X/Open Developers' Specification

Protocols for X/Open PC Interworking: (PC)NFS

X/Open Company, Ltd.

© *1990, X/Open Company Limited*

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.

X/Open Developers' Specification

Protocols for X/Open PC Interworking: (PC)NFS

ISBN: 1 872630 00 6
X/Open Document Number: XO/DEV/90/030

Set in Palatino by X/Open Company Ltd., U.K.
Printed by Maple Press, U.K.
Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.co.uk

Contents

PROTOCOLS FOR X/OPEN PC INTERWORKING: (PC)NFS

Chapter 1 INTRODUCTION

Chapter 2 INTRODUCTION TO NFS

2.1 OVERVIEW

2.2 AUDIENCE

2.3 PROTOCOL STACKS

2.4 REFERENCES

Chapter 3 XDR PROTOCOL SPECIFICATION

3.1 INTRODUCTION
3.1.1 A Canonical Standard
3.1.2 Byte Encoding
3.1.3 Basic Block Size

3.2 XDR DATA TYPES
3.2.1 Integer
3.2.2 Unsigned Integer
3.2.3 Enumeration
3.2.4 Boolean
3.2.5 Fixed-length Opaque Data
3.2.6 Variable-length Opaque Data
3.2.7 String
3.2.8 Fixed-length Array
3.2.9 Variable-length Array
3.2.10 Structure
3.2.11 Discriminated Union
3.2.12 Void
3.2.13 Constant
3.2.14 Typedef
3.2.15 Optional-data

3.3 The XDR LANGUAGE SPECIFICATION
3.3.1 Notational Conventions
3.3.2 Lexical Notes
3.3.3 Syntax Information
3.3.3.1 Syntax Notes

3.4 AN EXAMPLE OF AN XDR DATA DESCRIPTION

3.5 REFERENCES

Chapter 4 REMOTE PROCEDURE CALLS: PROTOCOL SPECIFICATION

4.1 INTRODUCTION
4.1.1 Terminology
4.1.2 The RPC Model
4.1.3 Transports and Semantics
4.1.4 Binding and Rendezvous Independence

4.2 RPC PROTOCOL REQUIREMENTS
4.2.1 Programs and Procedures
4.2.2 Authentication
4.2.3 Program Number Assignment

4.3 THE RPC MESSAGE PROTOCOL

4.4 AUTHENTICATION PROTOCOLS
4.4.1 Null Authentication
4.4.2 UNIX Authentication

4.5 THE RPC LANGUAGE
4.5.1 The RPC Language Specification
4.5.2 An Example Service Described in the RPC Language
4.5.3 Syntax Notes

4.6 REFERENCES

Chapter 5 NETWORK FILE SYSTEM: PROTOCOL SPECIFICATION

5.1 INTRODUCTION
5.1.1 Remote Procedure Call
5.1.2 External Data Representation
5.1.3 Stateless Servers and Idempotency

5.2 NFS PROTOCOL DEFINITION
5.2.1 File System Model

5.3 RPC INFORMATION
5.3.1 Sizes of XDR Structures
5.3.2 Basic Data Types
5.3.2.1 stat
5.3.2.2 ftype
5.3.2.3 nfscookie
5.3.2.4 fhandle
5.3.2.5 timeval
5.3.2.6 diropok
5.3.2.7 fattr
5.3.2.8 sattr
5.3.2.9 filename
5.3.2.10 path
5.3.2.11 attrstat
5.3.2.12 diropargs
5.3.2.13 diropres

5.4 NFS IMPLEMENTATION ISSUES

5.5 SERVER PROCEDURES
5.5.1 NFSPROC_NULL Specification - Do Nothing
5.5.1.1 RPC Data Descriptions
5.5.1.2 RPC Procedure Description
5.5.1.3 Description
5.5.1.4 Return Codes
5.5.2 NFSPROC_GETATTR Specification - Get File Attributes
5.5.2.1 RPC Data Descriptions
5.5.2.2 RPC Procedure Description
5.5.2.3 Description
5.5.2.4 Return Codes
5.5.3 NFSPROC_SETATTR Specification - Set File Attributes
5.5.3.1 RPC Data Descriptions
5.5.3.2 RPC Procedure Description
5.5.3.3 Description
5.5.3.4 Return Codes
5.5.4 NFSPROC_ROOT Specification - Get File System Root
5.5.4.1 RPC Data Descriptions
5.5.4.2 RPC Procedure Description
5.5.4.3 Description
5.5.4.4 Return Codes
5.5.5 NFSPROC_LOOKUP Specification - Look Up File Name
5.5.5.1 RPC Data Descriptions
5.5.5.2 RPC Procedure Description
5.5.5.3 Description
5.5.5.4 Return Codes
5.5.6 NFSPROC_READLINK Specification - Read From Symbolic Link
5.5.6.1 RPC Data Descriptions
5.5.6.2 RPC Procedure Description
5.5.6.3 Description
5.5.6.4 Return Codes
5.5.7 NFSPROC_READ Specification - Read From File
5.5.7.1 RPC Data Descriptions
5.5.7.2 RPC Procedure Description
5.5.7.3 Description
5.5.7.4 Return Codes
5.5.8 NFSPROC_WRITECACHE Specification - Write to Cache
5.5.8.1 RPC Data Descriptions
5.5.8.2 RPC Procedure Description
5.5.8.3 Description
5.5.8.4 Return Codes
5.5.9 NFSPROC_WRITE Specification - Write to File
5.5.9.1 RPC Data Descriptions
5.5.9.2 RPC Procedure Description

5.5.9.3 Description
5.5.9.4 Return Codes
5.5.10 NFSPROC_CREATE Specification - Create File
5.5.10.1 RPC Data Descriptions
5.5.10.2 RPC Procedure Description
5.5.10.3 Description
5.5.10.4 Return Codes
5.5.11 NFSPROC_REMOVE Specification - Remove File
5.5.11.1 RPC Data Descriptions
5.5.11.2 RPC Procedure Description
5.5.11.3 Description
5.5.11.4 Return Codes
5.5.12 NFSPROC_RENAME Specification - Rename File
5.5.12.1 RPC Data Descriptions
5.5.12.2 RPC Procedure Description
5.5.12.3 Description
5.5.12.4 Return Codes
5.5.13 NFSPROC_LINK Specification - Create Link to File
5.5.13.1 RPC Data Descriptions
5.5.13.2 RPC Procedure Description
5.5.13.3 Description
5.5.13.4 Return Codes
5.5.14 NFSPROC_SYMLINK Specification - Create Symbolic Link
5.5.14.1 RPC Data Descriptions
5.5.14.2 RPC Procedure Description
5.5.14.3 Description
5.5.14.4 Return Codes
5.5.15 NFSPROC_MKDIR Specification - Create Directory
5.5.15.1 RPC Data Descriptions
5.5.15.2 RPC Procedure Description
5.5.15.3 Description
5.5.15.4 Return Codes
5.5.16 NFSPROC_RMDIR Specification - Remove Directory
5.5.16.1 RPC Data Descriptions
5.5.16.2 RPC Procedure Description
5.5.16.3 Description
5.5.16.4 Return Codes
5.5.17 NFSPROC_READDIR Specification - Read From Directory
5.5.17.1 RPC Data Descriptions
5.5.17.2 RPC Procedure Description
5.5.17.3 Description
5.5.17.4 Return Codes
5.5.18 NFSPROC_STATFS Specification - Get File System Attributes
5.5.18.1 RPC Data Descriptions
5.5.18.2 RPC Procedure Description
5.5.18.3 Description
5.5.18.4 Return Codes

Chapter 6 ADJUNCT PROTOCOLS

6.1 INTRODUCTION

6.2 PORT MAPPER PROGRAM PROTOCOL
6.2.1 Introduction to Port Mapper Program Protocol
6.2.2 Port Mapper Protocol Specification (in RPC Language)
6.2.3 Port Mapper Procedures
6.2.4 PMAPPROC_NULL Specification - Do Nothing
6.2.4.1 RPC Data Descriptions
6.2.4.2 RPC Procedure Descriptions
6.2.4.3 Description
6.2.5 PMAPPROC_SET Specification - Set Mapping
6.2.5.1 RPC Data Descriptions
6.2.5.2 RPC Procedure Descriptions
6.2.5.3 Description
6.2.6 PMAPPROC_UNSET Specification - Unset Mapping
6.2.6.1 RPC Data Descriptions
6.2.6.2 RPC Procedure Descriptions
6.2.6.3 Description
6.2.7 PMAPPROC_GETPORT Specification - Get Port
6.2.7.1 RPC Data Descriptions
6.2.7.2 RPC Procedure Descriptions
6.2.7.3 Description
6.2.8 PMAPPROC_DUMP Specification - Dump Mappings
6.2.8.1 RPC Data Descriptions
6.2.8.2 RPC Procedure Descriptions
6.2.8.3 Description

6.3 PCNFSD PROTOCOL DEFINITION
6.3.1 Authentication
6.3.2 Print Spooling

6.4 RPC Information
6.4.1 Sizes of XDR Structures
6.4.2 Basic Data Types
6.4.2.1 ident
6.4.2.2 password
6.4.2.3 client
6.4.2.4 printername
6.4.2.5 username
6.4.2.6 spoolname
6.4.2.7 options
6.4.2.8 arstat
6.4.2.9 pirstat
6.4.2.10 psrstat
6.4.3 PCNFSD Server Procedures
6.4.4 PCNFSD_NULL Specification - Do Nothing
6.4.4.1 RPC Data Descriptions
6.4.4.2 RPC Procedure Description

6.4.4.3 Description
6.4.4.4 Return Codes
6.4.5 PCNFSD_AUTH Specification - Perform User Authentication
6.4.5.1 RPC Data Descriptions
6.4.5.2 RPC Procedure Description
6.4.5.3 Description
6.4.5.4 Return Codes
6.4.6 PCNFSD_PR_INIT Specification - Initialise Remote Printing
6.4.6.1 RPC Data Descriptions
6.4.6.2 RPC Procedure Description
6.4.6.3 Description
6.4.6.4 Return Codes
6.4.7 PCNFSD_PR_START Specification - Print a Spooled File
6.4.7.1 RPC Data Descriptions
6.4.7.2 RPC Procedure Description
6.4.7.3 Description
6.4.7.4 Return Codes

6.5 MOUNT PROTOCOL DEFINITION
6.5.1 Introduction

6.6 RPC Information
6.6.1 Sizes of XDR Structures
6.6.2 Basic Data Types
6.6.2.1 fhandle
6.6.2.2 fhstatus
6.6.2.3 dirpath
6.6.2.4 name
6.6.3 Server Procedures
6.6.4 MNTPROC_NULL Specification - Do Nothing
6.6.4.1 RPC Data Descriptions
6.6.4.2 RPC Procedure Description
6.6.4.3 Description
6.6.4.4 Return Codes
6.6.5 MNTPROC_MNT Specification - Add Mount Entry
6.6.5.1 RPC Data Descriptions
6.6.5.2 RPC Procedure Description
6.6.5.3 Description
6.6.5.4 Return Codes
6.6.6 MNTPROC_DUMP Specification - Return Mount Entries
6.6.6.1 RPC Data Descriptions
6.6.6.2 RPC Procedure Description
6.6.6.3 Description
6.6.6.4 Return Codes
6.6.7 MNTPROC_UMNT Specification - Remove Mount Entries
6.6.7.1 RPC Data Descriptions
6.6.7.2 RPC Procedure Description
6.6.7.3 Description
6.6.7.4 Return Codes

6.6.8 MNTPROC_UNMNTALL Specification - Remove all Mount Entries
6.6.8.1 RPC Data Description
6.6.8.2 RPC Procedure Descriptions
6.6.8.3 Return Codes
6.6.9 MNTPROC_EXPORT Specification - Return Export List
6.6.9.1 RPC Data Descriptions
6.6.9.2 RPC Procedure Description
6.6.9.3 Description
6.6.9.4 Return Codes

6.7 NETWORK LOCK MANAGER PROTOCOL DEFINITION
6.7.1 Introduction
6.7.2 Versions
6.7.2.1 Versions 1 and 2
6.7.2.2 Version 3
6.7.3 Synchronisation of Lock Managers
6.7.4 DOS File Sharing Support
6.7.5 RPC Information
6.7.5.1 Sizes of XDR Structures
6.7.5.2 Basic Data Types for Locking
6.7.6 DOS 3.1 File Sharing
6.7.6.1 fsh_mode
6.7.6.2 fsh_access
6.7.6.3 nlm_share
6.7.6.4 nlm_shareargs
6.7.6.5 nlm_shareres
6.7.6.6 nlm_notify
6.7.7 Server Procedures
6.7.8 NLM_NULL Specification - Do Nothing
6.7.8.1 RPC Data Descriptions
6.7.8.2 RPC Procedure Description
6.7.8.3 Description
6.7.8.4 Return Codes
6.7.9 NLM_UNLOCK Specification - Unlock File
6.7.9.1 RPC Data Descriptions
6.7.9.2 RPC Procedure Description
6.7.9.3 Description
6.7.9.4 Return Codes
6.7.10 NLM_SHARE Specification - Share a File
6.7.10.1 RPC Data Descriptions
6.7.10.2 RPC Procedure Description
6.7.10.3 Description
6.7.10.4 Return Codes
6.7.11 NLM_UNSHARE Specification - Unshare a File
6.7.11.1 RPC Data Descriptions
6.7.11.2 RPC Procedure Description
6.7.11.3 Description
6.7.11.4 Return Codes

6.7.12 NLM_NM_LOCK Specification - Non-monitored Lock
6.7.12.1 RPC Data Descriptions
6.7.12.2 RPC Procedure Description
6.7.12.3 Description
6.7.12.4 Return Codes
6.7.13 NLM_FREE_ALL Specification - Free All
6.7.13.1 RPC Data Descriptions
6.7.13.2 RPC Procedure Description
6.7.13.3 Description
6.7.13.4 Return Codes

Chapter 7 RPC INTERFACE TO UDP TRANSPORT SERVICES

7.1 INTRODUCTION

7.2 RPC AND TRANSPORT REQUIREMENTS

7.3 UDP AS A TRANSPORT PROTOCOL

7.4 RPC INTERFACE
7.4.1 The RPC request
7.4.2 The RPC reply
7.4.3 Receiving a UDP Reply Packet
7.4.4 Closing

Appendix A MAPPING FILENAMES AND ATTRIBUTES

A.1 INTRODUCTION

A.2 CONTEXT

A.3 MAPPING FILE NAMES

A.4 BACK-MAPPING FILENAMES

A.5 MAPPING FILE ATTRIBUTES

A.6 BACK-MAPPING FILE ATTRIBUTES

Appendix B NFS TRANSMISSION ANALYSIS

B.1 INTRODUCTION

B.2 DOS FUNCTIONS

Appendix C DEFINITIONS

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring greater value to users through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable system environment, called the *Common Applications Environment (CAE)*. This environment covers all the standards, above the hardware level, that are needed to support open systems. It ensures portability and connectivity of applications, and allows users to move between systems without retraining.

The interfaces identified as components of the Common Applications Environment are defined in the *X/Open Portability Guide*. This guide contains an evolving portfolio of practical applications programming interface standards (APIs), which significantly enhance portability of application programs at the source code level. The interfaces defined in the X/Open Portability Guide are supported by an extensive set of conformance tests and a distinct trademark - the X/Open brand - that is carried only on products that comply with the X/Open definitions.

X/Open is thus primarily concerned with standards selection and adoption. The policy is to use formal approved *de jure* standards, where they exist, and to adopt widely supported *de facto* standards in other cases.

Where formal standards do not exist, it is X/Open policy to work closely with standards development organizations to encourage the creation of formal standards covering the needed functionalities, and to make its own work freely available to such organizations. Additionally, X/Open has a commitment to align its definitions with formal approved standards.

The X/Open Product Family - XPG

There is a single family of X/Open products, which has the generic name ''XPG''.

XPG Versions

There are different numbered versions of XPG within the XPG family (XPG1, XPG2, XPG3). Each XPG version is an integrated set of elements supporting the development, procurement and implementation of open systems products, and each comprises its own:

- XPG Specifications
- XPG Verification Suite
- XPG descriptive guides

- XPG trademark licensing materials

The XPG trademark (or ''brand'') licensed by X/Open always contains a particular XPG version number (e.g., ''XPG3'') and, when associated with a vendor's system, communicates clearly and unambiguously to a procurer that the software bearing the trademark correctly implements the corresponding XPG specifications. Users specifying particular XPG versions in their procurements are therefore certain as to the XPG specifications to which vendors' systems conform.

XPG Specifications

There are four types of XPG specification:

- **XPG*n* Formal Specifications**

  These are the long-life XPG specifications that form the basis for conformant/branded X/Open systems, and are the only type of XPG specification released with an XPG version number (e.g., ''XPG3''). They are intended to be used widely within the industry for product development and procurement purposes. Currently, all XPG Formal Specifications are included in Issue 3 of the X/Open Portability Guide.

  Individual XPG specifications are released as Formal Specifications only as part of the formal release of the complete XPG version to which they belong. However, prior to the launch of that XPG version, they may be made available as:

- **XPG Developers' Specifications**

  These are specifically designed to allow developers to create X/Open-compliant products and applications in advance of the formal launch of a future version of the XPG.

  Developers' Specifications may be relied on by product developers as the final, base specification that will appear in a future XPG. They are made available beforehand in order to meet the need of product developers for advance notification of the contents of XPG Formal Specifications, to assist in their product planning and development activities.

  By providing such advance notification, X/Open makes it possible for products conforming to future XPG Formal Specifications to be developed as soon as practicable, enhancing the value of XPG itself as a procurement aid to users.

- **XPG Preliminary Specifications**

  These are XPG specifications, usually addressing an emerging area of technology, and consequently not yet supported by a base of conformant product implementations, that are released in a controlled manner for validation purposes. A Preliminary Specification is not a ''draft'' specification. Indeed, it is as stable as X/Open can make it, and on publication will have gone through the same rigorous X/Open development and review procedures as XPG Formal and Developers' Specifications.

  Preliminary Specifications are analogous with the ''trial-use'' standards issued by formal standards organizations, and product development teams are intended to develop product on the basis of them. Because of the nature of the technology they are addressing, they are untried in practice, and they may therefore change before being published as an XPG Formal or Developers' Specification.

- **Snapshot Specifications**

  These are ‘‘draft’’ documents, that provide a mechanism for X/Open to disseminate information on its current direction and thinking to a limited audience, in advance of formal publication, with a view to soliciting feedback and comment.

This Document

This document is a Developers’ Specification (see above), which defines protocols that have been adopted by X/Open as the means of interoperability between personal computers and X/Open-compliant systems.

For interoperability via asynchronous serial links, X/Open has already defined in XPG3 a file transfer protocol and a set of features provided on X/Open-compliant systems for terminal emulators. The protocols defined in this document are aimed at interoperability over local area networks (LANs).

X/Open has defined two protocols for PC interworking:

- (PC)NFS is intended for use in situations where personal computers are added to existing networks of X/Open-compliant systems which already have a distributed file system (the most widely adopted one being the Network File System (NFS) originally developed by Sun Microsystems).
- Server Message Block (SMB) is intended for use in situations where X/Open-compliant systems are added to an existing LAN consisting primarily of personal computers. (For personal computers running under DOS or OS/2, which is the vast majority, SMB is the generally accepted non-proprietary protocol.)

This document contains the X/Open (PC)NFS protocol.

Trademarks

AT&T is a registered trademark of American Telephone & Telegraph Company in the USA and other countries.

Diablo is a registered trademark of Xerox Corporation.

Ethernet is a registered trademark of Xerox Corporation.

IBM is a registered trademark of International Business Machines Corporation.

Lan Manager is a trademark of Microsoft Corporation.

MS-DOS is a registered trademark of Microsoft Corporation.

NFS is a registered trademark of Sun Microsystems, Inc.

OS/2 is a registered trademark of International Business Machines Corporation.

PC-NFS is trademark of Sun Microsystems, Inc.

PostScript is a registered trademark of Adobe Systems, Inc.

Sun Microsystems is a registered trademark of Sun Microsystems, Inc.

UNIX is a registered trademark of UNIX System Laboratories, Inc. in the USA and other countries.

VAX is a registered trademark of Digital Equipment Corporation.

VMS is a trademark of Digital Equipment Corporation.

X/Open is a trademark of the X/Open Company Ltd in the UK and other countries.

Acknowledgements

X/Open gratefully acknowledges Karl Auerbach and Avnish Aggarwal for permission to reprint RFCs 1001 and 1002.

Chapter 1

Introduction

Of all the types of computers, personal computers are the most abundant. Originally intended to be a personal productivity tool, an ever increasing number of them are being connected to computer networks, thus becoming parts of distributed information systems.

Personal computers normally run under single-user operating systems with interfaces differing from those specified in the **X/Open Portability Guide**. However, X/Open realises how important it is to facilitate interworking between personal computers and X/Open-compliant systems in a standardised way.

Two areas have to be addressed to achieve this goal; interoperability, and programming interfaces to server functions facilitating applications portability. Interoperability means that personal computers and X/Open-compliant systems can interchange information using the same network protocols. Standardisation of programming interfaces to server functions, in addition to standardisation of protocols, makes it possible to write Distributed Client/Server applications whose server component will be portable to all X/Open-compliant systems.

For interoperability via asynchronous serial links, X/Open has already defined in the **X/Open Portability Guide, Issue 3** a file transfer protocol and a set of features provided on X/Open-compliant systems for terminal emulators. Now it is time to address interworking in local area networks (LANs).

In the X/Open Developers' Specifications of the (PC)NFS and SMB protocols, interoperability of personal computers and X/Open-compliant systems is addressed. The applications portability component, containing definitions of programmatic interfaces to server functions, will appear in a future document.

When connecting personal computers and X/Open-compliant systems via standard transport protocols, there appear to be two possibly overlapping but distinct market segments. In the first one, personal computers are added to existing networks of X/Open-compliant systems which already have a distributed file system, the most widely adopted one being the Network File System originally designed by Sun Microsystems. In the second one, X/Open-compliant servers are added to LANs consisting primarily of personal computers. For personal computers running under DOS or OS/2 operating systems, which is the vast majority, the generally accepted non-proprietary protocol is the Server Message Block from Microsoft Corporation.

Therefore, for connecting personal computers to X/Open-compliant systems, both the (PC)NFS and the SMB protocols have been adopted by X/Open.

The resources accessed on the X/Open-compliant server are local to that system since X/Open has not yet specified a resource sharing method between X/Open-compliant systems. (PC)NFS and SMB protocol implementations on the same server are not required to interwork with respect to those additional features beyond those provided by XSI (e.g., extended DOS file open modes). The (PC)NFS and SMB protocols are only specified to run on a single LAN subnetwork, but interoperation in connected sub-networks is not precluded.

The following diagram illustrates the relationship of the service protocols (defined in the (PC)NFS and SMB specifications) to their underlying transport protocols. It also reflects the organisation of the two documents. The (PC)NFS specification describes the mandatory and optional protocols for NFS, RPC and XDR. The SMB specification describes the mandatory and optional protocols for SMB, the mapping of NetBIOS over an OSI transport (TOP/NetBIOS) and the mapping of NetBIOS over an Internet Protocol Suite transport (RFC1001/RFC1002).

<table>
<tr><td colspan="4" rowspan="2">SMB</td><td>NFS</td></tr>
<tr><td rowspan="2">RPC/XDR</td></tr>
<tr><td colspan="2">TOP/NetBIOS</td><td colspan="2">RFC 1001/1002</td></tr>
<tr><td rowspan="2">Connection-less
Transport
Services

Defined
outside
the XPG</td><td rowspan="2">Connection
orientated
Transport
Services
(T6212)

Defined
outside
the XPG</td><td>TCP

Defined
outside
the XPG</td><td colspan="2">UDP

Defined
outside
the XPG</td></tr>
<tr><td colspan="3">IP

Defined outside the XPG</td></tr>
</table>

Since SMB and NFS protocols do not easily map onto the seven layer OSI Reference Model, the diagram does not use it.

The text in this volume is divided into the specification itself and a number of appendices. Compliant implementations will provide functionality as defined by the specification. The text in the appendices is auxiliary information for implementors and is non-normative.

Throughout the specification DOS is used as an abbreviation for the Personal Computer's operating system instead of MS-DOS or PC DOS.

Chapter 2

Introduction to NFS

2.1 OVERVIEW

This Developers' Specification describes the Network File System (NFS) architecture and its applicability to interoperability for X/Open-compliant systems. The Network File System (NFS) is an operating system-independent service that allows users to mount directories across the network, and then to treat those directories as if they are local.

This specification contains protocol specifications for External Data Representation (XDR), Remote Procedure Call Protocol (RPC), the Network File System (NFS), and the NFS adjunct protocols which include the Portmap, PCNFSD, Mount and Network Lock Manager (NLM).

The XDR specification is included because this specification defines the NFS protocol in terms of XDR. Generally implementations of NFS make use of XDR, but this is not a requirement, merely one implementation strategy.

The RPC specification is included because the NFS protocols are implemented on top of it. This specification does not define any general RPC protocols or interfaces for distributed client-server applications. Such an RPC specification will appear in a future issue.

Although full NFS is operating system-independent, this specification describes only those parts required to implement a server for a single-user system client. Specific attention is given to issues unique to PC interworking. These include:

- authentication;
- the addition of a remote print spooling mechanism;
- server-only role, and
- DOS file sharing and locking support.

2.2 AUDIENCE

The description of NFS is targeted for those wishing to provide an NFS server on an X/Open-compliant system for a DOS client system.

2.3 PROTOCOL STACKS

In order to support NFS as a basis for PC Interworking, a server system must implement the following protocols:

- NFS (**Chapter 5**, **Network File System: Protocol Specification**)
- Portmap (**Section 6.2**, **Port Mapper Program Protocol**)
- PCNFSD (**Section 6.3**, **PCNFSD Protocol Definition**)
  (Not required, but highly desirable)
- Mount (**Section 6.4**, **Mount Protocol Definition**)
- NLM (**Section 6.5**, **Network Lock Manager Protocol Definition**)

These protocols are implemented on top of the Remote Procedure Call (**RPC**) protocol, which employs the External Data Representation (**XDR**) encoding to ensure operating system independence.

Although the RPC protocol is inherently independent of any particular transport service, existing implementations are generally based upon the **Internet Protocols**, popularly referred to as the TCP/IP suite. Since a major goal of this specification is to foster interoperability based upon *de facto* standards, this description is limited to the use of NFS and RPC over the Internet Protocols. The NFS and portmap protocols are explicitly accessed via well-known UDP transport addresses.

This specification is restricted to the protocol stack which corresponds to those implementations which are commercially available and in use today.

2.4 REFERENCES

This specification describes only those protocols which have not otherwise been standardised.

The following documents should be consulted for details of the Internet Protocol suite.

RFC 1011 - Official Internet Protocols

This specification is the authoritative reference, showing which document defines each protocol. RFC 1011 was published in May 1987; the reader is advised to check the **RFC Index** (available from the sources listed below) and, if necessary, obtain the latest version of **Official Internet Protocols**. The descriptions which follow are derived from RFC 1011.

RFC 791 - Internet Protocol (IP)

This is the universal protocol of the Internet. This datagram protocol provides the universal addressing of hosts in the Internet.

RFC 792 - Internet Control Message Protocol (ICMP)

This protocol describes the control messages and error reports that go with the Internet Protocol. Note that ICMP is defined to be an integral part of IP. An implementation of IP is not complete if it does not include ICMP.

RFC 768 - User Datagram Protocol (UDP)

This protocol provides a datagram service to applications. It adds port addressing to the IP services.

RFC 793 - Transmission Control Protocol (TCP)

This protocol provides a reliable end-to-end data stream service. Note that RFC 1011 includes many additions and clarifications to RFC 793, and refers to additional RFCs which go into greater detail on certain topics.

RFC 950 - Internet Subnet Protocol

This specification specifies procedures for the use of subnets, which are logical sub-sections of a single Internet network.

RFC 826 - Address Resolution Protocol (ARP)

This is a procedure for finding the network hardware address corresponding to an Internet Address.

RFC 997 - Internet Numbers

This specification describes the fields of network numbers and autonomous system numbers that are assigned specific values for actual use, and lists the currently assigned values.

RFC 1010 - Assigned Numbers

This specification describes the fields of various protocols that are assigned specific values for actual use, and lists the currently assigned values.

RFC 894 - Internet Protocol on Ethernet Networks

Describes the representation of Internet Protocol services on Ethernet networks.

RFC 1011 (includes) Internet Protocol on IEEE 802

This section of RFC 1011 describes the latest policy on the transmission of IP datagrams on IEEE 802 networks.

In addition, XDR, RPC and NFS are described in the following RFCs:

RFC 1014 - XDR: External Data Representation Standard

RFC 1057 - RPC: Remote Procedure Call Protocol Specification Version 2

This specification has the status of a proposal only, and includes the disclaimer that ‘‘This memo is not an Internet standard at this time’’.

RFC 1094 - NFS: Network File System Protocol Specification

For further information about Internet Protocols in general, please contact:

Joyce K. Reynolds
USC - Information Sciences Institute,
4676 Admiralty Way,
Marina del Rey,
CA 90292-6695,
U.S.A.

Phone: (+1) 213-822-1511

Electronic mail: jkreynolds@isi.edu

Chapter 3

XDR Protocol Specification

This chapter specifies a protocol that is used by many implementors of NFS. It is derived from a document designated **RFC1014** by the ARPA Network Information Centre.

This chapter includes only the subset of XDR that is required to define the NFS protocol.

3.1 INTRODUCTION

XDR 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 many diverse machines. XDR fits into the ISO presentation layer, and is roughly analogous in purpose to X.209 (previously X.409), ISO Abstract Syntax Notation. The major difference between these two is that XDR uses implicit typing, while X.209 uses explicit typing.

XDR uses a language to describe data formats. The language can only be used to describe data; it is not a programming language. This language allows description of 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 [**ref 1**], just as Courier [**ref 3**] is similar to Mesa. Protocols such as RPC (Remote Procedure Call) and the NFS (Network File System) use XDR to describe the format of their data.

The XDR standard assumes 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 [**ref 2**], or least significant bit first. A data-description language is used to define XDR rather than diagrams, as languages are more formal than diagrams and lead to less ambiguous descriptions of data. There is also 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. The language specification itself is an ASCII string that can be passed from machine to machine to perform on-the-fly data interpretation.

3.1.1 A Canonical Standard

XDR’s approach to standardising data representations is *canonical*. That is, XDR defines a single byte order (big-endian [**ref 2**]), 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.

No data-typing is provided in the XDR language as it has a relatively high cost (encoding and interpreting the type fields) and most protocols already know what data types they

are expecting. 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, describe the corresponding data type.

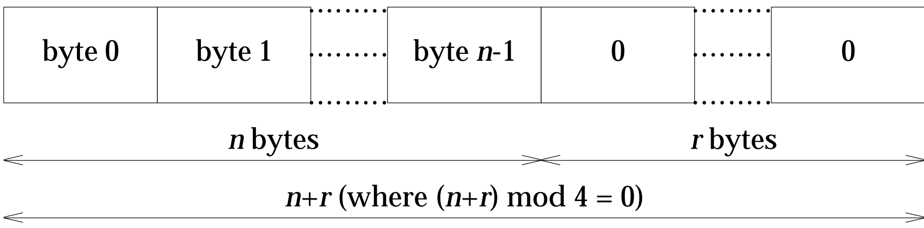
3.1.2 Byte Encoding

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 with the least significant bit first.

3.1.3 Basic Block Size

The representation of all items requires a multiple of four bytes (or 32 bits) of data. Four bytes is big enough to support most machine architectures efficiently, yet is small enough to keep the encoded data to a reasonable size. The bytes are numbered 0 to n-1. The bytes are read or written to a 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. Setting these residual bytes to zero enables the same data to be encoded to the same result on all machines, allowing encoded data to be meaningfully compared or checksummed.

A Block



3.2 XDR DATA TYPES

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

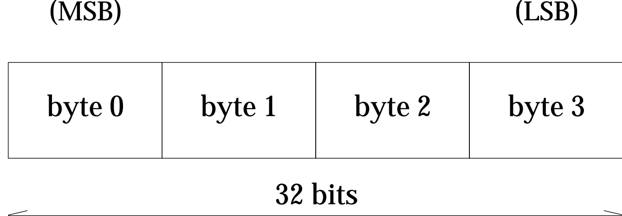
For each data type in the language a general paradigm declaration is shown. Note that angle brackets (< and >) denote variable length sequences of data and square brackets ([and]) denote fixed-length sequences of data. *n*, *m* and *r* denote integers. For the full language specification and more formal definitions of terms such as ‘‘identifier’’ and ‘‘declaration’’, refer to **Section 3.4**, **The XDR Language Specification**.

For some data types, more specific examples are included. A more extensive example of a data description is in **Section 3.5**, **An Example of an XDR Data Description**.

3.2.1 Integer

An XDR signed integer is a 32-bit datum that encodes an integer in the range [-2147483648,2147483647]. The integer is represented in two’s complement notation. The most and least significant bytes are 0 and 3, respectively. Integers are declared as follows:

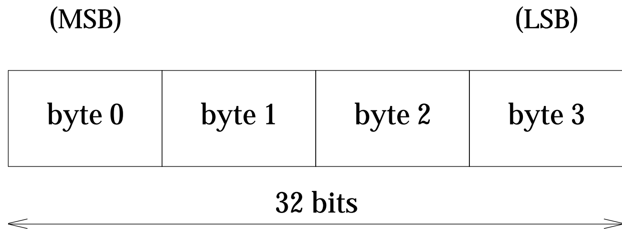
int identifier;



3.2.2 Unsigned Integer

An XDR unsigned integer is a 32-bit datum that encodes a non-negative integer in the range [0,4294967295]. It is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively. An unsigned integer is declared as follows:

unsigned int identifier;



3.2.3 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 colours 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 integer other than those that have been given assignments in the **enum** declaration.

3.2.4 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:

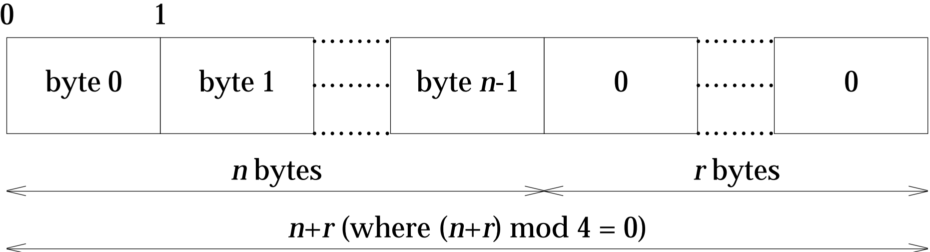
enum { FALSE = 0, TRUE = 1 } identifier;

3.2.5 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.



3.2.6 Variable-length Opaque Data

The standard also provides for variable-length (counted) opaque data, defined as a sequence of *n* (numbered 0 to *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). 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. Variable-length opaque data is declared in the following way:

opaque identifier<*m*>;

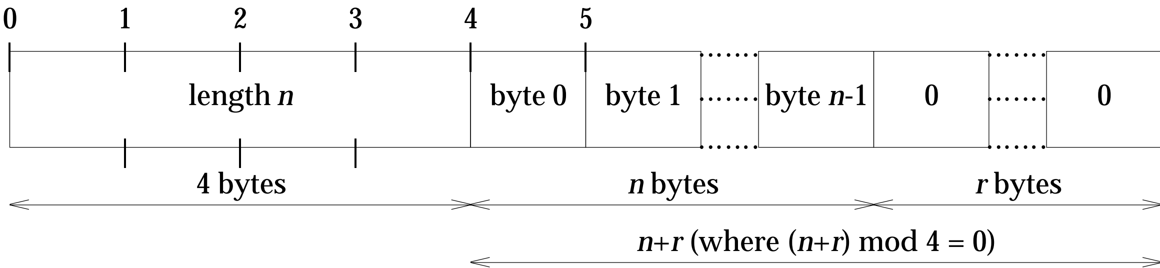
or

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:



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

3.2.7 String

The standard defines a string of *n* (numbered 0 to *n*-1) ASCII bytes to be the number *n* encoded as an unsigned integer (as described above), and followed by the *n* bytes of the string. Byte *m* of the string always precedes byte *m*+1 of the string, and byte 0 of the string always follows the string's length. If *n* is not a multiple of 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*>;

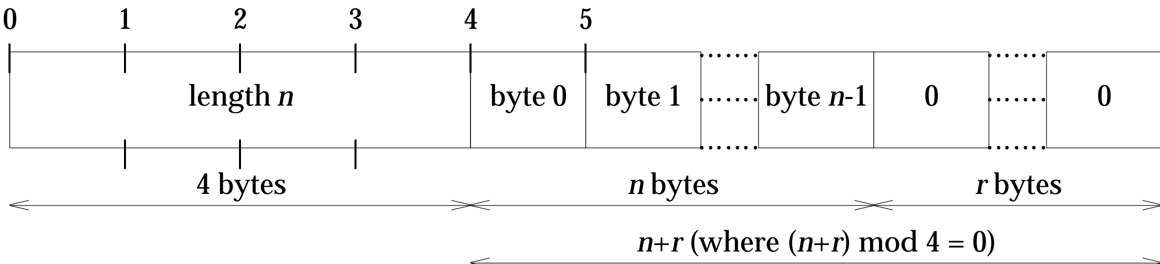
or

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* would normally be found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than 255 bytes, as follows:

string filename<255>;

This can be illustrated as:



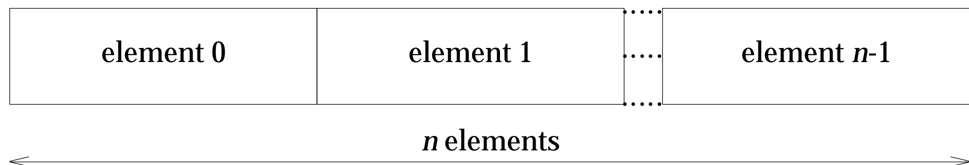
It is an error to encode a length greater than the maximum defined in the declaration.

3.2.8 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 to *n*-1, are encoded by individually encoding the elements of the array in their natural order, 0 to *n*-1. Each element's size is a multiple of four bytes. Though all elements are of the same type, the elements may have different sizes. For example, in a fixed-length array of strings, all elements are of type **string**, yet each element will vary in its length.



3.2.9 Variable-length Array

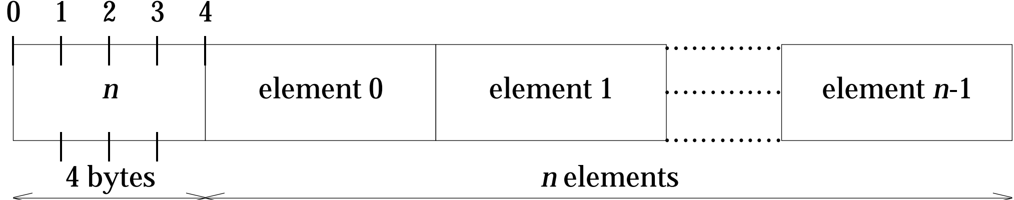
Counted arrays provide the ability to encode variable-length arrays of homogeneous elements. The array is encoded as the element count *n* (an unsigned integer) followed by the encoding of each of the array's elements, starting with element 0 and progressing through element *n*-1. The declaration for variable-length arrays follows this form:

type-name identifier<*m*>;

or

type-name identifier<>;

The constant *m* specifies the maximum acceptable element count of an array; if m is not specified, as in the second declaration, it is assumed to be 2^{32} -1.



It is an error to encode a value of *n* that is greater than the maximum described in the declaration.

3.2.10 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, though the components may be different sizes.

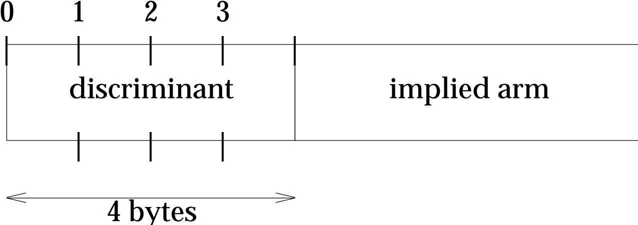
3.2.11 Discriminated Union

A discriminated union is a type composed of a discriminant followed by a type selected from a set of pre-arranged 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.

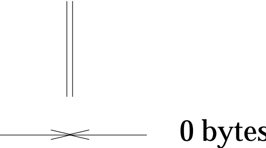


3.2.12 Void

An XDR **void** is a 0-byte quantity. **void**s are useful for describing operations that take no data as input or no data as output. They are also useful in **union**s, where some arms may contain data and others do not. The declaration is simply as follows:

```
void;
```

voids are illustrated as follows:



3.2.13 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 may be used anywhere a regular constant may be used. For example, the following defines a symbolic constant DOZEN, equal to 12.

```
const DOZEN = 12;
```

3.2.14 Typedef

typedef does not declare any data either, but serves to define new identifiers for declaring data. The syntax is:

```
typedef declaration;
```

The new type name is actually the variable name in the declaration part of the **typedef**. For example, the following defines a new type called *eggbox* using an existing type called *egg*:

```
typedef egg eggbox[DOZEN];
```

Variables declared using the new type name have the same type as the new type name would have in the **typedef**, if it was considered a variable. For example, the following two declarations are equivalent in declaring the variable *fresheggs*:

```
eggbox  fresheggs;
egg     fresheggs[DOZEN];
```

When a **typedef** involves a **struct**, **enum**, or **union** definition, there is another (preferred) syntax that may be used to define the same type. In general, a **typedef** of the following form:

```
typedef <<struct, union, or enum definition>> identifier;
```

may be converted to the alternative form by removing the ''typedef'' part and placing the identifier after the **struct**, **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 second syntax is preferred because it is not necessary to wait until the end of a declaration to find the name of the new type.

3.2.15 Optional-data

Optional-data is one kind of union that occurs so frequently that it is given a special syntax of its own. 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 the way in which 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.

3.3 The XDR LANGUAGE SPECIFICATION

3.3.1 Notational Conventions

This specification uses an extended Backus-Naur Form notation for describing the XDR language. Here is a brief description of the notation:

1. The characters |, **(**, **)**, **[**, **]**, ", and * are special.
2. Terminal symbols are strings of any characters surrounded by double quotes (").
3. Non-terminal symbols are strings of non-special characters.
4. Alternative items are separated by a vertical bar (|).
5. Optional items are enclosed in brackets.
6. Items are grouped together by enclosing them in parentheses.
7. A * following an item means zero 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’’

3.3.2 Lexical Notes

1. Comments begin with /* and terminate with */.
2. White space serves to separate items and is otherwise ignored.
3. An identifier is a letter followed by an optional sequence of letters, digits or underbar ‘‘_’’. The case (lower or upper) of identifiers is not ignored.
4. A constant is a sequence of one or more decimal digits, optionally preceded by a minus-sign ‘‘-’’.

3.3.3 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"
        | "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 *
```

Although XDR is used by many implementations of NFS, it has been defined in this document as a tool for use in later chapters. No implementation of the XDR language is required by a server.

3.3.3.1 Syntax Notes

- The following are keywords and cannot be used as identifiers: **bool**, **case**, **const**, **default**, **enum**, **opaque**, **string**, **struct**, **switch**, **typedef**, **union**, **unsigned** and **void**.
- Only unsigned constants may be used as size specifications for arrays. If an identifier is used, it must have been declared previously as an unsigned constant in a **const** definition.
- Constant and type identifiers within the scope of a specification are in the same name space and must be declared uniquely within this scope.
- Similarly, variable names must be unique within the scope of **struct** and **union** declarations. Nested **struct** and **union** declarations create new scopes.
- The discriminant of a **union** must be of a type that evaluates to an integer; that is, **int**, **unsigned int**, **bool**, an enumerated type or any **typedef**ed type that evaluates to one of these is legal. Also, the case values must be one of the legal values of the discriminant. Finally, a case value may not be specified more than once within the scope of a union declaration.

3.4 AN EXAMPLE OF AN XDR DATA DESCRIPTION

Here is a short XDR data description of an object called a *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’’ that contains just the data ‘‘(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 interpreter = 4
24	6c 69 73 70	lisp	Interpreter 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

If instead ‘‘john’’ stored the same file in the text file ‘‘sillytext’’, it 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 74 65 78	ytex	... and more characters ...
12	74 00 00 00	t...	... and 3 zero-bytes of fill
16	00 00 00 00	....	Filekind is TEXT = 0
			Note: no data encoded for void
20	00 00 00 04	....	Length of owner = 4
24	6a 6f 68 6e	john	Owner characters
28	00 00 00 06	....	Length of file data = 6
32	28 71 75 69	(qui	File data bytes ...
36	74 29 00 00	t)..	... and 2 zero-bytes of fill

3.5 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] ''Courier: The Remote Procedure Call Protocol'', XEROX Corporation, XSIS 038112, December 1981.

Chapter 4

Remote Procedure Calls: Protocol Specification

This chapter specifies a protocol that is used by many implementors of NFS. It is derived from a document designated **RFC1057** by the ARPA Network Information Centre.

4.1 INTRODUCTION

This chapter specifies a message protocol used in implementing a Remote Procedure Call (RPC) package. The message protocol is specified with the External Data Representation (XDR) language; see **Chapter 3**, **XDR Protocol Specification** for the details. It is assumed that the reader is familiar with XDR and no attempt is made to justify it or its uses. The paper by Birrell and Nelson [**ref 1**] is recommended as an excellent background to and justification of RPC.

4.1.1 Terminology

This chapter discusses servers, services, programs, procedures, clients and versions.

network 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.

network clients — pieces of software that initiate remote procedure calls to services.

A network server may support more than one version of a remote program in order to be forward compatible with changing protocols. For example, a network file service may be composed of two programs; one program may deal with high-level applications such as file system access control and locking; the other may deal with low-level file I/O and have procedures like *read* and *write*. A client machine of the network file service would call the procedures associated with the two programs of the service on behalf of a user on the client machine.

4.1.2 The RPC Model

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

In the remote case, 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 results of the procedure are extracted, and the caller's execution is resumed.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message. Note that in this model, only one of the two processes is active at any given time.

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

4.1.3 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 [**ref 2**], then most of the work is already done for it. If, however, it is running on top of an unreliable transport such as UDP/IP [**ref 3**], it must implement its 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 ensure 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 re-use its previous transaction ID when retransmitting a request. The server application, knowing this fact, may choose to remember this ID after granting a request and not re-grant 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.

However, 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 that although RPC is currently implemented on top of

both TCP/IP and UDP/IP transports, all known implementations of NFS are over connection-less protocols; therefore, RPC over connection-oriented protocols will not be discussed further in this specification.

4.1.4 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 **Section 6.2**, **Port Mapper Program Protocol**.)

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.

4.2 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, and
- provisions for authenticating the caller to service and *vice versa*.

Besides these requirements, RPC has features that detect the following:

- RPC protocol mismatches;
- remote program protocol version mismatches;
- protocol errors (such as incorrect specification of a procedure's parameters);
- reasons why remote authentication failed, and
- any other reasons why the desired procedure was not called.

4.2.1 Programs and Procedures

The RPC call message has three unsigned fields: remote program number (**Section 4.2.3, Program Number Assignment**), remote program version number, and remote procedure number. The three fields uniquely identify the procedure to be called. Program numbers are administered by some central authority. (Currently Sun Microsystems is responsible for administering program numbers.) Once an implementor has a program number, the remote program can be implemented; the first implementation would most likely have the version number of 1. Because most new protocols evolve into better, stable and mature protocols, a version field of the call message identifies which version of the protocol the caller is using. Version numbers make speaking old and new protocols through the same server process possible.

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

Just as remote program protocols may change over several versions, the actual RPC message protocol could also change. Therefore, the call message also contains the RPC version number, which is always 2 for the version of RPC described here.

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

- The remote implementation of RPC does not speak protocol version 2. The lowest and highest supported RPC version numbers are returned.
- The remote program is not available on the remote system.
- The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
- The requested procedure number does not exist. (This is usually a caller side protocol or programming error.)

- The parameters to the remote procedure appear to be uninterpretable from the server's point of view. (Again, this is usually caused by a disagreement about the protocol between client and service.)

4.2.2 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 4.4**, **Authentication Protocols**.

4.2.3 Program Number Assignment

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

Program Numbers	Description
0 - 1fffffff	Defined by Sun Microsystems
20000000 - 3fffffff	Defined by user
40000000 - 5fffffff	Transient
60000000 - 7fffffff	Reserved
80000000 - 9fffffff	Reserved
a0000000 - bfffffff	Reserved
c0000000 - dfffffff	Reserved
e0000000 - ffffffff	Reserved

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 generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

To obtain an assigned RPC program number contact:

RPC Administrator
Sun Microsystems
2550 Garcia Avenue,
Mountain View,
CA 94043,
U.S.A.

or via electronic mail:

rpc@sun.com

4.3 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 number */
        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 is not 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. N.B.: The xid
 * field may be used by clients to match reply messages with
 * call messages, or by servers detecting retransmissions; the
 * service side cannot treat this xid as any type of sequence
 * number.
 */
struct rpc_msg {
        unsigned int xid;
        union switch (msg_type mtype) {
                case CALL:
                        call_body cbody;
                case REPLY:
                        reply_body rbody;
        } body;
};

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

/*
 * Body of an RPC reply:
 * 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_ARGS
 * arms of the union are void.  The PROG_MISMATCH arm specifies
 * the lowest and highest version numbers of the remote program
 * supported by the server.
 */
struct accepted_reply {
        opaque_auth verf;
        union switch (accept_stat stat) {
                case SUCCESS:
                        opaque results[0];
                        /* procedure-specific results start here */
                case PROG_MISMATCH:
                        struct {
                                unsigned int low;
                                unsigned int high;
                        } mismatch_info;
                default:
                        /*
                        * Void.  Cases include PROG_UNAVAIL,
                        * PROC_UNAVAIL and GARBAGE_ARGS.
                        */
                        void;
        } reply_data;
};

/*
 * Reply to an RPC request that was rejected by the server:
 * The request can be rejected for two reasons: either the
 * server is not running a compatible version of the RPC
 * protocol (RPC_MISMATCH), or the server refuses to
 * authenticate the caller (AUTH_ERROR).  In case of an RPC
 * version mismatch, the server returns the lowest and highest
 * supported RPC version numbers.  In case of refused
 * authentication, failure status is returned.
 */
```

```
union rejected_reply switch (reject_stat stat) {
        case RPC_MISMATCH:
                struct {
                        unsigned int low;
                        unsigned int high;
                } mismatch_info;
        case AUTH_ERROR:
                auth_stat stat;
};
```

4.4 AUTHENTICATION PROTOCOLS

As previously stated, authentication parameters are opaque, but open-ended to the rest of the RPC protocol. This section defines two ‘‘flavours’’ of authentication. Other implementations are free to invent new authentication types, with the same rules of flavour number assignment as there are for program number assignment.

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,
        /* and more to be defined */
};

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

In other words, any *opaque_auth* structure is an *auth_flavor* enumeration followed by a sequence of 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.

If authentication parameters are rejected, the response message will contain information stating why they were rejected.

4.4.1 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 flavour 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.

4.4.2 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<8>;
};
```

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 (supplementary groups). An entry in the *gids* array whose value is 0xffffffff should be ignored. Although the supplementary group list *gids* is part of the NFS specification and is supported by (PC) NFS clients, it is only optional in the X/Open CAE. A server which does not implement supplementary groups may choose to ignore the *gids* field. The verifier accompanying the credentials (the *verf* field in *call_body* and *accept_reply*) should be of *AUTH_NULL* (defined above).

4.5 THE RPC LANGUAGE

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

4.5.1 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 ";"
```

4.5.2 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.

4.5.3 Syntax Notes

- The keywords **program** and **version** are added and cannot be used as identifiers.
- 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.

4.6 REFERENCES

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

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

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

Chapter 5

Network File System: Protocol Specification

This chapter specifies a protocol that Sun Microsystems and others are using. It is derived from a document designated **RFC1094** by the ARPA Network Information Center.

5.1 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 a client to attach remote directory trees to a local file system. The supporting mount protocol (see **Chapter 6**, **Adjunct Protocols**) is used by a client to obtain access to a particular file system or a subset thereof. The server will provide a ''handle'' which the client can use to identify the file system in subsequent NFS operations. Typically, the client will use the handle to arrange for the remote file system to appear to the user as part of the local file system.

5.1.1 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 4**, **Remote Procedure Calls: Protocol Specification**). 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 RPC protocol is described in **Chapter 4**, **Remote Procedure Calls: Protocol Specification**.

5.1.2 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 3**, **XDR 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.

5.1.3 Stateless Servers and Idempotency

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.

However, NFS deals with objects such as files and directories that inherently have state - what good would a file be if it did not keep its contents intact? 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 they can potentially be repeated). 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, neither is there any protocol associated with state recovery. However, NFS deals with objects such as files and directories which inherently have state. This apparent contradiction is resolved by introducing distributed state and by making operations idempotent.

Distributed state arises when an NFS server passes information such as a file handle or directory search ‘‘cookie’’ to a client. The server promises, in effect, that when the client passes this information back to the server at a later date it will still be valid and can be used to reconstruct the state needed to perform the requested operation. The state information must be valid even if the server has been rebooted in the interim, and thus must refer to objects held on stable server storage. (In practice, servers will employ caching techniques to accelerate the interpretation of this state in the normal case when no reboot has occurred.)

An idempotent operation is one which can be repeated several times without changing the results. For example, a request to write 5 bytes at offset 165 in a file is idempotent; a request to write 5 bytes at the current end-of-file is not. NFS employs idempotent operations wherever possible. Certain operations are inherently *not* idempotent - for example, deleting a file - so NFS server implementations will normally include mechanisms to attempt to detect duplicate requests and furnish the appropriate results. Occasionally this strategy will fail and a client will receive an unexpected error; NFS clients must be tolerant of such occurrences.

5.2 NFS PROTOCOL DEFINITION

Servers can change over time, and so can the protocol that they use. RPC therefore provides a version number with each RPC request. This chapter describes version 2 of the NFS protocol. It contains procedures and parameters which are unused (obsolete) but which are retained for compatibility purposes. NFS server implementations should be prepared to handle these appropriately.

5.2.1 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, travel 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. A Network Standard Pathname Representation could be defined, but then every pathname would have to be parsed and converted at each end. Other issues are discussed in **Section 5.4**, **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. However, directories can contain many entries, and a remote call to return each would lead to unacceptable performance.

Symbolic links

The NFS file system model includes the concept of symbolic links, in which a directory entry is associated with a piece of text instead of a file or directory. An NFS client which encounters a symbolic link while processing a path will normally issue an *NFSPROC_READLINK* to retrieve the text and will then treat this as a path and look up the components to locate the actual file or directory. An NFS server need not implement symbolic links; if it does not, it should be prepared to return a *PROC_UNAVAIL* error if a client invokes *NFSPROC_READLINK* or *NFSPROC_SYMLINK*. Similarly, an NFS client should only issue an *NFS_READLINK* if a lookup returns an entry typed as an **NFLNK**, and should be prepared to handle failures of any symbolic link operations.

5.3 RPC INFORMATION

Authentication

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

Transport Protocols

Current implementations of NFS are supported over 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 will use the ‘‘Portmapping’’ facility of RPC.

5.3.1 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 NFS_MAXDATA = 8192;

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

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

/* The size in bytes of the opaque ‘‘cookie’’ passed by READDIR */
const NFS_COOKIESIZE = 4;

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

5.3.2 Basic Data Types

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

5.3.2.1 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. The other values indicate some kind of error occurred on the server side during the servicing of the procedure.

NFSERR_PERM Not owner. The caller does not have the 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.

5.3.2.2 ftype

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

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

5.3.2.3 nfscookie

```
typedef opaque nfscookie[NFS_COOKIESIZE];
```

The **nfscookie** is an opaque value that identifies a particular piece of data, such as a directory entry in the *NFSPROC_READDIR* call.

5.3.2.4 fhandle

```
typedef opaque fhandle[NFS_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.

5.3.2.5 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.

5.3.2.6 diropok

```
struct diropok {
    fhandle file;
    fattr   attributes;
};
```

5.3.2.7 fattr

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

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

mode is the access mode encoded as a set of bits. Notice that the file type is specified both in the mode bits and in the file type; the server must ensure they are consistent.

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	Not used.
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.

Notes: The bits are the same as the mode bits returned by the *stat*() XSI system call with the addition of the socket and symbolic link combinations which are supported by NFS and some operating systems.

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

5.3.2.8 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** above. A value of 0xffffffff indicates a field that should be ignored. A *size* of zero means the file should be truncated to zero length.

5.3.2.9 filename

```
typedef string filename<NFS_MAXNAMLEN>;
```

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

5.3.2.10 path

```
typedef string path<NFS_MAXPATHLEN>;
```

The type **path** is a pathname to be used in the symbolic link operations *NFSPROC_SYMLINK* and *NFSPROC_READLINK*. The server must consider it as a string with no internal structure.

5.3.2.11 attrstat

```
union 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 succeeded, it also contains the attributes of the file on which the operation was done.

5.3.2.12 diropargs

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

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

5.3.2.13 diropres

```
union diropres switch (stat status) {
    case NFS_OK:
        struct diropok diropok;
    default:
        void;
};
```

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

5.4 NFS IMPLEMENTATION ISSUES

The NFS protocol is designed to be operating system independent, but since this version was designed in a UNIX environment, 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

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, it was decided not to let clients cross a server's mount point.

When a client does an *NFSPROC_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.

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 be resolved in interesting ways.

Using uid and gid implies that the client and server share the same uid list. Every server and client pair must have the same mapping from user to uid and from group to gid. Since every client can also be a server, this tends to imply that the whole network shares the same uid/gid space.

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 cannot 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 (with 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 client could gain access to all remote files. An X/Open-compliant server, by default, maps user id 0 to -2 (0xfffffffe) before doing its access checking. A server implementation may provide a mechanism to change this mapping.

5.5 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 synchronous. When a procedure returns to the client the operation has completed and any data associated with the request is now on stable storage. For example, a client *NFSPROC_WRITE* request will cause the server to update some or all of the following: data blocks, file system information blocks (such as indirect blocks), and file attribute information (size and modify times). When the *NFSPROC_WRITE* returns to the client, it can assume that the write is safe, even in case of a server crash, and it can discard the data written. This is a very important part of the statelessness of the server. If the server waited to flush data from remote requests, the client would have to save those requests so that it could resend them in case of a server crash.

```
/*
 * 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;
```

5.5.1 NFSPROC_NULL Specification - Do Nothing

5.5.1.1 RPC Data Descriptions

Call Arguments

None.

Return Arguments

None.

5.5.1.2 RPC Procedure Description

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

5.5.1.3 Description

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

5.5.1.4 Return Codes

None.

5.5.2 NFSPROC_GETATTR Specification - Get File Attributes

5.5.2.1 RPC Data Descriptions

Call Arguments

```
typedef opaque fhandle[NFS_FHSIZE];
```

Return Arguments

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

fattr and **sattr** are defined in **Section 5.3.2**, **Basic Data Types**.

5.5.2.2 RPC Procedure Description

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

5.5.2.3 Description

If the reply status is *NFS_OK*, then the reply attributes contains the attributes for the file given by the input fhandle. The file handle supplied to this procedure can refer to any of the supported file types. See the definition of **ftype** in **Section 5.3.2**, **Basic Data Types**.

5.5.2.4 Return Codes

NFS_OK — Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation-specific.

NFSERR_IO — Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

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.

5.5.3 NFSPROC_SETATTR Specification - Set File Attributes

5.5.3.1 RPC Data Descriptions

Call Arguments

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

fhandle and **sattr** are defined in **Section 5.3.2**, **Basic Data Types**.

Return Arguments

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

fattr and **sattr** are defined in **Section 5.3.2**, **Basic Data Types**.

5.5.3.2 RPC Procedure Description

```
attrstat
NFSPROC_SETATTR (sattrargs) = 2;
```

5.5.3.3 Description

The *attributes* argument contains fields which are either 0xffffffff or are the new value for the attributes of **file**. If the reply status is *NFS_OK*, then the reply attributes have the

attributes of the file after the *NFSPROC_SETATTR* operation has completed. The file handle supplied to this procedure can refer to any of the supported file types, but it may not be possible to set all attributes in the **sattr** structure for a particular file type.

Setting the *size* field to zero in the **sattr** structure means the file should be truncated. This operation should only be permitted on regular files.

5.5.3.4 Return Codes

NFS_OK — Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation-specific.

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

NFSERR_IO — Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFSERR_ACCES — Permission denied. The caller does not have the correct permission to perform the requested operation or is attempting to change an attribute which may not be modified for a particular file type.

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

NFSERR_ROFS — Read-only file system. Write attempted on a read-only file system.

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.

5.5.4 NFSPROC_ROOT Specification - Get File System Root

5.5.4.1 RPC Data Descriptions

Call Arguments

None.

Return Arguments

None.

5.5.4.2 RPC Procedure Description

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

5.5.4.3 Description

Obsolete. The function of looking up the root file handle is now handled by the mount protocol. 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 would require the definition of 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 6.4**, **Mount Protocol Definition**.)

5.5.4.4 Return Codes

None.

5.5.5 NFSPROC_LOOKUP Specification - Look Up File Name

5.5.5.1 RPC Data Descriptions

Call Arguments

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

fhandle and **filename** are defined in **Section 5.3.2**, **Basic Data Types**.

Return Arguments

```
union diropres switch (stat status) {
      case NFS_OK:
            struct diropok diropok;
      default:
            void;
};
```

fhandle, **fattr** and **sattr** are defined in **Section 5.3.2**, **Basic Data Types**.

5.5.5.2 RPC Procedure Description

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

5.5.5.3 Description

If the reply *status* is *NFS_OK*, then the reply *diropok.file* and reply *diropok.attributes* are the file handle and attributes for the file *name* in the directory given by *dir* in the argument. The file handle supplied to this procedure can refer to any of the supported file types.

5.5.5.4 Return Codes

NFS_OK Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation-specific.

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

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

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

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

5.5.6 NFSPROC_READLINK Specification - Read From Symbolic Link

5.5.6.1 RPC Data Descriptions

Call Arguments

```
typedef opaque fhandle[NFS_FHSIZE];
```

Return Arguments

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

path and **sattr** are defined in **Section 5.3.2**, **Basic Data Types**.

5.5.6.2 RPC Procedure Description

```
readlinkres
NFSPROC_READLINK(fhandle) = 5;
```

5.5.6.3 Description

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. The file handle supplied to this procedure must refer to a file of the symbolic link file type.

An NFS server need not implement symbolic links; if it does not, it should return an *PROC_UNAVAIL* error. An NFS client should only issue an *NFSPROC_READLINK* if a lookup returns an entry that is typed as *NFLNK*, and should be prepared to handle failures of any symbolic link operation.

Note that 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.

5.5.6.4 Return Codes

NFS_OK — Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation-specific.

NFSERR_IO — Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

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.

PROC_UNAVAIL — This procedure is not supported.

NFSERR_NXIO — The **fhandle** given in the argument does not refer to a symbolic link.

5.5.7 NFSPROC_READ Specification - Read From File

5.5.7.1 RPC Data Descriptions

Call Arguments

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

fhandle is defined in **Section 5.3.2**, **Basic Data Types**.

Return Arguments

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

fattr and **sattr** are defined in **Section 5.3.2**, **Basic Data Types**.

5.5.7.2 RPC Procedure Description

```
readres
NFSPROC_READ(readargs) = 6;
```

5.5.7.3 Description

Up to *count* bytes of *data* are returned from the file given by *file* starting at *offset* bytes from the beginning of the file. The first byte of the file is at offset zero. The file attributes after the read takes place are returned in *attributes*. Read operations should only be permitted on regular files. Reading directory files should be performed using the *NFSPROC_READDIR* procedure (see **Section 5.5.17**, **NFSPROC_READDIR Specification - Read From Directory**).

Note that the argument *totalcount* is unused.

5.5.7.4 Return Codes

NFS_OK — Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation specific.

NFSERR_IO — Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

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

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

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.

5.5.8 NFSPROC_WRITECACHE Specification - Write to Cache

5.5.8.1 RPC Data Descriptions

Call Arguments

None.

Return Arguments

None.

5.5.8.2 RPC Procedure Description

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

5.5.8.3 Description

Function not used.

5.5.8.4 Return Codes

None.

5.5.9 NFSPROC_WRITE Specification - Write to File

5.5.9.1 RPC Data Descriptions

Call Arguments

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

fhandle is defined in **Section 5.3.2**, **Basic Data Types**.

Return Arguments

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

fattr and **sattr** are defined in **Section 5.3.2**, **Basic Data Types**.

5.5.9.2 RPC Procedure Description

```
attrstat
NFSPROC_WRITE(writeargs) = 8;
```

5.5.9.3 Description

data is written, 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 has completed. The write operation is atomic. Data from this call to *NFSPROC_WRITE* will not be mixed with data from another client's calls. Write operations should only be permitted on regular files.

Note that the arguments *beginoffset* and *totalcount* are unused.

5.5.9.4 Return Codes

NFS_OK Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation-specific.

NFSERR_IO Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

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

5.5.10 NFSPROC_CREATE Specification - Create File

5.5.10.1 RPC Data Descriptions

Call Arguments

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

diropargs and **sattr** are defined in **Section 5.3.2**, **Basic Data Types**.

Return Arguments

```
union diropres switch (stat status) {
      case NFS_OK:
            struct diropok diropok;
      default:
            void;
};
```

fhandle, **fattr** and **stat** are defined in **Section 5.3.2**, **Basic Data Types**.

5.5.10.2 RPC Procedure Description

```
diropres
NFSPROC_CREATE(createargs) = 9;
```

5.5.10.3 Description

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

This procedure is used to create regular files only; directories may be created by the *NFSPROC_MKDIR* procedure (see **Section 5.5.15**, **NFSPROC_MKDIR Specification - Create Directory**).

Note that this call will succeed even if the file already exists.

5.5.10.4 Return Codes

NFS_OK — Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation-specific.

NFSERR_IO — Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

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

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

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_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.

5.5.11 NFSPROC_REMOVE Specification - Remove File

5.5.11.1 RPC Data Descriptions

Call Arguments

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

fhandle and **filename** are defined in **Section 5.3.2**, **Basic Data Types**.

Return Arguments

```
stat status;
```

stat is defined in **Section 5.3.2**, **Basic Data Types**.

5.5.11.2 RPC Procedure Description

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

5.5.11.3 Description

The file *name* is removed from the directory given by *dir*. A reply of *NFS_OK* means the directory entry was removed. Any other return value indicates an error, and the file was not removed. This procedure may be used to remove any of the supported file types except directories. Removal of directories should be performed using the *NFSPROC_RMDIR* procedure (see **Section 5.5.16**, **NFSPROC_RMDIR Specification - Remove Directory**).

Note that this is generally a non-idempotent operation. A server should attempt to provide this function in an idempotent fashion. X/Open-compliant 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 filesystem. It is impossible for a stateless server to implement these semantics.

5.5.11.4 Return Codes

NFS_OK — Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation-specific.

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

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

NFSERR_ROFS Read-only file system. Write attempted on a read-only file system.

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.

5.5.12 NFSPROC_RENAME Specification - Rename File

5.5.12.1 RPC Data Descriptions

Call Arguments

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

diropargs is defined in **Section 5.3.2**, **Basic Data Types**.

Return Arguments

```
stat status;
```

stat is defined in **Section 5.3.2, Basic Data Types**.

5.5.12.2 RPC Procedure Description

```
stat
NFSPROC_RENAME(renameargs) = 11;
```

5.5.12.3 Description

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 *NFSPROC_RENAME* operation is required to be atomic on the server; it cannot be interrupted in the middle, i.e., a link and unlink combination is not sufficient.

Note that this is possibly a non-idempotent operation. A server should attempt to provide this function in an idempotent fashion.

5.5.12.4 Return Codes

NFS_OK — Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation-specific.

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_ACCES — Permission denied. The caller does not have the correct permission to perform the requested operation.

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_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.

5.5.13 NFSPROC_LINK Specification - Create Link to File

5.5.13.1 RPC Data Descriptions

Call Arguments

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

fhandle and **diropargs** are defined in **Section 5.3.2**, **Basic Data Types**.

Return Arguments

stat status;

stat is defined in **Section 5.3.2, Basic Data Types**.

5.5.13.2 RPC Procedure Description

```
stat
NFSPROC_LINK(linkargs) = 12;
```

5.5.13.3 Description

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.

Note that this is generally a non-idempotent operation. A server should attempt to provide this function in an idempotent fashion.

5.5.13.4 Return Codes

NFS_OK Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation-specific.

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

NFSERR_IO Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

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_NOTDIR Not a directory. The caller specified a non-directory in a directory operation.

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_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.

5.5.14 NFSPROC_SYMLINK Specification - Create Symbolic Link

5.5.14.1 RPC Data Descriptions

Call Arguments

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

diropargs, **path** and **sattr** are defined in **Section 5.3.2**, **Basic Data Types**.

Return Arguments

```
stat status;
```

stat is defined in **Section 5.3.2, Basic Data Types**.

5.5.14.2 RPC Procedure Description

```
stat
NFSPROC_SYMLINK(symlinkargs) = 13;
```

5.5.14.3 Description

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.

An NFS server need not implement symbolic links; if it does not, it should to return an *PROC_UNAVAIL* error. An NFS client should be prepared to handle failures of any symbolic link operation. The *NFSPROC_READLINK* operation returns the data to the client for interpretation.

Note that servers may ignore the attributes depending on the symbolic link model they use.

5.5.14.4 Return Codes

NFS_OK	Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation-specific.

NFSERR_IO	Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.
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_NOTDIR	Not a directory. The caller specified a non-directory in a directory operation.
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_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.

5.5.15 NFSPROC_MKDIR Specification - Create Directory

5.5.15.1 RPC Data Descriptions

Call Arguments

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

diropargs and **sattr** are defined in **Section 5.3.2**, **Basic Data Types**.

Return Arguments

```
union diropres switch (stat status) {
      case NFS_OK:
            struct diropok diropok;
      default:
            void;
};
```

fhandle, **fattr** and **stat** are defined in **Section 5.3.2**, **Basic Data Types**.

5.5.15.2 RPC Procedure Description

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

5.5.15.3 Description

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

5.5.15.4 Return Codes

NFS_OK — Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation-specific.

NFSERR_IO — Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

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_NOTDIR — Not a directory. The caller specified a non-directory in a directory operation.

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_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.

5.5.16 NFSPROC_RMDIR Specification - Remove Directory

5.5.16.1 RPC Data Descriptions

Call Arguments

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

fhandle and **filename** are defined in **Section 5.3.2**, **Basic Data Types**.

Return Arguments

stat status;

stat is defined in **Section 5.3.2, Basic Data Types**.

5.5.16.2 RPC Procedure Description

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

5.5.16.3 Description

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

Note that this is possibly a non-idempotent operation. A server should attempt to provide this function in an idempotent fashion.

5.5.16.4 Return Codes

NFS_OK Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation-specific.

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

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

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_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.

5.5.17 NFSPROC_READDIR Specification - Read From Directory

5.5.17.1 RPC Data Descriptions

Call Arguments

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

fhandle and **nfscookie** are defined in **Section 5.3.2**, **Basic Data Types**.

Return Arguments

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

filename and **nfscookie** are defined in **Section 5.3.2**, **Basic Data Types**.

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

5.5.17.2 RPC Procedure Description

```
readdirres
NFSPROC_READDIR (readdirargs) = 16;
```

5.5.17.3 Description

A variable number of directory entries, with a total size of up to *count* bytes, are returned 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*s. 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 *NFSPROC_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. The *eof* flag has a value of *TRUE* if there are no more entries in the

directory. (See **Section 5.3.2**, **Basic Data Types**.)

5.5.17.4 Return Codes

NFS_OK Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation-specific.

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

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

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.

5.5.18 NFSPROC_STATFS Specification - Get File System Attributes

5.5.18.1 RPC Data Descriptions

Call Arguments

```
typedef opaque fhandle[NFS_FHSIZE];
```

Return Arguments

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

5.5.18.2 RPC Procedure Description

statfsres
NFSPROC_STATFS(fhandle) = 17;

5.5.18.3 Description

If the reply *status* is *NFS_OK*, then the reply *info* gives the attributes for the file system that contains the 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 NFSPROC_READ and NFSPROC_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.

5.5.18.4 Return Codes

NFS_OK — Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations. Other error values are considered implementation-specific.

NFSERR_IO — Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

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.

Chapter 6

Adjunct Protocols

This chapter specifies protocols currently in use in various implementations. It specifies protocols that are used as part of existing implementations of NFS.

6.1 INTRODUCTION

This chapter describes protocols that are related to but separate from the NFS protocol. These protocols are not included as part of the NFS protocol to provide implementational flexibility and to facilitate the development of new mechanisms without requiring the revision of related protocols.

The following protocols are described in this chapter: Port Mapper Protocol, Personal Computer NFS Daemon (PCNFSD) protocol, Mount Protocol, Network Lock Manager (NLM) protocol.

The Port Mapper protocol translates RPC program and version numbers into network transport addressing information.

NFS assumes that the host operating systems on client and server machines provide user authentication and access control mechanisms. To allow single-user personal computer systems the convenience of per-user authentication, the Personal Computer NFS Daemon (PCNFSD) protocol supports user authentication as well as print services.

The mount protocol provides operating system specific services for the look up of server pathnames, validation of user identity and checking of access permissions. Clients use the mount protocol to get the first file handle, which allows them entry into a remote filesystem. The mount protocol is separate from the NFS protocol to facilitate implementation flexibility and to allow new access checking and validation methods to be used.

The Network Lock Manager (NLM) protocol allows the implementation of DOS file locking and sharing. NFS is a stateless service, which means that it cannot provide file locking and access control synchronisation. The Network Lock Manager (see **Section 6.5**, **Network Lock Manager Protocol Definition**) protocol allows lock managers on client and server systems to implement these services.

Like NFS, these protocols are based on the RPC protocol, and the protocol specifications are written using the XDR data description language. (See **Chapter 3**, **XDR Protocol Specification** and **Chapter 4**, **Remote Procedure Calls: Protocol Specification**.)

6.2 PORT MAPPER PROGRAM PROTOCOL

6.2.1 Introduction to 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. For interoperation with personal computer clients, the port mapper program must support the UDP/IP protocol. The port mapper is contacted by talking to it on assigned port number 111.

6.2.2 Port Mapper Protocol Specification (in RPC Language)

```
const PMAP_PORT = 111;     /* port mapper 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<>;
};
```

6.2.3 Port Mapper Procedures

```
/*
* 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;
        } = 2;
} = 100000;
```

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 on either of these protocols. The following is a description of each of the port mapper procedures:

6.2.4 PMAPPROC_NULL Specification - Do Nothing

6.2.4.1 RPC Data Descriptions

Call Arguments

None.

Return Arguments

None.

6.2.4.2 RPC Procedure Descriptions

```
void
PMAPPROC_NULL(void) = 0;
```

6.2.4.3 Description

This procedure does no work. By convention, procedure zero of any RPC program takes no parameters and returns no results. It is made available to allow server response testing and timing.

6.2.5 PMAPPROC_SET Specification - Set Mapping

6.2.5.1 RPC Data Descriptions

Call Arguments

```
mapping        mapping;
```

Return Arguments

```
bool     ret_value;
```

6.2.5.2 RPC Procedure Descriptions

```
bool
PMAPPROC_SET(mapping) = 1;
```

6.2.5.3 Description

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, *mapping.prog*, version number, *mapping.vers*, transport protocol number, *mapping.prot*, and the port, *mapping.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*)’’.

6.2.6 PMAPPROC_UNSET Specification - Unset Mapping

6.2.6.1 RPC Data Descriptions

Call Arguments

```
mapping        mapping;
```

Return Arguments

```
bool     ret_val;
```

6.2.6.2 RPC Procedure Descriptions

bool
PMAPPROC_UNSET(mapping) = 2;

6.2.6.3 Description

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.

6.2.7 PMAPPROC_GETPORT Specification - Get Port

6.2.7.1 RPC Data Descriptions

Call Arguments

mapping   mapping;

Return Arguments

unsigned int   port;

6.2.7.2 RPC Procedure Descriptions

unsigned int
PMAPPROC_GETPORT(mapping) = 3;

6.2.7.3 Description

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

6.2.8 PMAPPROC_DUMP Specification - Dump Mappings

6.2.8.1 RPC Data Descriptions

Call Arguments

None.

Return Arguments

pmaplist   mappings;

6.2.8.2 RPC Procedure Descriptions

pmaplist
PMAPPROC_DUMP (void) = 4;

6.2.8.3 Description

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.

6.3 PCNFSD PROTOCOL DEFINITION

The purpose of the *PCNFSD* protocol is to provide a personal computer NFS client with the authentication and network printing services which are usually available in larger and more capable systems. Its use, while not necessary, is highly desirable. However personal computer NFS implementations must be prepared to work with NFS servers which do not support *PCNFSD*, and *vice versa*. The source code for the server implementation of *PCNFSD* is freely available from Sun Microsystems.

6.3.1 Authentication

The NFS file access control model is based upon the uid/gid mechanism used in X/Open-compliant systems. All NFS remote procedure calls must be made with *AUTH_UNIX* credentials from which a uid and gid can be extracted. If a client implementation supports the use of NFS services without any form of authentication, it should use the uid/gid pair (0xfffffffe, 0xfffffffe) (i.e., (-2, -2)), which is conventionally associated with the identity ‘‘nobody’’. Client and server support for access as ‘‘nobody’’ is an implementation or administrative option.

Operation as ‘‘nobody’’, while feasible, is undesirable, since the client can only access filesystem hierarchies with unlimited ‘‘other’’ permissions, and administrators of server systems have no way of controlling resource usage. For this reason, it is expected that personal computer NFS implementations will require or encourage users to establish valid access credentials. A typical implementation might be to prompt the user to enter a username and password, which could then be validated using the *PCNFSD_AUTH* procedure, which will return a uid/gid pair. The client can then use this information to synthesise the *AUTH_UNIX* credentials for subsequent RPC requests.

Since it is undesirable to pass clear-text passwords over a network, both the username and the password are mildly scrambled using a simple exclusive-or operation. The intent is not to be secure but to defeat ‘‘browsers’’.

6.3.2 Print Spooling

The availability of NFS file operations simplifies the print spooling mechanism. There are two remote procedures involved. The client begins by calling *PCNFSD_PR_INIT,* which returns the name of a directory on the server which is exported via NFS and in which the client may create spool files. When the client has written the text to be printed to a spool file in the designated directory, the *PCNFSD_PR_START* procedure instructs the server to print the file on a specific printer.

Version 1 of the PCNFSD protocol is used with any version of the NFS protocol.

6.4 RPC Information

Authentication

The PCNFSD service uses *AUTH_UNIX* style authentication only.

Transport Protocols

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

Port Number

Consult the server's port mapper, described in **Section 6.2**, **Port Mapper Program Protocol**, to find the port number on which the PCNFSD service is registered.

6.4.1 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 user name argument */
const IDENTLEN = 32;

/* The maximum number of bytes in a password argument */
const PASSWORDLEN = 64;

/* The maximum number of bytes in a print client name argument */
const CLIENTLEN = 64;

/* The maximum number of bytes in a printer name argument */
const PRINTERNAMELEN = 64;

/* The maximum number of bytes in a print user name argument */
const USERNAMELEN = 64;

/* The maximum number of bytes in a print spool file name argument */
const SPOOLNAMELEN = 64;

/* The maximum number of bytes in a print options argument */
const OPTIONSLEN = 64;

/* The maximum number of bytes in a print spool directory path */
const SPOOLDIRLEN = 255;
```

6.4.2 Basic Data Types

This section presents the data types used by the PCNFSD protocol.

6.4.2.1 ident

```
typedef string ident<IDENTLEN>;
```

The type **ident** is used for passing an encoded user name for authentication. The server should decode the string by replacing each octet with the value formed by performing an exclusive-or of the octet value with the value 0x5b, and *and*ing the result with 0x7f.

6.4.2.2 password

```
typedef string password<PASSWORDLEN>;
```

The type **password** is used for passing an encode password for authentication. The server should decode the password as described above.

6.4.2.3 client

```
typedef string client<CLIENTLEN>;
```

The type **client** is used for passing the hostname of a client for printing. The server may use this name in constructing the spool directory name.

6.4.2.4 printername

```
typedef string printername<PRINTERNAMELEN>;
```

The type **printername** is used for passing the name of a printer on which the client wishes to print.

6.4.2.5 username

```
typedef string username<USERNAMELEN>;
```

The type **username** is used for passing the user name for a print job. The server may use this in any way it chooses: it may attempt to change the effective identity with which it is running to **username** or may simply arrange for the text to be printed on the banner page.

6.4.2.6 spoolname

```
typedef string spoolname<SPOOLNAMELEN>;
```

The type **spoolname** is used for passing the name of a print spool file (a simple filename not a pathname) within the spool directory.

6.4.2.7 options

```
typedef string options<OPTIONSLEN>;
```

The type **options** is used for passing implementation-specific print control information.

The option string is a set of printable ASCII characters.

The first character should be ignored by the server; it is reserved for client use. The second character specifies the type of data in the print file. The following types are defined (an implementation may define additional values):

p PostScript data. The client will ensure that a valid PostScript header is included.

d Diablo 630 data.

x Generic printable ASCII text. The client will have filtered out all non-printable characters other than CR, LF, TAB, BS and VT.

r Raw print data. The client performs no filtering.

If diablo data (type *d*) is specified, a formatting specification string will be appended. This has the form:

ppnnnbbb

pp Pitch - 10, 12 or 15.

nnn The ‘‘normal’’ font to be used - encoded as follows:

PostScript font name	Value of *nnn*
Courier	crn
Courier-Bold	crb
Courier-Oblique	con
Courier-BoldOblique	cob
Helvetica	hrn
Helvetica-Bold	hrb
Helvetica-Oblique	hon
Helvetica-BoldOblique	hob
Times-Roman	trn
Times-Bold	trb
Times-Italic	ton
Times-BoldItalic	tob

bbb The ‘‘bold’’ font to be used - encoded in the same way.

For example, the string *nd10hrbcob* specifies that the print data is in Diablo 630 format, it should be printed at 10 pitch, ‘‘normal’’ text should be printed in Helvetica-Bold, and ‘‘bold’’ text should be printed in Courier-BoldOblique.

6.4.2.8 arstat

```
enum arstat {
    AUTH_RES_OK = 0,
    AUTH_RES_FAKE = 1,
    AUTH_RES_FAIL = 2
};
```

The type **arstat** is returned by *PCNFSD_AUTH*. A value of *AUTH_RES_OK* indicates that the server was able to verify the **ident** and **password** successfully. *AUTH_RES_FAIL* is returned if a verification failure occurred. The value *AUTH_RES_FAKE* may be used if the server wishes to indicate that the verification failed, but that the server has synthesised acceptable values for uid and gid which the client may use if it wishes.

6.4.2.9 pirstat

```
enum pirstat {
    PI_RES_OK = 0,
    PI_RES_NO_SUCH_PRINTER = 1,
    PI_RES_FAIL = 2
};
```

The type **pirstat** is returned by *PCNFSD_PR_INIT*. A value of *PI_RES_OK* indicates that the server has set up a spool directory for the client to use. *PI_RES_FAIL* is returned if spool

directory could not be created. The value *PI_RES_NO_SUCH_PRINTER* indicates that the printer name was not recognised.

6.4.2.10 psrstat

```
enum psrstat {
    PS_RES_OK = 0,
    PS_RES_ALREADY = 1,
    PS_RES_NULL = 2,
    PS_RES_NO_FILE = 3,
    PS_RES_FAIL = 4
};
```

The type **psrstat** is returned by *PCNFSD_PR_START*. A value of *PS_RES_OK* indicates that the server has started printing the job. It is possible that the reply from a *PCNFSD_PR_START* call may be lost, in which case the client will repeat the call. If the spool file is still in existence, the server will return *PS_RES_ALREADY* indicating that it has already started printing. If the file cannot be found, *PS_RES_NO_FILE* is returned. *PS_RES_NULL* indicates that the spool file was empty, while *PS_RES_FAIL* denotes a general failure. *PI_RES_FAIL* is returned if spool directory could not be created. The value *PI_RES_NO_SUCH_PRINTER* indicates that the printer name was not recognised.

6.4.3 PCNFSD Server Procedures

The following sections define the RPC procedures supplied by a PCNFSD server.

```
/*
Protocol description for the PCNFSD program
*/
program PCNFSDPROG {
/*
Version 1 of the PCNFSD protocol.
*/
!version PCNFSDVERS {
            void PCNFSD_NULL(void) = 0;
            auth_results PCNFSD_AUTH(auth_args) = 1;
            pr_init_results PCNFSD_PR_INIT(pr_init_args) = 2;
            pr_start_results PCNFSD_PR_START(pr_start_args) = 3;
      } = 1;
} = 150001;
```

6.4.4 PCNFSD_NULL Specification - Do Nothing

6.4.4.1 RPC Data Descriptions

Call Arguments

None.

Return Arguments

None.

6.4.4.2 RPC Procedure Description

```
void
PCNFSD_NULL(void) = 0;
```

6.4.4.3 Description

This procedure does no work. By convention, procedure zero of any RPC program takes no parameters and returns no results. It is made available to allow server response testing and timing.

6.4.4.4 Return Codes

None.

6.4.5 PCNFSD_AUTH Specification - Perform User Authentication

6.4.5.1 RPC Data Descriptions

Call Arguments

```
struct auth_args {

            ident       id;     /* user name (encoded) */
            password    pw;     /* user password (encoded) */

};
```

Return Arguments

```
struct auth_results {
    arstat stat;
    unsigned int uid;
    unsigned int gid;
};
```

6.4.5.2 RPC Procedure Description

```
auth_results
PCNFSD_AUTH(auth_args) = 1;
```

6.4.5.3 Description

This procedure is used to verify that the *id* and *pw* strings correspond to a user identity using the server operating system conventions. If the verification succeeds, the corresponding *uid* and *gid* values are returned. The *id* and *pw* are both encoded. The server should decode these strings by replacing each octet with the value formed by

performing an exclusive-or of the octet value with the value 0x5b, and *and*ing the result with 0x7f. If the verification succeeds, the corresponding *uid* and *gid* values are returned.

Caveat: This implies that the username and password are restricted to 7 bit ASCII characters. A future protocol revision may lift this 7 bit restriction, but due to the number of existing clients with this restriction, an X/Open-compliant server must accept 7 bit characters in the username and password.

6.4.5.4 Return Codes

When the procedure returns, *stat* is set to one of the following values:

AUTH_RES_OK — Indicates that the call completed successfully and that uid and gid are valid.

AUTH_RES_FAKE — Indicates that the call failed, but that the server has set uid and gid to acceptable values.

AUTH_RES_FAIL — Indicates that the authentication request failed. For security reasons the return status does not identify the precise reasons for the failure.

6.4.6 PCNFSD_PR_INIT Specification - Initialise Remote Printing

6.4.6.1 RPC Data Descriptions

Call Arguments

```
struct pr_init_args {
    client system;
    printername pn;
};
```

Return Arguments

```
struct pr_init_results {
    pirstat stat;
    spooldir dir;
};
```

6.4.6.2 RPC Procedure Description

```
pr_init_results
PCNFSD_PR_INIT(pr_init_args) = 2;
```

6.4.6.3 Description

This procedure is invoked by the client *system* when it wishes to spool print files to the server for subsequent printing on the printer *pn*. The server should create a directory which the client can mount using NFS and return its full pathname as *dir*.

6.4.6.4 Return Codes

When the procedure returns, *stat* is set to one of the following values:

PI_RES_OK — Indicates that the call completed successfully and that *dir* is valid.

PI_RES_NO_SUCH_PRINTER — Indicates that the call failed because *pn* did not name a suitable printer.

PI_RES_FAIL — Indicates that the call failed for some reason other than an invalid *pr*. The most common reason is that the server was unable to create a spool directory.

6.4.7 PCNFSD_PR_START Specification - Print a Spooled File

6.4.7.1 RPC Data Descriptions

Call Arguments

```
struct pr_start_args {
      client system;
      printername pr;
      username user;
      spoolname file;
      options opts;
};
```

Return Arguments

```
struct pr_start_results {
      psrstat stat;
};
```

6.4.7.2 RPC Procedure Description

```
pr_start_results
PCNFSD_PR_START(pr_start_args) = 3;
```

6.4.7.3 Description

This procedure is invoked by the client when it has placed a file containing printable data in the spool directory returned by *PCNFSD_PR_INIT* and wishes the server to print it. The arguments *system* and *pr* must be the same as passed to *PCNFSD_PR_INIT* since the server will normally use this information to locate the spool directory. *file* is a simple name (not a path) which identifies a file *within* this directory. The server should delete this file when it has been printed.

One possible implementation of *PCNFSD_PR_START* is as follows:

1. Change directory to the spool directory derived from *system* and *pr*.

2. If the file *file*.SPL exists, return *PS_RES_ALREADY* indicating that the file is already being printed.
3. If *file* does not exist, return *PS_RES_NO_FILE.*
4. Rename *file* to *file*.SPL.
5. Spawn a sub-process to perform the printing and return *PS_RES_OK.*
6. In the sub-process, change effective user identity to that given by *user* if possible.
7. Examine the *opts* string to determine if special processing is required.
8. Print the file *file*.SPL on the printer *pr*.
9. Remove the file *file*.SPL and terminate the sub-process.

Other errors which may occur result in the value *PS_RES_FAIL.*

6.4.7.4 Return Codes

When the procedure returns, *stat* is set to one of the following values:

PR_RES_OK — Indicates that the call completed successfully and that responsibility for printing the file has been transferred to the server's local print system.

PR_RES_ALREADY — Indicates that the server determined that the file was already being printed and that no further action was taken. the client will usually interpret this to mean that the request had been retransmitted because an original response with status *PR_RES_OK* was lost.

PR_RES_NULL — Indicates that the server determined that the file was empty, and was therefore not printed. The server deletes the file before returning this status.

PR_RES_NO_FILE — Indicates that the call failed because the print file could not be found.

PR_RES_FAIL — Indicates that the call failed for some other unspecified reason.

6.5 MOUNT PROTOCOL DEFINITION

6.5.1 Introduction

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

Notice that the protocol definition implies stateful servers because the server maintains a list of client's mount requests. This corresponds to current implementations which hold the mount list on stable storage. However, 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. The server must provide a mechanism to eliminate redundant information from the mount list.

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.

6.6 RPC Information

Authentication

The mount service uses *AUTH_UNIX* style authentication only.

Transport Protocols

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

Port Number

Consult the server's port mapper, described in **Section 6.2**, **Port Mapper Program Protocol**, to find the port number on which the mount service is registered.

6.6.1 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;
```

6.6.2 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.

6.6.2.1 fhandle

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 **Section 5.3.2**, **Basic Data Types**.

6.6.2.2 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 that

an error occurred. The following recommended error codes are derived from related UNIX error numbers.

```
const EPERM = 1;
const ENOENT = 2;
const EACCES = 13;
const EINVAL = 22;
```

6.6.2.3 dirpath

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

The type **dirpath** is a server pathname of a directory.

6.6.2.4 name

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

The type **name** is an arbitrary string used for various names.

6.6.3 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;
```

6.6.4 MNTPROC_NULL Specification - Do Nothing

6.6.4.1 RPC Data Descriptions

Call Arguments

None.

Return Arguments

None.

6.6.4.2 RPC Procedure Description

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

6.6.4.3 Description

This procedure does no work. By convention, procedure zero of any RPC program takes no parameters and returns no results. It is made available to allow server response testing and timing.

6.6.4.4 Return Codes

None.

6.6.5 MNTPROC_MNT Specification - Add Mount Entry

6.6.5.1 RPC Data Descriptions

Call Arguments

```
dirpath dirname;
```

Return Arguments

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

fhandle is defined in **Section 5.4.2**, **Basic Data Types**.

6.6.5.2 RPC Procedure Description

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

6.6.5.3 Description

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*.

6.6.5.4 Return Codes

<Zero> Indicates that the call completed successfully and the results are valid.

It is recommended that implementations return the following error codes for the following situations.

EPERM Indicates that the call failed because the mount server did not have the required privileges to perform the mount. (Most implementations require that the mount server run with uid 0.) This generally indicates a server configuration error.

EACCES Indicates that the call failed because access to the specified directory was denied. Either no directory in the path *dirname* is exported, or the client system is not permitted to mount this directory.

ENOENT Indicates that the call failed because the specified directory does not exist. If the server exports only **/a/b**, an attempt to mount **/a/b/c** will fail with *ENOENT* if the directory does not exist; on the other hand, an attempt to mount **/a/x** would fail with *EACCES*.

EINVAL Indicates that the call failed because the mount daemon was unable to translate the path into a file handle. This may indicate a server configuration error, or may occur if the directory is removed before the mount is complete.

6.6.6 MNTPROC_DUMP Specification - Return Mount Entries

6.6.6.1 RPC Data Descriptions

Call Arguments

None.

Return Arguments

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

name and **dirpath** are defined in **Section 5.3.2, Basic Data Types**.

6.6.6.2 RPC Procedure Description

```
mountlist
MNTPROC_DUMP(void) = 2;
```

6.6.6.3 Description

Returns the list of remote mounted file systems. The *mountlist* contains one entry for each *hostname* and *dirname* pair.

6.6.6.4 Return Codes

None.

6.6.7 MNTPROC_UMNT Specification - Remove Mount Entries

6.6.7.1 RPC Data Descriptions

Call Arguments

```
dirpath dirname;
```

Return Arguments

None.

6.6.7.2 RPC Procedure Description

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

6.6.7.3 Description

Removes the mount list entry for the input *dirname* that records the fact that *dirname* has been mounted by the client. *dirname* should be identical to the argument used in the corresponding *MNTPROC_MNT* call.

6.6.7.4 Return Codes

None.

6.6.8 MNTPROC_UNMNTALL Specification - Remove all Mount Entries

6.6.8.1 RPC Data Description

Call Arguments

None.

Return Arguments

None.

6.6.8.2 RPC Procedure Descriptions

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

Removes all of the mount list entries for this client.

6.6.8.3 Return Codes

None.

6.6.9 MNTPROC_EXPORT Specification - Return Export List

6.6.9.1 RPC Data Descriptions

Call Arguments

None.

Return Arguments

```
struct *expinfo {
        name    expitem;
        expinfo expnext;
};

struct *exportlist {
        dirpath filesys;
        expinfo expinfo;
        exportlist next;
};
```

name and **dirpath** are defined in **Section 5.3.2, Basic Data Types**.

6.6.9.2 RPC Procedure Description

```
exportlist
MNTPROC_EXPORT(void) = 5;
```

6.6.9.3 Description

Returns a variable number of export list entries. Each entry contains a filesystem name, *filesys*, and a list of text items describing how it may be mounted and by whom. Each item is encoded as an *expitem* in the list *expinfo*. The information is implementation specific, and while it may be meaningful to the user of the NFS client system it is not necessarily interpretable by client software. Typical information might include the names of systems, or groups of systems, which are allowed to mount the filesystem, or options describing access control or uid mapping.

6.6.9.4 Return Codes

None.

6.7 NETWORK LOCK MANAGER PROTOCOL DEFINITION

6.7.1 Introduction

The Network Lock Manager (NLM) is an RPC service that provides advisory locking of files across the network. There are multiple versions of the NLM; this specification describes version 3.

This service makes advisory DOS 3.1 file sharing and locking possible in an NFS environment. Its use is strongly encouraged but not mandatory, and NFS clients must be prepared to interoperate with servers which do not support this service. It is also recommended, but not required, that locks created by DOS processes be honoured by processes running on the server systems and vice versa.

Because the NFS protocol is stateless and has no knowledge of locks that may or may not have been granted, clients that wish exclusive access to a particular file must call the Network Lock Manager on the server to request access. The server Network Lock Manager is responsible for creating and destroying locks on files, as well as mediating requests for shared or exclusive file access.

6.7.2 Versions

6.7.2.1 Versions 1 and 2

Versions 1 and 2 of the Network Lock Manager are identical. Each provide the necessary services for implementing file and record locking across a network. However, these versions do not support personal computer clients.

6.7.2.2 Version 3

Version 3 includes the calls in version 2 of the Lock Manager and adds support for personal computers; non-monitored locks and DOS-compatible file sharing. These calls are used by personal computer implementations of NFS to access shared files while preserving the semantics of DOS versions 3.1 and later.

This specification describes version 3.

6.7.3 Synchronisation of Lock Managers

Due to the stateless nature of NFS servers it is difficult to incorporate a stateful service. The Lock Manager relies on the machine holding the locks as the keeper of the state. When an NFS server crashes and is rebooted, locks which it had granted may be recreated by the lock holders during a user-definable grace period. During the grace period no new locks are accepted, although NFS requests are accepted.

A client holding non-monitored locks or file shares may inform the lock manager, via the *NLM_FREE_ALL* procedure, that it has been rebooted and all locks and file shares are to be released.

Non-monitored locks

To support personal computer clients, the Network Lock Manager provides ‘‘non-monitored locks’’. This means that the server will not monitor the client to ensure that it

is still operational: it is the responsibility of the client system to advise the Network Lock Manager if it is rebooted so that the Network Lock Manager can discard any locks or file sharing reservations being held on behalf of the client.

6.7.4 DOS File Sharing Support

Version 3 of the protocol supports file locking and sharing for DOS machines on the net. File sharing is a mechanism which allows a DOS process to open or create a file and to restrict the way in which subsequent processes may access the file. For example, a DOS client may request that a file be opened for reading and writing, and that subsequent users may only open it for reading. To implement this feature an *NLM_SHARE* request is issued when a file is opened, and a corresponding *NLM_UNSHARE* is performed when it is closed. These procedures rely on the **nlm_share** structure, defined below. The precise semantics of the file sharing modes are described in the IBM **Disk Operating System Technical Reference**, **IBM part number 6138536**.

6.7.5 RPC Information

Authentication

The NLM service uses *AUTH_UNIX* style authentication only.

Transport Protocols

(PC)NFS clients use UDP/IP only.

Port Number

Consult the server's port mapper, described in **Section 6.2**, **Port Mapper Program Protocol**, to find the port number on which the NLM service is registered.

6.7.5.1 Sizes of XDR Structures

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

```
/* The maximum length of the string identifying the caller */
const LM_MAXSTRLEN = 1024;

/* The maximum number of bytes in the nlm_notify name argument */
const MAXNAMELEN = LM_MAXSTRLEN+1;

const MAXNETOBJ_SZ = 1024;
```

6.7.5.2 Basic Data Types for Locking

The following XDR definitions are the basic structures and types used in the parameters passed to and returned from the Network Lock Manager.

netobj

```
const netobj<MAXNETOBJ_SZ>
```

netobj is used to identify an object, generally a transaction, owner or file.

nlm_stats

```
enum nlm_stats {
      nlm_granted = 0,
      nlm_denied = 1,
      nlm_denied_nolocks = 2,
      nlm_blocked = 3,
      nlm_denied_grace_period = 4
};
```

nlm_stats are returned whenever the Network Lock Manager is called upon to create or test a lock on a file. Generally, the result is self explanatory; the only two codes that bear clarification are *nlm_denied_nolocks*, which is returned when a lock could not be created on the requested file (usually an error), and *nlm_denied_grace_period* which is returned when the server has recently been rebooted and is re-establishing existing locks.

nlm_holder

```
struct nlm_holder {
      bool exclusive;
      int uppid;
      netobj oh;
      unsigned l_offset;
      unsigned l_len;
};
```

The **nlm_holder** structure identifies the holder of a particular lock. The integer *uppid* provides a unique per-process identifier for lock differentiation. The values *l_offset* and *l_len* define the region of the file locked by this holder.

nlm_stat

```
struct nlm_stat {
      nlm_stats stat;
};
```

The **nlm_stat** structure returns lock status.

nlm_res

```
struct nlm_res {
      netobj cookie;
      nlm_stat stat;
};
```

The **nlm_res** structure is returned by the main lock routines of the Network Lock Manager. Note that clients should *not* rely upon the *cookie* being the same as that passed in the corresponding request.

nlm_lock

```
struct nlm_lock {
    string caller_name<LM_MAXSTRLEN>;
    netobj fh;            /* identify a file */
    netobj oh;            /* identify owner of a lock */
    int uppid;            /* generated from e.g., PSP in DOS */
    unsigned l_offset;    /* File offset (for record locking) */
    unsigned l_len;       /* Length (size of record) */
};
```

The **nlm_lock** structure defines the information needed to uniquely specify a lock. The netobj's *fh* and *oh* define the file and owner, *caller_name* uniquely identifies the host. *uppid* uniquely describes the process owning the file on the calling host, and the *offset* and *length* determine which bytes of the file are locked.

nlm_lockargs

```
struct nlm_lockargs {
    netobj cookie;
    bool block;                 /* Flag to indicate blocking behaviour */
    bool exclusive;             /* If exclusive access is desired */
    struct nlm_lock alock;      /* The actual lock data (see above) */
    bool reclaim;               /* used for recovering locks */
    int state;                  /* specify local status monitor state */
};
```

The **nlm_lockargs** structure defines the information needed to request a lock on a server. The **reclaim** field should only be set to true if the client is attempting to reclaim a lock held on a lock daemon which has been restarted (due to a server crash etc).

nlm_unlockargs

```
struct nlm_unlockargs {
    netobj cookie;
    struct nlm_lock alock;
};
```

The **nlm_unlockargs** structure defines the information needed to remove a previously established lock.

6.7.6 DOS 3.1 File Sharing

The following data types are used in version 3 of the lock manager to support DOS 3.1 compatible file sharing control.

6.7.6.1 fsh_mode

```
enum    fsh_mode {
        fsm_DN  = 0,            /* deny none */
        fsm_DR  = 1,            /* deny read */
        fsm_DW  = 2,            /* deny write */
        fsm_DRW = 3             /* deny read/write */
};
```

fsh_mode defines the legal sharing modes.

6.7.6.2 fsh_access

```
enum    fsh_access {
        fsa_NONE = 0,       /* for completeness */
        fsa_R    = 1,       /* read only */
        fsa_W    = 2,       /* write only */
        fsa_RW   = 3        /* read/write */
};
```

fsh_access defines the legal file access modes.

6.7.6.3 nlm_share

```
struct  nlm_share {
        string caller_name<LM_MAXSTRLEN>;
        netobj fh;
        netobj oh;
        fsh_mode    mode;
        fsh_access  access;
};
```

The **nlm_share** structure defines the information needed to uniquely specify a share operation. The netobj's define the file, *fh*, and owner, *oh*; *caller_name* uniquely identifies the host. *mode* and *access* define the file sharing and the access modes.

6.7.6.4 nlm_shareargs

```
struct  nlm_shareargs {
        netobj cookie;
        nlm_share share;    /* actual share data */
        bool reclaim;       /* used for recovering shares */
};
```

The **nlm_shareargs** structure encodes the arguments for a share or unshare request. The boolean *reclaim* should be TRUE if the client is attempting to reclaim a previously-granted sharing request and FALSE otherwise.

6.7.6.5 nlm_shareres

```
struct    nlm_shareres {
      netobj cookie;
      nlm_stats stat;
      int sequence;
};
```

The **nlm_shareres** structure encodes the results of a share or unshare request. The *cookie* and *sequence* should be ignored; they are required only for compatibility reasons. The result of the request is given by *stat*.

6.7.6.6 nlm_notify

```
struct    nlm_notify {
      string name<MAXNAMELEN>;
      long state;
};
```

This structure encodes the arguments for releasing all locks and shares a client holds.

6.7.7 Server Procedures

The following sections summarise the protocol used by the Network Lock Manager using RPC Language.

```
/*
 * Over-the-wire protocol used between the network lock managers
 */
program NLM_PROG {
      version NLM_VERS {
                  void            NLM_NULL(void) = 0;
                  nlm_res         NLM_UNLOCK(struct nlm_unlockargs) = 4;
                  nlm_shareres    NLM_SHARE(nlm_shareargs)       = 20;
                  nlm_shareres    NLM_UNSHARE(nlm_shareargs) = 21;
                  nlm_res         NLM_NM_LOCK(nlm_lockargs)   = 22;
                  void            NLM_FREE_ALL(nlm_notify)       = 23;
      } = 3;
} = 100021;
```

6.7.8 NLM_NULL Specification - Do Nothing

6.7.8.1 RPC Data Descriptions

Call Arguments

None.

Return Arguments

None.

6.7.8.2 RPC Procedure Description

```
void
NLM_NULL(void) = 0;
```

6.7.8.3 Description

This procedure does no work. By convention, procedure zero of any RPC program takes no parameters and returns no results. It is made available to allow server response testing and timing.

6.7.8.4 Return Codes

None.

6.7.9 NLM_UNLOCK Specification - Unlock File

6.7.9.1 RPC Data Descriptions

Call Arguments

```
struct nlm_unlockargs {
      netobj cookie;
      struct nlm_lock alock;
};
```

Return Arguments

```
struct nlm_res {
      netobj cookie;
      nlm_stat stat;
};
```

6.7.9.2 RPC Procedure Description

```
nlm_res
NLM_UNLOCK (nlm_unlockargs) = 4;
```

6.7.9.3 Description

This routine will remove the specified lock from the file identified by *alock*.

6.7.9.4 Return Codes

When the procedure returns, *stat* will be set to the following value:

nlm_granted Indicates that the call completed successfully.

6.7.10 NLM_SHARE Specification - Share a File

6.7.10.1 RPC Data Descriptions

Call Arguments

```
struct  nlm_shareargs {
        netobj      cookie;
        nlm_share   share;    /* actual share data */
        bool                  reclaim; /* used for recovering shares */
};
```

Return Arguments

```
struct  nlm_shareres {
        netobj cookie;
        nlm_stats stat;
        int sequence;
};
```

6.7.10.2 RPC Procedure Description

```
nlm_shareres
NLM_SHARE ( nlm_shareargs ) = 20;
```

6.7.10.3 Description

This procedure indicates that the client wishes to open the file *share_fh* for access *share.access* in sharing mode *share.mode*. If this does not conflict with existing use of the file, the request will be granted. If a conflict does exist, the request is rejected immediately. It is the responsibility of the client to retry any rejected requests. The server will examine any entry sharing reservations for this file to determine if the share is permitted. If it is, a sharing reservation is created for this file.

Once a sharing reservation has been established, the lock manager will make no attempt to verify that the reservation is still valid; if the client system crashes and restarts while the reservation is still in effect, it should call the *NLM_FREE_ALL* procedure to release the reservation.

6.7.10.4 Return Codes

When the procedure returns, *stat* will be set to the following value:

nlm_granted Indicates that the call completed successfully.

nlm_denied Indicates that the call failed because the request conflicted with existing sharing reservations for the file.

nlm_denied_no_locks Indicates that the call failed because the lock manager could not allocate the resources needed to process the request.

nlm_denied_grace_period
Indicates that the call failed because the lock manager was not yet ready to accept normal service requests.

6.7.11 NLM_UNSHARE Specification - Unshare a File

6.7.11.1 RPC Data Descriptions

Call Arguments

```
struct    nlm_shareargs {
      netobj       cookie;
      nlm_share    share;    /* actual share data */
      bool                   reclaim; /* used for recovering shares */
};
```

Return Arguments

```
struct    nlm_shareres {
      netobj cookie;
      nlm_stats stat;
      int sequence;
};
```

6.7.11.2 RPC Procedure Description

```
nlm_shareres
NLM_UNSHARE (nlm_shareargs) = 21;
```

6.7.11.3 Description

This procedure informs the lock manager that the client has closed the file *share.fh*, and any corresponding share reservations should be released.

6.7.11.4 Return Codes

When the procedure returns, *stat* will be set to the following value:

nlm_granted Indicates that the call completed successfully.

6.7.12 NLM_NM_LOCK Specification - Non-monitored Lock

6.7.12.1 RPC Data Descriptions

Call Arguments

```
struct nlm_lockargs {
    netobj cookie;
        bool block;
        bool exclusive;
        struct nlm_lock alock;
        bool reclaim;
        int state;
};
```

Return Arguments

```
struct nlm_res {
    netobj cookie;
    nlm_stat stat;
};
```

6.7.12.2 RPC Procedure Description

```
nlm_res
NLM_NM_LOCK (nlm_lockargs) = 22;
```

6.7.12.3 Description

This procedure establishes a non-monitored lock on *alock.l_len* bytes starting at offset *alock.l_offset* in the file identified by *alock.fh*. The phrase ''non-monitored'' refers to the fact that the lock manager will make no attempt to verify that the lock is still valid; if the client system crashes and restarts while the lock is still in effect, it should call the *NLM_FREE_ALL* procedure to release the lock.

6.7.12.4 Return Codes

When the procedure returns, *stat* will be set to the following value:

nlm_granted Indicates that the call completed successfully.

nlm_denied Indicates that the call failed because the request conflicted with existing locks for the file.

nlm_denied_no_locks Indicates that the call failed because the lock manager could not allocate the resources needed to process the request.

nlm_denied_grace_period
Indicates that the call failed because the lock manager was not yet ready to accept normal service requests.

6.7.13 NLM_FREE_ALL Specification - Free All

6.7.13.1 RPC Data Descriptions

Call Arguments

```
struct   nlm_notify {
      string name<MAXNAMELEN>;
      unsigned int state;
};
```

Return Arguments

```
void
```

6.7.13.2 RPC Procedure Description

```
void
NLM_FREE_ALL (nlm_notify) = 23;
```

6.7.13.3 Description

The *NLM_FREE_ALL* procedure informs the server that the client *name* has been rebooted and that all file sharing reservations and file locks currently being held on behalf of the client should be discarded. The *state* field is unused.

6.7.13.4 Return Codes

None.

Chapter 7

RPC Interface to UDP Transport Services

7.1 INTRODUCTION

The purpose of this specification is to describe how NFS interfaces with the underlying transport. The NFS protocol provides transparent remote access to shared filesystems over local networks. The NFS protocol is designed to be machine, operating system, network architecture and transport protocol independent. The independence is achieved through the use of Remote Procedure Call (RPC) primitives built on top of an eXternal Data Representation (XDR). This specification will deal with the interface between RPC and the underlying transport.

Though NFS is designed to be transport independent, this specification will only deal with the implementation of RPC on top of UDP/IP. It should also be noted that this specification contains no mention of the programmatic interface to UDP, as this is implementation-specific.

7.2 RPC AND TRANSPORT REQUIREMENTS

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 note 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 the application knows it is running on top of a reliable transport such as TCP/IP, then most of the work is already done for it. If, however, it is running on top of an unreliable transport such as UDP/IP, the application must implement its 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.

7.3 UDP AS A TRANSPORT PROTOCOL

UDP (User Datagram Protocol) is a datagram-based protocol that relies on the Internet Protocol (IP) transport for packet delivery. Because it is a datagram service without any connection, retransmission or ordering information, UDP delivery is unreliable. Although packets generally reach their destination, it cannot be guaranteed. They may be lost, duplicated or arrive out of order.

A UDP packet consists of a UDP header followed by data. The whole is passed to the IP layer for transmission. The IP layer delivers the data packet to the correct host specified by the destination IP address and the UDP layer targets the specific destination within the host, specified by a destination port number.

Full specifications of the UDP and IP protocols is contained in **RFC 768 User Datagram Protocol** and **RFC 791 Internet Protocol .**

7.4 RPC INTERFACE

7.4.1 The RPC request

A UDP packet containing an RPC request would be as follows:

0 15	16 31
Source Port	Destination Port
Length	Checksum
Data octets	

- Source Port
  The 16-bit port number the NFS client is using.
- Destination Port
  The 16-bit port number on the destination host. This must be specified by the protocol layer above UDP, either when the port is allocated or on a per-datagram basis depending on the protocol implementation. For NFS version 2, described in this specification, this is the port the NFS server resides at 2049 (decimal).
- Length
  The number of bytes in the packet. This includes the UDP header and the data (RPC packet in this case).
- Checksum
  The checksum is the 16 bit one's complement of the one's complement sum of all 16 bit words in the pseudo-header, UDP header and raw data.

  The UDP pseudo-header consists of the source and destination IP addresses, the Internet Protocol Number for UDP (17 decimal) and the UDP length (see **RFC 768**). An implementation may choose not to compute a UDP checksum when transmitting a packet, in which case it should set the checksum field to zero.
- Data Octets
  Provided by the protocol layer above UDP. In this case, this is the RPC request itself.

In addition, the destination of the UDP packet must be specified as an IP address.

7.4.2 The RPC reply

Once the RPC request has been received and processed by the NFS server, a reply packet must be constructed and sent to the NFS client.

In most implementations, the IP protocol layer will provide the upper layer protocols with the source and destination IP addresses of the request packet. This information can be used to construct the return packet. The source port and IP address from the RPC request become destination port and IP address of the RPC reply.

The data in the UDP packet is the RPC reply which will contain results and return data from the NFS server.

7.4.3 Receiving a UDP Reply Packet

Due to the unreliability inherent in a connectionless transport, implementations should handle waiting for a packet which will complete by receipt of a packet, a timeout or some form of exception (e.g., termination of the application).

7.4.4 Closing

Since UDP is a connectionless transport, no explicit actions are required to terminate the client/server relationship, although particular implementations may require the freeing of data structures, etc.

Appendix A

Mapping Filenames and Attributes

A.1 INTRODUCTION

In a homogeneous network - for example, one consisting entirely of DOS systems - filename and file attribute mapping is not an issue; there is a one-to-one correspondence between names and attributes on client and server. In a heterogeneous network - in the current context, one which includes DOS- and X/Open-compliant systems - the situation is more complicated. The semantics of the virtual file system provided by the server must be as close as possible to the semantics implemented by the client's local operating system. Since it is desirable to facilitate the sharing of files between applications running on the client and the server, any mapping should introduce as little inconsistency as possible, and should follow the principle of ''least surprise''. For example, users of DOS and UNIX will generally employ filenames which they can specify by typing a lower-case alphanumeric string, such as **abc**. In UNIX, the corresponding filename is **abc**, but under DOS it is translated to **ABC**. It is desirable that the same filename should identify the same file on each system.

There are two ways in which dissimilar file system semantics can be resolved. If the server performs the resolution, presenting a virtual file system view which is compatible with the client OS, the mapping becomes part of the server and protocol architecture and can be standardised. However, in NFS every server presents a common virtual file system model to all clients, and it is the responsibility of each client system to map the NFS semantics into its local operating system. Since the intent of this standard is to specify the server functionality necessary to create an X/Open-compliant system server for (PC)NFS clients, client implementation issues are formally outside the scope of this standard. Nonetheless, interoperability may be enhanced by reducing inconsistencies between implementations, and the following description is therefore presented as an illustration and recommendation.

A.2 CONTEXT

Filename and attribute translations occur in two ways.

1. When a client system retrieves information from the server, the server names and attributes must be translated into those of the client. This is referred to here as ‘‘mapping’’. For example, if a DOS application issues a series of DOS *FindFirst/FindNext* requests to retrieve the contents of a directory, each NFS filename and the corresponding file attributes must be mapped into the corresponding DOS elements.
2. When a client system changes the virtual file store by creating or renaming a file, or by changing a file's attributes, it is necessary to translate the DOS name and attributes into the equivalent NFS elements. This is referred to here as ‘‘back-mapping’’.

We now consider each case in turn.

A.3 MAPPING FILE NAMES

The following procedure may be used in mapping NFS files to DOS:

1. The special names ‘‘.’’ and ‘‘..’’ are not translated in any way.
2. The NFS name is case-inverted.
3. If the resulting name is a legal DOS name and contains no lower-case letters, the process is complete. A legal DOS name may not include the following characters:

   . , + [ ] * ? : \ / ; = < >

   A legal DOS filename must be 1 to 8 characters long, optionally followed by a period (.) and an extension of 1 to 3 characters. (To avoid back-mapping problems, any NFS name which includes a tilde (˜) as the sixth character is also mapped.)
4. If the resulting name has a legal DOS extension, the extension and period (.) are removed and saved. The corresponding characters are also trimmed from the NFS name.
5. The NFS name is then mapped to generate a new filename. The first five characters of the NFS name are copied to the new filename. All lower-case letters are translated to upper-case. All illegal DOS characters are replaced by tilde characters. If necessary, the new filename is padded to five characters with tilde characters.
6. The new filename is now extended to eight characters by the addition of a tilde and two legal DOS characters. These characters are the ‘‘mapping key’’. They should be chosen to facilitate the back-mapping process while minimising duplications. If the NFS name is already stored, the same mapping key should be used.
7. The NFS name is now stored using all or part of the new filename as a key.
8. Finally, if an extension was saved in step 4, it is appended to the new filename.

Note that this mapping process always yields a name with an eight-character filename component in which the sixth character is a tilde.

Sample UNIX file name	Mapped DOS file name	Notes
abc123.def	ABC123.DEF	No mapping required.
a	A	No mapping required.
A	A˜˜˜˜˜*XX*	Upper-case mapping.
a_long_name	A_LON˜*XX*	Using first 5 characters.
AB.c	AB˜˜˜˜*XX*.C	Using all characters, padding with tildes.
Ab.c	AB˜˜˜˜*XX*.C	Note that the XX value will be different from the last example.
a.b.c	A˜B˜˜˜*XX*.C	Illegal because of multiple extensions. The first period is replaced by a tilde; the extension is OK.
abcd.efgh	ABCD˜˜*XX*	The extension is illegal.
.login	˜LOGI˜XX	The ‘‘hidden’’ attribute will also be set.

A.4 BACK-MAPPING FILENAMES

The following procedure may be used in translating a DOS name into NFS form:

1. If the DOS name has an eight-character filename component in which the sixth character is a tilde, back-mapping is required. If not, all upper-case letters are converted to lower-case and the process is complete.
2. The ‘‘mapping key’’ is extracted from the DOS filename and used as a key to retrieve the NFS name. If a corresponding NFS name cannot be found, the DOS system call is terminated with an error.
3. If the DOS name included an extension, it is appended (with its leading period) to the NFS name, converting any upper-case letters to lower-case.

A.5 MAPPING FILE ATTRIBUTES

The following procedure may be used to map NFS file attributes into the DOS equivalent.

1. Using implementation-dependent authentication information, determine which set of NFS access permissions (owner, group or other) apply to this file for the current user.
2. If the NFS *read* and *execute* permission bits are both clear, hide this file completely from directory searches.
3. If the NFS *execute* permission is set but *read* is clear, do not hide the file from directory searches, but do not allow this file to be opened for reading except as part of an *exec* DOS function.
4. If the NFS *write* permission is clear, set the DOS *read-only* attribute.
5. If the NFS *setuid* attribute is set, set the DOS *hidden* attribute.
6. If the NFS *directory* attribute is set, set the DOS *directory* attribute.
7. Clear the DOS *system* and *volume label* attributes (but if requested in a *FindFirst* directory search, return the NFS server name as the volume label).
8. Set the DOS *archive* attribute. (This may be a configurable option.)

   In addition, if this mapping occurs as part of a directory search, and if the corresponding NFS name begins with a ‘‘ . ’’ and was mapped, the principle of least surprise suggests that the DOS *hidden* attribute be set.

A.6 BACK-MAPPING FILE ATTRIBUTES

The following procedure may be used to set NFS file attributes based on DOS settings.

1. If a file is being created, set the NFS access permissions based upon the current *umask*.
2. If the DOS *read-only* attribute is being set, clear the NFS *write* permission for all classes of access (owner, group, other). Note that if a file is created read-only, the client may choose to defer setting this attribute until file close.
3. If the DOS *read-only* attribute is being cleared, set the NFS *write* permission for all classes of access (owner, group, other), except where the current *umask* value would disallow this.
4. If the DOS *hidden* attribute is being set, set the NFS *setuid* attribute.
5. If the DOS *hidden* attribute is being cleared, clear the NFS *setuid* attribute.

Appendix B

NFS Transmission Analysis

B.1 INTRODUCTION

This appendix describes the DOS system calls which are intercepted by PC-NFS and translated into NFS Remote Procedure Calls. In most cases the need to intercept and emulate these calls is obvious, although one or two may be less intuitive. The general flow is indicated, but this is not claimed to be anywhere near an exhaustive description. Note that cache logic may cause the equivalent of a write function (0x40) to occur at almost any time.

It should also be noted that there are a number of features of the PC-NFS redirector which represent proprietary intellectual property. For example, PC-NFS intercepts and emulates a number of additional DOS system calls and software interrupts; in addition it processes certain combinations or varieties of DOS system calls using techniques which may differ from the basic model presented herein. Nonetheless, this description adequately captures the mapping between DOS file system services and NFS RPC requests, including some of the subtlety and complexity of the architecture, while at the same time preserving the confidentiality of any proprietary technology.

This material only refers to the DOS Functions available via DOS interrupt 0x21.

B.2 DOS FUNCTIONS

This section contains a table of DOS Functions ordered by function number, followed by a list of DOS Functions in alphabetic order.

Function Number	DOS Call
0x00	Terminate Program
0x01	Read Keyboard and Echo
0x02	Display Character
0x03	Auxiliary Input
0x04	Auxiliary Output
0x05	Print Character
0x06	Direct Console I/O
0x07	Direct Console Input
0x08	Read Keyboard
0x09	Display String
0x0a	Buffered Keyboard Input
0x0b	Check Keyboard Buffer Status
0x0c	Flush Buffer, Read Keyboard
0x0d	Reset Disk
0x0e	Select Disk
0x0f	Open File (FCB I/O)
0x10	Close File (FCB I/O)
0x11	Search For First Entry
0x12	Search For Next Entry
0x13	Delete File (FCB I/O)
0x14	Sequential Read (FCB I/O)
0x15	Sequential Write (FCB I/O)
0x16	Create File (FCB I/O)
0x17	Rename File (FCB I/O)
0x19	Get Current Disk
0x1b	Get Default Drive Data
0x1c	Get Drive Data
0x21	Random Read (FCB I/O)
0x22	Random Write (FCB I/O)
0x23	Get File Size (FCB I/O)
0x27	Random Block Read (FCB I/O)
0x28	Random Block Write (FCB I/O)
0x36	Get Disk Free Space
0x39	Create Directory
0x3a	Remove Directory
0x3b	Change Current Directory
0x3c	Create File Handle
0x3d	Open File Handle
0x3e	Close File Handle

Function Number	DOS Call
0x3f	Read Via File Handle
0x40	Write Via File Handle
0x41	Delete Directory Entry
0x42	Move File Pointer
0x43	Set/Get File Attributes
0x44	IOCTL
0x45	Duplicate File Handle
0x46	Force Duplicate File Handle
0x47	Get Current Directory
0x4b	Load and Execute Program/Load Overlay
0x4c	End Process
0x4e	Find First File
0x4f	Find Next File
0x56	Change Directory Entry
0x57	Set/Get Date/Time of File
0x59	Get Extended Error
0x5a	Create Temporary File Handle
0x5b	Create New File
0x5c	Unlock/Lock File
0x5e	Get Machine Name
0x5f	Get Assign List Entry
0x68	Flush Buffer

Auxiliary Input

Function number 0x03

Description Auxiliary Input

Reason Handles standard I/O redirection; mapped into function 0x14 if handle is remote.

Auxiliary Output

Function number 0x04

Description Auxiliary Output

Reason Handles standard I/O redirection; mapped into an internal version of function 0x15 if handle is remote.

Buffered Keyboard Input

Function number 0x0a

Description Buffered Keyboard Input

Reason Handles standard I/O redirection; mapped into an internal version of function 0x14 and/or 0x15 if one or both handles are remote.

Change Current Directory

Function number 0x3b

Description Change Current Directory

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_LOOKUP Lookup the path.

NFSPROC_READLINK If necessary, read a symbolic link.

Change Directory Entry

Function number 0x56

Description Change Directory Entry

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_LOOKUP Lookup the path.

NFSPROC_READLINK If necessary read any symbolic links.

NFSPROC_RENAME Rename the file or directory.

Check Keyboard Buffer Status

Function number 0x0b

Description Check Keyboard Buffer Status

Reason Handles standard I/O redirection; checks FCB file position if handle is remote.

Close File (FCB I/O)

Function number 0x10

Description Close File (FCB I/O)

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_WRITE Write file data.

NFSPROC_GETATTR Get file attributes.

NFSPROC_SETATTR Set file attributes.

If sharing/locking is configured, the following Network Lock Manager call is also made:

NLM_UNSHARE Release sharing mode access to file.

Close File Handle

Function number 0x3e

Description Close File Handle

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_GETATTR If the file was ‘‘create-RO’’, we must ...

NFSPROC_SETATTR ... update the file attributes.

If sharing/locking is configured, the following Network Lock Manager calls may also be made:

NLM_UNLOCK Release any locks.

NLM_UNSHARE Release sharing mode access to file.

Create Directory

Function number 0x39

Description Create Directory

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_LOOKUP Lookup the path.

NFSPROC_READLINK If necessary, read a symbolic link.

NFSPROC_MKDIR Create the directory.

Create File (FCB I/O)

Function number 0x16

Description Create File (FCB I/O)

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

First, close the file if it was open (see function 0x10). Then attempt to create it:

NFSPROC_CREATE Create a file.

Create File Handle

Function number 0x3c

Description Create File Handle

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_LOOKUP	Lookup the path.
NFSPROC_READLINK	If necessary read any symbolic links.
NFSPROC_REMOVE	Remove the file (if it exists).
NFSPROC_CREATE	Create the file.

If sharing/locking is configured, the following Network Lock Manager call is also made:

NLM_SHARE	Request sharing mode access to file.

Create New File

Function number 0x5b

Description Create New File

Reason Emulate DOS file system function; (see function 0x39 for details).

Create Temporary File Handle

Function number 0x5a

Description Create Temporary File Handle

Reason Emulate DOS file system function; (see function 0x39 for details).

Delete Directory Entry

Function number 0x41

Description Delete Directory Entry

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_LOOKUP	Lookup the path.
NFSPROC_READLINK	If necessary read any symbolic links.
NFSPROC_REMOVE	Remove (unlink) the file.

Delete File (FCB I/O)

Function number 0x13

Description Delete File (FCB I/O)

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls, depending on search criteria and directory contents:

NFSPROC_GETATTR Get file attributes (for directory).

NFSPROC_READDIR Read a directory (as often as needed).

NFSPROC_LOOKUP Lookup a name in a directory (for matches).

If the name was a symbolic link:

NFSPROC_READLINK If necessary read a symbolic link, and ...

NFSPROC_LOOKUP Lookup the name in a directory.

NFSPROC_REMOVE Remove (unlink) a file.

Direct Console I/O

Function number 0x06

Description Direct Console I/O

Reason Handles standard I/O redirection; mapped into an internal version of function 0x14 and/or 0x15 if one or both handles are remote.

Direct Console Input

Function number 0x07

Description Direct Console Input

Reason Handles standard I/O redirection; mapped into an internal version of function 0x14 and/or 0x15 if one or both handles are remote.

Display Character

Function number 0x02

Description Display Character

Reason Handles standard I/O redirection; mapped into an internal version of function 0x15 if handle is remote.

Display String

Function number 0x09

Description Display String

Reason Handles standard I/O redirection; mapped into an internal version of function 0x15 if handle is remote.

Duplicate File Handle

Function number 0x45

Description Duplicate File Handle

Reason Emulate DOS file system function; no network calls.

End Process

Function number 0x4c

Description End Process

Reason Emulate DOS file system function; invokes a variety of operations which may include function 0x3e.

Find First File

Function number 0x4e

Description Find First File

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

First, find the directory to be searched using:

NFSPROC_LOOKUP Lookup the path.

NFSPROC_READLINK If necessary read any symbolic links.

Then proceed as in function 0x11 (Search for list): depending on search criteria and directory contents:

NFSPROC_GETATTR Get file attributes (for directory).

NFSPROC_READDIR Read a directory (as often as needed).

NFSPROC_LOOKUP Lookup a name in a directory (for matches).

If the name is a symbolic link:

NFSPROC_READLINK If necessary read a symbolic link, and ...

NFSPROC_LOOKUP ... lookup the name in a directory.

Find Next File

Function number 0x4f

Description Find Next File

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls, depending on search criteria and directory contents:

NFSPROC_GETATTR Get file attributes (for directory).

NFSPROC_READDIR Read a directory (as often as needed).

NFSPROC_LOOKUP Lookup a name in a directory (for matches).

If the name is a symbolic link:

NFSPROC_READLINK If necessary read a symbolic link, and ...

NFSPROC_LOOKUP ... lookup the name in a directory.

Flush Buffer

Function number 0x68

Description Flush Buffer

Reason Emulate DOS file system function; may invoke the function 0x40.

Flush Buffer, Read Keyboard

Function number 0x0c

Description Flush Buffer, Read Keyboard

Reason Handles standard I/O redirection; mapped into an internal version of function 0x14 and/or 0x15 if one or both handles are remote.

Force Duplicate File Handle

Function number 0x46

Description Force Duplicate File Handle

Reason Emulate DOS file system function; no network calls.

Get Assign List Entry

Function number 0x5f

Subfunction number 0x02

Description Get Assign List Entry

Reason Emulated locally, no network calls.

Get Current Directory

Function number 0x47

Description Get Current Directory

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

First, loop back to the root of this drive using:

NFSPROC_GETATTR Get directory attributes.

NFSPROC_LOOKUP Lookup ‘‘..’’ to find the parent.

Next, loop back down to the current directory using:

NFSPROC_READDIR Read the names.

Get Current Disk

Function number 0x19

Description Get Current Disk

Reason Emulate DOS file system function; no network calls.

Get Default Drive Data

Function number 0x1b

Description Get Default Drive Data

Reason Emulate DOS file system function; subset of function 0x36.

Get Disk Free Space

Function number 0x36

Description Get Disk Free Space

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_STATFS Get file system status.

Get Drive Data

Function number 0x1c

Description Get Drive Data

Reason Emulate DOS file system function; subset of function 0x36.

Get Extended Error

Function number 0x59

Description Get Extended Error

Reason Emulate DOS extended error handling.

Get File Size (FCB I/O)

Function number 0x23

Description Get File Size (FCB I/O)

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_GETATTR Get file attributes.

Get Machine Name

Function number 0x5e

Subfunction number 0x00

Description Get Machine Name

Reason Emulated locally, no network calls.

IOCTL

Function number 0x44

Description IOCTL

Reason Emulate DOS file system function; no network calls.

Load and Execute Program/Load Overlay

Function number 0x4b

Description Load and Execute Program/Load Overlay

Reason Emulate DOS file system function; invokes a variety of operations which may include internal versions of function 0x3d (Open File Handle) and function 0x3f (Read via File Handle).

Move File Pointer

Function number 0x42

Description Move File Pointer

Reason Emulate DOS file system function; no network calls.

Open File (FCB I/O)

Function number 0x0f

Description Open File (FCB I/O)

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

First close the file if it was open (see function 0x10). Then open it:

NFSPROC_LOOKUP Lookup a name in a directory.

NFSPROC_READLINK If necessary read any symbolic links.

NFSPROC_GETATTR Get file attributes.

If sharing/locking is configured, the following Network Lock Manager call is also made:

NLM_SHARE Request sharing mode access to file.

Open File Handle

Function number 0x3d

Description Open File Handle

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_LOOKUP Lookup the path.

NFSPROC_READLINK If necessary read any symbolic links.

If sharing/locking is configured, the following Network Lock Manager call is also made:

NLM_SHARE Request sharing mode access to file.

Print Character

Function number 0x05

Description Print Character

Reason Not emulated.

Random Block Read (FCB I/O)

Function number 0x27

Description Random Block Read (FCB I/O)

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_READ Read file data.

Random Block Write (FCB I/O)

Function number 0x28

Description Random Block Write (FCB I/O)

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_WRITE Write file data.

Random Read (FCB I/O)

Function number 0x21

Description Random Read (FCB I/O)

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_READ Read file data.

Random Write (FCB I/O)

Function number 0x22

Description Random Write (FCB I/O)

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_WRITE Write file data.

Read Keyboard

Function number 0x08

Description Read Keyboard

Reason Handles standard I/O redirection; mapped into an internal version of 0x14 if handle is remote. (Same as function 0x07 except for ctrl-break processing).

Read Keyboard and Echo

Function number 0x01

Description Read Keyboard and Echo

Reason Handles standard I/O redirection; mapped into internal versions of function 0x14 and/or 0x15 if one or both handles are remote.

Read Via File Handle

Function number	0x3f
Description	Read Via File Handle
Reason	Emulate DOS file system function; since read data may be cached, the following NFS Remote Procedure Call may or may not be required:

NFSPROC_READ	Read file data.

Remove Directory

Function number	0x3a
Description	Remove Directory
Reason	Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_LOOKUP	Lookup the path; if necessary, we must ...
NFSPROC_READLINK	... read a symbolic link.
NFSPROC_RMDIR	Remove the directory.

Rename File (FCB I/O)

Function number	0x17
Description	Rename File (FCB I/O)
Reason	Emulate DOS file system function; performs the following NFS Remote Procedure Calls, depending on search criteria and directory contents:

NFSPROC_GETATTR	Get file attributes (for directory).
NFSPROC_READDIR	Read a directory (as often as needed).
NFSPROC_LOOKUP	Lookup a name in a directory (for matches).

If the name was a symbolic link:

NFSPROC_READLINK	If necessary read a symbolic link, and ...
NFSPROC_LOOKUP	... lookup the name in a directory
NFSPROC_RENAME	Rename a file or directory.

Reset Disk

Function number	0x0d
Description	Reset Disk
Reason	No emulation needed as NFS is synchronous.

Search For First Entry

Function number 0x11

Description Search For First Entry

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls, depending on search criteria and directory contents:

NFSPROC_GETATTR Get file attributes (for directory).

NFSPROC_READDIR Read a directory (as often as needed).

NFSPROC_LOOKUP Lookup a name in a directory (for matches).

If the name was a symbolic link:

NFSPROC_READLINK If necessary read a symbolic link, and ...

NFSPROC_LOOKUP ... lookup the name in a directory.

Search For Next Entry

Function number 0x12

Description Search For Next Entry

Reason Emulate DOS file system function; performs the same NFS Remote Procedure Calls as function 0x11 above.

Select Disk

Function number 0x0e

Description Select Disk

Reason Emulate DOS file system function; no network calls generated.

Sequential Read (FCB I/O)

Function number 0x14

Description Sequential Read (FCB I/O)

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_READ Read file data.

Sequential Write (FCB I/O)

Function number 0x15

Description Sequential Write (FCB I/O)

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_WRITE Write file data.

Set/Get Date/Time of File

Function number 0x57

Subfunction number 0x00

Description Get Date/Time of File

Reason Emulate DOS file system function; no network calls.

Subfunction number 0x01

Description Set Date/Time of File

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Call:

NFSPROC_SETATTR Set file attributes.

There are additional undocumented subfunctions, none of which cause network calls to be made.

Set/Get File Attributes

Function number 0x43

Subfunction number 0x00

Description Get File Attributes

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls:

NFSPROC_LOOKUP Lookup the path.

NFSPROC_READLINK If necessary read any symbolic links.

Subfunction number 0x01

Description Set File Attributes

Reason Emulate DOS file system function; performs the following NFS Remote Procedure Calls (but not if this is a request to set the volume label):

NFSPROC_LOOKUP Lookup the path.

NFSPROC_READLINK If necessary read any symbolic links.

NFSPROC_SETATTR Set file attributes.

Terminate Program

Function number 0x00

Description Terminate Program

Reason Process termination conditions, including releasing of locks, flushing buffers, etc. (See function 0x4c for more details).

Unlock/Lock File

Function number 0x5c

Subfunction number 0x00

Description Lock File

Reason Emulate DOS file system function; performs the following Network Lock Manager call:

NLM_NM_LOCK Request (non-monitored) byte range lock.

Subfunction number 0x01

Description Unlock File

Reason Emulate DOS file system function; performs the following Network Lock Manager call:

NLM_UNLOCK Release byte range lock.

Write Via File Handle

Function number 0x40

Description Write Via File Handle

Reason Emulate DOS file system function; since read data may be cached, the following NFS Remote Procedure Call may or may not be required:

NFSPROC_WRITE Write file data.

In addition, the following calls may be needed to update the file size:

NFSPROC_GETATTR Get file attributes.

NFSPROC_SETATTR Set file attributes.

Definitions

ACL

(Access Control List)

A list used to control access to a file or resource. The list contains the user IDs and/or group IDs that are allowed access to the file or resource.

ARP

(Address Resolution Protocol)

The protocol used to bind a high level Internet Address to a low level physical hardware address. It can only be used on networks that support hardware broadcast. The protocol is only across a single physical network.

ARPA

(Advanced Research Project Agency)

Part of the U.S. Department of Defense. This agency funded the ARPANET and DARPA Internet. Its present name is DARPA. It is located at 1400 Wilson Blvd, Arlington, VA, U.S.A..

ARPANET

A network built by BBN (Bolt, Beranek, and Newman, Incorporated) and funded by ARPA. It was one of the first large scale packet switched networks, and was used to link academic institutes involved with ARPA work. It helped with the early network research and formed a basis for Internet.

baud

The unit of signaling speed, i.e, one signal unit per second. It is the number of times per second the signal can change on a transmission line.

big-endian

The name of a particular byte order (coined by Danny Cohen). When looking at addresses in increasing order, the most significant byte comes first. The Internet protocols use big-endian byte order.

bps

(bits per second)

Measurement of the rate of data transmission.

bridge

A device that interconnects two or more networks at the MAC-layer (see **router**, **gateway**).

broadcast

The function of delivering a given packet to all hosts that are attached to the broadcasting delivery system. Broadcasting is implemented both at the hardware and the software levels.

byte

8 bits.

CAE

(Common Applications Environment)

chaining

Transmission of more than one SMB request in a single transport PDU.

client-server

The distributed system model where a requesting program (the client) interacts with a program that can satisfy the request (the server). The client initiates the interaction and may wait for the server to respond.

connection oriented service

A service provided between two endpoints along which data is passed in a sequenced and reliable way.

connection-less service

In a connection-less service each packet is a separate entity containing a source and destination address; therefore packets may be dropped or delivered out of sequence. The delivery service offered by the Internet Protocol (IP) is a connection-less service.

CRC

(Cyclic Redundancy Check)

An integer calculated from a sequence of octets used to check that errors have not occurred during their transmission. The CRC is calculated and transmitted with the octets. At the receiving end the CRC is re-calculated and compared with the value sent. If the values are identical the data is assumed to be error free.

DARPA

(Defense Advanced Projects Research Agency)

Formerly ARPA.

data encapsulation

The way a lower level protocol accepts a message from a higher level protocol and places it in the data portion of the low level frame.

daemon

A process that is not associated with any user. This sort of process performs system-wide functions, e.g., administration, control of networks and execution dependent activities.

datagram

A packet sent independently of the others in the network. It contains the source and destination addresses as well as the data.

DIB

(Directory Information Base)

distributed database

A distributed database which is split up into several components, with each component on a different computer. The end-user, however, is given the impression that only a single local database is used.

DSA

(ISO Directory Service Agent)

DSN

(Domain Service Name)

Part of the OSI naming hierarchy.

DUA

(ISO Directory User Agent)

effective group id

An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime.

effective user id

An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime.

ES

(End System)

Ethernet

A local area network developed by Digital Equipment Corporation, Intel and Xerox Corporation. The Ethernet is a passive coaxial cable with the interconnections containing all the active components.

expedited data

Data that is considered urgent. The specified semantics of expedited data are defined by the transport provider.

exec

The XSI system call that is used to start a process running.

FCB

(File Control Block)

The area of memory holding the file information and status. It is a term associated with DOS.

FID

(File ID)

A unique number associated with a file to enable it to be identified.

fifo

(First In First Out)

One of the file types supported on an XSI system. A fifo, the alternative name for a pipe, differs from a regular file because its data is transient, i.e., once data is read from the pipe it cannot be read again.

fork

The XSI system call which is used to create a new process. The process created is a duplicate of the calling process.

full-duplex

A transmission channel that can carry signals in both directions simultaneously.

gateway

A mechanism for interconnecting two or more networks which may use dissimilar protocols. The interconnection usually occurs at or above the transport layer (see router, bridge).

half-duplex

A transmission channel which can carry signals in both directions but not simultaneously.

ICMP

(Internet Control and Monitoring Protocol)

Part of the Internet Protocol Suite. ICMP is used to provide network layer management facilities, providing an error reporting facility and routing suggestions. ICMP also includes an echo request/reply, used to test whether a destination is reachable and responding.

IGP

(Interior Gateway Protocol)

Any protocol used to propagate network reachability and routing information within an autonomous system.

internet

A large virtual network made up of a series of networks interconnected by routers.

Internet, The

The cooperative virtual network that uses the TCP/IP protocol and includes the ARPANET, MILNET and NSFnet. It provides universal connectivity and reaches many universities, government, military and commercial establishments.

interoperability

The ability of software and hardware on multiple machines and from multiple vendors to communicate effectively.

ioctl

A system call which allows a process to specify control information to control a device. This function exists in both XSI and DOS.

IP

(Internet Protocol)

The protocol from the Internet Protocol Suite that provides the basis for internet communications.

IP

(Interworking Protocol)

The OSI protocol which supports the interconnection of separate OSI networks.

IPC

(Inter-Process Communication)

Methods by which two or more processes can communicate, e.g., formatted data streams or shared memory.

IS

(Intermediate System)

LAN

(Local Area Network)

A physical network that operates at a high speed over short distances, e.g., Ethernet.

LAP

(Link Access Procedure)

A subset of the high-level data link control for use as the link level in X.25 networks.

little-endian

The name of a particular byte order (coined by Danny Cohen). When looking at addresses in increasing order, the least significant byte comes first.

LMX

(LAN Manager UNIX)

The implementation of the LAN Manager on UNIX systems.

LSAP

(Link Layer Service Access Point)

MAC

(Media Access Control)

Low level interface to the hardware protocol.

MBZ

(Must Be Zero)

Reserved fields are often defined MBZ.

MID

(Multiplex Identifier)

A number which uniquely identifies a protocol request and response within a process.

multicast

A method by which copies of a single packet are passed to a selected subset of all destinations. Broadcast is a special case of multicast whereby the subset of destinations receiving a copy of the packet is the entire set of destinations.

named pipe

An interprocess communication mechanism defined by the extended SMB specification. Also a fifo.

NBDD

(NetBIOS Datagram Distribution Server)

Like NBNS this is part of the underlying NetBIOS mechanism and is invisible to the applications. The NBDD extends the NetBIOS datagram distribution service to support broadcasting or multicasting. NBDD is defined as a separate entity from NetBIOS name server.

NBNS

(NetBIOS Name Server)

NDSE

(NetBIOS Directory Service Entity)

NDUA

(NetBIOS Directory User Agent)

NetBIOS

(Network Basic Input Output System)

The *de facto* standard programatic interface to networks for DOS systems.

NFS

(Network File System)

A protocol which allows a set of computers access to each others file systems. NFS was developed by Sun Microsystems and is used primarily on UNIX systems.

NLM

(Network Lock Manager)

An RPC based service which provides advisory DOS file locking and access control synchronisation across the network. This service is used in conjunction with NFS.

non-secured NBNS (NetBIOS Name Server)

This server is part of the underlying NetBIOS mechanism and is invisible to the applications. The server manages the NetBIOS names. The server does not check for name consistency, it leaves this to the end-nodes.

NSFnet

(National Science Foundation NETwork)

The collection of networks across the U.S. sponsored by NSF.

NSP

(NetBIOS Name Service Protocol)

NSAP

(Network Service Access Point)

NSPDU

(NetBIOS Name Service Protocol Data Unit)

octet

8 bits

opportunistic lock

The server will notify the client, allowing it to flush its dirty buffers and unlock the file, when another client attempts to open the file.

OSI

(Open Systems Interconnect)

ISO standards for the interconnection of cooperative (open) computer systems.

packet

A block of data sent across a packet switching network.

PCNFSD

(Personal Computer NFS Daemon)

The daemon that provides personal computer NFS clients with authentication and printing services which are usually available in larger and more capable systems.

PDU

(Protocol Data Unit)

The basic unit of data manipulated by a protocol.

PID

(Process ID)

The number assigned to a process so that it can be uniquely identified.

psel

(presentation selector)

remote mount

The process by which one machine can mount a file system that exists on a remote machine so it can be accessed as if it were a local file system.

responder

An entity with which an initiator wishes to establish a transport connection.

RFC

(Request For Comments)

The name of a series of notes that contain surveys, measurements, ideas, techniques and observations, as well as proposed and accepted Internet protocol standards.

root (of filesystem)

The top directory in the directory hierarchical structure.

router

A mechanism for interconnection of two or more networks at the network layer (see **bridge**, **gateway**).

RPC

(Remote Procedure Call)

secured NBNS (NetBIOS Name Server)

Part of the underlying NetBIOS mechanism not visible to the applications above. Unlike the un-secured NBNS, the server monitors and participates in name activity to ensure consistency.

SMB

(Server Message Block)

A protocol which allows a set of computers to access shared resources as if they were local. The core protocol was developed by Microsoft and Intel, and the extended protocol was developed by Microsoft.

SNPA

(Subnetwork Point of Attachment)

SNPDU

(Subnetwork Protocol Data Unit)

socket

A program-defined endpoint for network communication between processes. Sockets are a particular paradigm used for interprocess communication.

ssel

(session selector)

stateful server

A stateful server is a server that maintains information about the state of the transactions it has processed; for example, whether or not a file is currently open.

stateless server

A stateless server is a server that does not maintain state information from one transaction to another.

TBD

(To Be Defined)

Further detail will be provided at a later time.

TCP

(Transmission Control Protocol)

The Internet standard transport level connection-oriented protocol. It provides a full duplex, reliable stream service which allows a process on one machine to send a stream of data to a process on another. Part of the Internet Protocol Suite.

TELNET

The protocol for remote terminal connection service that allows a user from one site to use a remote timesharing system on another site as if the terminal was connected directly to the remote machine. Generally implemented over TCP.

TPDU

(Transport Protocol Data Unit)

The basic unit of data manipulated by the transport layer of a protocol stack.

TSDU

(Transport Service Data Unit)

tsel

(transport selector)

TTL

(Time To Live)

Used to stop the existence of endlessly looping packets. Each packet is assigned an integer which is decremented each time it passes through a router. If the integer reaches zero the router discards the packet.

UDP

(User Datagram Protocol)

The Internet connection-less protocol. Part of the Internet Protocol Suite.

UID

(User Identifier)

A token representing an authenticated <*username, password*> tuple. UIDs are registered by the redirectors.

umask

The XSI process' file mode creation mask used during file and directory creation. Bit positions that are set in the umask are cleared in the mode of the newly created file or directory. The umask is set using the *umask*() call.

VC

(Virtual Circuit)

The path between two communicating systems that provides a reliable, sequenced data delivery service.

WORD

Consists of two bytes, ordered such that the low-order byte precedes the high byte. (e.g., like a VAX).

working directory

A directory, associated with a process, that is used in pathname resolution for pathnames that do not begin with a slash.

XDR

(External Data Representation)

A machine independent data representation scheme developed by Sun Microsystems.

X.25

The CCITT standard protocol for network level network service.

Index

adjunct protocols: 87
authentication using PCNFSD: 98
basic data types: 56
External Data Representation: 53
file system: 55
file system model: 55
mount data types: **103**
- dirpath: 104
- fhandle: 103
- fhstatus: 103
- name: 104

mount protocol: **102**
- basic data types: 103
- RPC information: 103
- XDR structure sizes: 103

mount server procedures: **104**
- MNTPROC_DUMP: 106
- MNTPROC_EXPORT: 108
- MNTPROC_MNT: 105
- MNTPROC_NULL: 104
- MNTPROC_UMNT: 107
- MNTPROC_UMNTALL: 107

Network File System: 53
network lock manager protocol: 109
NFS: **53**, 55
- implementation: 62
- permission issues: 62
- server/client relationship: 62

NFS data types: **56**
- attrstat: 61
- diropargs: 61
- diropres: 61
- fattr: 59
- fhandle: 58
- filename: 60
- ftype: 58
- path: 60
- sattr: 60
- stat: 56
- timeval: 58

NFS protocol definition: 55
NFS server procedures: **64**
- NFSPROC_CREATE: 73
- NFSPROC_GETATTR: 65
- NFSPROC_LINK: 77
- NFSPROC_LOOKUP: 68
- NFSPROC_MKDIR: 80
- NFSPROC_NULL: 65
- NFSPROC_READDIR: 83
- NFSPROC_READLINK: 69
- NFSPROC_READ: 70
- NFSPROC_REMOVE: 75
- NFSPROC_RENAME: 76
- NFSPROC_RMDIR: 81
- NFSPROC_ROOT: 67
- NFSPROC_SETATTR: 66
- NFSPROC_STATFS: 84
- NFSPROC_SYMLINK: 79
- NFSPROC_WRITECACHE: 71
- NFSPROC_WRITE: 72

NFS version-2 protocol specification: 53
NFS RPC information: 56
NLM data types: 110, 113
- fsh_access: 113
- fsh_mode: 113
- netobj: 110
- nlm_holder: 111
- nlm_lock: 112
- nlm_lockargs: 112
- nlm_res: 111
- nlm_share: 113
- nlm_shareargs: 113
- nlm_shareres: 114
- nlm_stat: 111
- nlm_stats: 111
- nlm_unlockargs: 112

NLM protocol:
- additional data types: 113
- basic data types: 110
- RPC information: 110

NLM server procedures: 114
- NLM_FREE_ALL: 119
- NLM_NM_LOCK: 118
- NLM_NULL: 114
- NLM_SHARE: 116
- NLM_UNLOCK: 115
- NLM_UNSHARE: 117

PCNFSD data types: 94

- arstat: 96
- client: 95
- ident: 94
- options: 95
- password: 95
- pirstat: 96
- printername: 95
- psrstat: 97
- spoolname: 95
- username: 95

PCNFSD protocol: 93
- basic data types: 94
- RPC information: 94
- XDR structure sizes: 94

PCNFSD server procedures: 97
- PCNFSD_AUTH: 98
- PCNFSD_NULL: 98
- PCNFSD_PR_INIT: 99
- PCNFSD_PR_START: 100

print spooling:
- initialisation: 99
- printing a file: 100

Remote Procedure Call: 53

stateless servers: 54

XDR: **0**
- array, fixed length: 28
- array, variable length: 28
- basic block size: 24
- block size: 24
- boolean: 26
- constant: 30
- data types: 25
- discriminated union: 29
- enumeration: 26
- fixed-length array: 28
- fixed-length opaque data: 26
- integer: 25
- integer, unsigned: 25
- opaque data, fixed length: 26
- opaque data, variable length: 26
- protocol specification: 0
- string: 27
- structure: 29
- typedef: 30
- union discriminated: 29
- unsigned integer: 25
- variable-length array: 28
- variable-length opaque data: 26
- void: 30
- RFC status: 0

XDR language: **33**
- notation: 33
- syntax: 33, 35

XDR structure sizes: 56

XDR RFC: 0