks is the number
                                  of blocks the file takes up on
                                  disk */

     unsigned int fsid;        /* file system identifier for the
                                  file system containing the
                                  file */

     unsigned int fileid;      /* number that uniquely
                                  identifies the file within its
                                  file system */

     timeval atime;            /* time when the file was last
                                  accessed for either
                                  read or write */

     timeval mtime;            /* time when the file data was
                                  last modified (written) */

     timeval ctime;            /* time when the status of the
                                  file was last changed */
};
```

nlink also indicates the number of different names for the same file. Writing to the file also changes ctime if the size of the file changes.

The descriptions given below specify the bit positions using octal numbers.

| Bit | Description |
|---|---|
| 0040000 | This is a directory; "type" field should be NFDIR. |
| 0020000 | This is a character special file; "type" field should be NFCHR. |
| 0060000 | This is a block special file; "type" field should be NFBLK. |
| 0100000 | This is a regular file; "type" field should be NFREG. |
| 0120000 | This is a symbolic link file; "type" field should be NFLNK. |
| 0140000 | This is a named socket; "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. |

The bits are the same as the mode bits returned by the stat(2) system call. The file type is specified both in the mode bits and in the file type. This is fixed in future versions.

The rdev field in the attributes structure is an operating system specific device specifier.

### sattr

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

The sattr structure contains the file attributes which can be set from the client. The fields are the same as for fattr previously described. A "size" of zero indicates the file will be truncated. A value of -1 indicated a field that will 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 without internal structure, but to the client it is the name of a node in a file system tree.

### attrstat

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

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

### diropres

```
union diropres switch (stat status) {
    case NFS_OK:
        struct {
            fhandle file; /* new file handle */
           fattr   attributes; /* attributes associated with
                                  the new file handle */
        } diropok;
    void;
};
```

The results of a directory operation are returned in a diropres structure. If the call succeeds, a new file handle file and the fattr attributes are returned along with the status.

## Server Procedures

The protocol definition is given as a set of procedures with arguments and results defined using the RPC language. A brief description of the function of each procedure should provide enough information to allow implementation.

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, file system information blocks (such as indirect blocks), 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 would wait 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.

```
/* Remote file service routines */
program NFS_PROGRAM {
    version NFS_VERSION {
        void        NFSPROC_NULL(void)              = 0;
        attrstat    NFSPROC_GETATTR(fhandle)        = 1;
        attrstat    NFSPROC_SETATTR(sattrargs)      = 2;
        void        NFSPROC_ROOT(void)              = 3;
        diropres    NFSPROC_LOOKUP(diropargs)       = 4;
        readlinkres NFSPROC_READLINK(fhandle)       = 5;
        readres     NFSPROC_READ(readargs)          = 6;
        void        NFSPROC_WRITECACHE(void)        = 7;
        attrstat    NFSPROC_WRITE(writeargs)        = 8;
        diropres    NFSPROC_CREATE(createargs)      = 9;
        stat        NFSPROC_REMOVE(diropargs)       = 10;
        stat        NFSPROC_RENAME(renameargs)      = 11;
        stat        NFSPROC_LINK(linkargs)          = 12;
        stat        NFSPROC_SYMLINK(symlinkargs)    = 13;
        diropres    NFSPROC_MKDIR(createargs)       = 14;
        stat        NFSPROC_RMDIR(diropargs)        = 15;
        readdirres  NFSPROC_READDIR(readdirargs)    = 16;
        statfsres   NFSPROC_STATFS(fhandle)         = 17;
    } = 2;
} = 100003;
```

### Do Nothing

```
void
NFSPROC_NULL(void) = 0;
```

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

### Get File Attributes

```
attrstat
NFSPROC_GETATTR (fhandle) = 1;
```

If the reply status is NFS_OK, then the reply attributes contain the attributes for the file given by the input fhandle.

### Set File Attributes

```
struct sattrargs {
    fhandle file;
    sattr attributes;
};

attrstat
NFSPROC_SETATTR (sattrargs) = 2;
```

The attributes argument contains fields which are either -1 or are the new value for the attributes of the fhandle file. If the reply status is NFS_OK, then the reply attributes have the attributes of the file after the SETATTR operation completes.

**Note**

**The use of -1 to indicate an unused field in "attributes" is changed in the next version of the protocol.**

### Get file system Root

```
void
NFSPROC_ROOT(void) = 3;
```

Obsolete. This procedure is no longer used because finding the root file handle of a file system requires moving pathnames between client and server. To do this correctly, we would need 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 *Mount Protocol Definition* later in this chapter for details.)

### Look Up File Name

```
diropres
NFSPROC_LOOKUP(diropargs) = 4;
```

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

### Read From Symbolic Link

```
union readlinkres switch (stat status) {
     case NFS_OK:
          path data;
     default:
          void;
};

readlinkres
NFSPROC_READLINK(fhandle) = 5;
```

If "status" has the value NFS_OK, then the reply "data" is the data in the symbolic link given by the file referred to by the fhandle argument.

**Note**

**Since NFS always parses pathnames on the client, the pathname in a symbolic link may mean something different (or be meaningless) on a different client or on the server if a different pathname syntax is used.**

### Read From File

```
struct readargs {
    fhandle file;
    unsigned offset;
    unsigned count;
    unsigned totalcount;
};

union readres switch (stat status) {
    case NFS_OK:
        fattr attributes;
        opaque data<NFS_MAXDATA>;
    default:
        void;
};

readres
NFSPROC_READ(readargs) = 6;
```

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 fattr structure "attributes."

**Note**

**The argument "totalcount" is unused and is removed in the next protocol revision.**

### Write to Cache

```
void
NFSPROC_WRITECACHE(void) = 7;
```

To be used in the next protocol revision.

### Write to File

```
struct writeargs {
    fhandle file;
    unsigned beginoffset;
    unsigned offset;
    unsigned totalcount;
    opaque data<NFS_MAXDATA>;
};

attrstat
NFSPROC_WRITE(writeargs) = 8;
```

Writes "data" beginning "offset" bytes from the beginning of "file." The first byte of the file is at offset zero. If the reply "status" is NFS_OK, then the reply "attributes" contains the attributes of the file after the write is finished. The write operation is atomic. Data from this call to WRITE will not be mixed with data from another client's calls.

**Note**

**The arguments "beginoffset" and "totalcount" are ignored and are removed in the next protocol revision.**

### Create File

```
struct createargs {
    diropargs where;
    sattr attributes;
};

diropres
NFSPROC_CREATE(createargs) = 9;
```

The file "name" is created in the directory given by "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 the replies "file" and "attributes" are its file handle and attributes, respectively. Any other reply status means that the operation failed, and no file was created.

**Note**

**This routine should pass an exclusive create flag, meaning "create the file only if it is not already there". This will be added in the next protocol revision.**

### Remove File

```
stat
NFSPROC_REMOVE(diropargs) = 10;
```

The file "name" is removed from the directory given by "dir." A reply of NFS_OK means the directory entry was removed.

**Note**

**Possibly non-idempotent operation. A non-idempotent operation interferes with operations with other clients by retrying unsuccessful calls.**

### Rename File

```
struct renameargs {
  diropargs from;
  diropargs to;
};

stat
NFSPROC_RENAME(renameargs) = 11;
```

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 the reply is *NFS_OK*, the file was renamed. The RENAME operation is atomic on the server; it cannot be interrupted.

**Note**

**Possibly non-idempotent operation.**

### Create Link to File

```
struct linkargs {
  fhandle from;
  diropargs to;
};

stat
FSPROC_LINK(linkargs) = 12;
```

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

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

**Note**

**Possibly non-idempotent operation.**

### Create Symbolic Link

```
struct symlinkargs {
    diropargs from;
    path to;
    sattr attributes;
};

stat
NFSPROC_SYMLINK(symlinkargs) = 13;
```

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

A symbolic link is a pointer to another file. The name given in "to" is not interpreted by the server, only stored in the newly created file. When the client references a file that is a symbolic link, the contents of the symbolic link are normally transparently reinterpreted as a pathname to substitute. A READLINK operation returns the data to the client for interpretation.

**Note**

**On CONVEX servers the attributes are never used, since symbolic links always have mode 0777.**

### Create Directory

```
diropres
NFSPROC_MKDIR (createargs) = 14;
```

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

**Note**

**Possibly non-idempotent operation.**

### Remove Directory

```
stat
NFSPROC_RMDIR(diropargs) = 15;
```

The existing empty directory "name" in the directory given by "dir" is removed. If the reply is NFS_OK, the directory was removed.

**Note**

**Possibly non-idempotent operation.**

### Read From Directory

```
struct readdirargs {
    fhandle dir;
    nfscookie cookie;
    unsigned count;
};

struct entry {
    unsigned fileid;
    filename name;
    nfscookie cookie;
    entry *nextentry;
};

union readdirres switch (stat status) {
    case NFS_OK:
        struct {
            entry *entries;
             bool eof;
        } readdirok;
    default:
        void;
};

readdirres
NFSPROC_READDIR (readdirargs) = 16;
```

Returns a variable number of directory entries, with a total size of up to "count" bytes, from the directory given by "dir." If the returned value of "status" is NFS_OK, then it is followed by a variable number of "entry" entries. Each "entry" contains a "fileid" which consists of a unique number to identify the file within a file system, 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 attributes of the file. (See section *Basic Data Types*.) The eof flag has a value of TRUE, if there are no more entries in the directory.

### Get file system Attributes

```
union statfsres (stat status) {
    case NFS_OK:
        struct {
            unsigned tsize;
            unsigned bsize;
            unsigned blocks;
            unsigned bfree;
            unsigned bavail;
         } info;
    default:
        void;
};

statfsres
NFSPROC_STATFS(fhandle) = 17;
```

If the reply status is NFS_OK, then the reply "info" gives the attributes for the file system that contains file referred to by the input fhandle. 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 file system.

blocks:
The total number of "bsize" blocks on the file system.

bfree:
The number of free "bsize" blocks on the file system.

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

**Note**

**This call does not work well if a file system has variable size blocks.**

## NFS Implementation Issues

The NFS protocol is designed to be operating system independent, but many operations have semantics similar to the operations of the UNIX file system. This section discusses some of the implementation-specific semantic issues.

### 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 file system semantics.

For example, some operating systems allow 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 file system. It is impossible for a stateless server to implement these semantics. The client can do some tricks such as renaming the file on remove, and only removing it on close. The server provides enough functionality to implement most file system semantics on the client.

Every NFS client can also potentially be a server, and remote and local mounted file systems can be freely intermixed. This leads to some interesting problems when a client travels down the directory tree of a remote file system and reaches the mount point on the server for another remote file system. Allowing the server to follow the second remote mount would require loop detection, server lookup, and user revalidation. Instead, clients are not permitted to cross a server's mount point. When a client does a LOOKUP on a directory on which the server has mounted a file system, 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.

### Pathname Interpretation

There are a few complications to the rule that pathnames are always parsed on the client. For example, symbolic links can have different interpretations on different clients. Another common problem for some implementations is the special interpretation of the pathname "..", which indicates the parent of a given directory. The next revision of the protocol uses an explicit flag to indicate the parent instead.

### 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 operating system 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. There are various problems with this method that can been 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. AUTH_DES (and the next revision of the NFS protocol) uses string names instead of numbers, but there are still complex problems to be solved.

Another problem arises due to the usually stateful open operation. Most operating systems check permission at open time, and then check that the file is open on each read and write request. With stateless servers, the server has no idea that the file is open and must do permission checking on each read and write call. On a local file system, a user can open a file and then change the permissions so that no one is allowed to touch it, but will still be able to write to the file because it is open. On a remote file system, by contrast, the write would fail. To get around this problem, the server's permission checking algorithm should allow the owner of a file to access it regardless of the permission setting.

A similar problem has to do with paging in from a file over the network. The operating system usually checks for execute permission before opening a file for demand paging, and then reads blocks from the open file. The file may not have read permission, but after it is opened it doesn't matter. An NFS server can not 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 most operating systems, a particular user (on the user ID zero) has access to all files no matter what permission and ownership they have. This "super-user" permission may not be allowed on the server, since anyone who can become super-user on their workstation could gain access to all remote files. The server by default maps user id 0 to -2 before doing its access checking. This works except for NFS root file systems, where super-user access cannot be avoided.

## Setting RPC Parameters

Various file system parameters and options should be set at mount time. The mount protocol is described in the section *Mount Protocol Definition*. For example, soft as well as hard mounts are usually both provided. Soft mounted file systems return errors when RPC operations fail (after a given number of optional retransmissions), while hard mounted file systems continue to retransmit forever. Clients and servers may need to keep caches of recent operations to help avoid problems with non-idempotent operations.

## Mount Protocol Definition

The mount protocol is separate from, but related to, the NFS protocol. It provides operating system specific services to get the NFS started looking up server path names, validating user identities, and checking access permissions. Clients use the mount protocol to get the first file handle, which allows them entry into a remote file system.

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 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 of the mount protocol is used with version 2 of the NFS protocol. The only connecting point is the fhandle structure, which is the same for both protocols.

### RPC Information

Authentication:
The mount service uses AUTH_UNIX() and AUTH_DES() style authentication only.

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

Port Number:
Consult the server's portmapper, described in chapter *Remote Procedure Calls: Protocol Specification* to find the port number on which the mount service is registered.

### Sizes of XDR Structures

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

```
/* The maximum number of bytes in a pathname argument */
const MNTPATHLEN = 1024;

/* The maximum number of bytes in a name argument */
const MNTNAMLEN = 255;

/* The size in bytes of the opaque file handle */
const FHSIZE = 32;
```

## Basic Data Types

This section presents the data types used by the mount protocol. In many cases they are similar to the types used in NFS.

### fhandle

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

The type 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 previous section *Basic Data Types*.

### fhstatus

```
union fhstatus switch (unsigned status) {
    case 0:
    fhandle directory;
default:
    void;
};
```

The type fhstatus is a union. 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 an error number.

### dirpath

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

The type dirpath is a server 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.

```
/*
* Protocol description for the mount program
*/
program MOUNTPROG {
/*
* Version 1 of the mount protocol used with
* version 2 of the NFS protocol.
*/
    version MOUNTVERS {
        void        MOUNTPROC_NULL(void)      = 0;
        fhstatus    MOUNTPROC_MNT(dirpath)    = 1;
        mountlist   MOUNTPROC_DUMP(void)      = 2;
        void        MOUNTPROC_UMNT(dirpath)   = 3;
        void        MOUNTPROC_UMNTALL(void)   = 4;
        exportlist  MOUNTPROC_EXPORT(void)    = 5;
    } = 1;
} = 100005;
```

### Do Nothing

```
void
MNTPROC_NULL(void) = 0;
```

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

### Add Mount Entry

```
fhstatus
MNTPROC_MNT(dirpath) = 1;
```

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

### Return Mount Entries

```
struct *mountlist {
    name       hostname;
    dirpath    directory;
    mountlist  nextentry;
};

mountlist
MNTPROC_DUMP(void) = 2;
```

Returns the list of remote mounted file systems. Mountlist contains one entry for each hostname and directory pair.

### Remove Mount Entry

```
void
MNTPROC_UMNT(dirpath) = 3;
```

Removes the mount list entry for the input dirpath.

### Remove All Mount Entries

```
void
MNTPROC_UMNTALL(void) = 4;
```

Removes all of the mount list entries for this client.

### Return Export List

```
struct *groups {
    name grname;
    groups grnext;
};

struct *exportlist {
    dirpath filesys;
    groups groups;
    exportlist next;
};

exportlist
MNTPROC_EXPORT(void) = 5;
```

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

**Note**

**The exportlist should contain more information about the status of the file system, such as a read-only flag.**

# Remote Procedure Calls: Protocol Specification

# 2

## Introduction

This chapter specifies a message protocol used in implementing Sun's Remote Procedure Call (RPC) package and discusses servers, services, programs, procedures, clients, and versions. (The message protocol is specified with the External Data Representation (XDR) language. See chapter *XDR Standard: Protocol Specification* for the details. Here, we assume that the reader is familiar with XDR and do not attempt to justify it or its uses). The paper by Birrell and Nelson [1] is recommended as an excellent background to and justification of RPC.

### Terminology

- *server*: a piece of software where network services are implemented
- *network service*: a collection of one or more remote programs
- *remote program*: implements one or more remote procedures. The procedures, their parameters, and results are documented in the specific program's protocol specification (see the Port Mapper Program Protocol, below, for an example).
- *network clients*: 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, imagine two programs that compose a network file service. One program deals with high-level applications such as file system access control and locking. The other deals with low-level file IO and has procedures such as "read" and "write." A client machine of the network file service calls the procedures associated with the two programs of the service on behalf of a user on the client machine.

### The RPC Model

The remote procedure call model is similar to the local procedure call model. In the local procedure call model, the caller places arguments to a procedure in a specified location. The caller transfers control to the procedure, which it eventually regains. At that point, the caller extracts the results of the procedure from the specified location and continues execution.

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

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

Note that in this model, only one of the two processes is active at any given time. However, we give this model only as an example. The RPC protocol makes no restrictions on the implemented concurrency model, so others are possible. For example, an implementation may have asynchronous RPC calls, so that the client can do useful work while waiting for the reply from the server. Another possibility is to have the server create a task to process an incoming request, so that the server can be free to receive other requests.

## Transports and Semantics

The RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. The protocol deals only with specification and interpretation of messages.

It is important to point out that RPC does not try to implement any kind of reliability and that the application must be aware of the type of transport protocol underneath RPC. If it knows it is running on top of a reliable transport such as TCP/IP[6], then most of the work is already done for it. On the other hand, if it is running on top of an unreliable transport such as UDP/IP[7], it must implement is own retransmission and time-out policy as the RPC layer does not provide this service.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, consider RPC running on top of an unreliable transport such as UDP/IP. If an application retransmits RPC messages after short time-outs, the only thing it can infer if it receives no reply is that the procedure was executed zero or more times. If it does receive a reply, then it can infer that the procedure was executed at least once.

A server may wish to remember previously granted requests from a client and not regrant them in order to insure some degree of execute-at-most-once semantics. A server can do this by taking advantage of the transaction ID that is packaged with every RPC request. The main use of this transaction is by the client RPC layer in matching replies to requests. However, a client application may choose to reuse its previous transaction ID when retransmitting a request. The server application, knowing this fact, can choose to remember this ID after granting a request and not regrant requests with the same ID in order to achieve some degree of execute-at-most-once semantics. The server is not allowed to examine this ID in any other way except as a test for equality.

On the other hand, if using a reliable transport such as TCP/IP, the application can infer from a reply message that the procedure was executed exactly once, but if it receives no reply message, it cannot assume the remote procedure was not executed. Note that even if a connection-oriented protocol like TCP is used, an application still needs time-outs and reconnection to handle server crashes.

There are other possibilities for transports besides datagram- or connection-oriented protocols. For example, a request-reply protocol such as VMTP[2] is perhaps the most natural transport for RPC.

**Note**

**In ConvexOS, an application can use either TCP/IP or UDP/IP transport services.**

## 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 the Port Mapper Program Protocol, below).

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.

## 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. Several different authentication protocols can be supported. A field in the RPC header indicates which protocol is being used. More information on specific authentication protocols can be found in section *Authentication Protocols*.

## RPC Protocol 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 (such as misspecification of a procedure's parameters)
- reasons why remote authentication failed
- any other reasons why the desired procedure was not called

### 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. Once an implementor has a program number, she can implement her remote program; the first implementation would most likely have the version number of 1. Because most new protocols evolve into better, stable, and mature protocols, a version field of the call message identifies which version of the protocol the caller is using. Version numbers make speaking old and new protocols through the same server process possible.

The procedure number identifies the procedure to be called. These numbers are documented in the specific program's protocol specification. For example, a file service's protocol specification may state that its procedure number 5 is "read" and procedure number 12 is "write."

Just as remote program protocols can change over several versions, the actual RPC message protocol can also change. Therefore, the call message also has in it the RPC version number, which is always equal to 2 for the current version of RPC.

The reply message to a request message has enough information to distinguish the following error conditions:

- The remote implementation of RPC does speak protocol version 2. The lowest and highest supported RPC version numbers are returned.
- The remote program is not available on the remote system.
- The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
- The requested procedure number does not exist. (This is usually a caller side protocol or programming error.)
- The parameters to the remote procedure appear to be garbage from the server's point of view. (Again, this is usually 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 has two authentication fields, the credentials and verifier. The reply message has one authentication field, the response verifier. The RPC protocol specification defines all three fields to be the following opaque type:

```
enum auth_flavor {
    AUTH_NULL         = 0,
    AUTH_UNIX         = 1,
    AUTH_SHORT        = 2,
    AUTH_DES          = 3
    /* and more to be defined */
};

struct opaque_auth {
    auth_flavor flavor;
    opaque body<400>;
};
```

Any opaque_auth structure is an auth_flavor enumeration followed by bytes which are opaque to the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields is specified by individual, independent authentication protocol specifications. (See section *Authentication Protocols* for definitions of the various authentication protocols.)

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

## Program Number Assignment

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

| Program Numbers | Description |
|---|---|
| 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 sites. The second range is for applications peculiar to a particular site. This range is intended primarily for debugging new programs. When a site 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 dynamically generate program numbers. The final groups are reserved for future use, and should not be used.

## Other Uses of the RPC Protocol

The intended use of this protocol is for calling remote procedures. That is, each call message is matched with a response message. However, the protocol itself is a message-passing protocol with which other (non-RPC) protocols can be implemented. Sun currently uses the RPC message protocol for the following two (non-RPC) protocols: batching (or pipelining) and broadcast RPC. These two protocols are discussed but not defined below.

### Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server. Batching typically uses reliable byte stream protocols (like TCP/IP) for its 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 RPC uses unreliable, packet-based protocols (like UDP/IP) as its transports. Servers that support broadcast protocols only respond when the request is successfully processed, and are silent in the face of errors. Broadcast RPC uses the Port Mapper RPC service to achieve its semantics. See section *Port Mapper Program Protocol* for more information.

## The RPC Message Protocol

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

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

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

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

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

/*
* Why authentication failed:
*/
enum auth_stat {
    AUTH_BADCRED      = 1,/* bad credentials */
    AUTH_REJECTEDCRED = 2,/* client must begin new session */
    AUTH_BADVERF      = 3,/* bad verifier */
    AUTH_REJECTEDVERF = 4,/* verifier expired or replayed  */
    AUTH_TOOWEAK      = 5 /* rejected for security reasons */
};
```

```
/*
* The  RPC  message:
* All  messages  start with  a transaction  identifier,  xid,
* followed  by a  two-armed  discriminated union.  The union's
* discriminant is a  msg_type which switches to  one of the two
* types   of the message.   The xid  of a REPLY  message always
* matches  that of the initiating CALL   message.   NB: The xid
* field is only  used for clients  matching reply messages with
* call messages  or for servers detecting  retransmissions;
* the service side  cannot treat this id  as any type of
* sequence number.
*/
struct rpc_msg {
    unsigned int xid;
    union switch (msg_type mtype) {
        case CALL:
            call_body cbody;
        case REPLY:
            reply_body rbody;
    } body;
};

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

/*
* Body of a reply to an RPC request:
* The call message was either accepted or rejected.
*/
union reply_body switch (reply_stat stat) {
    case MSG_ACCEPTED:
        accepted_reply areply;
    case MSG_DENIED:
        rejected_reply rreply;
} reply;
```

```
/*
* Reply to an RPC request  that  was accepted  by the server
* there could be an error even though the request was
* accepted.
* The first field is an authentication verifier that the server
* generates in order to  validate itself  to the caller.  It is
* followed by    a  union whose    discriminant  is   an  enum
* accept_stat.  The  SUCCESS  arm of   the union  is  protocol
* specific.  The PROG_UNAVAIL, PROC_UNAVAIL, and GARBAGE_ARGP
* arms of the union are void.   The PROG_MISMATCH arm specifies
* the lowest and highest version numbers of the remote program
* supported by the server.
*/
struct accepted_reply {
    opaque_auth verf;
    union switch (accept_stat stat) {
        case SUCCESS:
            /* procedure-specific results start here */
            opaque results[0];
        case PROG_MISMATCH:
            struct {
                unsigned int low;
                unsigned int high;
            } mismatch_info;
    default:
       /* Void.  Cases include PROG_UNAVAIL, PROC_UNAVAIL, *
        *  and GARBAGE_ARGS.
        */
        void;
    } reply_data;
};

/*
* Reply to an RPC request that was rejected by the server:
* The request  can   be rejected for  two reasons:  either the
* server   is not  running a  compatible  version  of the  RPC
* protocol    (RPC_MISMATCH), or   the  server   refuses   to
* authenticate the  caller  (AUTH_ERROR).  In  case of  an  RPC
* version mismatch, the server returns the  lowest and highest
* supported    RPC  version    numbers.  In   case   of refused
* authentication, failure status is returned.
*/
union rejected_reply switch (reject_stat stat) {
    case RPC_MISMATCH:
        struct {
            unsigned int low;
            unsigned int high;
        } mismatch_info;
    case AUTH_ERROR:
        auth_stat stat;
};
```

## Authentication Protocols

As previously stated, authentication parameters are opaque, but open-ended to the rest of the RPC protocol. This section defines some "flavors" of authentication implemented at (and supported by) Sun. Other sites are free to invent new authentication types, with the same rules of flavor number assignment as there is for program number assignment.

### Null Authentication

Often calls must be made where the caller does not know who he is or the server does not care who the caller is. In this case, the flavor value (the discriminant of the opaque_auth's union) of the RPC message's credentials, verifier, and response verifier is AUTH_NULL. The bytes of the opaque_auth's body are undefined. It is recommended that the opaque length be zero.

### UNIX Authentication

The caller of a remote procedure may wish to identify himself as he is identified on a UNIX system. The value of the credential's discriminant of an RPC call message is AUTH_UNIX. The bytes of the credential's opaque body encode the following structure:

```
struct auth_unix {
    unsigned int stamp;
    string machinename<255>;
    unsigned int uid;
    unsigned int gid;
    unsigned int gids<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 caller's 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 discriminant of the response verifier received in the reply message from the server may be AUTH_NULL or AUTH_SHORT. In the case of AUTH_SHORT the bytes of the response verifier's string encode an opaque structure. This new opaque structure may now be passed to the server instead of the original AUTH_UNIX flavor credentials. The server keeps a cache which maps shorthand opaque structures (passed back by way of an AUTH_SHORT style response verifier) to the original credentials of the caller. The caller can save network bandwidth and server cpu cycles by using the new credentials.

The server may flush the shorthand opaque structure at any time. If this happens, the remote procedure call message will be rejected due to an authentication error. The reason for the failure will be AUTH_REJECTEDCRED. At this point, the caller may wish to try the original AUTH_UNIX style of credentials.

## DES Authentication

UNIX authentication suffers from two major problems:

1. The naming is too UNIX-system oriented.

2. There is no verifier, so credentials can easily be faked.

DES authentication attempts to fix these two problems.

### Naming

The first problem is handled by addressing the caller by a simple string of characters instead of by an operating system specific integer. This string of characters is known as the "netname" or network name of the caller. The server is not allowed to interpret the contents of the caller's name in any other way except to identify the caller. Thus, netnames should be unique for every caller in the internet.

It is up to each operating system's implementation of DES authentication to generate netnames for its users that insure this uniqueness when they call upon remote servers. Operating systems already know how to distinguish users local to their systems. It is usually a simple matter to extend this mechanism to the network. For example, a user at Sun with a user ID of 515 might be assigned the following netname: "unix.515@sun.com." This netname contains three items that serve to insure it is unique. Going backwards, there is only one naming domain called "sun.com" in the Internet. Within this domain, there is only one user with user ID 515. However, there may be another user on another operating system, for example VMS, within the same naming domain that, by coincidence, happens to have the same user ID. To insure that these two users can be distinguished we add the operating system name. So one user is "unix.515@sun.com" and the other is "vms.515@sun.com".

The first field is actually a naming method rather than an operating system name. It just happens that today there is almost a one-to-one correspondence between naming methods and operating systems. If the world could agree on a naming standard, the first field could be the name of that standard, instead of an operating system name.

### DES Authentication Verifiers

Unlike UNIX authentication, DES authentication does have a verifier so the server can validate the client's credential (and vice-versa). The contents of this verifier is primarily an encrypted timestamp. The server can decrypt this timestamp, and if it is close to what the real time is, then the client must have encrypted it correctly. The only way the client could encrypt it correctly is to know the "conversation key" of the RPC session. And if the client knows the conversation key, then it must be the real client.

The conversation key is a DES [5] key which the client generates and notifies the server of in its first RPC call. The conversation key is encrypted using a public key scheme in this first transaction. The particular public key scheme used in DES authentication is Diffie-Hellman [3] with 192-bit keys. The details of this encryption method are described later.

The client and the server need the same notion of the current time in order for all of this to work. If network time synchronization cannot be guaranteed, then client can synchronize with the server before beginning the conversation, perhaps by consulting the Internet Time Server (TIME[4]).

The way a server determines if a client timestamp is valid is somewhat complicated. For any other transaction but the first, the server just checks for two things:

1. The timestamp is greater than the one previously seen from the same client.

2. The timestamp has not expired.

A timestamp is expired if the server's time is later than the sum of the client's timestamp plus what is known as the client's "window." The "window" is a number the client passes (encrypted) to the server in its first transaction. You can think of it as a lifetime for the credential.

This explains everything but the first transaction. In the first transaction, the server checks only that the timestamp has not expired. If this was all that was done though, then it would be quite easy for the client to send random data in place of the timestamp with a fairly good chance of succeeding. As an added check, the client sends an encrypted item in the first transaction known as the "window verifier" which must be equal to the window minus 1, or the server will reject the credential.

The client too must check the verifier returned from the server to be sure it is legitimate. The server sends back to the client the encrypted timestamp it received from the client, minus one second. If the client gets anything different than this, it will reject it.

### Nicknames and Clock Synchronization

After the first transaction, the server's DES authentication subsystem returns in its verifier to the client an integer "nickname" which the client may use in its further transactions instead of passing its netname, encrypted DES key and window every time. The nickname is most likely an index into a table on the server which stores for each client its netname, decrypted DES key and window.

Although they originally were synchronized, the client's and server's clocks can get out of synchronization again. When this happens the client RPC subsystem most likely will get back RPC_AUTHERROR at which point it should resynchronize.

A client may still get the RPC_AUTHERROR error even though it is synchronized with the server. The reason is that the server's nickname table is a limited size, and it may flush entries whenever it wants. A client should resend its original credential in this case and the server will give it a new nickname. If a server crashes, the entire nickname table gets flushed, and all clients will have to resend their original credentials.

### DES Authentication Protocol (in XDR language)

```
/*
 * There are two kinds of credentials: one in which the client
 * uses its full network name, and one in which it uses its
 * "nickname"(unsigned integer) given to it by the server. The
 * client must use its full name in its first transaction with
 * the server, in which the server will return to the client its
 * nickname. The client may use its nickname in all further
 * transactions with the server. There is no requirement to use
 * the nickname, but it is wise to use it for performance
 * reasons.
 */
enum authdes_namekind {
    ADN_FULLNAME = 0,
    ADN_NICKNAME = 1
};

/* A 64-bit block of encrypted DES data */
typedef opaque des_block[8];

/* Maximum length of a network user's name */
const MAXNETNAMELEN = 255;

/*
 * A fullname contains the network name of the client, an
 * encrypted conversation key and the window.  The window is
 * actually a lifetime for the credential.  If the time
 * indicated in the verifier timestamp plus the window has past,
 * then the server should expire the request and not grant it.
 * To insure that requests are not replayed, the server should
 * insist that timestamps are greater than the previous one
 * seen, unless it is the first transaction.  In the first
 * transaction, the server checks instead that the window
 * verifier is one less than the window.
 */
struct authdes_fullname {
   string name<MAXNETNAMELEN>;  /* name of client */
   des_block key;          /* PK encrypted conversation key */
    unsigned int window;          /* encrypted window */
};

/*
 * A credential is either a fullname or a nickname
 */
union authdes_cred switch (authdes_namekind adc_namekind) {
    case ADN_FULLNAME:
        authdes_fullname adc_fullname;
    case ADN_NICKNAME:
        unsigned int adc_nickname;
};
```

```
/*
* A timestamp encodes the time since midnight,
* January 1, 1970.
*/
struct timestamp {
    unsigned int seconds;    /* seconds */
    unsigned int useconds;   /* and microseconds */ ;
};

/*
* Verifier: client variety
* The window verifier is only used in the first transaction.  In
* conjunction with a fullname credential, these items are
* packed into the following structure before being encrypted:
*
* struct {
*     adv_timestamp;            -- one DES block
*     adc_fullname.window;      -- one half DES block
*     adv_winverf;              -- one half DES block
* }
* This structure is encrypted using CBC mode encryption with
* an input vector of zero.  All other encryptions of
* timestamps use ECB mode encryption.
*/
struct authdes_verf_clnt {
    timestamp adv_timestamp;    /* encrypted timestamp      */
    unsigned int adv_winverf;   /* encrypted window verifier */
};

/*
* Verifier: server variety
* The server returns (encrypted) the same timestamp the client
* gave it minus one second.  It also tells the client its
* nickname to be used in future transactions (unencrypted).
*/
struct authdes_verf_svr {
    timestamp adv_timeverf;     /* encrypted verifier      */
    unsigned int adv_nickname;  /* new nickname for client */
};
```

### Diffie-Hellman Encryption

In this scheme, there are two constants, BASE and MODULUS. The particular values Sun has chosen for these for the DES authentication protocol are:

```
const BASE = 3;
const MODULUS=
  "d4a0ba0250b6fd2ec626e7efd6df76c716e22d0944b88b"; /* hex */
```

The way this scheme works is best explained by an example. Suppose there are two people "A" and "B" who want to send encrypted messages to each other. So, A and B both generate "secret" keys at random which they do not reveal to anyone. Let these keys be represented as SK(A) and SK(B). They also publish in a public directory their "public" keys. These keys are computed as follows:

```
PK(A) = ( BASE ** SK(A) ) mod MODULUS
PK(B) = ( BASE ** SK(B) ) mod MODULUS
```

The "**" notation is used here to represent exponentiation. Now, both A and B can arrive at the "common" key between them, represented here as CK(A, B), without revealing their secret keys.

A computes:

```
CK(A, B) = (PK(B) ** SK(A)) mod MODULUS
```

while B computes:

```
CK(A, B) = (PK(A) ** SK(B)) mod MODULUS
```

```
These two can be shown to be equivalent:
```

```
(PK(B) ** SK(A)) mod MODULUS = (PK(A) ** SK(B)) mod MODULUS
```

We drop the "mod MODULUS" parts and assume modulo arithmetic to simplify things:

```
PK(B) ** SK(A) = PK(A) ** SK(B)
```

Then, replace PK(B) by what B computed earlier and likewise for PK(A).

```
((BASE ** SK(B)) ** SK(A)) = ((BASE ** SK(A)) ** SK(B))
```

which leads to:

```
BASE ** (SK(A) * SK(B)) = BASE ** (SK(A) * SK(B))
```

This common key CK(A, B) is not used to encrypt the domesticates used in the protocol. Rather, it is used only to encrypt a conversation key which is then used to encrypt the timestamps. The reason for doing this is to use the common key as little as possible, for fear that it could be broken. Breaking the conversation key is a far less serious offense, since conversations are relatively short-lived.

The conversation key is encrypted using 56-bit DES keys, yet the common key is 192 bits. To reduce the number of bits, 56 bits are selected from the common key as follows. The middle-most 8-bytes are selected from the common key, and then parity is added to the lower order bit of each byte, producing a 56-bit key with 8 bits of parity.

## 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). Sun uses this RM/TCP/IP transport for passing RPC messages on TCP streams. One RPC message fits into one RM record.

A record is composed of one or more record fragments. A record fragment is a 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 that this record specification is NOT in XDR standard form!)

## The RPC Language

Just as there was a need to describe the XDR data-types in a formal language, there is also need to describe the procedures that operate on these XDR data-types in a formal language as well. We use the RPC Language for this purpose. It is an extension to the XDR language. The following example is used to describe the essence of the language.

### An Example Service Described in the RPC Language

Here is an example of the specification of a simple ping program.

```
/*
* Simple ping program
*/
program PING_PROG {
    /* Latest and greatest version */
    version PING_VERS_PINGBACK {
    void
    PINGPROC_NULL(void) = 0;
    /*
    * Ping the caller, return the round-trip time
    * (in microseconds). Returns -1 if the operation
    * timed out.
    */
    int
    PINGPROC_PINGBACK(void) = 1;
} = 2;

/*
* Original version
*/
version PING_VERS_ORIG {
    void
    PINGPROC_NULL(void) = 0;
    } = 1;
} = 1;

const PING_VERS = 2;      /* latest version */
```

The first version described is PING_VERS_PINGBACK() with two procedures, PINGPROC_NULL() and PINGPROC_PINGBACK(). PINGPROC_NULL() takes no arguments and returns no results, but it is useful for computing round-trip times from the client to the server and back again. By convention, procedure 0 of any RPC protocol should have the same semantics, and never require any kind of authentication. The second procedure is used for the client to have the server do a reverse ping operation back to the client, and it returns the amount of time (in microseconds) that the operation used. The next version, PING_VERS_ORIG() is the original version of the protocol and it does not contain PINGPROC_PINGBACK() procedure. It is useful for compatibility with old client programs, and as this program matures it may be dropped from the protocol entirely.

## The RPC Language Specification

The RPC language is identical to the XDR language, except for the added definition of a program-def described below.

```
program-def:
    "program" identifier "{"
        version-def
        version-def *
    "}" "=" constant ";"

version-def:
  "version" identifier "{"
      procedure-def
      procedure-def *
"}" "=" constant ";"

procedure-def:
  type-specifier identifier "(" type-specifier ")"
  "=" constant ";"
```

## Syntax Notes

- The following keywords are added and cannot be used as identifiers: "program" and "version."
- A version name cannot occur more than once within the scope of a program definition. Nor can a version number occur more than once within the scope of a program definition.
- A procedure name cannot occur more than once within the scope of a version definition. Nor can a procedure number occur more than once within the scope of version definition.
- Program identifiers are in the same name space as constant and type identifiers.
- Only unsigned constants can be assigned to programs, versions and procedures.

## Port Mapper Program Protocol

The port mapper program maps RPC program and version numbers to transport-specific 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 also aids in broadcast RPC. A given RPC program will usually have different port number bindings on different machines, so there is no way to directly broadcast to all of these programs. The port mapper, however, does have a fixed port number. So, to broadcast to a given program, the client actually sends its message to the port mapper located at the broadcast address. Each port mapper that picks up the broadcast then calls the local service specified by the client. When the port mapper gets the reply from the local service, it sends the reply on back to the client.

### Port Mapper Protocol Specification (in RPC Language)

```
const PMAP_PORT = 111;      /* portmapper port number */

/*
* A mapping of (program, version, protocol) to port number
*/
struct mapping {
    unsigned int prog;
    unsigned int vers;
    unsigned int prot;
    unsigned int port;
};

/*
* Supported values for the "prot" field
*/
const IPPROTO_TCP = 6;      /* protocol number for TCP/IP */
const IPPROTO_UDP = 17;     /* protocol number for UDP/IP */

/*
* A list of mappings
*/
struct *pmaplist {
    mapping map;
    pmaplist next;
};

/*
* Arguments to callit
*/
struct call_args {
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque args<>;
};
```

```
/*
* Results of callit
*/
struct call_result {
    unsigned int port;
    opaque res<>;
};

/*
* Port mapper procedures
*/
program PMAP_PROG {
    version PMAP_VERS {
        void  PMAPPROC_NULL(void)          = 0;
        bool PMAPPROC_SET(mapping)         = 1;
        bool PMAPPROC_UNSET(mapping)       = 2;
        unsigned int PMAPPROC_GETPORT(mapping)    = 3;
        pmaplist PMAPPROC_DUMP(void)           = 4;
        call_result PMAPPROC_CALLIT(call_args)   = 5;
    } = 2;
} = 100000;
```

## Port Mapper Operation

The port mapper program currently supports two protocols (UDP/IP and TCP/IP). The port mapper is contacted by talking to it on assigned port number 111 (SUNRPC [8]) on either of these protocols. The following is a description of each of the port mapper procedures:

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

PMAPPROC_SET:
When a program first becomes available on a machine, it registers itself with the 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 a boolean response whose value is TRUE if the procedure successfully established the mapping and FALSE otherwise. The procedure refuses to establish a mapping if one already exists for the tuple "(prog, vers, prot)".

PMAPPROC_UNSET:
When a program becomes unavailable, it should unregister itself with the port mapper program on the same machine. The parameters and results have meanings identical to those of PMAPPROC_SET The protocol and port number fields of the argument are ignored.

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

PMAPPROC_DUMP:
This procedure enumerates all entries in the port mapper's database. The procedure takes no parameters and returns a list of program, version, protocol, and port values.

PMAPPROC_CALLIT:
This procedure allows a caller to call another remote procedure on the same machine without knowing the remote procedure's port number. It is intended for supporting broadcasts to arbitrary remote programs via the well-known port mapper's port. The parameters "prog," "vers," "proc," and the bytes of "args" are the program number, version number, procedure number, and parameters of the remote procedure.

**Note**

**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 using UDP/IP only.**

**The procedure returns the remote program's port number, and the bytes of results are the results of the remote procedure.**

## References

[1] Birrell, Andrew D. & Nelson, Bruce Jay; "Implementing Remote Procedure Calls"; XEROX CSL-83-7, October 1983.

[2] Cheriton, D.; "VMTP: Versatile Message Transaction Protocol", Preliminary Version 0.3; Stanford University, January 1987.

[3] Diffie & Hellman; "New Directions in Cryptography"; IEEE Transactions on Information Theory IT-22, November 1976.

[4] Harrenstien, K.; "Time Server", RFC 738; Information Sciences Institute, October 1977.

[5] National Bureau of Standards; "Data Encryption Standard"; Federal Information Processing Standards Publication 46, January 1977.

[6] Postel, J.; "Transmission Control Protocol - DARPA Internet Program Protocol Specification", RFC 793; Information Sciences Institute, September 1981.

[7] Postel, J.; "User Datagram Protocol", RFC 768; Information Sciences Institute, August 1980.

[8] Reynolds, J. & Postel, J.; "Assigned Numbers", RFC 923; Information Sciences Institute, October 1984.

# Remote Procedure Call: Programming Guide

This document assumes a working knowledge of network theory. It is intended for programmers who wish to write network applications using remote procedure calls, and who want to understand the RPC mechanisms usually hidden by the rpcgen(1) protocol compiler. rpcgen is described in detail in *rpcgen Programming Guide.*

**Note**

**Before attempting to write a network application or convert an existing non-network application to run over the network, you will want to understand the material in this chapter. However, for most applications, you can circumvent the need to cope with the details presented here by using rpcgen. *Generating XDR Routines* section of *rpcgen Programming Guide* contains the complete source for a working RPC service—a remote directory listing service which uses rpcgen to generate XDR routines as well as client and server stubs.**

What are remote procedure calls? Remote procedure calls are the high-level communications paradigm used in the UNIX operating system. RPC presumes the existence of low-level networking mechanisms (such as TCP/IP and UDP/IP), and upon them it implements a logical client to serve a communications system designed specifically for the support of network applications. With RPC, the client makes a procedure call to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, sends back the reply, and the procedure call returns to the client.

## Layers of RPC

The RPC interface can be seen as being divided into three layers.[1]

### The Highest Layer

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

For example, at the highest level, users can simply make calls to rnusers(), a C routine that returns the number of users on a remote machine. The users are not explicitly aware of using RPC — they simply call a procedure, just as they would call malloc(), for instance.

### The Middle Layer

The middle layer is really "RPC proper." Here, the users don't need to consider details about sockets, the UNIX system, or other low-level implementation mechanisms. They simply make remote procedure calls to routines on other machines. The selling point here is simplicity. The middle-layer routines are used for most applications.

RPC calls are made with the system routines registerrpc(), callrpc(), and svc_run(). The first two of these are the most fundamental: registerrpc() obtains a unique system-wide procedure-identification number, and callrpc() actually executes a remote procedure call. At the middle level, a call to rnusers() is implemented by way of these two routines.

The middle layer is, unfortunately, rarely used in serious programming due to its inflexibility (simplicity). It does not allow timeout specifications or the choice of transport. It allows no UNIX process control or flexibility in case of errors. It doesn't support multiple kinds of call authentication. The programmer rarely needs all these kinds of control, but one or two of them is often necessary.

### The Lowest Layer

The lowest layer allows details to be controlled by the programmer, and for that reason it is often necessary. Programs written at this level are also most efficient, but this is rarely a real issue — since RPC clients and servers rarely generate heavy network loads.

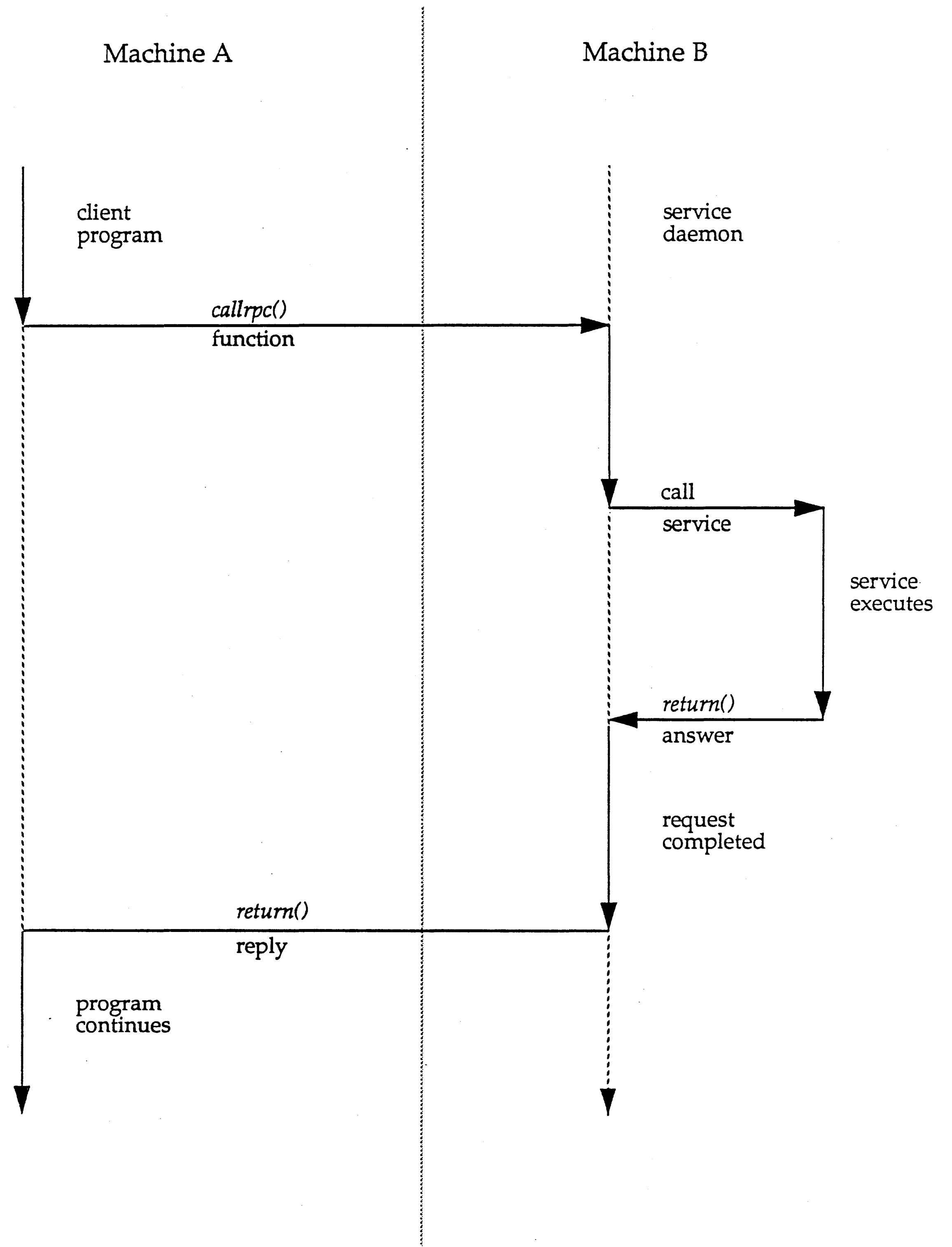
Although this document only discusses the interface to C, remote procedure calls can be made from any language. Even though this document discusses RPC when it is used to communicate between processes on different machines, it works just as well for communication between different processes on the same machine.

1. For a complete specification of the routines in the remote procedure call library, see the rpc(3N) manual pages.

## The RPC Paradigm

Below is a diagram of the RPC paradigm.

**Figure 1-1**
Network Communication with RPC



## Higher Layers of RPC

### Highest Layer

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

```
#include <stdio.h>

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

	if (argc != 2) {
		fprintf(stderr, "usage: rnusers hostname\n");
		exit(1);
	}

	if ((num = rnusers(argv[1])) < 0) {
		fprintf(stderr, "error: rnusers\n");
		exit(-1);
	}

	printf("%d users on %s\n", num, argv[1]);
	exit(0);
}
```

RPC library routines, such as rnusers(), are in the RPC services library librpcsvc.a. Thus, you compile the program listed above by entering

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

rnusers(), like the other RPC library routines, is documented in section 3R of the *System Interface Manual for the Sun Workstation*, the same section which documents the standard Sun RPC services. See the intro(3R) manual page for an explanation of the documentation strategy for these services and their RPC protocols.

Below are some of the RPC service library routines available to the C programmer.

| Routine | Description |
|---|---|
| rnusers() | Return number of users on remote machine |
| rusers() | Return information about users on remote machine |
| havedisk() | Determine if remote machine has disk |
| rstats() | Get performance data from remote kernel |
| rwall() | Write to specified remote machines |
| yppasswd() | Update user password in Yellow Pages |

Other RPC services — for example ether(), mount(), rquota(), and spray() are not available to the C programmer as library routines. They do, however, have RPC program numbers so they can be invoked with callrpc(), which will be discussed in the next section. Most of them also have compilable rpcgen(1) protocol description files. (The rpcgen protocol compiler radically simplifies the process of developing network applications. See the *rpcgen Programming Guide* for detailed information about rpcgen and rpcgen protocol description files).

## Intermediate Layer

The simplest interface, which explicitly makes RPC calls, uses the functions callrpc() and registerrpc(). Using this method, the number of remote users can be determined as follows:

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

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

    if (argc != 2) {
        fprintf(stderr, "usage: nusers hostname\n");
        exit(-1);
    }

    if (stat = callrpc(argv[1],   RUSERSPROG, RUSERSVERS,
        RUSERSPROC_NUM,   xdr_void, 0, xdr_u_long,
        &nusers) != 0) {
        clnt_perrno(stat);
        exit(1);
    }

    printf("%d users on %s\n", nusers, argv[1]);
    exit(0);
}
```

Each RPC procedure is uniquely defined by a program number, version number, and procedure number. The program number specifies a group of related remote procedures, each of which has a different procedure number. Each program also has a version number; so, when a minor change is made to a remote service (adding a new procedure, for example), a new program number doesn't have to be assigned. When you want to call a procedure to find the number of remote users, you look up the appropriate program, version, and procedure numbers in a manual, just as you look up the name of a memory allocator when you want to allocate memory.

The simplest way of making remote procedure calls is with the RPC library routine callrpc(). It has eight parameters. The first is the name of the remote server machine. The next three parameters are the program, version, and procedure numbers—together they identify the procedure to be called. The fifth and sixth parameters are an XDR filter and an argument to be encoded and passed to the remote procedure. The final two parameters are a filter for decoding the results returned by the remote procedure and a pointer to the place where the procedure's results are to be stored. Multiple arguments and results are handled by embedding them in structures. If callrpc() completes successfully, it returns zero; otherwise, it returns a nonzero value. The return codes (of type cast into an integer) can be found in the file <rpc/clnt.h>.

Since data types may be represented differently on different machines, callrpc() needs both the type of the RPC argument, as well as a pointer to the argument itself (and similarly for the result). For RUSERSPROC_NUM, the return value is an unsigned long, so callrpc() has xdr_u_long() as its first return parameter. This indicates that the result is of type unsigned long, and &nusers is 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.

If callrpc() gets no answer after trying several times to deliver a message, it returns an error code. The delivery mechanism is UDP, which stands for User Datagram Protocol. Methods for adjusting the number of retries or for using a different protocol require you to use the lower layer of the RPC library, discussed later in this document. The remote server procedure corresponding to the above might look like this:

```
char *
nuser(indata)

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

It takes one argument, which is a pointer to the input of the remote procedure call (ignored in our example), and it returns a pointer to the result. In the current version of C, character pointers are 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 the following:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.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 registers a C procedure as corresponding to a given RPC procedure number. The first three parameters, RUSERPROG, RUSERSVERS, and RUSERSPROC_NUM are the program, version, and procedure numbers, respectively, of the remote procedure to be registered. nuser() is the name of the local procedure that implements the remote procedure, and xdr_void() and xdr_u_long() are the XDR filters for the remote procedure's arguments and results, respectively. (Multiple arguments or multiple results are passed as structures.)

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

**Caution**

**The UDP transport mechanism can only deal with arguments and results less than 8K bytes in length.**

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

## Assigning Program Numbers

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

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

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

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

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

Please include a compilable rpcgen ".x" file describing your protocol. You will be given a unique program number in return.

The RPC program numbers and protocol specifications of standard Sun RPC services can be found in the include files in /usr/include/rpcsvc. These services, however, constitute only a small subset of those which have been registered. On the following page is a complete list of registered programs, at the time of this printing.

| RPC Number | Program | Description |
|---|---|---|
| 100000 | PMAPPROG | portmapper |
| 100001 | RSTATPROG | remote stats |
| 100002 | RUSERSPROG | remote users |
| 100003 | NFSPROG | NFS |
| 100004 | YPPROG | Yellow Pages |
| 100005 | MOUNTPROG | mount daemon |
| 100006 | DBXPROG | remote dbx |
| 100007 | YPBINDPROG | YP binder |
| 100008 | WALLPROG | shutdown msg |
| 100009 | YPPASSWDPROG | yppasswd server |
| 100010 | ETHERSTATPROG | ether stats |
| 100011 | RQUOTAPROG | disk quotas |
| 100012 | SPRAYPROG | spray packets |
| 100013 | IBM3270PROG | 3270 mapper |
| 100014 | IBMRJEPROG | RJE mapper |
| 100015 | SELNSVCPROG | selection service |
| 100016 | RDATABASEPROG | remote database access |
| 100017 | REXECPROG | remote execution |
| 100018 | ALICEPROG | Alice Office Automation |
| 100019 | SCHEDPROG | scheduling service |
| 100020 | LOCKPROG | local lock manager |
| 100021 | NETLOCKPROG | network lock manager |
| 100022 | X25PROG | x.25 inr protocol |
| 100023 | STATMON1PROG | status monitor 1 |
| 100024 | STATMON2PROG | status monitor 2 |
| 100025 | SELNLIBPROG | selection library |
| 100026 | BOOTPARAMPROG | boot parameters service |
| 100027 | MAZEPROG | mazewars game |
| 100028 | YPUPDATEPROG | YP update |
| 100029 | KEYSERVEPROG | key server |
| 100030 | SECURECMDPROG | secure login |
| 100031 | NETFWDIPROG | NFS net forwarder init |
| 100032 | NETFWDTPROG | NFS net forwarder trans |
| 100033 | SUNLINKMAP_PROG | sunlink MAP |
| 100034 | NETMONPROG | network monitor |
| 100035 | DBASEPROG | lightweight database |
| 100036 | PWDAUTHPROG | password authorization |
| 100037 | TFSPROG | translucent file svc |
| 100038 | NSEPROG | nse server |
| 100039 | NSE_ACTIVATE_PROG | nse activate daemon |
| 150001 | PCNFSDPROG | pc passwd authorization |
| 200000 | PYRAMIDLOCKINGPROG | Pyramid-locking |
| 200001 | PYRAMIDSYS5 | Pyramid-sys5 |
| 200002 | CADDS_IMAGE | CV cadds_image |
| 300001 | ADT_RFLOCKPROG | ADT file locking |

## 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_wrapstring()
xdr_char()         xdr_u_char()
```

**Note**

**The routine xdr_string() exists, but cannot be used with callrpc() and registerrpc(), which only pass two parameters to their XDR routines. xdr_wrapstring() has only two parameters, and is thus OK. It calls xdr_string().**

As an example of a user-defined type routine, if you want to send the structure

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

you can call callrpc() through

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

where xdr_simple() is

```
#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; otherwise, it returns 0.

The existing XDR routines handle both encoding and decoding in the same routine, and all SDR routines are expected to do the same. A complete description of XDR is in section *XDR Protocol Specification*. Only few implementation examples are given here.

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

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

To send a variable array of integers, you might package them 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; {
       return (xdr_array(xdrsp, &arrp->data, &arrp->arrlnth,
            MAXLEN, sizeof(int), xdr_int));
    }
```

The xdr_array routine above takes the following 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,
- an XDR routine for handling each array element

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

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

XDR always converts quantities to 4-byte multiples when serializing. Thus, if either of the examples mentioned previously involved characters instead of integers, each character would occupy 32 bits. That is the reason for the XDR routine xdr_bytes(), which is like xdr_array() except that it packs characters. xdr_bytes() has four parameters, similar to the first four parameters of xdr_array(). For null-terminated strings, there is also the xdr_string() routine, which is the same as xdr_bytes() without the length parameter. When serializing, it gets the string length from strlen(); and, when deserializing, it creates a null-terminated string.

Here is a final example that calls the previously written xdr_simple() routine, 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;
{

    if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
        return (0);

    if (!xdr_reference(xdrsp, &finalp->simplep,
        sizeof(struct simple), xdr_simple);
        return (0);

    return (1);
}
```

**Note**

**We could as easily call xdr_simple() here instead of xdr_reference().**

## Lowest Layer of RPC

In the examples given so far, RPC takes care of many details automatically for you. In this section, we'll show you how you can change the defaults by using lower layers of the RPC library. It is assumed that you are familiar with sockets and the system calls for dealing with them.

There are several occasions when you may need to use lower layers of RPC. A few of them are listed below.

- You may need to use TCP since the higher layer uses UDP, which restricts RPC calls to 8K bytes of data. Using TCP permits calls to send long streams of data. For an example, see section *TCP*.

- You may want to allocate and free memory while serializing or deserializing with XDR routines. There is no call at the higher level to let you free memory explicitly. For more explanation, see section *Memory Allocation with XDR*.

- You may need to perform authentication on either the client or server side, by supplying credentials or verifying them. See the explanation in section *Authentication*.

## More on the Server Side

The server for the nusers() program shown below does the same thing as the one on page 7using registerrpc(), but is written using a lower layer of the RPC package.

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

main()
{
    SVCXPRT *transp;
    int nuser();

    transp = svcudp_create(RPC_ANYSOCK);

    if (transp == NULL){
        fprintf(stderr, "can't create an RPC server\n");
           exit(1);
    }

    pmap_unset(RUSERSPROG, RUSERSVERS);

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

    svc_run();  /* Never returns */
    fprintf(stderr, "should never reach this point\n");
}
```

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

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

First, the server gets a transport handle, which is used for receiving and replying to RPC messages. registerrpc() uses svcudp_create() to get a UDP handle. If you require a more reliable protocol, call svctcp_create() instead. If the argument to svcudp_create() is RPC_ANYSOCK, the RPC library creates a socket on which to receive and reply to RPC calls. Otherwise, svcudp_create() expects its argument to be a valid socket number. If you specify your own socket, it can be bound or unbound. If it is bound to a port by the user, the port numbers of svcudp_create() and clnttcp_create() (the low-level client routine) must match.

If the user specifies the RPC_ANYSOCK argument, the RPC library routines will open sockets. Otherwise, they will expect the user to do so. The routines svcudp_create() and clntudp_create() will cause the RPC library routines to call bind() to bind their socket if it is not bound already.

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

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

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

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

1. Procedure NULLPROC (currently zero) returns without results. This can be used as a simple test for detecting if a remote program is running.

2. 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. As an example of how a server handles an RPC program that receives data, we can add a procedure RUSERSPROC_BOOL, which has an argument nusers, and returns TRUE or FALSE depending on whether there are nusers logged on. It 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, "can't reply to RPC call\n");
        return (1);
    }

    return;
}
```

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

## Memory Allocation with XDR

XDR routines can allocate memory via malloc on your behalf, if you wish. 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));
}
```

If space has already been allocated in chararr[], it can be called from a server like this:

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

If you want XDR to do the allocation, you must rewrite this routine in the following way:

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

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

Then the RPC call will look something like this:

```
char *arrptr;
arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);

/*
 * Use the result here
*/

svc_freeargs(transp, xdr_chararr2, &arrptr);
```

Note that after being used, the character array can be freed with svc_freeargs(). svc_freeargs() will not attempt to free any memory if the pointer to it has a value of NULL. For example, in the routine xdr_finalexample() given earlier, if finalp->string is NULL, then it will not be freed. The same is true for finalp->simplep.

To summarize, each XDR routine is responsible for serializing, deserializing, and freeing memory. When an XDR routine is called from callrpc(), the serializer is used. When called from svc_getargs(), the deserializer is used. And, when called from svc_-freeargs(), the memory deallocator is used. When building simple examples like those in this section, a user doesn't have to worry about the three modes. See *External Data Representation: Sun Technical Notes* for examples of more sophisticated XDR routines that determine their own mode and adjust their behavior accordingly.

## The Calling Side

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

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

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

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

    if (argc != 2) {
        fprintf(stderr, "usage: nusers hostname\n");
        exit(-1);
    }

    if ((hp = gethostbyname(argv[1])) == NULL) {
       fprintf(stderr, "can't get addr for %s\n",argv[1]);
        exit(-1);
    }

    pertry_timeout.tv_sec = 3;
    pertry_timeout.tv_usec = 0;
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
        hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port =  0;

    if ((client = clntudp_create(&server_addr, RUSERSPROG,
        RUSERSVERS, pertry_timeout, &sock)) == NULL) {
        clnt_pcreateerror("clntudp_create");
        exit(-1);
    }
```

```
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_void,
        0, xdr_u_long, &nusers, total_timeout);

    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "RPC");
        exit(-1);
    }

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

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

The CLIENT pointer is encoded with the transport mechanism. callrpc() uses UDP, thus it calls clntudp_create() to get a CLIENT pointer. To get TCP (Transmission Control Protocol), you would use clnttcp_create()

The parameters to clntudp_create() are 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.

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

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

```
clnttcp_create(&server_addr, prognum, versnum, &sock,
   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()

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

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

## Other RPC Features

This section discusses some other aspects of RPC that are occasionally useful.

### Select on the Server Side

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

```
void
svc_run()
{
    fd_set readfds;
    int dtbsz = getdtablesize();

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

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

### Broadcast RPC

The portmapper is a daemon that converts RPC program numbers into DARPA protocol port numbers; see the portmap man page. You can't do broadcast RPC without the portmapper. Here are the main differences between broadcast RPC and normal RPC calls:

- Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answer from each responding machine).
- Broadcast RPC can only be supported by packet-oriented (connectionless) trans-

port protocols like UPD/IP.

- The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC never knows.

- All broadcast messages are sent to the portmap port. Thus, only services that register themselves with their portmapper are accessible via the broadcast RPC mechanism.

- Broadcast requests are limited in size to the MTU (Maximum Transfer Unit) of the local network. For Ethernet, the MTU is 1500 bytes.

### Broadcast RPC Synopsis

```
#include <RPC/pmap_clnt.h>
. . .
enum clnt_statclnt_stat;
. . .
clnt_stat = clnt_broadcast(prognum, versnum, procnum,
    inproc, in, outproc, out, eachresult)
u_long     prognum;          /* program number */
u_long     versnum;          /* version number */
u_long     procnum;          /* procedure number */
xdrproc_t  inproc;           /* XDR routine for args */
caddr_t    in;               /* pointer to args */
xdrproc_t  outproc;          /* XDR routine for results */
caddr_t    out;              /* pointer to results */
bool_t     (*eachresult)();  /* call with each result */
```

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

```
bool_t done;
. . .
    done = eachresult(resultsp, raddr)

caddr_t resultsp;
struct sockaddr_in *raddr; /* Addr of responding machine */
```

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

## 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

- Each RPC call in the pipeline requires no response from the server, and the server does not send a response message.
- The pipeline of calls is transported on a reliable byte stream transport such as TCP/IP.

Since the server does not respond to every call, the client can generate new calls in parallel with the server executing previous calls. 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.

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

A contrived example of batching follows. Assume a string rendering service (like a window system) has two similar calls: one renders a string and returns void results, while the other renders a string and remains silent. The service (using the TCP/IP transport) may look like

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

void windowdispatch();

main()
{
    SVCXPRT *transp;

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);

    if (transp == NULL){
        fprintf(stderr, "can't create an RPC server\n");
        exit(1);
    }

    pmap_unset(WINDOWPROG, WINDOWVERS);

    if (!svc_register(transp, WINDOWPROG, WINDOWVERS,
        windowdispatch, IPPROTO_TCP)) {
        fprintf(stderr, "can't register WINDOW service\n");
        exit(1);
    }

    svc_run();  /* Never returns */
    fprintf(stderr, "should never reach this point\n");
}
```

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

    switch (rqstp->rq_proc) {
        case NULLPROC:
            if (!svc_sendreply(transp, xdr_void, 0))
            fprintf(stderr, "can't reply to RPC call\n");
            return;
        case RENDERSTRING:
            if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                fprintf(stderr, "can't decode arguments\n");

                /*
                 *  Tell caller he screwed up
                 */
                svcerr_decode(transp);
                break;
            }

            /*
             *  Code here to render the string s
             */
            if (!svc_sendreply(transp, xdr_void, NULL))
              fprintf(stderr, "can't reply to RPC call\n");
            break;
        case RENDERSTRING_BATCHED:
            if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                fprintf(stderr, "can't decode arguments\n");

                /*
                 *  We are silent in the face of
                 *  protocol errors
                 */
                break;
            }

            /*
             *  Code here to render string s, but send
             *  no reply!
             */
            break;
        default:
            svcerr_noproc(transp);
            return;
    }

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

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:

- the result's XDR routine must be zero NULL
- 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 (EOF).

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

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

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

    if ((client = clnttcp_create(&server_addr,
        WINDOWPROG, WINDOWVERS, &sock, 0, 0)) == NULL) {
        perror("clnttcp_create");
        exit(-1);
    }

    total_timeout.tv_sec = 0;
    total_timeout.tv_usec = 0;

    while (scanf("%s", s) != EOF) {
        clnt_stat = clnt_call(client, RENDERSTRING_BATCHED,
            xdr_wrapstring, &s, NULL, NULL, total_timeout);

        if (clnt_stat != RPC_SUCCESS) {
            clnt_perror(client, "batched RPC");
            exit(-1);
        }
    }
```

```
        /* Now flush the pipeline */
        total_timeout.tv_sec = 20;
        clnt_stat = clnt_call(client, NULLPROC, xdr_void, NULL,
            xdr_void, NULL, total_timeout);

        if (clnt_stat != RPC_SUCCESS) {
            clnt_perror(client, "RPC");
            exit(-1);
        }

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

Since the server sends no message, the clients cannot be notified of any of the failures that may occur. Therefore, clients are on their own when it comes to handling errors.

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

| Configuration | Result |
|---|---|
| machine to itself, regular RPC | 50 seconds |
| machine to itself, batched RPC | 16 seconds |
| machine to another, regular RPC | 52 seconds |
| machine to another, batched RPC | 10 seconds |

Running fscanf() on /etc/termcap only requires six seconds. These timings show the advantage of protocols that allow for overlapped execution, although they are often difficult to design.

## Authentication

In the examples presented so far, the caller never identified itself to the server, and the server never required an ID from the caller. Clearly, some network services, such as a network filesystem, require stronger security than what has been presented so far.

In reality, every RPC call is authenticated by the RPC package on the server, and similarly, the RPC client package generates and sends authentication parameters. Just as different transports (TCP/IP or UDP/IP) can be used when creating RPC clients and servers, different forms of authentication can be associated with RPC clients; the default authentication type used as a default is type none

The authentication subsystem of the RPC package is open ended. That is, numerous types of authentication are easy to support.

### UNIX Authentication

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 defaults the associate authentication handle to be

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

The RPC client can choose to use UNIX style authentication by setting clnt->cl_auth after creating the RPC client handle:

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

This causes each RPC call associated with clnt to carry with it the following authentication credentials structure:

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

These fields are set by authunix_create_default() by invoking the appropriate system calls. Since the RPC user created this new style of authentication, the user is responsible for destroying it with:

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

This should be done in all cases, to conserve memory.

The Server Side

Service implementors have a harder time dealing with authentication issues since the RPC package passes the service dispatch routine a request that has an arbitrary authentication style associated with it. Consider the fields of a request handle passed to a service dispatch routine:

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

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

```
/*
 * Authentication info. Mostly opaque to the programmer.
 */
struct opaque_auth {
    enum_t  oa_flavor;     /* style of credentials */
    caddr_t oa_base;       /* address of more auth stuff */
    u_int   oa_length;     /* not to exceed MAX_AUTH_BYTES */
};
```

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

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. Remember that only unix style is currently supported, so (currently) rq_clntcred could be cast to a pointer to an authunix_parms structure. If rq_clntcred is NULL the service implementor may wish to inspect the other (opaque) fields of rq_cred in case the service knows about a new type of authentication that the RPC package does not know about.

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

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

    /*
     * we don't care about authentication for null proc
     */
    if (rqstp->rq_proc == NULLPROC) {
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can't reply to RPC call\n");
              return (1);
        }
        return;
    }

    /*
     * now get the uid
     */
    switch (rqstp->rq_cred.oa_flavor) {
        case AUTH_UNIX:
            unix_cred =
               (struct authunix_parms *)rqstp->rq_clntcred;
            uid = unix_cred->aup_uid;
            break;
        case AUTH_NULL:
        default:
            svcerr_weakauth(transp);
            return;
    }
```

```
    switch (rqstp->rq_proc) {
        case RUSERSPROC_NUM:

            /*
             * make sure caller is allowed to call this proc
             */
            if (uid == 16) {
                svcerr_systemerr(transp);
                return;
            }

            /*
             * Code here to compute the number of users
             * and assign it to the variable nusers
             */
          if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
              fprintf(stderr, "can't reply to RPC call\n");
               return (1);
            }

            return;
        default:
            svcerr_noproc(transp);
            return;
    }
}
```

A few things should be noted here. First, it is customary not to check the authentication parameters associated with the NULLPROC (procedure number zero). Second, if the authentication parameter's type is not suitable for your service, you should call svcerr_weakauth(). And finally, the service protocol itself should return status for access denied. In the case of our example, the protocol does not have such a status, so we call the service primitive svcerr_systemerr() instead.

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

## DES Authentication

UNIX authentication is quite easy to defeat. Instead of using authunix_create_default(), you can call authunix_create() and then modify the RPC authentication handle authunix_create() returns by furnishing the desired user ID and hostname they wish the server to think they have. DES authentication is thus recommended for people who want more security than UNIX authentication offers.

The details of the DES authentication protocol are complicated and are not explained here. See *Remote Procedure Calls: Protocol Specification* for the details.

In order for DES authentication to work, the keyserv(8c) daemon must be running on both the server and client machines. The users on these machines need public keys assigned by the network administrator in the publickey(5) database. They also need to have decrypted their secret keys using their login password. This automatically happens when you log in using login(1). You can also do this manually using keylogin(1). The section *Network Services* explains more about establishing secure networking.

### Client Side

If a client wishes to use DES authentication, it must set its authentication handle appropriately. Here is an example:

```
cl->cl_auth = authdes_create(servername, 60, &server_addr,
    NULL);
```

The first argument is the network name or "netname" of the owner of the server process. Typically, server processes are root processes, and their netname can be derived using the following call:

```
char servername[MAXNETNAMELEN];

host2netname(servername, rhostname, NULL);
```

Here, rhostname is the hostname of the machine the server process is running on. host2netname() furnishes servername to contain this root process's netname. If the server process was run by a regular user, one could use the call user2netname() instead. Here is an example for a server process with the same user ID as the client.

```
char servername[MAXNETNAMELEN];

user2netname(servername, getuid(), NULL);
```

The last argument to both user2netname() and host2netname() is the name of the naming domain where the server is located. The NULL used here means "use the local default domain name."

The second argument to authdes_create() is a lifetime for the credential. Here it is set to sixty seconds. This means that the credential will expire 60 seconds from when authdes_create() is called. If some mischievous user tries to reuse the credential, the server RPC subsystem will recognize that it has expired and not grant any requests. If the same mischievous user tries to reuse the credential within the sixty second lifetime, he will still be rejected because the server RPC subsystem remembers which credentials it has already seen in the near past, and will not grant requests to duplicates.

The third argument to authdes_create() is the address of the host with which to synchronize. In order for DES authentication to work, the server and client must agree upon the time. Here we pass the address of the server itself, so the client and server will both be using the server's time. A NULL argument indicates that further synchronization is not necessary. You should only do this if you are sure the client and server are already synchronized.

The final argument to authdes_create() is the address of a DES encryption key to use for encrypting timestamps and data. If this argument is NULL, as it is in this example, a random key will be chosen. The client may find out the encryption key being used by consulting the ah_key field of the authentication handle.

### Server Side

The server side is a lot simpler than the client side. Here is the previous example re-written to use AUTH_DES instead of AUTH_UNIX.

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

    /*
     * we don't care about authentication for null proc
     */
    if (rqstp->rq_proc == NULLPROC) {   /* same as before */
    }

    /*
     * now get the uid
     */
    switch (rqstp->rq_cred.oa_flavor) {
        case AUTH_DES:
            des_cred =
                (struct authdes_cred *) rqstp->rq_clntcred;

            if (! netname2user(des_cred->adc_fullname.name,
                &uid, &gid, &gidlen, gidlist)) {
                fprintf(stderr, "unknown user: %s\n",
                    des_cred->adc_fullname.name);
                svcerr_systemerr(transp);
                return;
            }
            break;
        case AUTH_NULL:
        default:
            svcerr_weakauth(transp);
            return;
    }
/*
 * The rest is the same as before
 */
```

**Note**

**The use of the routine netname2user() the inverse of user2netname() it takes a network ID and converts to a unix ID. netname2user() also supplies the group IDs which we don't use in this example, but which may be useful to other ConvexOS programs.**

## Using inetd(8C)

An RPC server can be started from inetd(8C). The only difference from the usual code is that the service creation routine should be called in the following form since inetd() passes a socket as file descriptor 0:

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

Also, svc_register() should be called as

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

with the final flag as 0, since the program will already be registered by inetd(). Remember, if you want to exit from the server process and return control to inetd(), you need to explicitly exit, since svc_run() never returns.

The format of entries in /etc/inetd.conf for RPC services is in one of the following two forms:

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

where `p_name` is the symbolic name of the program as it appears in RPC(5). `server` is the program implementing the server, and `program` and `version` are the program and version numbers of the service. For more information, see inetd.conf(5).

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

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

### Versions

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

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

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

Both versions can be handled by the same C procedure, as in this example.

```
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;
    unsigned short nusers2;

    switch (rqstp->rq_proc) {

        case NULLPROC:
            if (!svc_sendreply(transp, xdr_void, 0)) {
                fprintf(stderr,
                   "can't reply to RPC call\n");
                return (1);
            }
            return;

        case RUSERSPROC_NUM:
            /*
            * Code here to compute the number of users
             * and assign it to the variable nusers
             */
            nusers2 = nusers;

            switch (rqstp->rq_vers) {
                case RUSERSVERS_ORIG:
                    if (!svc_sendreply(transp,
                        xdr_u_long,&nusers)){
                        fprintf(stderr,
                            "can't reply to RPC call\n");
                     }
                     break;
                case RUSERSVERS_SHORT:
                    if (!svc_sendreply(transp,
                       xdr_u_short,&nusers2)){
                        fprintf(stderr,
                           "can't reply to RPC call\n");
                    }
                    break;
            }

          default:
              svcerr_noproc(transp);
              return;
    }
}
```

## TCP

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

```
/*
 * 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_rpc(xdrs, fp)
    XDR *xdrs;
    FILE *fp;

{
    unsigned long size;
    char buf[BUFSIZ], *p;

    if (xdrs->x_op == XDR_FREE)/* nothing to free */
        return 1;

    while (1) {
        if (xdrs->x_op == XDR_ENCODE) {
            if ((size = fread(buf,sizeof(char),
                BUFSIZ,fp)) == 0
                && ferror(fp)) {
                fprintf(stderr, "can't fread\n");
                return (1);
            }
        }
        p = buf;

        if (!xdr_bytes(xdrs, &p, &size, BUFSIZ))
            return 0;

        if (size == 0)
            return 1;

        if (xdrs->x_op == XDR_DECODE) {
            if (fwrite(buf, sizeof(char), size,   fp)
                != size) {
                fprintf(stderr, "can't fwrite\n");
                return (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 xdr_rpc();
    int err;

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

   if ((err = callrpctcp(argv[1], RPCPROG, RPCPROC, RPCVERS,
        xdr_rpc, stdin, xdr_void, 0) != 0)) {
        clnt_perrno(err);
        fprintf(stderr, "can't make RPC call\n");
        exit(1);
    }

    exit(0);
}

callrpctcp(host, prognum, procnum, versnum, inproc, in,
    outproc, out)
    char *host, *in, *out;
    xdrproc_t inproc, outproc;

{
    struct sockaddr_in server_addr;
    int socket = RPC_ANYSOCK;
    enum clnt_stat clnt_stat;
    struct hostent *hp;
    register CLIENT *client;
    struct timeval total_timeout;

    if ((hp = gethostbyname(host)) == NULL) {
        fprintf(stderr, "can't get addr for '%s'\n", host);
        return (-1);
    }

    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
        h->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port =  0;
```

```
    if ((client = clnttcp_create(&server_addr, prognum,
        versnum, &socket, BUFSIZ, BUFSIZ)) == NULL) {
        perror("rpctcp_create");
        return (-1);
    }

    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, procnum, inproc, in,
    outproc, out, total_timeout);
    clnt_destroy(client);
    return (int)clnt_stat;
}



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

main()
{
    register SVCXPRT *transp;
    int RPC_service(),
    xdr_rpc();

   if ((transp = svctcp_create(RPC_ANYSOCK, BUFSIZ, BUFSIZ))
        == NULL) {
        fprintf("svctcp_create: error\n");
        exit(1);
    }

    pmap_unset(RPCPROG, RPCVERS);

   if (!svc_register(transp, RPCPROG, RPCVERS, rpc_service,
        IPPROTO_TCP)) {
        fprintf(stderr, "svc_register: error\n");
        exit(1);
    }

    svc_run();  /* never returns */
    fprintf(stderr, "svc_run should never return\n");
}
```

```
rpc_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: rpc_service");
                return (1);
            }

            return;
        case rpcPROC_FP:

            if (!svc_getargs(transp, xdr_rpc, stdout)) {
                svcerr_decode(transp);
                return;
            }

            if (!svc_sendreply(transp, xdr_void, 0)) {
                fprintf(stderr, "can't reply\n");
                return;
            }

            return (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 that 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 break point, 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 break point has been reached.

In order to do an RPC callback, you need a program number on which to make the RPC call. Since this will be a dynamically generated program number, it should be in the transient range

```
0x40000000 - 0x5fffffff
```

The routine gettransient() returns a valid program number in the transient range, and registers it with the portmapper. It only talks to the portmapper running on the same machine as the gettransient() routine itself. The call to pmap_set() is a test and set operation in that it indivisibly tests whether a program number has already been registered, and if it has not, then reserves it. On return, the sockp argument 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 proto, vers, *sockp;
{
    static int prognum = 0x40000000;
    int s, len, socktype;
    struct sockaddr_in addr;


    switch(proto) {
        case IPPROTO_UDP:
            socktype = SOCK_DGRAM;
            break;
        case IPPROTO_TCP:
            socktype = SOCK_STREAM;
            break;
        default:
            fprintf(stderr, "unknown protocol type\n");
            return 0;
    }

    if (*sockp == RPC_ANYSOCK) {

        if ((s = socket(AF_INET, socktype, 0)) < 0) {
            perror("socket");
            return (0);
        }

        *sockp = s;
    }

    else s = *sockp;

    addr.sin_addr.s_addr = 0;
    addr.sin_family = AF_INET;
    addr.sin_port = 0;
    len = sizeof(addr);
```

```
	/*
	 * may be already bound, so don't check for error
	 */
	bind(s, &addr, len);

	if (getsockname(s, &addr, &len)< 0) {
		perror("getsockname");
		return (0);
	}

	while (!pmap_set(prognum++, vers, proto,
		ntohs(addr.sin_port)))
		continue; return (prognum-1);

}
```

**Note**

**The call to ntohs() is necessary to ensure that the port number in addr.sin_port, which is in network byte order, is passed in host byte order (as pmap_set() expects).** See the byteorder(3N) man page for more details on the conversion of network addresses from network to host byte order.

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. The client then waits 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. 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()
{
    int x, ans, s;
    SVCXPRT *xprt;

    gethostname(hostname, sizeof(hostname));
    s = RPC_ANYSOCK;
    x = gettransient(IPPROTO_UDP, 1, &s);
    fprintf(stderr, "client gets prognum %d\n", x);

    if ((xprt = svcudp_create(s)) == NULL) {
        fprintf(stderr, "rpc_server: svcudp_create\n");
        exit(1);
    }

    /* protocol is 0 - gettransient does registering  */
    (void)svc_register(xprt, x, 1, callback, 0);
    ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
    EXAMPLEPROC_CALLBACK, xdr_int, &x, xdr_void, 0);

    if ((enum clnt_stat) ans != RPC_SUCCESS) {
        fprintf(stderr, "call: ");
        clnt_perrno(ans);
        fprintf(stderr, "\n");
    }

    svc_run();
    fprintf(stderr, "Error: svc_run shouldn't return\n");
}
```

```
callback(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;

{
    switch (rqstp->rq_proc) {
        case 0:
            if (!svc_sendreply(transp, xdr_void, 0)) {
                fprintf(stderr, "err: exampleprog\n");
                return (1);
            }
            return (0);
        case 1:
            if (!svc_getargs(transp, xdr_void, 0)) {
                svcerr_decode(transp);
                return (1);
            }
            fprintf(stderr, "client got callback\n");

            if (!svc_sendreply(transp, xdr_void, 0)) {
                fprintf(stderr, "err: exampleprog");
                return (1);
            }
    }
}


/*
 * server
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>

char *getnewprog();
char hostname[256];
int docallback();
int pnum;     /* program number for callback routine */

main()
{
    gethostname(hostname, sizeof(hostname));
    registerrpc(EXAMPLEPROG, EXAMPLEVERS,
       EXAMPLEPROC_CALLBACK, getnewprog, xdr_int, xdr_void);
    fprintf(stderr, "server going into svc_run\n");
    signal(SIGALRM, docallback);
    alarm(10);
    svc_run();
    fprintf(stderr, "Error: svc_run shouldn't return\n");
}
```

```
char *
getnewprog(pnump)
    char *pnump;

{
    pnum = *(int *)pnump;
    return NULL;
}

docallback()
{
    int ans;

    ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0,
        xdr_void, 0);

    if (ans != 0) {
        fprintf(stderr, "server: ");
        clnt_perrno(ans);
        fprintf(stderr, "\n");
    }
}
```

# rpcgen Programming Guide

# 4

## The rpcgen Protocol Compiler

The details of programming applications that use Remote Procedure Calls can be overwhelming. Perhaps most daunting is the writing of the XDR routines necessary to convert procedure arguments and results into their network format and vice-versa.

Fortunately, rpcgen(1) exists to help programmers write RPC applications simply and directly. rpcgen does most of the dirty work, allowing programmers to debug the main features of their applications, instead of requiring them to spend most of their time debugging their network interface code.

rpcgen is a compiler. It accepts a remote program interface definition written in a language, called RPC Language, which is similar to C. It produces a C language output that includes stub versions of the following:

- client routines
- a server skeleton
- XDR filter routines for both parameters and results
- a header file that contains common definitions

The client stubs interface with the RPC library and effectively hide the network from their callers. The server stub similarly hides the network from the server procedures that are to be invoked by remote clients. rpcgen output files can be compiled and linked in the usual way. The developer writes server procedures—in any language that observes ConvexOS "C" calling conventions—and links them with the server skeleton produced by rpcgen to get an executable server program. To use a remote program, a programmer writes an ordinary main program that makes local procedure calls to the client stubs produced by rpcgen Linking this program with rpcgen stubs creates an executable program. (At present the main program must be written in C). rpcgen options can be used to suppress stub generation and to specify the transport to be used by the server stub.

Like all compilers, rpcgen reduces development time that would otherwise be spent coding and debugging low-level routines. All compilers, including rpcgen, do this at a small cost in efficiency and flexibility. However, many compilers allow escape hatches for programmers to mix low-level code with high-level code. rpcgen is no exception. In speed-critical applications, hand-written routines can be linked with the rpcgen output without any difficulty. Also, one may proceed by using rpcgen output as a starting point, and then rewriting it as necessary. (If you need a discussion of RPC programming without rpcgen see the *Remote Procedure Call Programming Guide.*)

## Converting Local Procedures into Remote Procedures

Assume an application that runs on a single machine, one which we want to convert to run over the network. Here we will demonstrate such a conversion by way of a simple example—a program that prints a message to the console:

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

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

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

    message = argv[1];

    if (!printmessage(message)) {
        fprintf(stderr, "%s: couldn't print your message\n",
            argv[0]);
        exit(1);
    }

    printf("Message Delivered!\n");
    exit(0);
}

/*
 * Print a message to the console.
 * Return a boolean indicating whether the message was
 * actually printed.
 */
printmessage(msg)
    char *msg;
{
    FILE *f;

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

And then, of course:

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

If printmessage() was turned into a remote procedure, then it could be called from anywhere in the network. Ideally, one would just like to stick a keyword like *remote* in front of a procedure to turn it into a remote procedure. Unfortunately, we have to live within the constraints of the C language, since it existed long before RPC did. Even without language support, it's not very difficult to make a procedure remote.

In general, it's necessary to determine the types for all procedure inputs and outputs. In this case, we have a procedure printmessage() which takes a string as input, and returns an integer as output. Knowing this, we can write a protocol specification in RPC language that describes the remote version of printmessage, shown below.

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

Remote procedures are part of remote programs, so we actually declared an entire remote program here which contains the single procedure PRINTMESSAGE. This procedure is declared to be in version 1 of the remote program. Null procedure (procedure 0) is unnecessary because rpcgen generates it automatically.

Notice that procedures and variables are declared with all capital letters. This is not required, but is a good convention to follow.

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

There are just two more things to write. First, there is the remote procedure itself. Here's the definition of a remote procedure to implement the PRINTMESSAGE procedure we declared earlier:

```
/*
 * msg_proc.c: implementation of the remote procedure
 * "printmessage"
 */
#include <stdio.h>
#include <RPC/RPC.h>      /* always needed */
#include "msg.h"          /* need this too: msg.h will be */
                          /* generated by rpcgen          */

/*
 * Remote verson of "printmessage"
 */
int *
printmessage_1(msg)
    char **msg;
{
    static int result;  /* must be static! */
    FILE *f;

    f = fopen("/dev/console", "w");

    if (f == NULL) {
        result = 0;
        return (&result);
    }

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

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

1. It takes a pointer to a string instead of a string itself. This is true of all remote procedures; they always take pointers to their arguments rather than the arguments themselves.

2. It returns a pointer to an integer instead of an integer itself. This is also generally true of remote procedures. They always return a pointer to their results.

3. It has an "_1" appended to its name. In general, all remote procedures called by rpcgen are named by the following rule: the name in the program definition (here PRINTMESSAGE) is converted to all lower-case letters. An underbar ("_") is appended to it, and finally the version number (here 1) is appended.

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

```
/*
 * rprintmsg.c: remote version of "printmsg.c"
 */
#include <stdio.h>
#include <RPC/RPC.h>        /* always needed */
#include "msg.h"            /* need this too: msg.h will be */
                            /* generated by rpcgen          */

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

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

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

    /*
     * Create client "handle" used for calling MESSAGEPROG
     * on the server designated on the command line.
     * We tell the RPC package to use the "tcp" protocol
     * when contacting the server.
    */
   cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");

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

```
        /*
         * Call the remote procedure "printmessage" on the
         * server
         */
        result = printmessage_1(&message, cl);

        if (result == NULL) {

        /*
         * An error occurred while calling the server.
         * Print error message and die.
         */
            clnt_perror(cl, server);
            exit(1);
        }

        /*
         * Okay, we successfully called the remote procedure.
         */
        if (*result == 0) {

        /*
         * Server was unable to print our message.
         * Print error message and die.
         */
            fprintf(stderr,
                "%s: %s couldn't print your message\n",
                argv[0], server);
            exit(1);
        }

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

There are two things to note here:

1. First, a client "handle" is created using the RPC library routine clnt_create. This client handle will be passed to the stub routines which call the remote procedure.

2. The remote procedure printmessage_1() is called exactly the same way as it is declared in msg_proc.c, except for the inserted client handle as the first argument.

Here's how to put all of the pieces together:

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

Two programs are compiled here: the client program rprintmsg and the server program msg_server. Before doing this, rpcgen is used to fill in the missing pieces.

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

- It created a header file called msg.h that contained #define for MESSAGEPROG, MESSAGEVERS, and PRINTMESSAGE for use in the other modules.

- It created client "stub" routines in the msg_clnt.c file. In this case there is only one, the printmessage_1() that was referred to from the printmsg client program. The name of the output file for client stub routines is always formed in this way: if the name of the input file is FOO.x the client stubs output file is called FOO_clnt.c

It created the server program which calls printmessage_1() in msg_proc.c. This server program is named msg_svc.c. The rule for naming the server output file is similar to the previous one: for an input file called FOO.x, the output server file is named FOO_svc.c.

Now, copy the server to a remote machine and run it. For this example, the machine is called "moon." Server processes are run in the background, because they never exit.

```
moon% msg_server &
```

Then, on our local machine, we can print a message on moon's console.

```
sun% printmsg moon "Hello, moon."
```

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

## Generating XDR Routines

The previous example only demonstrated the automatic generation of client and server RPC code. rpcgen may also be used to generate XDR routines, that is, the routines necessary to convert local data structures into network format and vice-versa. This example presents a complete RPC service—a remote directory listing service, which uses rpcgen not only to generate stub routines, but also to generate the XDR routines. Here is the protocol description file:

```
/*
 * dir.x: Remote directory listing protocol
 */

/* maximum length of a directory entry */
const MAXNAMELEN = 255;

typedef string nametype<MAXNAMELEN>; /* a directory entry */
typedef struct namenode *namelist;/* a link in the listing */

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

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

/*
 * The directory program definition
 */
program DIRPROG {
    version DIRVERS {
        readdir_res READDIR(nametype) = 1;
    } = 1;
} = 76;
```

### Note

**Types (like readdir_res in the example above) can be defined using the "struct,""union," and "enum" keywords, but these keywords should not be used in subsequent declarations of variables of those types. For example, if you define a union "foo," you should declare using only "foo" and not "union foo." In fact, rpcgen compiles RPC unions into C structures and it is an error to declare them using the "union" keyword.**

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

Here is the implementation of the READDIR procedure.

```
/*
 * dir_proc.c: remote readdir implementation
 */
#include <RPC/RPC.h>
#include <sys/dir.h>
#include "dir.h"

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

readdir_res *
readdir_1(dirname)
    nametype *dirname;
{
    DIR *dirp;
    struct direct *d;
    namelist nl;
    namelist *nlp;
    static readdir_res res; /* must be static! */

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

    if (dirp == NULL) {
        res.errno = errno;
        return (&res);
    }

    /*
     * Free previous result
     */
    xdr_free(xdr_readdir_res, &res);

    /*
     * Collect directory entries.
     * Memory allocated here will be freed by xdr_free
     * next time readdir_1 is called
     */
    nlp = &res.readdir_res_u.list;

    while (d = readdir(dirp)) {
        nl = *nlp = (namenode *) malloc(sizeof(namenode));
        nl->name = strdup(d->d_name);
        nlp = &nl->next;
    }

    *nlp = NULL;
```

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

Finally, there is the client side program to call the server.

```
/*
 * rls.c: Remote directory listing client
 */
#include <stdio.h>
#include <RPC/RPC.h> /* always need this */
#include "dir.h"     /* will be generated by rpcgen */

extern int errno;
main(argc, argv)
    int argc;
    char *argv[];
{
    CLIENT *cl;
    char *server;
    char *dir;
    readdir_res *result;
    namelist nl;

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

    /*
     * Remember what our command line arguments refer to
     */
    server = argv[1];
    dir = argv[2];

    /*
     * Create client "handle" used for calling MESSAGEPROG
     * on the server designated on the command line. We tell
     * the RPC package to use the "tcp" protocol when
     * contacting the server.
     */
    cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");
```

```
    if (cl == NULL) {

        /*
         * Couldn't establish connection with server.
         * Print error message and die.
         */
        clnt_pcreateerror(server);
        exit(1);
    }

    /*
     * Call the remote procedure readdir on the server
     */
    result = readdir_1(&dir, cl);

    if (result == NULL) {

        /*
         * An error occurred while calling the server.
         * Print error message and die.
         */
        clnt_perror(cl, server);
        exit(1);
    }

    /*
     * Okay, we successfully called the remote procedure.
     */
    if (result->errno != 0) {

        /*
         * A remote system error occurred.
         * Print error message and die.
         */
        errno = result->errno;
        perror(dir);
        exit(1);
    }

    /*
     * Successfully got a directory listing.
     * Print it out.
     */
    for (nl = result->readdir_res_u.list; nl != NULL;
        nl = nl->next) {
        printf("%s\n", nl->name);
    }

    exit(0);
}
```

Compile everything, and run.

```
sun%  rpcgen dir.x
sun%  cc rls.c dir_clnt.c dir_xdr.c -o rls
sun%  cc dir_svc.c dir_proc.c dir_xdr.c -o dir_svc
sun%  dir_svc &
moon%  rls sun /usr/pub
.
..
ascii
eqnchar
greek
kbd
marg8
tabclr
tabs
tabs4
moon%
```

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

**Note**

**If you do this, you may want to comment out calls to RPC library routines and have client-side routines call server routines directly.**

## The C-preprocessor

The C-preprocessor is run on all input files before they are compiled, so all the preprocessor directives are legal within a ".x" file. Four symbols may be defined, depending upon which output file is getting generated. The symbols are:

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

Also, rpcgen does a little preprocessing of its own. Any line that begins with a percent sign is passed directly into the output file without any interpretation. Here is a simple example that demonstrates the preprocessing features.

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

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

The "%" feature is not generally recommended, as there is no guarantee that the compiler will place the output where you intended.

## Timeout Changes

RPC sets a default timeout of 25 seconds for RPC calls when clnt_create() is used. This timeout may be changed using clnt_control(). Here is a small code fragment to demonstrate use of clnt_control().

```
struct timeval tv;

CLIENT *cl;
cl = clnt_create("somehost", SOMEPROG, SOMEVERS, "tcp");

if (cl == NULL) {
    exit(1);
}

tv.tv_sec = 60;/* change timeout to 1 minute */
tv.tv_usec = 0;
clnt_control(cl, CLSET_TIMEOUT, &tv);
```

## Handling Broadcast on the Server Side

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

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

Here is an example of a procedure that replies only if it thinks it is an NFS server:

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

    if (access("/etc/exports", F_OK) < 0) {
        return (NULL);/* prevent RPC from replying */
    }

    /*
     * return non-null pointer so RPC will send out a reply
     */
    return ((void *)&notnull);
}
```

**Note**

**If procedure returns type "void *," they must return a non-NULL pointer if they want RPC to reply for them.**

## Other Information Passed to Server Procedures

Server procedures will often want to know more about an RPC call than just its arguments. For example, getting authentication information is important to procedures that want to implement some level of security. This extra information is actually supplied to the server procedure as a second argument. Here is an example to demonstrate its use. What we've done here is rewrite the previous printmessage_1() procedure to only allow root users to print a message to the console.

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

    aup = (struct authunix_parms *)rq->rq_clntcred;

    if (aup->aup_uid != 0) {
        result = 0;
        return (&result);
    }

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

## RPC Language

RPC language is an extension of XDR language. The sole extension is the addition of the program type. For a complete description of the XDR language syntax, see *External Data Representation Standard: Protocol Specification*. For a description of the RPC extensions to the XDR language, see *Remote Procedure Calls: Protocol Specification*.

The XDR language is so close to C that if you know C, you know most of it already. We describe here the syntax of the RPC language, showing a few examples along the way. We also show how the various RPC and XDR type definitions get compiled into C-type definitions in the output header file.

### Definitions

An RPC language file consists of a series of definitions, as shown below.

```
definition-list:
   definition ";"
   definition ";" definition-list
```

It recognizes five types of definitions, listed below.

```
definition:
   enum-definition
   struct-definition
   union-definition
   typedef-definition
   const-definition
   program-definition
```

### Structures

An XDR struct is declared almost exactly like its C counterpart. It looks like the following:

```
struct-definition:
    "struct" struct-ident "{"
        declaration-list
    "}"

declaration-list:
    declaration ";"
    declaration ";" declaration-list
```

As an example, here is an XDR structure to a two-dimensional coordinate, and the C structure that it gets compiled into in the output header file.

```
struct coord {          ☞      struct coord {
    int x;                          int x;
    int y;                          int y;
};                              };
                                typedef struct coord coord;
```

The output is identical to the input, except for the added typedef at the end of the output. This allows one to use "coord" instead of "struct coord" when declaring items.

## Unions

XDR unions are discriminated unions, and look quite different from C unions. They are more analogous to Pascal variant records than they are to C unions.

```
union-definition:
  "union" union-ident "switch" "(" declaration ")" "{"
      case-list
  "}"

case-list:
  "case" value ":" declaration ";"
  "default" ":" declaration ";"
  "case" value ":" declaration ";" case-list
```

Here is an example of a type that might be returned as the result of a "read data" operation. If there is no error, then it returns a block of data. Otherwise, it doesn't return anything.

```
  union read_result switch (int errno) {
case 0:
    opaque data[1024];
default:
    void;
};
```

It gets compiled into the following:

```
  struct read_result {
    int errno;
    union {
        char data[1024];
    } read_result_u;
};
typedef struct read_result read_result;
```

Notice that the union component of the output struct has the name as the type name, except for the trailing "_u."

## Enumerations

XDR enumerations have the same syntax as C enumerations, as shown.

```
enum-definition:
    "enum" enum-ident "{"
        enum-value-list
    "}"

enum-value-list:
    enum-value
    enum-value "," enum-value-list

enum-value:
    enum-value-ident
    enum-value-ident "=" value
```

Here is a short example of an XDR enum, and the C enum that it gets compiled into.

```
enum colortype {              enum colortype {
     RED = 0,                      RED = 0,
     GREEN = 1,        ☞           GREEN = 1,
     BLUE = 2                      BLUE = 2,
 };                           };
                              typedef enum colortype colortype;
```

## Typedef

XDR typedefs have the same syntax as C typedefs.

```
typedef-definition:
    "typedef" declaration
```

Here is an example that defines a fname_type used for declaring file name strings that have a maximum length of 255 characters.

```
typedef string fname_type<255>;     typedef char *fname_type;
```

## Constants

XDR constants are symbolic constants that may be used wherever a integer constant is used, for example, in array size specifications.

```
const-definition:
    "const" const-ident "=" integer
```

For example, the following defines a constant DOZEN equal to 12:

```
const DOZEN = 12;    ☞    #define DOZEN 12
```

## Programs

RPC programs are declared using the following syntax:

```
program-definition:
    "program" program-ident "{"
        version-list
    "}" "=" value

version-list:
    version ";"
    version ";" version-list

version:
    "version" version-ident "{"
        procedure-list
    "}" "=" value

procedure-list:
    procedure ";"
    procedure ";" procedure-list

procedure:
  type-ident procedure-ident "(" type-ident ")" "=" value
```

For example, here is the time protocol, revisited.

```
/*
 * time.x: Get or set the time. Time is represented as number
 * of seconds since 0:00, January 1, 1970.
 */
program TIMEPROG {
    version TIMEVERS {
        unsigned int TIMEGET(void) = 1;
         void TIMESET(unsigned) = 2;
    } = 1;
} = 44;
```

This file compiles into #defines in the output header file.

```
#define TIMEPROG 44
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

## Declarations

In XDR, there are only four kinds of declarations.

1. simple-declaration:

Simple-declarations are just like simple C declarations.

```
type-ident variable-ident
```

Example:

```
colortype color;      ☞      colortype color;
```

2. fixed-array-declaration:

Fixed-length Array Declarations are just like C array declarations.

```
type-ident variable-ident "[" value "]"
```

Example:

```
colortype palette[8];      ☞      colortype palette[8];
```

3. variable-array-declaration:

Variable-Length Array Declarations have no explicit syntax in C, so XDR invents its own using angle-brackets.

```
type-ident variable-ident "<" value ">"
type-ident variable-ident "<" ">"
```

The maximum size is specified between the angle brackets. The size may be omitted, indicating that the array may be of any size.

```
int heights<12>;     /* at most 12 items */
int widths<>;        /* any number of items */
```

Since variable-length arrays have no explicit syntax in C, these declarations are actually compiled into structures. For example, the "heights" declaration gets compiled into the following structure:

```
struct {
   u_int heights_len;  /* # of items in array */
   int *heights_val;   /* pointer to array */
} heights;
```

**Note**

**The number of items in the array is stored in the "_len" component and the pointer to the array is stored in the "_val" component. The first part of each of these component's names is the same as the name of the declared XDR variable.**

4. pointer-declaration:

Pointer Declarations are made in XDR exactly as they are in C. You can't really send pointers over the network, but you can use XDR pointers for sending recursive data types such as lists and trees. The type is actually called "optional-data", not "pointer", in XDR language.

```
type-ident "*" variable-ident
```

Example:

```
listitem *next;      ☞      listitem *next;
```

## Special Cases

There are a few exceptions to the rules previously described. These exceptions are examined below.

Booleans: C has no built-in boolean type. However, the RPC library has a boolean type called bool_t that is either TRUE or FALSE. Variables declared as type bool in XDR language are compiled into bool_t in the output header file.

Example:

```
bool married;      ☞      bool_t married;
```

Strings: C has no built-in string type, but instead uses the null-terminated char * convention. In XDR language, strings are declared using the string keyword, and compiled into char * in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the NULL character). The maximum size can be left off, indicating a string of arbitrary length.

Examples:

```
string name<32>;            char *name;
string longname<>;    ☞     char *longname;
```

Opaque Data: Opaque data is used in RPC and XDR to describe untyped data, that is, sequences of arbitrary bytes. It can be declared either as a fixed or variable length array.

Examples:

```
opaque diskblock[512];         char diskblock[512];
                        ☞
opaque filedata<1024>;         struct {
                                   u_int filedata_len;
                                   char *filedata_val;
                               } filedata;
```

Voids: In a void declaration, the variable is not named. The declaration is simply "void" and nothing else. Void declarations can only occur in two places: union definitions and program definitions (as the argument or result of a remote procedure).

# XDR Standard: Protocol Specification

# 5

## Introduction

The External Data Representation (XDR) standard is a standard for the description and encoding of data. It is useful for transferring data between different computer architectures, and has been used to communicate data between such diverse machines as the Sun Workstation, VAX, IBM-PC, and CONVEX machines. XDR fits into the ISO presentation layer, and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference between these two is that XDR uses implicit typing, while X.409 uses explicit typing.

XDR uses a language to describe data formats. The language can only be used only to describe data. It is not a programming language. This language allows you to describe intricate data formats in a concise manner. The alternative of using graphical representations (itself an informal language) quickly becomes incomprehensible when faced with complexity. The XDR language itself is similar to the C language [1], just as Courier [4] is similar to Mesa. Protocols such as Sun RPC (Remote Procedure Call) and the NFS (Network File System) use XDR to describe the format of their data.

The XDR standard makes the following assumption: that bytes (or octets) are portable, where a byte is defined to be 8 bits of data. A given hardware device should encode the bytes onto the various media in such a way that other hardware devices may decode the bytes without loss of meaning. For example, the Ethernet standard suggests that bytes be encoded in "little-endian" style [2], or least significant bit first.

### 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. The bytes are read or written to some byte stream such that byte m always precedes byte m+1. If the n bytes needed to contain the data are not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of 4.

We include the familiar graphic box notation for illustration and comparison. In most illustrations, each box (delimited by a plus sign at the 4 corners and vertical bars and dashes) depicts a byte. Ellipses (...) between boxes show zero or more additional bytes where required.

**Figure 5-1**
A block

```
+--------+--------+...+--------+--------+...+--------+
| byte 0 | byte 1 |...|byte n-1|    0   |...|    0   |
+--------+--------+...+--------+--------+...+--------+
|<-----------n bytes---------->|<------r bytes------>|
|<-----------n+r (where (n+r) mod 4 = 0)>----------->|
```

## XDR Data Types

Each section that follows describes a data type defined in the XDR standard. Each section shows how the data type is declared in the language, and includes a graphic illustration of its encoding.

For each data type in the language we show a general paradigm declaration. Note that angle brackets (<>) denote variable length sequences of data and square brackets ([ ]) denote fixed-length sequences of data. "n," "m," and "r" denote integers. For the full language specification and more formal definitions of terms such as "identifier" and "declaration," refer to section *XDR Language Specification.*

For some data types, more specific examples are included. A more extensive example of a data description is in section *An Example of an XDR Data Description.*

### 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. Integers are declared as follows:

Figure 5-2
Integer

```
(MSB)                   (LSB)
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
+-------+-------+-------+-------+
<------------32 bits------------>
```

### 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. An unsigned integer is declared as follows:

Figure 5-3
Unsigned integer

```
(MSB)                   (LSB)
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
+-------+-------+-------+-------+
<------------32 bits------------>
```

### Enumeration

Enumerations have the same representation as signed integers. Enumerations are handy for describing subsets of the integers. Enumerated data is declared as follows:

```
enum { name-identifier = constant, ... } identifier;
```

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

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

It is an error to encode as an enum any other integer than those that have been given assignments in the enum declaration.

## Boolean

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Booleans are declared as follows:

```
bool identifier;
```

This is equivalent to:

```
enum { FALSE = 0, TRUE = 1 } identifier;
```

## Hyper Integer and Unsigned Hyper Integer

The standard also defines 64-bit (8-byte) numbers called hyper integer and unsigned hyper integer. Their representations are the obvious extensions of integer and unsigned integer defined above. They are represented in two's complement notation. The most and least significant bytes are 0 and 7, respectively. Their declarations:

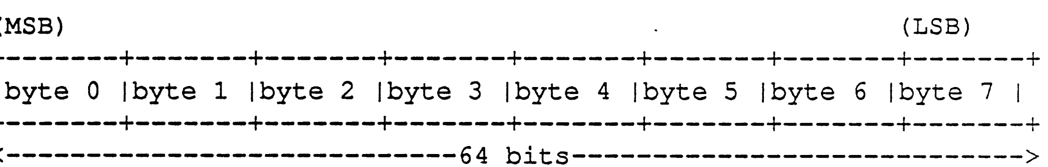



**Figure 5-4**
Hyper integer and unsigned hyper integer

## Floating-point

The standard defines the floating-point data type "float" (32 bits or 4 bytes). The encoding used is the IEEE standard for normalized single-precision floating-point numbers [3].

The floating-point number is described by:

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

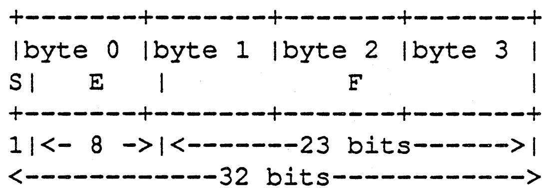
where the following three fields describe the single-precision floating-point number:

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

E The exponent of the number, base 2. 8 bits are devoted to this field. The exponent is biased by 127.

F The fractional part of the number's mantissa, base 2. 23 bits are devoted to this field.

It is declared as follows:



```
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
S|   E   |          F            |
+-------+-------+-------+-------+
1|<- 8 ->|<-------23 bits------>|
<------------32 bits------------>
```

**Figure 5-5**
Single-precision floating-point

Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a single-precision floating- point number are 0 and 31. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 9, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow) [3]. According to IEEE specifications, the "NaN" (not a number) is system dependent and should not be used externally.

## Double-precision Floating-point

The standard defines the encoding for the double-precision floating- point data type "double" (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized double-precision floating-point numbers [3]. The standard encodes the following three fields, which describe the double-precision floating-point number:

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

E: The exponent of the number, base 2. 11 bits are devoted to this field. The exponent is biased by 1023.

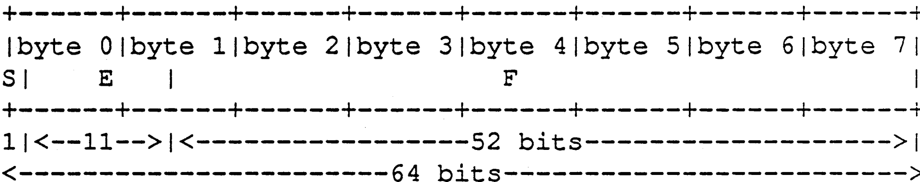
F: The fractional part of the number's mantissa, base 2. 52 bits are devoted to this field.

Therefore, the floating-point number is described by:

(-1)**S * 2**(E-Bias) * 1.F

It is declared as follows:

**Figure 5-6**
Double precision floating-point




Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a double-precision floating- point number are 0 and 63. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 12, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow) [3]. According to IEEE specifications, the "NaN" (not a number) is system dependent and should not be used externally.

## Fixed-length Opaque Data

At times, fixed-length uninterpreted data needs to be passed among machines. This data is called "opaque" and is declared as follows:

```
opaque identifier[n];
```

where the constant n is the (static) number of bytes necessary to contain the opaque data. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes r to make the total byte count of the opaque object a multiple of four.

Figure 5-7
Fixed-length opaque

```
0        1    ...
+--------+--------+...+--------+--------+...+--------+
| byte 0 | byte 1 |...|byte n-1|    0   |...|    0   |
+--------+--------+...+--------+--------+...+--------+
|<-----------n bytes---------->|<------r bytes------>|
|<-----------n+r (where (n+r) mod 4 = 0)------------>|
```

## Variable-length Opaque Data

The standard also provides for variable-length (counted) opaque data, defined as a sequence of n (numbered 0 through n-1) arbitrary bytes to be the number n encoded as an unsigned integer (as described below), and followed by the n bytes of the sequence.

Byte m of the sequence always precedes byte m+1 of the sequence, and byte 0 of the sequence always follows the sequence's length (count). enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four. Variable-length opaque data is declared in the following way:

```
opaque identifier<m>;
```

or

```
opaque identifier<>;
```

The constant m denotes an upper bound of the number of bytes that the sequence may contain. If m is not specified, as in the second declaration, it is assumed to be (2**32) - 1, the maximum length. The constant m would normally be found in a protocol specification.

For example, a filing protocol may state that the maximum data transfer size is 8192 bytes, as follows:

```
opaque filedata<8192>;
```

This can be illustrated as follows:

Figure 5-8
Variable-length opaque

```
0     1     2     3     4     5   ...
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|        length n       |byte0|byte1|...| n-1 |  0  |...|  0  |
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|<-------4 bytes------->|<------n bytes------>|<---r bytes--->|
|<----n+r (where (n+r) mod 4 = 0)---->|
```

It is an error to encode a length greater than the maximum described in the specification.

## String

The standard defines a string of n (numbered 0 through n-1) ASCII bytes to be the number n encoded as an unsigned integer (as described above) followed by the n bytes of the string. Byte m of the string always precedes byte m+1 of the string, and byte 0 of the string always follows the string's length. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four. Counted byte strings are declared as follows:

```
string object<m>;
```

or

```
string object<>;
```

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

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

Which can be illustrated as:

Figure 5-9
A string

```
0     1     2     3     4     5   ...
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|        length n       |byte0|byte1|...| n-1 |  0  |...|  0  |
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|<-------4 bytes------->|<------n bytes------>|<---r bytes--->|
|<----n+r (where (n+r) mod 4 = 0)---->|
```

It is an error to encode a length greater than the maximum described in the specification.

## Fixed-length Array

Declarations for fixed-length arrays of homogeneous elements are in the following form:

```
type-name identifier[n];
```

Fixed-length arrays of elements numbered 0 through n-1 are encoded by individually encoding the elements of the array in their natural order, 0 through n-1. Each element's size is a multiple of four bytes. Even though all elements are of the same type, the elements may have different sizes. For example, in a fixed-length array of strings, all elements are of type string, yet each element will vary in its length.

**Figure 5-10**
Fixed-length array

```
+---+---+---+---+---+---+---+---+...+---+---+---+---+
|   element 0   |   element 1   |...|  element n-1  |
+---+---+---+---+---+---+---+---+...+---+---+---+---+
|<--------------------n elements------------------->|
```

## Variable-length Array

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 declaration for variable-length arrays follows the form

```
type-name identifier<m>;
```

or

```
type-name identifier<>;
```

The constant m specifies the maximum acceptable element count of an array. If m is not specified, as in the second declaration, it is assumed to be

```
(2**32) - 1
```

**Figure 5-11**
Variable-length array

```
0 1 2 3
+--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
|     n     | element 0 | element 1 |...|element n-1|
+--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
|<-4 bytes->|<--------------n elements------------->|
```

It is an error to encode a value of n that is greater than the maximum described in the specification.

## Structure

Structures are declared as follows:

```
struct {
    component-declaration-A;
    component-declaration-B;
     ...
} identifier;
```

The components of the structure are encoded in the order of their declaration in the structure. Each component's size is a multiple of four bytes, even though the components may be different sizes.

**Figure 5-12**
A structure

```
+-------------+-------------+...
| component A | component B |...
+-------------+-------------+...
```

## Discriminated Union

A discriminated union is a type composed of a discriminant followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of discriminant is either int, unsigned int, or an enumerated type, such as bool. The component types are called arms of the union, and are preceded by the value of the discriminant which implies their encoding. Discriminated unions are declared as follows:

```
union switch (discriminant-declaration) {
    case discriminant-value-A:
        arm-declaration-A;
    case discriminant-value-B:
        arm-declaration-B;
     ...
    default:
        default-declaration;
} identifier;
```

Each case keyword is followed by a legal value of the discriminant. The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. The size of the implied arm is always a multiple of four bytes.

The discriminated union is encoded as its discriminant followed by the encoding of the implied arm.

**Figure 5-13**
Discriminated union

```
  0   1   2   3
+---+---+---+---+---+---+---+---+
| discriminant  | implied arm |
+---+---+---+---+---+---+---+---+
|<---4 bytes--->|
```

## Void

An XDR void is a 0-byte quantity. Voids are useful for describing operations that don't take data as input or don't return data as output. They are also useful in unions, where some arms may contain data and others do not. The declaration is simply as follows:

```
void;
```

Voids are illustrated as follows:

**Figure 5-14**
Void

```
  ++
  ||
  ++
--><-- 0 bytes
```

## Constant

The data declaration for a constant follows this form:

```
const name-identifier = n;
```

Const is used to define a symbolic name for a constant. It does not declare any data. The symbolic constant can be used anywhere a regular constant is used. For example, the following defines a symbolic constant DOZEN, equal to 12.

```
const DOZEN = 12;
```

## Typedef

Typedef does not declare any data, but serves to define new identifiers for declaring data. The syntax is:

```
typedef declaration;
```

The new type name is actually the variable name in the declaration part of the typedef. For example, the following defines a new type called eggbox using an existing type called egg:

```
typedef egg eggbox[DOZEN];
```

Variables declared using the new type name have the same type as the new type name in the typedef if it is a variable. For example, the following two declarations are equivalent in declaring the variable fresheggs:

```
eggbox fresheggs;
egg     fresheggs[DOZEN];
```

When a typedef involves a struct, enum, or union definition, there is another (preferred) syntax that may be used to define the same type. In general, a typedef of the following form:

```
typedef <<struct, union, or enum definition>> identifier;
```

can be converted to the alternative form by removing the typedef part and placing the identifier after the struct, union, or enum keyword, instead of at the end. For example, here are the two ways to define the type bool:

```
 typedef enum {    /* using typedef */
      FALSE = 0,
      TRUE = 1
} bool;

 enum bool {        /* preferred alternative */
      FALSE = 0,
      TRUE = 1
};
```

The reason this syntax is preferred is one does not have to wait until the end of a declaration to figure out the name of the new type.

## Optional-data

Optional-data is one kind of union that occurs so frequently that we give it a special syntax of its own for declaring it. It is declared as follows:

```
type-name *identifier;
```

This is equivalent to the following union:

```
 union switch (bool opted) {
      case TRUE:
           type-name element;
      case FALSE:
           void;
} identifier;
```

It is also equivalent to the following variable-length array declaration, since the boolean opted can be interpreted as the length of the array:

```
type-name identifier<1>;
```

Optional-data is not so interesting in itself, but it is very useful for describing recursive data-structures such as linked-lists and trees. For example, the following defines a type stringlist that encodes lists of arbitrary length strings:

```
 struct *stringlist {
      string item<>;
      stringlist next;
};
```

It could have been equivalently declared as the following union:

```
union stringlist switch (bool opted) {
    case TRUE:
        struct {
            string item<>;
            stringlist next;
        } element;
    case FALSE:
        void;
};
```

or as a variable-length array:

```
struct stringlist<1> {
    string item<>;
    stringlist next;
};
```

Both of these declarations obscure the intention of the stringlist type, so the optional-data declaration is preferred over both of them. The optional-data type also has a close correlation to how recursive data structures are represented in high-level languages such as Pascal or C by use of pointers. In fact, the syntax is the same as that of the C language for pointers.

## Areas for Future Enhancements

The XDR standard lacks representations for bit fields and bitmaps, since the standard is based on bytes. Also missing are packed (or binary-coded) decimals.

The intent of the XDR standard was not to describe every kind of data that people have ever sent or will ever want to send from machine to machine. Rather, it only describes the most commonly used data-types of high-level languages such as Pascal or C, so that applications written in these languages will be able to communicate easily over some medium.

Perhaps you can imagine extensions to XDR that would let it describe almost any existing protocol, such as TCP. The minimum necessary for this is support for different block sizes and byte-orders. The XDR discussed here can then be considered the 4-byte big-endian member of a larger XDR family.

# Discussion

## Why a Language for Describing Data?

There are many advantages in using a data-description language such as XDR, versus using diagrams. Languages are more formal than diagrams and lead to less ambiguous descriptions of data. Languages are also easier to understand, and allow you to think of other issues instead of the low-level details of bit-encoding. Also, there is a close analogy between the types of XDR and a high-level language such as C or Pascal. This makes the implementation of XDR encoding and decoding modules an easier task. Finally, the language specification itself is an ASCII string that can be passed from machine to machine to perform on-the-fly data interpretation.

## Why Only One Byte-Order for an XDR Unit?

Supporting two byte-orderings requires a higher level protocol for determining in which byte-order the data is encoded. Since XDR is not a protocol, this can't be done. The advantage of this, however, is that data in XDR format can be written to a magnetic tape, for example, and any machine will be able to interpret it since no higher level protocol is necessary for determining the byte-order.

## Why does XDR use Big-Endian Byte-Order?

Yes, it is unfair, but having only one byte-order means you have to be unfair to somebody. Many architectures, such as the Motorola 68000 and IBM 370, support the big-endian byte-order.

## Why is the XDR Unit Four Bytes Wide?

There is a trade-off in choosing the XDR unit size. Choosing a small size such as two makes the encoded data small, but causes alignment problems for machines that aren't aligned on these boundaries. A large size such as eight means the data will be aligned on virtually every machine, but causes the encoded data to grow too big. We chose four as a compromise. Four is big enough to support most architectures efficiently, except for rare machines such as the eight-byte aligned Cray. Four is also small enough to keep the encoded data restricted to a reasonable size.

## Why must Variable-Length Data be Padded with Zeros?

It is desirable that the same data encode into the same thing on all machines, so that encoded data can be meaningfully compared or checksummed. Forcing the padded bytes to be zero ensures this.

## Why is there No Explicit Data-Typing?

Data-typing has a relatively high cost for what small advantages it may have. One cost is the expansion of data due to the inserted type fields. Another is the added cost of interpreting these type fields and acting accordingly. And most protocols already know what type they expect, so data-typing supplies only redundant information. However, one can still get the benefits of data-typing using XDR. One way is to encode two things: first a string which is the XDR data description of the encoded data, and then the encoded data itself. Another way is to assign a value to all the types in XDR, and then define a universal type which takes this value as its discriminant and for each value, describes the corresponding data type.

# The XDR Language Specification

## Notational Conventions

This specification uses an extended Backus-Naur Form notation for describing the XDR language. Here is a brief description of the notation:

- The characters | () [] " , and * are special.
- Terminal symbols are strings of any characters surrounded by double quotes.
- Non-terminal symbols are strings of non-special characters.
- Alternative items are separated by a vertical bar ( | ).
- Optional items are enclosed in brackets.
- Items are grouped together by enclosing them in parentheses.
- A * following an item means 0 or more occurrences of that item.

For example, consider the following pattern:

"a " "very" (", " " very")* [" cold " "and"] " rainy " ("day" | "night")

An infinite number of strings match this pattern. A few of them are:

a very rainy day
a very, very rainy day
a very cold and rainy day
a very, very, very cold and rainy night

## Lexical Notes

- Comments begin with '/*' and terminate with '*/', as shown below

```
/* This is a comment */
```

- White space serves to separate items and is otherwise ignored.
- An identifier is a letter followed by an optional sequence of letters, digits or underbars (_). The case of identifiers is not ignored.
- A constant is a sequence of one or more decimal digits, optionally preceded by a minus-sign (-).

## Syntax Information

```
declaration:
    type-specifier identifier
    | type-specifier identifier "[" value "]"
    | type-specifier identifier "<" [ value ] ">"
    | "opaque" identifier "[" value "]"
    | "opaque" identifier "<" [ value ] ">"
    | "string" identifier "<" [ value ] ">"
    | type-specifier "*" identifier
    | "void"

value:
    constant
    | identifier

type-specifier:
    [ "unsigned" ] "int"
    | [ "unsigned" ] "hyper"
    | "float"
    | "double"
    | "bool"
    | enum-type-spec
    | struct-type-spec
    | union-type-spec
    | identifier

enum-type-spec:
    "enum" enum-body

enum-body:
    "{"
    ( identifier "=" value )
    ( "," identifier "=" value )*
    "}"

struct-type-spec:
    "struct" struct-body

struct-body:
    "{"
    ( declaration ";" )
    ( declaration ";" )*
    "}"

union-type-spec:
    "union" union-body

union-body:
    "switch" "(" declaration ")" "{"
    ( "case" value ":" declaration ";" )
    ( "case" value ":" declaration ";" )*
    [ "default" ":" declaration ";" ]
    "}"
```

```
constant-def:
    "const" identifier "=" constant ";"

type-def:
    "typedef" declaration ";"
    | "enum" identifier enum-body ";"
    | "struct" identifier struct-body ";"
    | "union" identifier union-body ";"

definition:
    type-def
    | constant-def

specification:
    definition *
```

### Syntax Notes

- The following are keywords and cannot be used as identifiers:

  -bool
  -case
  -const
  -default
  -double
  -enum
  -float
  -hyper
  -opaque
  -string
  -struct
  -switch
  -typedef
  -union
  -unsigned
  -void

- Only unsigned constants can be used as size specifications for arrays. If an identifier is used, it must have been declared previously as an unsigned constant in a const definition.

- Constant and type identifiers within the scope of a specification are in the same name space and must be declared uniquely within this scope.

- Similarly, variable names must be unique within the scope of struct and union declarations. Nested structure and union declarations create new scopes.

- The discriminant of a union must be of a type that evaluates to an integer. That is, int, unsigned int, bool, an enumerated type or any typedef type that evaluates to one of these is legal. Also, the case value must be one of the legal values of the discriminant. Finally, a case value may not be specified more than once within the scope of a union declaration.

## Example of an XDR Data Description

Here is a short XDR data description of a data structure named file, which might be used to transfer files from one machine to another.

```
const MAXUSERNAME = 32;      /* max length of a user name */
const MAXFILELEN = 65535;    /* max length of a file      */
const MAXNAMELEN = 255;      /* max length of a file name */

/*
 * Types of files:
 */
enum filekind {
    TEXT = 0,        /* ascii data */
    DATA = 1,        /* raw data   */
    EXEC = 2         /* executable */
};

/*
 * File information, per kind of file:
 */
union filetype switch (filekind kind) {
    case TEXT:
        void; /* no extra information */
    case DATA:
        string creator<MAXNAMELEN>; /* data creator */
    case EXEC:
      string interpretor<MAXNAMELEN>;/*program interpretor*/
};

/*
 * A complete file:
 */
struct file {
    string filename<MAXNAMELEN>; /* name of file */
    filetype type;               /* info about file */
    string owner<MAXUSERNAME>;    /* owner of file   */
    opaque data<MAXFILELEN>;      /* file data       */
};
```

Suppose now that there is a user named john who wants to store his lisp program, sillyprog, which only contains the statement

```
(quit)
```

His file would be encoded as follows:

| Offset | Hex Bytes | ASCII | Description |
|---|---|---|---|
| 0 | 00 00 00 09 | .... | Length of filename = 9 |
| 4 | 73 69 6c 6c | sill | Filename characters |
| 8 | 79 70 72 6f | ypro | ...and more characters ... |
| 12 | 67 00 00 00 | g... | ... and 3 zero-bytes of fill |
| 16 | 00 00 00 02 | .... | Filekind is EXEC = 2 |
| 20 | 00 00 00 04 | .... | Length of interpretor = 4 |
| 24 | 6c 69 73 70 | lisp | Interpretor characters |
| 28 | 00 00 00 04 | .... | Length of owner = 4 |
| 32 | 6a 6f 68 6e | john | Owner characters |
| 36 | 00 00 00 06 | .... | Length of file data = 6 |
| 40 | 28 71 75 69 | (qui | File data bytes ... |
| 44 | 74 29 00 00 | t).. | ... and 2 zero-bytes of fill |

## References

[1] Brian W. Kernighan & Dennis M. Ritchie, "The C Programming Language", Bell Laboratories, Murray Hill, New Jersey, 1978.

[2] Danny Cohen, "On Holy Wars and a Plea for Peace", IEEE Computer, October 1981.

[3] "IEEE Standard for Binary Floating-Point Arithmetic", ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August 1985.

[4] "Courier: The Remote Procedure Call Protocol", XEROX Corporation, XSIS 038112, December 1981.

# XDR Standard: Technical Notes

This chapter contains technical notes on Sun's implementation of the External Data Representation (XDR) standard, a set of library routines that allow a C programmer to describe arbitrary data structures in a machinex-independent fashion. For a formal specification of the XDR standard, see *XDR Standard: Protocol Specification.*

XDR is the backbone of Sun's 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. [1]

This chapter contains a short tutorial overview of the XDR library routines, a guide to accessing currently available XDR streams, and information on defining new streams and data types. XDR was designed to work across different languages, operating systems, and machine architectures. Most users (particularly RPC users) will only need the information in the *Number Filters Floating Point Filters* and *Enumeration Filters* sections. Programmers wishing to implement RPC and XDR on new machines will be interested in this chapter in its entirety, as well as the *XDR Standard: Protocol Specification*, which is our primary reference to XDR.

**Note**

***rpcgen* can be used to write XDR routines even in cases where no RPC calls are being made.**

On Sun systems, C programs that want to use XDR routines must include the file <rpc/rpc.h>, which contains all the necessary interfaces to the XDR system. Since the C library libc.a contains all the XDR routines, compile the C programs as usual by entering the following:

```
% cc program.c
```

where ***program*** is the C program you want to compile.

1. For a compete specification of the system External Data Representation routines, see the xdr(3N) manual pages.

## Justification

Consider the following two programs, "writer":

```
#include <stdio.h>

main()                          /* writer.c */
{
        long i;

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

and "reader"

```
#include <stdio.h>

main()                          /* reader.c */
{
        long i, j;

        for (j = 0; j < 8; j++) {
                if (fread((char *)&i, sizeof (i), 1, stdin) != 1) {
                        fprintf(stderr, "failed!\n");
                        exit(1);
                }
                printf("%ld ", i);
        }
        printf("\n");
        exit(0);
}
```

These two programs appear to be portable, for the following reasons:

- they pass lint checking
- they exhibit the same behavior when executed on two different hardware architectures, a Sun and a VAX

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 and 4.2BSD came the concept of "network pipes" — a process producing data on one machine, and a second process consuming data on another machine. You can construct a network pipe with writer and reader. Here are the results of producer and consumer programs. The first programs produces data on a Sun, and the second program 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 $2 sup 24$ — when four bytes are reversed, the 1 winds up in the 24th bit.

Whenever data is shared by two or more machine types, there is a need for portable data. Programs can be made data-portable by replacing the read() and write() calls with calls to an XDR library routine xdr_long(), a filter that knows the standard representation of a long integer in its external form.

Here are the revised versions of "writer"

```
#include <stdio.h>
#include <rpc/rpc.h>        /* XDR is a sub-library of RPC */

main()                      /* writer.c */
{
    XDR xdrs;
    long i;

    xdrstdio_create(&xdrs, stdout, XDR_ENCODE);
    for (i = 0; i < 8; i++) {
        if (!xdr_long(&xdrs, &i)) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
    }
    exit(0);
}
```

and "reader"

```
#include <stdio.h>
#include <rpc/rpc.h>        /* XDR is a sub-library of RPC */

main()                      /* reader.c */
{
    XDR xdrs;
    long i, j;

    xdrstdio_create(&xdrs, stdin, XDR_DECODE);
    for (j = 0; j < 8; j++) {
        if (!xdr_long(&xdrs, &i)) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
        printf("%ld ", i);
    }
    printf("\n");
    exit(0);
}
```

The new programs are 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%
```

**Note**

**Integers are just the tip of the portable-data iceberg. Arbitrary data structures present portability problems, particularly with respect to alignment and pointers. Alignment on word boundaries may cause the size of a structure to vary from machine to machine. And pointers, which are very convenient to use, have no meaning outside the machine where they are defined.**

## A Canonical Standard

XDR's approach to standardizing data representations is canonical. That is, XDR defines a single byte order (Big Endian), a single floating-point representation (IEEE), and so on. Any program running on any machine can use XDR to create portable data by translating its local representation to the XDR standard representations. Similarly, any program running on any machine can read portable data by translating the XDR standard representations to its local equivalents.

The single standard completely decouples programs that create or send portable data from those that use or receive portable data. The advent of a new machine or a new language has no effect upon the community of existing portable data creators and users. A new machine joins this community by being "taught" how to convert the standard representations and its local representations. The local representations of other machines are irrelevant. Conversely, to existing programs running on other machines, the local representations of the new machine are also irrelevant; such programs can immediately read portable data produced by the new machine because such data conforms to the canonical standards that they already understand.

There are strong precedents for XDR's canonical approach. For example, TCP/IP, UDP/IP, XNS, Ethernet, and, indeed, all protocols below layer five of the ISO model, are canonical protocols. The advantage of any canonical approach is simplicity. In the case of XDR, a single set of conversion routines is written once and is never touched again. The canonical approach has a disadvantage that is unimportant in real-world data transfer applications. Suppose two Little-Endian machines are transferring integers according to the XDR standard. The sending machine converts the integers from Little-Endian byte order to XDR (Big-Endian) byte order. The receiving machine performs the reverse conversion. Because both machines observe the same byte order, their conversions are unnecessary. The point, however, is not necessity, but cost as compared to the alternative.

The time spent converting to and from a canonical representation is insignificant, especially in networking applications. Most of the time required to prepare a data structure for transfer is not spent in conversion but in traversing the elements of the data structure. To transmit a tree, for example, each leaf must be visited and each element in a leaf record must be copied to a buffer and aligned there. Storage for the leaf may have to be deallocated as well. Similarly, to receive a tree, storage must be allocated for each leaf, data must be moved from the buffer to the leaf and properly aligned, and pointers must be constructed to link the leaves together. Every machine pays the cost of traversing and copying data structures whether or not conversion is required. In networking applications, communications overhead—the time required to move the data down through the sender's protocol layers, across the network and up through the receiver's protocol layers—dwarfs conversion overhead.

## The XDR Library

The XDR library not only solves data portability problems, it also allows you to write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it can make sense to use the library even when the data is not shared among machines on a network.

The XDR library has filter routines for strings (null-terminated arrays of bytes), structures, unions, and arrays, to name a few. Using more primitive routines, you can write your own specific XDR routines to describe arbitrary data structures, including elements of arrays, arms of unions, or objects pointed at from other structures. The structures themselves may contain arrays of arbitrary elements or pointers to other structures.

Let's examine the two programs more closely. There is a family of XDR stream creation routines in which each member treats the stream of bits differently. In our example, data is manipulated using standard I/O routines, so we use xdrstdio_create. The parameters to XDR stream creation routines vary according to their function. In our example, xdrstdio_create() takes a pointer to an XDR structure that it initializes, a pointer to a FILE on which the input or output is performed, and the operation. The operation may be XDR_ENCODE for serializing in the writer program, or XDR_DECODE for deserializing in the reader program.

## Note

**RPC users never need to create XDR streams; the RPC system itself creates these streams, which are then passed to the users.**

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

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

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

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

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

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

The corresponding XDR routine describing this structure is as follows:

```
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**

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

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

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

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

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

This document uses both coding styles.

## XDR Library Primitives

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

### Number Filters

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

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

Specifically, the eight primitives are:

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

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

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

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

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

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

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

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

The first parameter, xdrs, is an XDR stream handle. The second parameter is the address of the number that provides data to the stream or receives data from it. All routines return TRUE if they complete successfully, and FALSE if they do not.

## Floating Point Filters

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

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

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

The first parameter, xdrs, is an XDR stream handle. The second parameter is the address of the floating point number that provides data to the stream or receives data from it. Both routines return TRUE if they complete successfully, and FALSE if they do not.

**Note**

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

## Enumeration Filters

The XDR library provides a primitive for generic enumerations. The primitive assumes that a C type of enum has the same representation inside the machine as a C type of integer. The boolean type is an important instance of the enum. The external representation of a boolean is always TRUE (1) or FALSE (0).

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

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

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

The second parameters, ep and bp, are addresses of the associated type that provides data to, or receives data from, the stream xdrs.

## No Data

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 previously discussed. This section includes primitives for strings, arrays, unions, and pointers to structures.

Constructed data type primitives may utilize 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.

The xdr_string routine converts between the two representations of strings.

```
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 can be no longer than 255 characters.

The routine returns FALSE if the number of characters exceeds maxlength and TRUE if it doesn't.

**Caution**

**Keep maxlength small. If it is too big you can blow the heap, since xdr_string() will call malloc() for space.**

The behavior of xdr_string() is similar to the behavior of other routines discussed in this section. The direction XDR_ENCODE is easiest to understand. The parameter sp points to a string of a certain length. If the string does not exceed maxlength, the bytes are serialized.

The effect of deserializing a string is subtle. First, the length of the incoming string is determined. (It must not exceed maxlength.) Next, sp is dereferenced. If the value is NULL, then a string of the appropriate length is allocated and *sp is set to this string. If the original value of *sp is non-null, then the XDR package assumes that a target area has been allocated, which can hold strings no longer than maxlength. In either case, the string is decoded into the target area. The routine then appends a null character to the string.

In the XDR_FREE operation, the string is obtained by dereferencing sp. If the string isn't NULL, it is freed, and *sp is set to NULL. In this operation, xdr_string() ignores the maxlength parameter.

### Byte Arrays

Often variable-length arrays of bytes are preferable to strings. Byte arrays differ from strings in the following three ways:

1. The length of the array (the byte count) is explicitly located in an unsigned integer.
2. The byte sequence is not terminated by a null character.
3. The external representation of the bytes is the same as their internal representation.

The primitive xdr_bytes() converts between the internal and external representations of byte arrays:

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

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

### Arrays

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

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

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

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

Example #1. A user on a networked machine can be identified by

- the machine name, such as krypton (see the gethostname man page)
- the user's UID (see the geteuid man page)
- the group numbers to which the user belongs (see the getgroups man page)

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

```
struct netuser {
    char *nu_machinename;
    int   nu_uid;
    u_int nu_glen;
    int   *nu_gids;
};

#define NLEN 255     /* machine names < 256 chars */
#define NGRPS 20     /* user can't be in > 20 groups */

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

Example #2. 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 #3. The well-known parameters to main argc and argv can be combined into a structure. An array of these structures can make up a history of commands. The declarations and XDR routines might look like:

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

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

struct history {
    u_int h_len;
    struct cmd *h_cmds;
};

#define NCMDS 75     /* history is no more than 75 commands */

bool_t
xdr_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));
}
```

The most confusing part of this example is that the routine xdr_wrap_string() is needed to package the xdr_string() routine. This is because the implementation of xdr_array() passes only 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. Let's continue with more constructed data types.

### Opaque Data

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

```
bool_t
xdr_opaque(xdrs, p, len)
XDR *xdrs;
char *p;    /* the location of the bytes */
u_int len;  /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 provides a primitive, xdr_vector() for fixed-length arrays.

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

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


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

    if (!xdr_string(xdrs, &nup->nu_machinename, NLEN))
         return(FALSE);

    if (!xdr_int(xdrs, &nup->nu_uid))
         return(FALSE);

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

### 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 of type enum_t. Next, the union located at *unp is translated. The parameter arms is a pointer to an array of xdr_discrim structures. Each structure contains an ordered pair of [value,proc]. If the union's discriminant is equal to the associated value then the procedure 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 default arm procedure is called, if it is non-null; otherwise the routine returns FALSE.

Example #4. Suppose the type of a union can be integer, character pointer (a string), or a gnumbers structure. Also, assume the union and its current type are declared in a structure. The declaration is:

```
enum utype { INTEGER=1, STRING=2, GNUMBERS=3 };
struct u_tag {
    enum utype utype;    /* the union's discriminant */
    union {
        int ival;
        char *pval;
        struct gnumbers gn;
    }
    uval;
};
```

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

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

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

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

### Pointers

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

```
bool_t
xdr_reference(xdrs, pp, size, proc)
XDR *xdrs;
char **pp;   /* address of the pointer to the structure */
u_int ssize; /* size in bytes of the structure */
bool_t
(*proc)(); /* XDR routine that describes the structure */
```

To obtain the value of the parameter ssize, use the C function sizeof(). When decoding data, storage is allocated if *pp is NULL.

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

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

Example #5. 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) indicates 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 #5, 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() doesn't 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 serialing data will most likely cause a memory fault and, depending on your system, a core dump.

xdr_pointer() correctly handles NULL pointers. For more information about its use, see section *Linked Lists*.

Exercise: After reading the section *Linked Lists*, return here and extend Example #5 so that it can correctly handle NULL pointer values.

Exercise: Using the xdr_union, xdr_reference(), and xdr_void() primitives, implement a generic pointer handling primitive that implicitly handles NULL pointers. (Implement xdr_pointer.)

## 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.

**Caution**

**In some XDR streams, the returned value of xdr_getpos() is meaningless. The routine returns a –1 in this case (even though –1 should be a legitimate value).**

The routine xdr_setpos() sets a stream position to pos.

**Caution**

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

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

## XDR Operation Directions

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

### Standard I/O Streams

XDR streams can be interfaced to standard I/O using the xdrstdio_create() routine as follows:

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

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

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

### Memory Streams

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

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

void
xdrmem_create(xdrs, addr, len, x_op)
XDR *xdrs;
char *addr;  /* pointer to local memory address */
u_int len;    /* length of memory, in bytes */
enum xdr_op x_op;
```

The routine xdrmem_create() initializes an XDR stream in local 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, is 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 part of RPC */

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

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

The parameter xdrs is similar to the corresponding parameter described above. The stream does its own data buffering similar to that of standard I/O. The parameters sendsize and recvsize determine the size in bytes of the output and input buffers, respectively. If their values are zero (0), then predetermined defaults are used.

When a buffer needs to be filled or flushed, the routine readproc() or writeproc() is called, respectively. The usage and behavior of these routines are similar to the UNIX system calls read() and write(). However, the first parameter to each of these routines is the opaque parameter iohandle. The other two parameters buf and nbytes and the results (byte count) are identical to the system routines. If *xxx* is readproc() or writeproc(), then it has the following form:

```
 /*
* returns the actual number of bytes transferred.
* -1 is an error
*/

 int
 xxx(iohandle, buf, len)
 char *iohandle;
 char *buf;
 int nbytes;
```

The XDR stream provides means for delimiting records in the byte stream. The implementation details of delimiting records in a stream are discussed in section *Advanced Topics*. 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. (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 added param */
    struct xdr_ops {
        bool_t (*x_getlong)();   /* get long from stream */
        bool_t (*x_putlong)();   /* put long to stream */
        bool_t (*x_getbytes)();  /*get bytes from stream */
        bool_t (*x_putbytes)();  /* put bytes to stream */
        u_int (*x_getpostn)();   /* return stream offset */
        bool_t (*x_setpostn)();  /* reposition offset */
        caddr_t (*x_inline)();   /* ptr to buffered data */
        VOID (*x_destroy)();     /* free private area */
    }
    *x_ops;
    caddr_t x_public;      /* users' data */
    caddr_t x_private;     /* pointer to private data */
    caddr_t x_base;        /* private for position info */
    int     x_handy;       /* extra private word */
}
XDR;
```

The x_op field is the current operation being performed on the stream. This field is important to the XDR primitives, but should not affect a stream's implementation. That is, a stream's implementation should not depend on this value. The fields x_private, x_base, and x_handy are private to the particular stream's implementation. The field x_public is for the XDR client and should never be used by the XDR stream implementations or the XDR primitives. x_getpostn(), x_setpostn(), and x_destroy() are macros for accessing operations. The operation x_inline() takes two parameters: an XDR * and an unsigned integer, which is a byte count. The routine returns a pointer to a piece of the stream's internal buffer. The caller can then use the buffer segment for any purpose. From the stream's point of view, the bytes in the buffer segment have been consumed 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 if they are not. The routines have identical parameters (replace *xxx*).

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

The operations x_getlong() and x_putlong() receive and put long numbers from and to the data stream. It is the responsibility of these routines to translate the numbers between the machine representation and the (standard) external representation. The primitives htonl() and ntohl() can be helpful in accomplishing this. The higher-level XDR implementation assumes that signed and unsigned long integers contain the same number of bits, and that nonnegative integers have the same bit representations as unsigned integers. The routines return TRUE if they succeed, and FALSE otherwise. They have identical parameters:

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

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

## Advanced Topics

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

### Linked Lists

The last example in the pointers topic earlier in this chapter presented a C data structure and its associated XDR routines for a individual's gross assets and liabilities. The example is duplicated below:

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

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

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

```
struct gnumbers_node {
    struct gnumbers gn_numbers;
    struct gnumbers_node *gn_next;
};

typedef struct gnumbers_node *gnumbers_list;
```

The head of the linked list can be thought of as the data object. That is, the head is not merely a convenient shorthand for a structure. Similarly the gn_next field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the gn_next field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

The XDR data description of this linked list is described by the recursive declaration of gnumbers_list

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

struct gnumbers_node {
    gnumbers gn_numbers;
    gnumbers_node *gn_next;
};
```

In this description, the boolean indicates whether there is more data following it. If the boolean is FALSE, 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.

**Note**

**The C declaration has no boolean explicitly declared in it (even though the gn_next field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.**

**Hints for writing the XDR routines for a gnumbers_list follow easily from the XDR description above. The primitive xdr_pointer() is used to implement the XDR union above.**

```
bool_t
xdr_gnumbers_node(xdrs, gn)
XDR *xdrs;
gnumbers_node *gn;
{
    return(xdr_gnumbers(xdrs, &gn->gn_numbers) &&
        xdr_gnumbers_list(xdrs, &gp->gn_next));
}

bool_t
xdr_gnumbers_list(xdrs, gnp)
XDR *xdrs;
gnumbers_list *gnp;
{
    return(xdr_pointer(xdrs, gnp,
        sizeof(struct gnumbers_node), xdr_gnumbers_node));
}
```

The unfortunate side effect of using XDR on a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list. This is due to recursion.

The following routine collapses the previous two, mutually recursive routines into a single, non-recursive one.

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

    for (;;) {
        more_data = (*gnp != NULL);
        if (!xdr_bool(xdrs, &more_data))
            return(FALSE);

        if (! more_data)
            break;

        if (xdrs->x_op == XDR_FREE)
            nextp = &(*gnp)->gn_next;

        if (!xdr_reference(xdrs, gnp,
            sizeof(struct gnumbers_node), xdr_gnumbers))
            return(FALSE);

        np = (xdrs->x_op == XDR_FREE) ? nextp :
            &(*gnp)->gn_next;
    }
    *gnp = NULL;
    return(TRUE);
}
```

The first task is to find out whether there is more data or not, so that the boolean information can be serialized. Notice that this statement is unnecessary in the XDR_DECODE case, since the value of more_data is not known until we deserialize it in the next statement.

The next statement uses XDR for the more_data field of the XDR union. Then, if there is truly no more data, we set this last pointer to NULL to indicate the end of the list and return TRUE indicating we are done. Note that setting the pointer to NULL is only important in the XDR_DECODE case, since it is already NULL in the XDR_ENCODE and XDR_FREE cases.

Next, if the direction is XDR_FREE, we set the value of nextp to indicate the location of the next pointer in the list. We do this at this point because we need to dereference gnp to find the location of the next item in the list. After the next statement, the storage to which gnp points will be freed and will no longer be valid. We can't do this for all directions for in the XDR_DECODE direction, the value of gnp won't be set until the next statement.

Next, we XDR the data in the node using the primitive xdr_reference(). xdr_reference() is like xdr_pointer(), which we used before, but it does not send over the boolean indicating whether there is more data. We use it instead of xdr_pointer() because we have already referenced this information ourselves using XDR. Notice that the XDR routine passed is not the same type as an element in the list. The routine passed is xdr_gnumbers for referencing gnumbers using XDR, but each element in the list is actually of type gnumbers_node. We don't pass xdr_gnumbers_node() because it is recursive, and, instead, we use xdr_gnumbers(), which references all of the non-recursive part using XDR. Note that this trick will work only if the gn_numbers field is the first item in each element, so that their addresses are identical when passed to xdr_reference.

Finally, we update gnp to point to the next item in the list. If the direction is XDR_FREE, we set it to the previously saved value, otherwise we can dereference gnp to get the proper value. Even though it is harder to understand than the recursive version, this non-recursive routine is far less likely to blow the C stack. It will also run more efficiently since a lot of procedure call overhead has been removed. Most lists are small (in the hundreds of items or less) and the recursive version should be sufficient for them.

# Yellow Pages: Protocol Specification

## Introduction and Terminology

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

Translating or mapping a name to its value is one of the most common operations performed in computer systems. Common examples are translations of a variable names to virtual memory addresses, the translation of user names to system IDs or list of capabilities, and the translation of network node names to Internet addresses. There are two fundamental read-only operations that can be performed on a map: match and enumerate. *Match* is to look up a name, called a *key*, and return its current value. *Enumerate* is 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 availability and reliability by replicating both databases and database servers on multiple nodes within a single local net, and within the Internet. The database is replicated. All changes are made at a single server and eventually propagate to the remaining servers without locking. The YP is appropriate for an environment in which changes to the mapping databases occur on the order of tens per day.

The YP operates on an arbitrary number of map databases. Map names provide the lower of two levels of a naming hierarchy. Maps are themselves grouped into named sets, called domains. Domain names provide a second, higher level of naming. Map names must be unique within a domain, but can 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 that 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 Remote Procedure Call (RPC) mechanism defines a paradigm for interprocess communication modeled on function calls. Clients call functions that optionally return values. All inputs and outputs to the functions are in the client's address space. The function is executed by a server program.

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

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

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

## External Data Representation (XDR)

The External Data Representation (XDR) specification establishes standard representations for basic data types (such as strings, signed and unsigned integers, and structures and unions) in a way that allows them to be transferred among machines with varying architectures.

XDR provides primitives to encode (that is, translate from the local host's representation to the standard representation) and decode (translate from the standard representation to the local host's representation) basic data types.

Constructor primitives allow arbitrarily complex data types to be made from the basic types.

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

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

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

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

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

states that the byte sequence 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

## Maps and Map Operations

### Map Structure

Maps are named sets of key-value pairs. Keys and their values are counted binary objects, and may be ASCII information, but need not be. The data comprising a map is determined by the client applications that are the final customers for the data, not by the YP. YP has no syntactic nor semantic knowledge of the map contents. YP doesn't determine or know any map's name. Map names are managed by the YP 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 map database is an implementation detail and is unspecified by the protocol.

### Match Operation

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

### Map Entry Enumeration

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

### Entire Map Retrieval

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

### Map Update

The update of YP maps is an implementation detail that's outside the specification of the YP service. A separate YP update is used to achieve very limited update capability to YP maps for Secure NFS.

## Master and Slave YP Database Servers

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

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

## Map Propagation and Consistency

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

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

### Functions to Aid in Map Propagation

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

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

## Domains

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

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

## Restrictions

The following capabilities are not included in the current YP protocols.

### Map Update Within the YP

All write and delete accesses to the YP map database are assumed to be outside of the YP subsystem. It is probable that write access to the map database will be included in later versions of the YP protocols. A limited form of YP update protocol is provided for Secure NFS.

### Version Commitment Across Multiple Requests

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

### Guaranteed Global Consistency

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

### Access Control

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

## YP Database Server Protocol Definition

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

### RPC Constants

The following numbers are in decimal:

- YP database server protocol program number
- current YP protocol version

### Other Manifest Constants

The following numbers are in decimal:

- total maximum size of key and value for any pair. (The absolute sizes of the key and value may divide this maximum arbitrarily.)
- maximum number of characters in a domain name
- maximum number of characters in a map name
- maximum number of characters in a YP host name

### Remote Procedure Return Values

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

ypstat:

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

ypxfrstat:

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

## Basic Data Structures

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

domainname: `typedef string domainname<YPMAXDOMAIN>`

mapname: `typedef string mapname<YPMAXMAP>`

peername: `typedef string peername<YPMAXPEER>`

keydat: `typedef string keydat<YPMAXRECORD>`

valdat: `typedef string valdat<YPMAXRECORD>`

ypmap_parms:

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

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

ypreq_xfr:

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

ypresp_val:

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

ypresp_key_val:

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

ypresp_master:

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

ypresp_order:

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

ypresp_all:

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

ypresp_xfr:

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

ypmaplist:

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

ypresp_maplist:

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

### YP Database Server Remote Procedures

This section contains a specification for each function that can be called as a remote procedure in Version 2. The input and output parameters are described using the XDR data definition language.

Procedure 0: Do Nothing

```
YPPROC_NULL() returns ()
```

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

Procedure 1: Do You Serve This Domain?

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

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

Procedure 2: Answer Only If You Serve This Domain

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

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

The current implementation currently returns the FALSE case by forcing an RPC decode error.

Procedure 3: Return Value of a Key

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

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

Procedure 4: Get First Key-Value Pair in Map

```
PPROC_FIRST (req) returns (resp)
    ypreq_key req;
    ypresp_key_val resp;
```

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

Procedure 5: Get Next Key-Value Pair in Map

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

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

Procedure 6: Transfer Map

```
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 is the YP server process, or some other process) to call back the requester with a summary status.

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

Procedure 7: Reinitialize Internal State

```
YPPROC_CLEAR () returns ()
```

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

Procedure 8: Get All Key-Value Pairs in Map

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

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

Procedure 9: Get Map Master Name

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

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

Procedure 10: Get Map Order Number

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

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

Procedure 11: Get All Maps in Domain

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

This 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 chapter describes the YP binder, an optional element in the YP subsystem that supplies YP database server addressing information to potential YP clients.

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

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

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

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

### YP Binder Protocol Definition

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

#### RPC Constants

All numbers are decimal.

- the YP binder protocol program number
- the current YP binder protocol version

#### Other Manifest Constants

All numbers are decimal.

- the maximum number of characters in a domain name

This is identical to the constant defined above within the YP database server protocol section.

ypbind_resptype:

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

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

ypbinderr:

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

The error case of most interest to a YP binder client is YPBIND_ERR_NOSERV. It indicates 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.

Procedure 0: Do Nothing

```
YPBINDPROC_NULL () returns ()
```

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

Procedure 1: Get Current Binding for a Domain

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

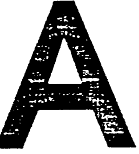
This returns the binding information necessary to address a YP database server within the Internet environment.

Procedure 2: Set Domain Binding

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

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

# Reporting Problems



This appendix introduces the CONVEX Technical Assistance Center (TAC) and the `contact` utility.

The `contact` utility is an online system for reporting problems to the TAC. To use it, enter **`contact`** at the system prompt and answer the questions as they appear on the screen.

This appendix describes:

- Prerequisites for using `contact`
- Tips for using `contact`
- The step-by-step process through which `contact` takes you through

## Technical Assistance Center

The CONVEX Technical Assistance Center (TAC) is staffed by technical specialists who can address diverse questions and problems that arise in a supercomputing environment. If you have a hardware, software, or documentation question, contact the TAC. This group stands ready to solve such problems.

## The `contact` Utility

The TAC recommends using the `contact` utility to report a hardware, software, or documentation problem. The `contact` utility is an interactive program that helps the TAC track reports and route them to the CONVEX personnel most qualified to fix a problem.

After you invoke `contact`, it prompts you for information about the problem. When you finish your report, `contact` electronically mails it to the TAC. The TAC notifies you within 48 hours that your report has been received.

To use `contact` requires:

- UNIX-to-UNIX Communication Protocol (UUCP) connection to the TAC
- Full path name of the program or utility in question
- Version number of the program or utility in question

## UUCP Connection

Before using contact, ask your system administrator if your site has a UUCP connection to the TAC. A UUCP connection allows files to be copied from one UNIX-based system to another. The uucp (UNIX-to-UNIX copy) command relies on either a dial-up or hard-wired UUCP communication line.

## Finding the Program Path Name

To determine the full path name of the program or utility in question, use the which command. Figure A-1 illustrates use of the which command to find the full path name of the loader (ld) utility.

Figure A-1
Using the which command

```
> which ld
/bin/ld
>
```

In this example, the full path name of the loader is /bin/ld.

If you use the C shell (csh), you can also use the whence command to find the program path name. The whence command works like which, but faster.

For more information on the which command, refer to the which(1) man page. You can also use the info online information system by entering **info which** at the system prompt.

## Finding the Program Version Number

To determine the version number of the program or utility in question, use the vers command. Figure A-7 illustrates use of the vers command to find the version number of the loader (ld) utility. Enter **vers**, then the path name of the program or utility.

Figure A-2
Using the vers command

```
> vers /bin/ld
/bin/ld: 7.0
>
```

In this example, the loader version number is 7.0.

For more information on the vers command, refer to the vers(1) man page. You can also use the info online information system by entering **info vers** at the system prompt.

## Using `contact`

The `contact` utility prompts for the following information:

- Your name, title, phone number, and corporate name
- Name and version of the product
- One-line summary of the problem
- Detailed description of the problem
- Priority of the problem
- Instructions on how to reproduce the problem
- Comments about the problem
- Comments about the documentation relating to the problem
- Files to include in the `contact` report

Following is a step-by-step discussion of these prompts.

### Step 1a

To invoke the `contact` utility, enter **`contact`** at the system prompt. The system responds with a welcome message and a series of questions regarding your hardware, software, or documentation question. Figure A-3 illustrates use of the `contact` command and the resulting system response.

**Figure A-3**
Beginning a `contact` session

```
> contact
Welcome to contact version 0.11 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
>
```

### Step 1b

If there is a .contact file in your home directory, `contact` skips the first prompt. (Refer to "Using a .contact File" on page 6 for more information.) Figure A-4 illustrates the `contact` command and the system response when you have a .contact file in your home directory.

**Figure A-4**
Beginning a `contact` session
with a .contact file

```
> contact
Welcome to contact version 0.11 ()

Enter the name of the product involved
>
```

### Step 2

The `contact` utility prompts for the version number of the product. If you do not know the version number, press **CTRL-Z** to suspend the session.

Use the `which` (or `whence` if you use `csh`) and `vers` commands to find the version number of the product. Use the `fg` command to return to the session, and enter the version number in the form X.X or X.X.X.X.

### Step 3

The `contact` utility prompts for a one-line summary of the problem. This summary is the subject header in any further correspondence regarding the problem. Please make this summary as descriptive as possible in one line.

### Step 4

The `contact` utility prompts for a detailed description of the problem. Please make this description as complete as possible. Include source code and a stack backtrace when possible. (Refer to the adb(1) or csd(1) man page for information on obtaining a stack backtrace.) The more information you provide, the quicker the TAC can isolate and solve the problem.

### Step 5

The `contact` utility prompts for the priority of the problem. Figure A-5 illustrates this prompt and priority levels from which to choose. You must enter a priority number.

Figure A-5
Specifying priority of a problem

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

### Step 6

The `contact` utility prompts for an explanation of how to reproduce the problem. Please include the command syntax and options you used and anything else you did to make the program run.

### Step 7

The `contact` utility prompts for any other pertinent comments. Please include all relevant information.

### Step 8

The `contact` utility prompts for suggestions regarding documentation supporting the product. Indicate whether the documentation could be revised to address the problem.

## Step 9

The contact utility prompts for names of files necessary to reproduce the problem. Figure A-6 illustrates this prompt and sample user response.

Figure A-6
Including files in a contact report

```
Are there any files that should be included in this report (yes | no)?
> yes
Please enter the names of the files, one to a line (^D to terminate)
> test.f
> ~/subroutines/sub.f
>
```

### Note

**"Tilde-Escape Sequences" on page 7 are not recognized in responses to this prompt. In contact, a tilde in this section indicates your home directory. This convention is based on use of the tilde for expanding file names in csh.**

If files specified are small text files, they are automatically included in the contact report. If the files are too large to be included in this report, contact gives further instructions on how to submit these files.

To specify a directory, combine directory files into a single file using the tar command (refer to the tar(1) man page for further information) or enter each file name in the directory on a single line in the contact report.

## Step 10

The contact utility prompts you to review, edit, submit, or abort the report. Figure A-7 illustrates this prompt.

Figure A-7
Prompt to review, edit, submit, or abort report

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

Choose the number of the option you want to select. These options let you do the following:

Review — review the text of the contact report. You are then prompted again to select an option.

Edit — edit the text of the contact report. If you choose to edit the report, contact opens your default text editor.

Submit — sends the report to the CONVEX TAC. The TAC notifies you within 48 hours that your report has been received. Choosing this option exits the contact utility and returns you to the shell.

Abort — saves the text of the report in a file named ~/dead.report. Choosing this option exits contact and returns you to the shell.

## Tips for Using contact

The contact utility is interactive and easy to use. This section lists tips to help use it efficiently. In particular, this section tells how to:

- Use a .contact file.
- Abort a contact session.
- Resubmit an aborted report.
- Suspend a contact session.
- Move within contact from one prompt to another.
- Use tilde-escape sequences in the contact utility.

### Using a .contact File

When you invoke contact, it first prompts for your name, title, phone number, and company name. You can, however, create a .contact file to skip this first prompt.

Follow these steps to create a .contact file.

1. Create a .contact file in your home directory.
2. Enter your name, job title, phone number, and company name, each on a new line.

When you invoke contact, it automatically includes the .contact file as input for the first prompt and proceeds to the next prompt.

### Aborting the Report

To abort a contact report, either press the interrupt key (usually **CTRL-C**) or choose the abort option when prompted by the contact utility. Using **CTRL-C** to abort does not save the contents of the report. Using the abort option saves the contents of the report in a file named ~/dead.report.

### Submitting the dead.report File

After you abort a contact session, the contact utility saves the report in a file named ~/dead.report. Using the contact command with the -r option automatically merges the contents of the ~/dead.report file into the new contact session. Enter

```
contact -r
```

and contact finds the ~/dead.report file and merges it into the contact report. You can then edit the report. When you end the editing session, contact resumes at the final prompt, which asks you to review, edit, submit, or abort the report.

## Suspending a Report

Sometimes it is necessary to stop in the middle of a `contact` report and return to the shell (for instance, to suspend the `contact` session to find the program path name or version number). To suspend the `contact` session, press **CTRL-Z**.

To return to the `contact` session, type **`fg`**. Using **CTRL-Z** and the `fg` (foreground) command, you can switch between the `contact` utility and the shell. You cannot, however, use **CTRL-Z** and `fg` to switch back and forth in the Bourne shell (`sh`).

## Ending a Response

The `contact` utility prompts for information pertinent to your hardware, software, or documentation question. Some prompts require one-line responses; to move to the next prompt, press **RETURN**. Other prompts require more than a one-line response; to move to the next prompt, press **CTRL-D**.

## Tilde-Escape Sequences

The `contact` utility treats input beginning with a tilde (~) as a special sequence. The character following the tilde is considered a request for a special function. You can use the following tilde sequences within `contact`:

~e start the text editor (defined in the EDITOR environment variable)

~h display a list of available tilde-escape sequences

~p print the `contact` report to the terminal screen

~r *filename* read the contents of *filename* as a response to the current prompt. Some prompts require only a one-line response. This tilde-escape sequence works only for prompts that allow more than a one-line response.

~~ insert a single tilde as the first character in the line

# Master Index


## A

administration, RPC 3-8
arbitrary data types 3-10
arrays 5-8
assigning program numbers 3-8
auth_flavor 2-5
AUTH_NULL 2-10
AUTH_REJECTEDCRED 2-10
AUTH_SHORT 2-10
AUTH_UNIX 2-10
authentication 3-26
authentication protocols, RPC 2-10
authentication, RPC 2-3, 2-5

## B

Backus-Naur Form 5-14
batching 3-22
batching, RPC 2-6
binder remote procedures, YP 7-15
binders, YP 7-13
binding, RPC 2-3
block size, basic XDR 5-1
boolean type declaration 4-21
booleans 5-3
broadcast RPC 2-6, 3-21, 3-22, 4-14

## C

callback procedures, RPC 3-39
canonical standard, XDR 6-6
client handle, used by rpcgen 4-6
clnt_control() 4-14
compiling rcpgen programs 4-12
constants 4-18, 5-10
conventions, rpcgen 4-3
C-preprocessor 4-13
credentials, RPC 2-5
C-type definitions 4-16

## D

data description example, XDR 5-17
data structures, basic YP binder 7-14, 7-15
data types
  attrstat, NFS 1-15
  basic YP 7-7
  boolean, XDR 5-3
  constant, XDR 5-10
  constants, XDR 5-10
  diropres, NFS 1-15
  dirpath, mount 1-27
  discriminated unions, XDR 5-9
  double precision floating point, XDR 5-5
  enumeration, XDR 5-2
  fattr, NFS 1-13
  fhandle, mount 1-27
  fhandle, NFS 1-12
  fhstatus, mount 1-27
  filename, NFS 1-14
  fixed-length arrays, XDR 5-8
  floating-point, XDR 5-4
  ftype, NFS 1-12
  hyper integers, XDR 5-3
  integer, XDR 5-2
  name, mount 1-27
  NFS 1-10
  opaque, XDR 5-6
  optional-data, XDR 5-11
  path, NFS 1-15
  sattr, NFS 1-14
  stat, NFS 1-10
  string, XDR 5-7
  structures, XDR 5-9
  timeval, NFS 1-12
  typedef, XDR 5-10
  unsigned integer, XDR 5-2
  variable-length arrays, XDR 5-8
  void, XDR 5-10
  XDR 5-2
  XDR definitions 1-10
data types, passing arbitrary RPC 3-10
database servers 7-4
debugging with rpcgen 4-12
declarations, rpcgen 4-20
default timout, changing using rpcgen 4-14
definitions in an RPC language file 4-16
DES authentication 2-11, 3-30
  naming 2-11
  record marking 2-16
  verifiers 2-11
DES authentication protocol example 2-13
deserialization, XDR 3-10
difference, remote and local procedures 4-4
Diffie-Hellman encryption 2-15
direction of XDR operations 6-21
discriminated union 5-9
discriminated unions 4-17
domainname 7-7, 7-14
double-precision floating- point 5-5

## E

encoding files, XDR 5-18
enum clnt_stat (in RPC programming) 3-6
enumeration filters 6-10
enumerations 4-17, 5-2
error codes, NFS 1-11, 1-12
External Data Representation 1-7
External Data Representation (see XDR) 6-1

## F

file attributes field values 1-23
file system model, NFS 1-9
fixed size arrays, XDR 6-16
fixed-array-declaration 4-20
floating point filters, XDR 6-10
floating-point 5-4

## G

generating XDR routines 4-8
gettransient() 3-40, 3-42

## H

highest layer of RPC 3-4
hyper integer 5-3

## I

implementation issues, NFS 1-24
inetd(8C) 3-33
integer representation 5-2
intermediate layer of RPC 3-5

## K

keydat 7-7
keywords, XDR 5-16

## L

languages, RPC 2-17
library primitives, XDR 6-9
library, XDR 6-7

## M

map propagation and consistency 7-5
mapname 7-7
memory allocation with XDR 3-17
message protocol, RPC 2-7
model, RPC 2-1
mount data types 1-27
  dirpath 1-27
  fhandle 1-27
  fhstatus 1-27
mount protocol
  basic data types 1-27
  definition 1-26
  RPC information 1-26
  XDR structure sizes 1-26
mount server procedures 1-28
  MNTPROC_DUMP() 1-28
  MNTPROC_EXPORT() 1-29
  MNTPROC_MNT() 1-28
  MNTPROC_NULL() 1-28
  MNTPROC_UMNT() 1-29
  MNTPROC_UMNTALL() 1-29
mount server procedures to
  add mount entry 1-28
  do nothing 1-28
  remove all mount entries 1-29
  remove mount entry 1-29
  return export list 1-29
  return mount entries 1-28

## N

network clients, definition 2-1
network communication with RPC 3-3
Network File System 1-7
network service, definition 2-1
NFS
- basic data types 1-10
- error codes 1-11, 1-12
- file attributes field values 1-23
- file system model 1-9
- pathname interpretation 1-24
- permission issues 1-24
- protocol definition 1-9
- RPC information 1-9
- server/client relationship 1-24
- setting RPC parameters 1-25

NFS data types 1-10
- attrstat 1-15
- diropres 1-15
- fattr 1-13
- fhandle 1-12
- filename 1-14
- ftype 1-12
- path 1-15
- sattr 1-14
- stat 1-10
- timeval 1-12

NFS server procedures 1-15
- file service routines 1-16
- NFSPROC_CREATE() 1-19
- NFSPROC_GETATTR() 1-16
- NFSPROC_LINK() 1-20
- NFSPROC_LOOKUP() 1-17
- NFSPROC_MKDIR() 1-21
- NFSPROC_NULL() 1-16
- NFSPROC_READ 1-18
- NFSPROC_READDIR 1-22
- NFSPROC_READDIR() 1-22
- NFSPROC_READLINK() 1-17
- NFSPROC_REMOVE() 1-19
- NFSPROC_RENAME() 1-19
- NFSPROC_RMDIR() 1-21
- NFSPROC_ROOT 1-17
- NFSPROC_SETATTR() 1-16
- NFSPROC_STATFS() 1-23
- NFSPROC_SYMLINK() 1-20
- NFSPROC_WRITE() 1-18
- NFSPROC_WRITECACHE() 1-18

NFS server procedures to
- create directory 1-21
- create file 1-19
- create link to file 1-20
- create symbolic link 1-20
- do nothing 1-16
- get file attributes 1-16
- get filesystem attributes 1-23
- get filesystem root 1-17
- look up file name 1-17
- read from directory 1-22
- read from files 1-18
- read from symbolic link 1-17
- remove directory 1-21
- remove file 1-19
- rename file 1-19
- set file attributes 1-16
- write to cache 1-18
- write to file 1-18

no data, XDR 6-10
normalized single-precision floating-point numbers 5-4
null authentication 2-10
null procedure 4-3

## O

opaque data 5-6, 5-7
opaque data declaration 4-21
opaque data, XDR 6-16
opaque_auth 2-5
optional-data 5-11

## P

parameters, setting RPC 1-25
pathname interpretation, NFS 1-24
peername 7-7
permission issues, NFS 1-24
PING_PROG example 2-17
PING_VERS_PINGBACK() 2-17
PINGPROC_NULL() 2-17
PINGPROC_PINGBACK() 2-17
PMAPPROC_CALLIT 2-21
PMAPPROC_DUMP 2-21
PMAPPROC_GETPORT 2-21
PMAPPROC_NULL 2-21
PMAPPROC_SET 2-21
PMAPPROC_UNSET 2-21
pointer semantics and XDR 6-20
pointer-declaration 4-21
port mapper program 2-19, 2-21
printmessage() 4-3
procedure conversion, local to remote 4-2
procedure numbers 2-4
program number assignment, RPC 2-6, 3-8
program type 4-16
programs, declaring RPCusing rpcgen 4-19
protocol requirements, RPC 2-4

## R

readproc() 6-23
record fragments 2-16
record marking (RM) 2-16
registered programs, RPC 3-9
remote directory listing client 4-10
remote directory listing protocol 4-8
Remote Procedure Call 1-7
remote procedure definition, rpcgen 4-4
remote program, definition 2-1
remote readdir implementation 4-9
rendezvous independence, RPC 2-3
reply message 2-5
reply_if_nfsserver() 4-14
replying, preventing servers from 4-14
RPC 1-7, 7-2
- administration 3-8
- assigning program numbers 3-8
- auth_flavor enumeration 2-5
- AUTH_NULL 2-10
- AUTH_REJECTEDCRED 2-10
- AUTH_SHORT 2-10
- AUTH_UNIX 2-10
- authentication 2-3, 2-5, 3-26, 3-30
- authentication protocols 2-10
- authentication, general 1-9
- batching 2-6, 3-22
- binding 2-3
- broadcast 2-6, 3-21
- broadcast synopsis 3-22
- built-in XDR routines 3-10
- callback procedures 3-39
- calling side 3-19
- clock synchronization, DES authentication 2-12
- credentials 2-5
- definition 3-1
- DES authentication 2-11, 3-30
- DES authentication example 2-13, 2-14
- DES authentication verifiers 2-11
- Diffie-Hellman encryption 2-15
- error detection, reply messages 2-5
- example service described in 2-17
- guarantees 3-28
- highest layer 3-2, 3-4
- intermediate layer 3-5
- language 2-17
- language specification 2-18
- layers of 3-2
- local procedure calls, similarities to 2-1
- lowest layer 3-2, 3-13
- lowest layer program example 3-14
- lowest layer, reasons for using 3-13
- message protocol 2-7
- message protocol example 2-8, 2-9
- middle layer 3-2
- model 2-1
- naming, DES authentication 2-11
- network clients 2-1
- network clients (definition) 2-1
- network service (definition) 2-1
- nicknames, DES authentication 2-12
- null authentication 2-10
- opaque_auth structure 2-5
- paradigm 3-3
- port mapper operation 2-21
- port mapper protocol 2-19
- port mapper protocol example 2-20
- port number, general 1-9
- port protocols, general 1-9
- prefabricated XDR routines 3-11
- procedure numbers 2-4
- procedures and programs 2-4
- program number, assignment of 2-6
- program numbers 2-4
- program numbers, assignment of 2-6
- programs 4-19
- programs and procedures 2-4
- protocol requirements 2-4
- protocol, other uses of 2-6
- protocols, batching 2-6
- protocols, other uses of broadcast RPC 2-6
- record fragments 2-16
- record marking standard 2-16
- registered program list 3-9
- remote program (definition) 2-1
- rendezvous independence 2-3
- reply message error detection 2-5
- RPC_AUTHERROR 2-12
- select() 3-21
- semantics 2-2
- semantics and transports 2-2

sending variable integer arrays 3-11
server (definition)r 2-1
server side 3-14
service library routines 3-4
services 3-4
terms 2-1
timestamps, DES authentication 2-12
TPC 3-36
transports 2-2
transports and semantics 2-2
UNIX authentication 2-10, 3-27
user-defined type routine 3-10
verifier 2-5
verifiers 2-5
versions 3-34
XDR deserialization 3-10
XDR memory allocation 3-18
XDR serialization 3-10, 3-18
RPC and XDR type definitions 4-16
RPC language 4-16
RPC_AUTHERROR 2-12
rpcgen 3-1
"%" feature 4-13
array types 4-20
boolean type 4-21
broadcast RPC 4-14
compiling 4-12
constants 4-18
conventions 4-3
C-preprocessing symbols 4-13
debugging 4-12
declarations 4-20
default timeout changes 4-14
definitions 4-16
difference beween remote and local procedures 4-4
enumerations 4-17
generating XDR routines 4-8
input file 4-7
opaque data type 4-21
output files 4-8
passing extra info to server procedures 4-15
pointer type 4-21
preprocessing 4-13
preventing servers from replying 4-14
programs 4-19
purpose of 4-1
remote procedure definition 4-4
simple types 4-20
special case declarations 4-21
string type 4-21
structures 4-16
stub versions 4-1
timeout changes 4-14
typedef 4-18
unions 4-17
voids 4-22

## S

select() 3-21
semantics and transports, RPC 2-2
sendto() 6-22
serialization, XDR 3-10
server procedures, mount 1-28
server procedures, NFS 1-15
server procedures, passing other info to 4-15
server, definition 2-1
server/client relationship, NFS 1-24
servers, stateless 1-7
service library routines, RPC 3-4
simple-declaration 4-20
size specifications for arrays 5-16
stateless servers 1-7
stream implementation in XDR 6-25
string type 4-21
strings, XDR 5-7, 6-11
structures 4-16, 5-9
XDR, sizes of 1-10
stub versions 4-1
symbolic constants 4-18
syntax notes, RPC 2-18
syntax of the RPC language 4-16

## T

TCP 3-36
timestamp, server determination of client 2-12
transports and semantics, RPC 2-2
typedef 4-18, 5-10, 5-11

## U

UDP 8K warning 3-7
unions 4-17
UNIX authentication 3-27
UNIX authentication, RPC 2-10

## V

valdat 7-7
variable-array-declaration 4-20
verifiers, RPC 2-5
versions 3-34
void declaration 4-22

## W

writeproc() 6-23

## X

x_destroy() 6-25
x_getbytes() 6-26
x_getlong() 6-26
x_getpostn() 6-25
x_putbytes() 6-26
x_putlong() 6-26
x_setpostn() 6-25
XDR 1-7, 5-1, 6-1, 7-2
array, fixed length 5-8
array, variable length 5-8
arrays 6-12
block size, basic 5-1
boolean 5-3
byte arrays 6-12
byte order rationale 5-13
canonical standard 6-6
collapsing mutually recursive routines 6-29
constant 5-10
constructed data type filters 6-11
data description example 5-17
data types 5-2
data typing rationale 5-13
data, optional 5-11
discriminated union 5-9
discussions 5-13
double-precision floating-point integer 5-5
enhancements, future areas for 5-12
enum and C equivalent 4-18
enumeration 5-2
enumeration filters 6-10
file encoding example 5-18
fixed sized arrays 6-16
fixed-length array 5-8
fixed-length opaque data 5-6
floating point filters 6-10
floating-point integer 5-4
hyper integer 5-3
integer 5-2
integer, double-precision floating point 5-5
integer, floating point 5-4
integer, hyper 5-3
integer, unsigned 5-2
justification 6-2
justifications 5-13
keywords, list of 5-16
language rationale 5-13
lexical notes 5-14
library 6-7
library primitives 6-9
linked lists 6-27
memory allocation 3-17
memory allocation, example 3-17
memory streams 6-22
no data 6-10
non-filter primitives 6-21
notational conventions 5-14
number filters 6-9
object 6-25
opaque data 6-16
opaque data, fixed length 5-6
opaque data, variable length 5-6
operation directions 6-21
optional data 5-11
pointer semantics 6-20
pointers 6-19
portable data 6-2
protocol description file 4-8
reader and writer revisions 6-4
record (TCP/IP) streams 6-23
standard I/O streams 6-22
standardizing data representations 6-6
stream access 6-22
stream implementation 6-25
string 5-7
strings 6-11
structure 5-9
structures, sizes of 1-10
syntax 5-15, 5-16
typedef 5-10
union 5-9
unit size rationale 5-13
unsigned integer 5-2
variable-length array 5-8
variable-length data padding 5-13
variable-length opaque data 5-6
void 5-10
XDR library

constructed data type filters 6-11
discriminated unions 6-17
enumeration filters 6-10
floating point filters 6-10
no data 6-10
number filters 6-9
strings 6-11
xdr_array() 3-11, 6-12
xdr_bytes() 6-12
XDR_DECODE 6-11, 6-21, 6-25, 6-29
xdr_destroy() 6-21
xdr_element() 6-12
XDR_ENCODE 6-11, 6-21, 6-25, 6-29
XDR_FREE 6-11, 6-21, 6-25, 6-29
xdr_getpos() 6-21
xdr_long() 6-7
xdr_opaque() 6-16
xdr_pointer() 6-20
xdr_reference() 6-19, 6-20, 6-30
xdr_reference(), program example 3-12
xdr_setpos() 6-21
xdr_string() 6-11
xdr_string(), program example 3-12
xdr_vector() 3-12, 6-16
xdrmem_create() 6-22
xdrrec_create() 6-23
xdrrec_endofrecord() 6-24
xdrrec_eof() 6-24
xdrrec_skiprecord() 6-24
xdrstdio_create() 6-7, 6-22

# Y

Yellow Pages (YP) 7-1
YP
binder basic data structures 7-14
binder protocol definition 7-13
binder remote procedures 7-15
binders 7-13
data structures 7-7
database server protocol definition 7-6
database server remote procedures 7-9, 7-10, 7-11, 7-12
database servers 7-4
domains 7-5
map operations 7-4
map propagation and consistency 7-5
restrictions 7-5
RPC 7-2
terms 7-1
XDR 7-2
ypbind_binding 7-14
ypbind_resp 7-15
ypbind_resptype 7-14
ypbind_setdom 7-15
ypbinderr 7-14
ypmap_parms 7-8
ypmaplist 7-9
ypreq_xfr 7-8
ypresp_all 7-9
ypresp_key_val 7-8
ypresp_maplist 7-9
ypresp_master 7-8
ypresp_order 7-8
ypresp_val 7-8
ypresp_xfr 7-9
ypstat 7-6
ypxfrstat 7-7