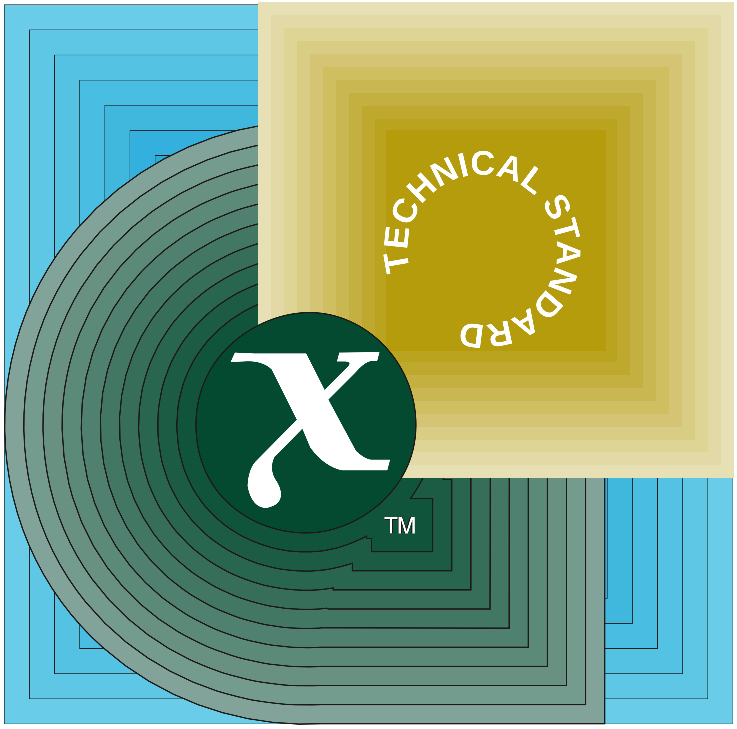
Technical Standard

Protocols for Interworking: XNFS, Version 3W



THE Open GROUP

[This page intentionally left blank]

Open Group Technical Standard

Protocols for Interworking: XNFS, Version 3W

The Open Group

Open Group Technical Standard

Protocols for Interworking: XNFS, Version 3W

ISBN: 1-85912-184-5
Document Number: C702

Published in the U.K. by The Open Group, February 1998.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

List of Figures

Preface

The Open Group

The Open Group is the leading vendor-neutral, international consortium for buyers and suppliers of technology. Its mission is to cause the development of a viable global information infrastructure that is ubiquitous, trusted, reliable, and as easy-to-use as the telephone. The essential functionality embedded in this infrastructure is what we term the *IT DialTone*. The Open Group creates an environment where all elements involved in technology development can cooperate to deliver less costly and more flexible IT solutions.

Formed in 1996 by the merger of the X/Open Company Ltd. (founded in 1984) and the Open Software Foundation (founded in 1988), The Open Group is supported by most of the world's largest user organizations, information systems vendors, and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to meet its new mission, as well as to assist user organizations, vendors, and suppliers in the development and implementation of products supporting the adoption and proliferation of systems which conform to standard specifications.

With more than 200 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- Consolidating, prioritizing, and communicating customer requirements to vendors
- Conducting research and development with industry, academia, and government agencies to deliver innovation and economy through projects associated with its Research Institute
- Managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- Adopting, integrating, and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- Licensing and promoting the Open Brand, represented by the ‘‘X’’ Device, that designates vendor products which conform to Open Group Product Standards
- Promoting the benefits of the IT DialTone to customers, vendors, and the public

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development, and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trademark on behalf of the industry.

The Development of Product Standards

This process includes the identification of requirements for open systems and, now, the IT DialTone, development of Technical Standards (formerly CAE and Preliminary Specifications) through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product.

The ''X'' Device is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the Open Brand they guarantee, through the Open Brand Trade Mark License Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical Standards and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys, and business titles.

There are several types of specification:

- *Technical Standards* (formerly *CAE Specifications*)

  The Open Group Technical Standards form the basis for our Product Standards. These Standards are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement a Technical Standard can enjoy the benefits of a single, widely supported industry standard. Where appropriate, they can demonstrate product compliance through the Open Brand. Technical Standards are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *CAE Specifications*

  CAE Specifications and Developers' Specifications published prior to January 1998 have the same status as Technical Standards (see above).

- *Preliminary Specifications*

  Preliminary Specifications have usually addressed an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

  Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organizations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a Technical Standard. While the intent is to progress Preliminary Specifications to corresponding Technical Standards, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

  The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif, and CDE.

  Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as Technical Standards, in which case the relevant Technology Specification is superseded by a Technical Standard.

In addition, The Open Group publishes:

- *Product Documentation*

  This includes product documentation—programmer's guides, user manuals, and so on—relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

  These provide information that is useful in the evaluation, procurement, development, or management of open systems, particularly those that relate to the Technical Standards or Preliminary Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

  Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Program. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

Versions and Issues of Specifications

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.opengroup.org/corrigenda**.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.opengroup.org/pubs**.

This Document

This document describes XNFS, the X/Open NFS Specification. This document supersedes the previous X/Open publication *Protocols for X/Open Interworking, XNFS, Version 3*, Document Number C525, August 1996. The previous version was aligned with Sun's NFS Version 3 (RFC 1813). This revised version (XNFS, Version 3W) incorporates the Sun WebNFS^{TM} optional extensions (RFCs 1738, 1808, 2054, 2055).

The process of accessing remote files and directories as though they were part of the local file system hierarchy is commonly known as ‘‘transparent file access’’ (TFA). The most widely used heterogeneous TFA architecture is the Network File System (NFS), originally developed by Sun Microsystems.

XNFS provides a means of access to files and directories that are physically stored on remote systems, by extending the semantics of local system interfaces so that applications and end users can (as far as possible) ignore the distinctions between local and remote objects.

NFS has been implemented on a wide range of architectures, from personal computers to mainframes and supercomputers. The specifications for the protocols associated with NFS have been published, and there have been several independent implementations.

With the XNFS specification, X/Open offers the market a temporary but complete solution to the problem of transparent file access between X/Open-compliant systems. Temporary, because X/Open recognises the TFA standardisation effort ongoing within the IEEE P1003.1f committee, and X/Open intends to be compliant with 1003.1f TFA at such time as it becomes an IEEE standard. Complete, because X/Open now offers both protocols for interoperability (via this XNFS specification) and interfaces for application/user portability (via the XSI specifications, in conjunction with the semantic differences defined in the appendices to this document).

This specification is based in part on the X/Open **(PC)NFS** Specification.

The X/Open **(PC)NFS** Specification defines the protocols for communication between a PC client running DOS or OS/2 and an X/Open-compliant server.

The XNFS specification defines:

- The transparent file access service provided by XNFS
- The protocols that support this service between X/Open-compliant machines, which can take the role of either servers or clients
- The differences in semantics of the X/Open System Interfaces and Headers Specification (see reference **XSH**) and the X/Open Commands and Utilities Specification (see reference **XCU**) when they are used ‘‘transparently’’ using XNFS rather than locally

Since many of the protocols used are the same for PC and X/Open-compliant system clients, there is obviously a great deal of overlap between these specifications.

In the event of any inconsistency or disagreement between the two documents, this document is to be treated as authoritative. At some future date, the X/Open **(PC)NFS** Specification will be revised to include only those elements which are specific to PC clients, such as the *pcnfsd* protocol, filename and attribute mapping, and the transmission analysis.

Intended Audience

There are two intended audiences for this specification. The first includes anyone who wishes to implement XNFS as part of an X/Open-compliant system. This specification defines the protocols that are to be implemented by a conforming system, so that it may interoperate with other conforming systems. It does not, however, define the way in which these protocols are to be implemented, nor the way in which XNFS is to be integrated into the rest of the system.

The second audience is the developers of X/Open CAE applications. This group relies upon the semantics of the XSI as defined in the X/Open System Interfaces and Headers Specification (see reference **XSH**) and the X/Open Commands and Utilities Specification (see reference **XCU**) and needs to be aware of any semantic changes which the use of XNFS may introduce. These changes are described in Appendix A, Appendix B and Appendix C. Obviously an XNFS implementor must be aware of this material so that their implementation does not behave in an unexpected manner.

Structure

- Chapter 1 introduces the context for NFS.
- Chapter 2 describes the basic XNFS Service Model, in terms of abstract objects and operations which are implementation-independent.
- Chapter 3 specifies the subset of the XDF protocol used by XNFS.
- Chapter 4 specifies a message protocol (using XDR language) used in implementing an RPC package.
- Chapter 5 explains how protocols defined as part of XNFS interface with the underlying transport.
- Chapter 6 describes the port mapper protocol, which maps RPC program and version numbers to transport-specific port numbers.
- Chapter 7 specifies the NFS protocol that Sun Microsystems, Inc. and others use to provide transparent remote access to shared file systems over local area networks.
- Chapter 8 specifies the Mount protocol, which is separate from but related to the NFS protocol.
- Chapter 9 defines the Network Lock Manager (NLM) and Network Status Monitor (NSM) which together provide stateful services for NFS.
- Chapter 10 defines the NLM protocol.
- Chapter 11 defines the NSM protocol.
- Chapter 12 specifies the additional NFS, Version 3 protocol which must be supported in addition to that specified in Chapter 7. It includes a summary of the changes introduced in the Version 3 protocol.
- Chapter 13 specifies an additional Mount, Version 3 protocol which must be supported in addition to that specified in Chapter 8.
- Chapter 14 specifies an additional NLM, Version 4 protocol which must be supported in addition to that specified in Chapter 10 in order to support NFS, Version 3.
- Appendix A summarises semantic differences for access of files on remote systems using XNFS.

- Appendix B describes semantic differences for system interfaces and headers (XSH) which operate differently when used with XNFS.
- Appendix C describes semantic differences for commands and utilities (XCU) which operate differently when used with XNFS.
- Appendix D gives a general description of how a system interface (XSH) function is executed by a sequence of RPCs.
- Appendix E describes the WebNFS extensions to the NFS protocol.

XNFS Version 2 → Version 3 Changes

The changes to the XNFS Issue 4 specification (document number C218, which described Sun NFS Version 2) to add the NFS Version 3 protocol comprise miscellaneous modifications to sections in the XNFS Issue 4 specification, plus addition of three new chapters 12, 13 and 14.

A summary of these changes follows:

- Referenced Documents updated.
- Section 1.2 (Scope), 2nd paragraph extended to add:

  > Protocols corresponding to both NFS Version 2 and NFS Version 3 are specified. Systems conforming to this document must support both protocol sets.

- Section 1.4 (Protocol Stacks and Conformance), first bullet list extended to add:

  — NFS Version 3 (see Chapter 12 on page 175)

  — Mount Version 3 (see Chapter 13 on page 255)

  — NLM Version 4 (see Chapter 14 on page 263)

- Section 1.4 (Protocol Stacks and Conformance), XPG3/XPG4 Compliance description extended to read:

  > This specification applies to Issue 3, Issue 4 and Issue 4, Version 2 of XPG, except where explicitly stated otherwise.

- New section (Hyper Integer and Unsigned Hyper Integer) added after section 3.2.2, with subsequent sections renumbered accordingly.
- Section 3.3.3 (Syntax Information), in the BNF grammar, add:

  ```
  | [ "unsigned" ] "hyper"
  ```

- Section 3.3.4 (Syntax Note), first bullet point, add ‘‘hyper’’ to the list of keywords which cannot be used as identifiers.
- Section 4.4 (Authentication Protocols), in the first paragraph, change:

  two ‘‘flavours’’ of authentication.

  to

  four ‘‘flavours’’ of authentication.

- Section 4.4 (Authentication Protocols), add:

  ```
  AUTH_DES      = 3,
  AUTH_KERB     = 4,
  ```

  to the **auth_flavor** definition.

- After section 4.4.2, add a new subsection (4.4.3 DES and Kerberos Authentication).
- Section 7.3.2 (Basic Data Types), clarify the meaning of the ''blocks'' and ''blocksize'' fields in the **fattr** structure.
- Section 7.5.6 (NFSPROC_READLINK Specification), change the Return Code NFSERR_NXIO to NFSERR_INVAL (to match existing server practice).
- Section 10.2.2 (Basic Data Types for Locking), add to the description for **nlm_holder**:

  > The ''oh'' field is an opaque object that identifies the host, or a process on the host, that is holding the lock.

- Section 10.2.2 (Basic Data Types for Locking), replace the sentence which describes ''fh'' and ''oh'' with:

  > The ''fh'' field identifies the file to lock. The ''oh'' field is an opaque object that identifies the host, or a process on the host, that is making the request.

- Add Chapter 12 ''Network File System: Protocol Specification, Version 3''.
- Add Chapter 13 ''Mount Protocol, Version 3''.
- Add Chapter 14 ''Network Lock Manager Protocol, Version 4''.
- Appendix A is expanded to include description of those semantic differences for interfaces, commands and utilities which were previously included in man page descriptions in Appendices B and C.
- Appendix B is reduced to listing those functions which show no semantic differences (that is, behave the same) when running in a local file system environment, and also those which may show a semantic difference when invoked in an XNFS environment. The man page definitions which described these differences are deleted, since Appendix A now covers these differences.
- Appendix C is reduced to listing those commands and utilities that show no semantic differences (that is, behave the same) when running in a local file system environment, and also those which may show semantic differences when invoked in an XNFS environment. The man page definitions which described these differences are deleted, since Appendix A now covers these differences.
- Appendix D revision for NFS Version 3 is represented by an additional bullet item (Version 2 versus Version 3) in Section D.1.

XNFS Version 3 → Version 3W Changes

The changes to the XNFS Version 3 specification (document number C525) to add the WebNFS extensions to the NFS protocol, comprise various additions to sections 1.6, 2.4.1, 7.2.1, 7.3, 7.3.2, 12.2, 12.2.2, 12.3.8, 12.4.4, and a new Appendix E detailing the WebNFS extensions.

Trade Marks

AT&T® is a registered trademark of AT&T in the U.S.A. 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 and Network File System™ is a trademark of Sun Microsystems, Inc.

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

PC-NFS™ is a trademark of Sun Microsystems, Inc..

Postscript® is a registered trademark of Adobe Systems Incorporated.

VAX® is a registered trademark of Digital Equipment Corporation.

VMS® is a registered trademark of Digital Equipment Corporation.

Motif,® OSF/1,® UNIX,® and the ‘‘X Device’’® are registered trademarks and IT DialTone™ and The Open Group™ are trademarks of The Open Group in the U.S. and other countries.

WebNFS™ is a trademark of Sun Microsystems, Inc.

Referenced Documents

The following documents are referenced in this specification:

Open Group Documents

BSFT
: CAE Specification, December 1991, Byte Stream File Transfer (BSFT) (ISBN: 1-872630-27-8, C194), published by The Open Group.

Internationalisation Guide
: Guide, July 1993, Internationalisation Guide, Version 2 (ISBN: 1-859120-02-4, G304), published by The Open Group.

(PC)NFS
: Developers' Specification, August 1990, Protocols for X/Open PC Interworking: (PC)NFS (ISBN: 1-872630-00-6, D030), published by The Open Group.

XBD, Issue 4, Version 2
: CAE Specification, August 1994, System Interface Definitions, Issue 4, Version 2 (ISBN: 1-85912-036-9, C434), published by The Open Group.

XCU, Issue 4, Version 2
: CAE Specification, August 1994, Commands and Utilities, Issue 4, Version 2 (ISBN: 1-85912-034-2, C436), published by The Open Group.

XNS (Networking Services, Issue 4)
: CAE Specification, August 1994, Networking Services, Issue 4 (ISBN: 1-85912-049-0, C438), published by The Open Group.

XPG3
: X/Open Specification, 1988, 1989, February 1992 (ISBN: 1-872630-43-X, T921); this specification was formerly X/Open Portability Guide, seven volumes, January 1989 (ISBN: 0-13-685819-8, XO/XPG/89/000).

XSH, Issue 4, Version 2
: CAE Specification, August 1994, System Interfaces and Headers, Issue 4, Version 2 (ISBN: 1-85912-037-7, C435), published by The Open Group.

International Standards

ISO 8859-1
: ISO 8859-1: 1987, Information Processing — 8-bit Single-byte Coded Graphic Character Sets — Part 1: Latin Alphabet No. 1.

ISO / IEC 9945-1
: ISO/IEC 9945-1: 1990, Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language] (identical to IEEE Std 1003.1-1990).

Access to IETF RFCs

RFCs may be obtained via Email or FTP from many RFC repositories. Many of these repositories also now have World Wide Web servers. Try one of the following URLs as a starting point:

```
http://www.isi.edu/rfc-editor/
http://ds.internic.net/ds/rfc-index.html/
```

RFCs can be obtained via FTP from DS.INTERNIC.NET, NIS.NSF.NET, NISC.JVNC.NET, FTP.ISI.EDU, WUARCHIVE.WUSTL.EDU, SRC.DOC.IC.AC.UK, FTP.NCREN.NET, FTP.SESQUI.NET, FTP.NIC.IT, or FTP.IMAG.FR, using the FTP username *anonymous* and the FTP password *guest* For further information about Internet Protocols in general, please contact:

USC - Information Sciences Institute,
4676, Admiralty Way,
Marina del Rey, CA 90292-6695,
USA

Tel: (+1) 213-822-1511

Internet Protocol Suite RFCs

RFC 1140 - IAB Official Protocol Standards
IETF RFC 1140 gives the state of standardisation of protocols used in the Internet as determined by the Internet Activities Board (IAB). RFC 1140 was published in May 1990; this document is reissued on a regular basis, and the reader should obtain the current version as described above.

RFC 1011 - Official Internet Protocols
IETF RFC 1011 is the authoritative reference as to which document defines each protocol. RFC 1011 was published in May 1987; this document is reissued on a regular basis, and the reader should obtain the current version as described above.

The descriptions which follow are derived from RFC 1011.

RFC 791 - Internet Protocol (IP)
Status: Standard. IETF RFC 791 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)
Status: Standard. IETF RFC 792 describes the control messages and error reports that go with the Internet Protocol.

Note: ICMP is defined to be an integral part of IP. An implementation of IP is incomplete if it does not include ICMP.

RFC 768 - User Datagram Protocol (UDP)
Status: Standard. IETF RFC 768 provides a datagram service to applications. It adds port addressing to the IP services.

RFC 793 - Transmission Control Protocol (TCP)
Status: Standard. IETF RFC 793 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
Status: Standard. IETF RFC 950 specifies procedures for the use of subnets, which are logical sub-sections of a single Internet network.

RFC 826 - Address Resolution Protocol (ARP)
Status: Standard. IETF RFC 826 is a procedure for finding the network hardware address corresponding to an Internet Address.

RFC 997 - Internet Numbers
IETF RFC 997 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 1060 - Assigned Numbers
Status: Historic. IETF RFC 1060 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
Status: Standard. IETF RFC 894 describes the representation of Internet Protocol services on Ethernet networks.

RFC 1011 (includes) Internet Protocol on IEEE 802
IETF RFC 1011 includes description of the latest policy on the transmission of IP datagrams on IEEE 802 networks.

All of the preceding material should be interpreted in accordance with the following two documents, which provide an authoritative policy on the way in which the various protocols should be implemented and administered:

RFC 1122 - Requirements for Internet Hosts - Communication Layers
IETF RFC 1122, R T Braden, October 1989. Status: Standard.

RFC 1123 - Requirements for Internet Hosts - Application and Support
IETF RFC 1123, R T Braden, October 1989. Status: Standard.

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

RFC 1014 - XDR: External Data Representation Standard
IETF RFC 1014, Sun Microsystems, June 1987.

RFC 1057 - RPC: Remote Procedure Call Protocol Specification, Version 2
IETF RFC 1057, Sun Microsystems. June 1988, Status: Informational.

RFC 1094 - NFS: Network File System Protocol Specification
IETF RFC 1094, Sun Microsystems. Mar-01-1989, Status: Informational.

RFC 1813 - NFS: Network File System Prococol, Version 3 Specification
IETF RFC 1813, B Callaghan, B Pawlowski & P Staubach, June 1995. Status: Informational.

RFC 1831 - RPC: Remote Procedure Call Protocol Specification Version 2
IETF RFC 1831, R Srinivasan, August 1995. Status: Proposed Standard.

RFC 1832 - XDR: External Data Representation Standard.
IETF RFC 1832, R Srinivasan, August 1995. Status: Proposed Standard.

RFC 1833 - Binding Protocols for ONC RPC Version 2
IETF RFC 1833, R Srinivasan, August 1995. Status: Proposed Standard.

WebNFS-relevant RFCs

RFC 1738 - Uniform Resource Locators (URL)
IETF RFC 1738, T Berners-Lee, L Masinter & M McCahill. December 1994, Status: Proposed Standard. This document describes the syntax and semantics of absolute Uniform Resource Locators, which can be used for the location and access of resources via the Internet.

RFC 1808 - Relative Uniform Resource Locators
IETF RFC 1808, R Fielding. June 1995, Status: Proposed Standard. This document describes the syntax and semantics of relative Uniform Resource Locators. Relative URLs can be used for the location and access of resources relative to an absolute URL.

RFC 2054 - WebNFS Client Specification
IETF RFC 2054, B Callaghan. October 1996, Status: Informational. This document describes the procedure that a WebNFS client uses to access an NFS server.

RFC 2055 - WebNFS Server Specification
IETF RFC 2055, B Callaghan, October 1996, Status: Informational. This document describes the features that are required of a WebNFS server.

RFC 2224 - NFS URL Scheme
IETF RFC 2224, B Callaghan, October 1997, Status: Informational. This document describes a URL scheme used to refer to files and directories on NFS servers using the general URL syntax defined in RFC 1738.

Other Documents

Andrew D. Birrell and Bruce Jay Nelson, *Implementing Remote Procedure Calls*, XEROX CSL-83-7, October 1983.

Danny Cohen, *On Holy Wars and a Plea for Peace*, IEEE Computer, October 1981.

Courier: *The Remote Procedure Call Protocol*, XEROX Corporation, XSIS 038112, December 1981.

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Bell Laboratories, Murray Hill, New Jersey, 1978.

J. Postel, Transmission Control Protocol — DARPA Internet Program Protocol Specification, RFC 793, Information Sciences Institute, September 1981.

J. Postel, User Datagram Protocol, RFC 768, Information Sciences Institute, August 1980.

Disk Operating System Technical Reference, IBM part no. 6138536.

Chapter 1

Introduction

1.1 Overview

The vast majority of open-systems-compliant systems, now and in the future, will be connected to other systems in local or wide area networks. To exploit the capabilities of these networks, two types of mechanism have been used. In the first traditional model, the existence of the network is explicitly recognised, and interfaces are provided which allow applications or end users to perform operations across the network. In the second, the semantics of local system interfaces are extended to allow access to objects on remote systems, so that applications and end users can (as far as possible) ignore the distinctions between local and remote objects. The first approach is the subject of the X/Open Networking Services Specification (see reference **XNS**). This document is concerned with the second. More specifically, it is concerned with access to files and directories which are physically stored on remote systems. It does not address operations on, or communications between, processes within a network.

The process of accessing remote files and directories as though they were part of the local file system hierarchy has been termed ‘‘transparent file access’’ (TFA). TFA is an attractive concept for a number of reasons. Using TFA, it becomes possible to use a single set of applications, utilities, and so on, for local and remote operations. For example, instead of invoking a network application such as BSFT (see the X/Open **BSFT** Specification) to copy a file from one system to another, the XSI *cp* utility may be used. Accessing a remote file directly, rather than copying it to the local system and working on the copy, has two obvious benefits. First, the single copy of the file can be updated without the need to update the copies. Second, the total disk storage used within the network is reduced. From here it is a short step to using TFA to support a diskless system, in which *all* file service is networked. Such a system may offer certain benefits in the areas of cost, simplicity and administration.

The most widely used heterogeneous TFA architecture is the Network File System (NFS), originally developed by Sun Microsystems, Inc. NFS has been implemented on a wide range of architectures, from personal computers to mainframes and supercomputers. The specifications for the protocols associated with NFS have been published, and there have been several independent implementations.

With the XNFS specification, The Open Group offers the market a temporary but complete solution to the problem of transparent file access between open-systems-compliant systems. Temporary, because The Open Group recognises the TFA standardisation effort ongoing within the IEEE P1003.1f committee, and The Open Group intends to be compliant with P1003.1f TFA at such time as it becomes an IEEE standard. Complete, because The Open Group now offers both protocols for interoperability (via this XNFS specification) and interfaces for application/user portability (via the XSI specifications, in conjunction with the semantic differences defined in Appendix A on page 271, Appendix B on page 287 and Appendix C on page 295.

There is a certain overlap between the X/Open **(PC)NFS** Specification and this specification. However, the X/Open **(PC)NFS** Specification contains a large number of PC-specific definitions, and the two documents address different markets (PC-X/Open Interworking and X/Open-X/Open interworking, respectively).

1.2 Scope

This document is a specification for XNFS. XNFS incorporates all of the protocols defined for NFS, and formalises the semantics for XSI applications and utilities operating on remote file systems. A system compliant to an Open Brand for XNFS will be able to access files on any other conforming system, and will be able to make its local files accessible to conforming systems. It will also be able to act as a file server to personal computers running (PC)NFS, as described in the X/Open **(PC)NFS** Specification. It is intended that it will also be able to interoperate with implementations of NFS on systems which are not compliant with an Open Brand for XNFS, but such interoperation is outside the scope of this specification. Throughout this specification, XNFS will refer to the complete specification, while NFS will refer only to the file-sharing protocol component.

This document contains protocol specifications for External Data Representation (XDR), Remote Procedure Call Protocol (RPC), the Network File System (NFS) and the XNFS adjunct protocols, which include Portmap, Mount, Network Status Monitor (NSM) and Network Lock Manager (NLM). Protocols corresponding to both NFS Version 2 and NFS Version 3 are specified. Systems conforming to this document must support both protocol sets.

This document introduces the WebNFS extensions in an optional appendix.

The RPC specification is included because the NFS protocols are defined in terms of it. The XDR specification is included because the NFS protocols and RPC are defined in terms of it.

The inclusion of these specifications does not mandate the use of distinct XDR or RPC implementation components, nor does it define or mandate any general RPC protocols or interfaces for distributed client/server applications. However, the RPC and XDR specifications define unambiguously the encoding of NFS and adjunct protocol requests and responses when transmitted across a network, as described in Chapter 5 on page 57.

This specification does not define the utilities and interfaces used for the administration of XNFS. However it is impractical to avoid the issue of configuration entirely. XNFS implementations include a number of configuration options, and many of the semantic issues discussed in the appendices are inherently dependent upon the way in which XNFS is configured. The solution adopted in this document is to define an abstract XNFS Service Model (see Chapter 2 on page 9) which specifies a set of administrative operations and the attributes of the entities to be managed. The attributes are those defined in the most widely-used NFS reference source from Sun Microsystems, Inc., ‘‘NFSSRC4.0’’. The implementation of the administrative model, in terms of programming interfaces, utilities, data bases and so on, is not specified, and need not be patterned on NFSSRC4.0. It is intended that this specification is compatible and interoperable with current practice as described in RFC 1094 and RFC 1813 (see Section 1.7 on page 7). The behaviour described in Appendix A on page 271, Appendix B on page 287 and Appendix C on page 295, is presented in terms of the attributes introduced in the Service Model.

These appendices document the semantic differences experienced when using NFS instead of a local file system.

1.3 Audience

There are two intended audiences for this specification. The first includes anyone who wishes to implement XNFS as part of an open-systems-compliant scheme. This specification defines the protocols which are to be implemented by a conforming system, so that it may interoperate with other conforming systems. It does not, however, define the way in which these protocols are to be implemented, nor the way in which XNFS is to be integrated into the rest of the system. If, for example, an implementation performs a *read*( ) on a remote file, this specification defines completely the format of the remote request and the corresponding response. It does not define the circumstances under which an implementation should issue a remote *read*( ) request. However, as a guide for implementors, a (non-normative) transmission analysis is included as Appendix D on page 299.

The second audience is the developers of applications. This group relies upon the semantics of the XSI as defined in the X/Open System Interfaces and Headers Specification (see reference **XSH**) and the X/Open Commands and Utilities Specification (see reference **XCU**), and needs to be aware of any semantic changes which the use of XNFS may introduce. These changes are described in Appendix A on page 271, Appendix B on page 287 and Appendix C on page 295. Obviously an XNFS implementor must be aware of this material so that his or her implementation does not behave in an unexpected manner.

1.4 Terminology

The following common english words have special meaning when used in this document:

can
This describes a permissible optional feature or behaviour available to the user or application; all systems support such features or behaviour as mandatory requirements.

implementation-dependent
The value or behaviour is not consistent across all implementations. The provider of an implementation normally documents the requirements for correct program construction and correct data in the use of that value or behaviour. When the value or behaviour in the implementation is designed to be variable or customisable on each instantiation of the system, the provider of the implementation normally documents the nature and permissible ranges of this variation. Applications that are intended to be portable must not rely on implementation-dependent values or behaviour.

legacy
Certain features are *legacy*, which means that they are being retained for compatibility with older applications, but have limitations which make them inappropriate for developing portable applications. New applications should use alternative means of obtaining equivalent functionality. Legacy features are marked **LEGACY**.

may
With respect to implementations, the feature or behaviour is optional. Applications should not rely on the existence of the feature. To avoid ambiguity, the reverse sense of *may* is expressed as *need not*, instead of *may not*.

must
This means that the behavior described is a requirement on the implementation.

should
With respect to implementations, the feature is recommended, but it is not mandatory. Applications should not rely on the existence of the feature.

With respect to users or applications, the word means recommended programming practice that is necessary for maximum portability.

undefined
A value or behaviour is undefined if this document imposes no portability requirements on applications for erroneous program constructs or erroneous data. Implementations may specify the result of using that value or causing that behaviour, but such specifications are not guaranteed to be consistent across all implementations. An application using such behaviour is not fully portable to all systems.

unspecified
A value or behaviour is unspecified if this document imposes no portability requirements on applications for correct program construct or correct data. Implementations may specify the result of using that value or causing that behaviour, but such specifications are not guaranteed to be consistent across all implementations. An application requiring a specific behaviour, rather than tolerating any behaviour when using that functionality, is not fully portable to all systems.

will
This means that the behavior described is a requirement on the implementation.

1.5 Protocol Stacks and Conformance

An open-systems-conformant XNFS implementation must support the following protocols:

- NFS (see Chapter 7 on page 69)
- Mount (see Chapter 8 on page 107)
- NLM (see Chapter 10 on page 127)
- NSM (see Chapter 11 on page 161)
- Portmap (see Chapter 6 on page 61).
- NFS Version 3 (see Chapter 12 on page 175)
- Mount Version 3 (see Chapter 13 on page 255)
- NLM Version 4 (see Chapter 14 on page 263)

These protocols are defined in terms of the Remote Procedure Call (RPC) protocol, which may employ the External Data Representation (XDR) encoding to ensure operating system independence.

Although the RPC protocol is inherently independent of any particular transport service, an open-systems-conformant NFS implementation must support at least one protocol suite, and support it in the manner defined in this specification.

This specification defines a single transport service, UDP over IP from the Internet Protocol Suite (IPS). All XNFS protocols are implemented over this transport.

Note: While an implementation may elect to use some other transport such as TCP for communication between XNFS components on a single machine, this is an implementation issue and is not required or described in this specification.

Provided that an interoperable transport service and the same physical connection mechanism are chosen for two systems, conformance to this specification by both system's NFS implementations guarantees that both systems can make their local files accessible for each other and that they can access each other's files.

In addition to the protocols described in the chapters mentioned above, the following chapters are normative for open-systems-compliant NFS implementations:

- Chapter 2 apart from the *MntStdFileSysOp* and *UnMntAllFileSys* operations
- Chapter 9
- Appendix A
- Appendix B
- Appendix C.

Compliance

This specification applies to the X/Open Portability Guide, Issue 3 (XPG3) and the Single UNIX Specification, Issue 4, Version 2 (XCU, XSH and XBD — see **Referenced Documents** on page xix), except where explicitly stated otherwise.

1.6 Relationship to other Open Group Specifications

This specification is based in part on the X/Open **(PC)NFS** Specification.

The X/Open **(PC)NFS** Specification defines the protocols for communication between a PC client running DOS or OS/2 and an open-systems-compliant server.

The XNFS specification defines:

- The transparent file access service provided by XNFS
- The protocols that support this service between open-systems-compliant machines, which can take the role of either servers or clients
- The differences in semantics of the X/Open System Interfaces and Headers Specification (see reference **XSH**) and the X/Open Commands and Utilities Specification (see reference **XCU**) when they are used ‘‘transparently’’ using XNFS rather than locally.

Since many of the protocols used are the same for PC and open-systems-compliant system clients, there is obviously a great deal of overlap between these specifications.

In the event of any inconsistency or disagreement between the two documents, this document is to be treated as authoritative. At some future date, the X/Open **(PC)NFS** Specification will be revised to include only those elements which are specific to PC clients, such as the *pcnfsd* protocol, filename and attribute mapping, and the transmission analysis.

This specification describes additional return codes and changed behaviour for some of the system calls and utilities described in the X/Open System Interfaces and Headers Specification (see reference **XSH**) and the X/Open Commands and Utilities Specification (see reference **XCU**). These are documented in Appendix A on page 271, Appendix B on page 287 and Appendix C on page 295.

1.7 References to RFCs

This document describes only those protocols which have not otherwise been standardised. The RFCs listed in the **Referenced Documemts** section in the front pages of this document should be consulted for details of the Internet Protocol Suite.

Certain RFC documents are reissued periodically, with each issue receiving a new RFC number. To determine whether a particular RFC is current or not, the reader should consult a copy of the latest RFC Index. Guidance on where to find this information is included in the **Referenced Documents** section.

Chapter 2

XNFS Service Model

2.1 Introduction

This chapter describes the the basic XNFS Service Model (XNFSSM). Its purpose is to provide a high-level description of the XNFSSM in terms of abstract objects and operations which are independent of any particular implementation.

The XNFSSM describes the way in which cooperating XNFS client and server systems interact in order to:

1. (on a server) make a file system available for use by clients
2. (on a client) gain access to a file system on a server and make it usable by local processes
3. (on a client) access files and directories stored on a server
4. (on a client) make a remote file system unusable by local processes and give up access to the file system
5. (on a server) make a file system unavailable to clients.

Before the formal elements of the XNFSSM are defined, an informal overview may be helpful.

2.2 Informal Overview of XNFS

This section provides a slightly simplified example of the use of XNFS to access remote files. Figure 2-1 illustrates a portion of the file hierarchy on an X/Open-compliant system with the system name **alpha**.

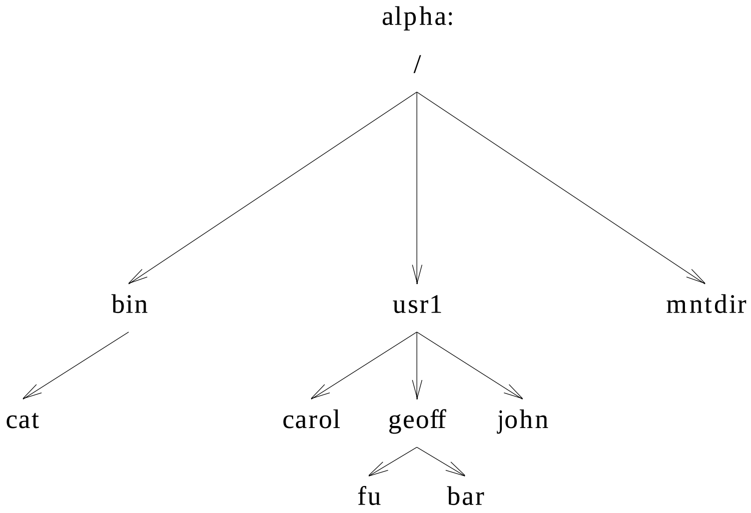


Figure 2-1 File Hierarchy on Example System Called alpha

Users and applications on system **alpha** can refer to files within this hierarchy with pathnames such as **/bin/cat** and **/usr1/geoff/fu**.

Similarly, Figure 2-2 illustrates part of the file hierarchy on the system **beta**.

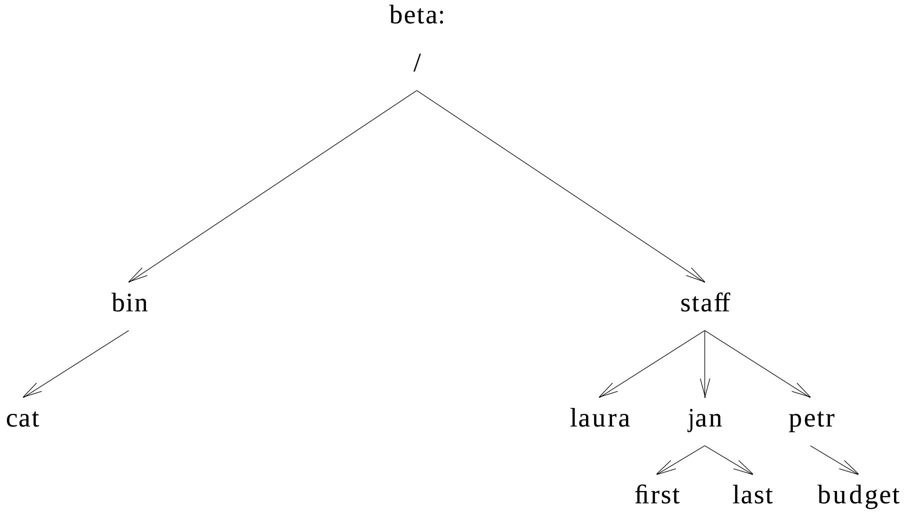


Figure 2-2 File Hierarchy on Example System Called beta

The administrator of system **beta** decides to make the portion of the file system hierarchy under the directory **/staff** available to other systems within the network. This is accomplished by exporting the directory **/staff**.

The administrator of system **alpha** now decides to make this file system available to users and applications on **alpha**. This process is referred to as mounting the remote file system. To mount a file system, three pieces of information must be provided:

- the name of the system exporting the file system
- the name of the directory to be mounted
- the name of a directory within the local file system to be used as the mount point.

The choice of mount point is important, since it determines the name by which the remote directory, and everything below it, will be known on the client system. In this case, the administrator chooses to mount **/staff** from system **beta** on the (empty) directory **/mntdir**. To a user of system **alpha** the local file system hierarchy now appears as shown in Figure 2-3.

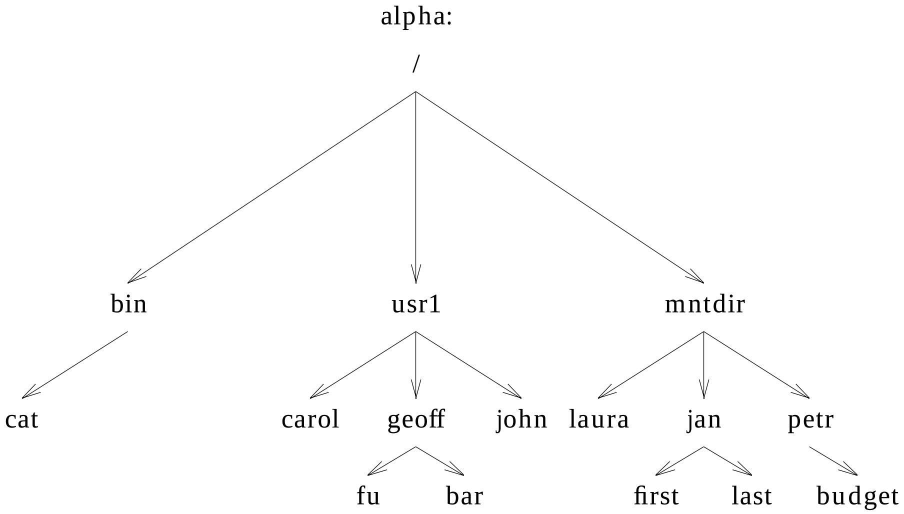


Figure 2-3 View from System alpha of Example File Hierarchy on System beta

If a user types the command **/bin/cat /mntdir/jan/first**, the result is to list the contents of the file **/staff/jan/first** stored on the system **beta**. The command **cd /mntdir/petr** will select the directory **/staff/petr** on **beta** as the current directory; typing **ls** will show the file **budget**, and **df .** will display the amount of free space on the remote file system.

One effect of mounting a file system on a directory is that the contents of the local mount point directory become inaccessible while the mount is in effect. Since an empty directory is usually chosen as the mount point, this is not normally a problem. Suppose, however, that the local file system on **alpha** was as shown in Figure 2-4.

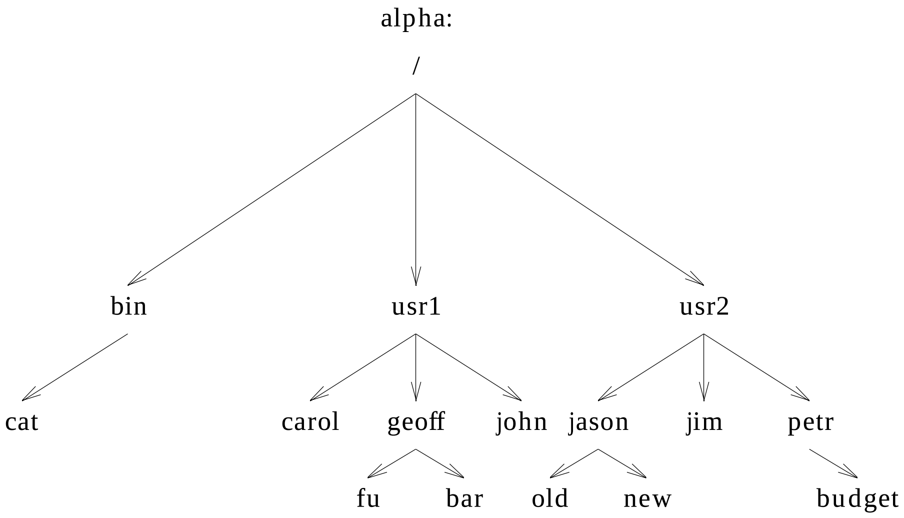


Figure 2-4 Example 2 File Hierarchy on System Called alpha

If the directory **/staff** on **beta** is mounted on **/usr2**, the resulting file system hierarchy will be as shown in Figure 2-5.

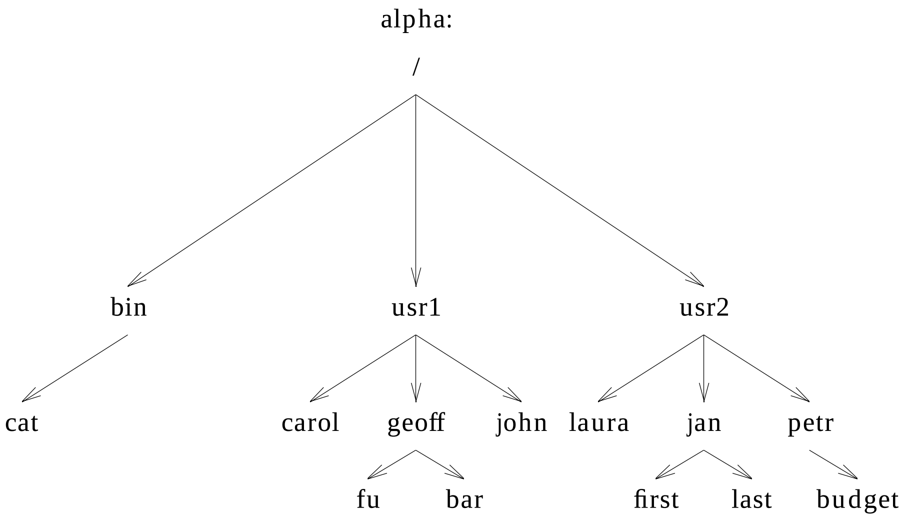


Figure 2-5 Resulting File Hierarchy on System alpha from Example 2

Note that, for example, the file **/usr2/jason/old** is now inaccessible. Furthermore, the path **/usr2/petr/budget** is still valid, but it now refers to the file **/staff/petr/budget** on **beta**.

2.3 Elements of the XNFS Service Model

The XNFS Service Model incorporates the following elements:

- the *ExportedFileSystem* and *MountedFileSystem* objects
- the *ExpFileSysOp* and *UnExpFileSysOp* administrative operations and their derivatives
- the *ShowExpFileSysOp*, *MntFileSysOp* and *UnMntFileSysOp* administrative operations and their derivatives
- file and directory access.

The operations and objects are abstractions only. It is not expected that an XNFS implementation will incorporate procedures and data structures which correspond to these definitions. The (informal) definition of these attributes is not to be interpreted as implying a specific representation. The process of mounting a file relies on retrieving from the server a token (*FileHandle*) which corresponds to the remote file system, and storing an association of this token with the point in the local file system where the remote file system appears. This may be implemented in a variety of ways by using, for example, a command line utility, a graphical network resource browser, or ‘‘on the fly’’ using some kind of ‘‘automounter’’. However the notion of the *FileHandle* attribute of a mounted file system must be stored in some form.

2.4 XNFS Objects

2.4.1 ExportedFileSystem

An *ExportedFileSystem* object is created on an XNFS server as a result of a successful *ExpFileSysOp* operation. It is added to the set of exported file systems known to the *mount* server.

An XNFS server must implement the following set of *ExportedFileSystem* attributes, but is not precluded from adding others.

Attribute	**Type**	**Recommended Default**
PathName	path	NO DEFAULT
Mode	ReadOnly or ReadWrite	ReadWrite
AnonMapping	uid or −1	Implementation-dependent
Root	list of hostnames	No root access
Access	list of hosts or groups	Unlimited

A WebNFS server must implement the following additional attribute:

Attribute	**Type**	**Recommended Default**
Public	boolean	NO DEFAULT

A fuller description of each attribute follows.

PathName=pathname

This attribute identifies the file system object to be exported. This attribute must be specified.

Mode={ReadOnly or ReadWrite}

A value of ReadOnly specifies that the named file system object is not writable by XNFS clients. ReadWrite (the default) means that read and write access are granted.

AnonMapping=uid

Specifies the UID to be used for NFS accesses from an ‘‘unknown’’ user. Users with UID 0 are always considered unknown by the NFS server unless they are included in the *Root* attribute below. The recommended default value of this attribute is −2. Specifying *AnonMapping=-1* disables anonymous access to the file system.

Root=hostname[:hostname] . . .

If this attribute is specified, NFS accesses from the specified systems with UID 0 are processed without being subject to the *AnonMapping* process. If this attribute is not specified, all UID 0 accesses are mapped.

Access={hostname or groupname}[:{hostname or groupname}] . . .

This attribute may be used to explicitly identify those systems which may have access to the file system. Each element in the associated list identifies a host, or a group of hosts, which is to be allowed to mount the file system. The mechanism whereby a *hostname* or *groupname* is resolved is not specified. If this attribute is not specified, the file system is accessible to any client system. Note that the access control provided by the *Access* attribute is applied only at the time that the client mounts the file system object; subsequent NFS accesses using the file handle acquired by the mount will not be affected by any changes to the *Access* list.

Public

A WebNFS server may designate a single ExportedFileSystem as the public file system. References to the public filehandle (see Appendix E on page 307) are taken to mean the root of this file system. Alternatively, the server may designate a single directory in a non-exported file system. In this case, references to the public filehandle are taken to mean this directory. If the server designates a non-exported directory, clients must ensure that paths relative to the public filehandle refer to objects in an ExportedFileSystem.

2.4.2 MountedFileSystem

A *MountedFileSystem* object is created on an XNFS client as a result of a successful *MntFileSysOp* operation. The local mount point object is marked as remote, as discussed in Section 2.7 on page 24.

An XNFS client must implement the following set of *MountedFileSystem* attributes, but is not precluded from adding others.

Attribute	**Type**	**Supplied in** *MntFileSysOp* **Operation**	**Retrieved from Server**	**Recommended Default**
FileHandle	fhtype	—	X	NO DEFAULT
PathName	path	X	—	NO DEFAULT
Server	hostname	X	—	NO DEFAULT
MountPoint	path	X	—	NO DEFAULT
Mode	ReadOnly or ReadWrite	X	—	ReadWrite
GrpID	boolean	X	—	False
SetUID	boolean	X	—	False
ReadSize	integer	X	—	UNSPECIFIED
WriteSize	integer	X	—	UNSPECIFIED
NFSTimeOut	time	X	—	700 milliseconds
NFSRetransmissions	integer	X	—	3
RetrySemantics	Soft or Hard	X	—	Hard
Intr	boolean	X	—	False
NFSServerPort	UDP port	X	—	2049
AttribCaching	boolean	X	—	False
ACRegMin	time	X	—	3 seconds
ACRegMax	time	X	—	60 seconds
ACDirMin	time	X	—	30 seconds
ACDirMax	time	X	—	60 seconds

A fuller description of each attribute follows.

FileHandle

This attribute identifies the file system object on the server.

PathName=pathname

This attribute identifies the file system object to be mounted. This attribute must be specified.

Server=hostname

This attribute identifies the XNFS server system from which the object is to be mounted. This attribute must be specified.

MountPoint=pathname

This attribute identifies the local file system object on which the XNFS object is to be mounted. This attribute must be specified. The mountpoint must exist. Upon completion of the *MntFileSysOp* operation, the previous contents of the mountpoint will be inaccessible and the mountpoint pathname will refer to the XNFS mounted object. Note that if this object is a regular file rather than a directory the effects of directory-related operations are undefined.

Mode={ReadOnly or ReadWrite}

A value of ReadOnly specifies that the named file system object will be mounted for reading only. ReadWrite (the recommended default) means that writes will be allowed (although the server may reject them if the file system object was exported in ReadOnly mode).

GrpID={True or False}

If True, any file or directory created on the file system will inherit the group ID of the parent directory. If False, the group ID is set to the group of the current process or the parent directory, as specified by XSI. The recommend default is False.

SetUID={True or False}

If True, programs stored on the XNFS server which have the SetUID attribute may be executed with the corresponding effective UID. If false, such programs are executed with the current UID. The recommended default is False.

ReadSize=nn

When reading data from a remote file, the maximum amount of data that will be requested in a single RPC call is *nn* bytes. The recommended default value for this attribute is implementation-dependent. In addition, the XNFS client may limit the maximum transfer size to the *info.tsize* advertised by the server in the **statfsres** attributes (see Section 7.5.0 on page 104).

WriteSize=nn

When writing data to a remote file, the maximum amount of data that will be written in a single RPC call is *nn* bytes. The recommended default value for this attribute is implementation-dependent. In addition, the XNFS client may limit the maximum transfer size to the *info.tsize* advertised by the server in the **statfsres** attributes (see Section 7.5.0 on page 104).

NFSTimeOut=nn

Specifies that the initial timeout for an NFS RPC request is *nn* tenths of a second. (The timeout for subsequent retransmissions will depend on an implementation-dependent backoff algorithm.) The recommended default is 7, equivalent to a timeout of 700 milliseconds.

NFSRetransmissions=nn

If the *RetrySemantics* attribute is *Soft*, this specifies the number of times that an NFS RPC request will be retransmitted before the corresponding XSI request fails. The recommended default is 3.

RetrySemantics={Soft or Hard}

If a file system is mounted Hard (the recommended default), a failure of the server to respond to an NFS RPC causes the request to be retried until the server responds, or until the local process is interrupted (if *Intr*=*True*). If the file system is mounted with *RetrySemantics*=*Soft*, the corresponding XSI call will fail after the number of retries specified by *NFSRetransmissions* (see below).

Intr={True or False}

If True, a keyboard signal to a process will cause any operation which is being retried *RetrySemantics*=*Hard* to be abandoned.

NFSServerPort=nn

Specifies that NFS RPC requests are directed to UDP port *nn* on the server. The recommended default is 2049.

AttribCaching={True or False}

If True, no attribute, access, or directory caching will be performed. Otherwise the attributes (including permissions, size and timestamps) for files and directories may be cached to reduce the need to perform over-the-wire *GETATTR* or *ACCESS* RPCs. The *ACRegMin*, *ACRegMax*, *ACDirMin* and *ACDirMax* attributes control the length of time for which the cached values will be retained.

ACRegMin=nn

Hold cached attributes for at least *nn* seconds after file modification. The recommended default is 3.

ACRegMax=nn

Hold cached attributes for no more than *nn* seconds after file modification. The recommended default is 60.

ACDirMin=nn

Hold cached attributes for at least *nn* seconds after directory update. The recommended default is 30.

ACDirMax=nn

Hold cached attributes for no more than *nn* seconds after directory update. The recommended default is 60.

2.5 XNFS Server Operations

The four XNFS server administration operations are used to export or unexport local file system objects. All require appropriate privileges. These operations merely update the set of *ExportedFileSystem* objects; it is the responsibility of the *mount* server to interpret incoming *MNTPROC_MNT* requests and determine whether or not the request is to be granted (see Section 2.6.2 on page 21). The mechanism by which the XNFS server operations communicate with the *mount* server is not specified.

It should be noted that the file system objects must be local to the server system; XNFS does not support third party or proxy operations.

2.5.1 The *ExpFileSysOp* Operation

An XNFS server implementation will provide a mechanism by which a local file system object - a directory or file - is made available for mounting over the network by XNFS clients. The result of invoking this operation is the creation of an *ExportedFileSystem* object which is added to the set of exported file systems known to the *mount* server.

The *ExpFileSysOp* operation is invoked with a set of attributes which specify what is to be exported, which clients may mount the object, and how NFS accesses to the object are to be handled. These attributes are drawn from the attributes of the *ExportedFileSystem* object defined above; at a minimum, the *PathName* must be specified.

It is not permitted to export a file system object which is either a parent or a sub-directory of one which is currently exported and *is within the same physical file system*. This is because security considerations dictate that an XNFS server must disallow access to the parent directory of the exported file system object; if this constraint is violated, the server cannot enforce such a policy.

The *ExpFileSysOp* operation does not involve any direct protocol interaction with client systems.

2.5.2 The *UnExpFileSysOp* Operation

The *UnExpFileSysOp* operation may be used to make a previously exported file system object unavailable. The effect is to remove the specified *ExportedFileSystem* from the set of exported file systems known to the mount server and to destroy the object.

Only one attribute may be given.

Attribute	**Type**	**Recommended Default**
PathName	path	NO DEFAULT

The *PathName* attribute uniquely identifies the *ExportedFileSystem* object which is to be unexported.

If XNFS clients have previously mounted this file system, they may retain valid file handles for objects within the file system which is being unexported. The *UnExpFileSysOp* operation does not invalidate these file handles, which means that NFS operations to the file system may continue after the *UnExpFileSysOp* operation has been completed. An XNFS implementation may include a separate mechanism to forcibly invalidate existing file handles.

The execution of the *UnExpFileSysOp* operation does not involve any direct protocol interactions with XNFS client systems.

2.5.3 The *ExpStdFileSysOp* Operation

An XNFS server implementation may optionally provide a mechanism whereby a predefined sequence of *ExpFileSysOp* operations is executed. This sequence is performed by invoking the *ExpStdFileSysOp* operation, which will usually be a part of the system start-up sequence. To support this operation, the XNFS server will implement a *StandardExports* database. Each record in this database consists of a set of *ExpFileSysOp* attributes as described above. When the *ExpStdFileSysOp* operation is invoked, an *ExpFileSysOp* operation is performed for each record of the *StandardExports* database.

This specification does not describe the format of the *StandardExports* database, nor the mechanisms which may be used to manage it.

2.5.4 The *UnExpStdFileSysOp* Operation

The *UnExpStdFileSysOp* operation may be used to unexport every element of the *StandardExports* database. For each element of the database, it causes an *UnExpFileSysOp* operation to be invoked with the corresponding *PathName* attribute.

2.6 XNFS Client Operations

The five XNFS client administration operations are used to mount or unmount remote file system objects, or to determine what file systems are exported by a server. All except for *ShowExpFileSysOp* require appropriate privileges.

2.6.1 The *ShowExpFileSysOp* Operation

An XNFS client implementation will provide a mechanism by which the user may interrogate an XNFS server system to determine what file system objects are exported. The *ShowExpFileSysOp* operation is invoked with the following attribute:

Attribute	**Type**	**Recommended Default**
Server	hostname	NO DEFAULT

For each *ExportedFileSystem* object exported by the XNFS server *hostname*, *ShowExpFileSysOp* will output the pathname together with a list of the hosts or groups listed in the *Access* attribute of the exported file system.

The *ShowExpFileSysOp* operation causes a *MNTPROC_EXPORT* RPC call to be performed to the mount server on the XNFS server system (see Section 8.3.0 on page 115). The mount server responds by sending a list of exported file system entries, which the *ShowExpFileSysOp* operation will render into a suitable form for output. This specification does not define this format.

2.6.2 The *MntFileSysOp* Operation

An XNFS client implementation will provide a mechanism by which a remote file system object - a directory or file exported by an XNFS server - is mounted. Mounting a remote file system consists of creating a *MountedFileSystem* object with the desired attributes, and associating it with a local file system object - the *mountpoint* - so that subsequent XSI references to the mountpoint will be interpreted as references to the remote file system object.

The attributes of the *MntFileSysOp* operation fall into two groups. The first consists of attributes for the *MountedFileSystemObject* as described above. The second is drawn from the following set of options which control the way in which the *MntFileSysOp* call is performed.

Attribute	**Type**	**Recommended Default**
ReMount	boolean	False
MountRetryCount	integer	0
MountRetry	Background or Foreground	Foreground

A fuller description of each attribute follows.

ReMount={True or False}

If True, the file system must already be mounted, otherwise an error results. If the *Mode* with which the file system was previously mounted is the same as that specified in this *MntFileSysOp* request, the operation has no effect. Otherwise the file system is unmounted and remounted.

MountRetryCount=nn, *MountRetry={Background or Foreground}*

If the first attempt to mount the file system fails due to a recoverable error (such as the server being temporarily inaccessible), and the *MountRetryCount* is non-zero, the request will be retried until it is successful or *MountRetryCount* attempts have been made. If *MountRetry* is specified as Background, a new process will be forked to perform the retries in the background, and the *MntFileSysOp* operation will terminate successfully.

The execution of the *MntFileSysOp* operation involves the acquisition of an NFS file handle for the exported file system object. This is realised by executing a *MNTPROC_MNT* RPC call to the mount server program on the XNFS server system, providing as an argument the pathname of the object (see Section 8.3.0 on page 111). The mount server interrogates the local XNFS server to retrieve a file handle for the object; if this is successful it examines the set of *ExportedFileSystem* objects established by *ExpFileSysOp* operations to find one which corresponds to that provided by the client. Having identified the exported file system, the mount server will verify access by the client based upon the attributes of the exported file system. If the mount is allowed, the file handle is returned to the XNFS client system. A *MountedFileSystem* object is created which includes the file handle and the attributes supplied in the *MntFileSysOp* invocation. The final procedure - binding the *MountedFileSystem* and file handle to the mountpoint on the client - is implementation-dependent, and is not specified by this standard.

2.6.3 The *UnMntFileSysOp* Operation

The *UnMntFileSysOp* operation will be used to unmount a previously mounted *MountedFileSystem* object. The file system may be identified by giving a (remote) *PathName* and *Server*, or by specifying the (local) *MountPoint*. Upon completion of the *UnMntFileSysOp* operation, the previous contents of the mountpoint will once more be accessible.

An implementation is not required to support both mechanisms to specify the file system to be unmounted.

Attribute	**Type**	**Recommended Default**
PathName	path	NO DEFAULT
Server	hostname	NO DEFAULT
MountPoint	path	NO DEFAULT

A fuller description of each attribute follows.

PathName=pathname

This attribute identifies the remote file system object to be unmounted.

Server=hostname

This attribute identifies the XNFS server system from which the object was mounted.

MountPoint=pathname

This attribute identifies the local file system object on which the XNFS object was mounted.

If the mounted file system is in use, the *UnMntFileSysOp* operation will fail with an appropriate error indication. The definition of ‘‘in use’’ is implementation-dependent.

The execution of the *UnMntFileSysOp* operation causes a *MNTPROC_UMNT* RPC to be made to the mount server process on the XNFS server system with the pathname as an argument (see Section 8.3.0 on page 113).

2.6.4 The *MntStdFileSysOp* Operation

An XNFS client implementation may provide a mechanism by which a predefined sequence of *MntFileSysOp* operations is executed. This sequence is performed by invoking the *MntStdFileSysOp* operation, which will usually be a part of the system start-up sequence. To realise this mechanism, a *StandardMounts* database will exist. Each record in this database consists of a set of *MntFileSysOp* attributes as described above. When the *MntStdFileSysOp* operation is invoked, a *MntFileSysOp* operation is performed for each record of the *StandardMounts* database.

This specification does not describe the format of the *StandardMounts* database, nor the mechanisms which may be used to manage it.

2.6.5 The *UnMntAllFileSys* Operation

An XNFS client implementation may include an *UnMntAllFileSys* operation which may be used to unmount some or all mounted file systems.

Attribute	**Type**	**Recommended Default**
Server	hostname	All servers

A fuller description of each attribute follows.

Server=hostname

If this attribute is given, all file systems mounted from the given XNFS server are unmounted. If it is not, all NFS file systems which are mounted by the client are unmounted.

For each *MountedFileSystem* object which is to be unmounted, an *UnMntFileSysOp* operation is invoked with the corresponding *PathName* and *Server* attributes.

2.7 File and Directory Operations

Unlike the administrative part of the model, file and directory access is precisely specified. The client system must implement a mechanism whereby every active file or directory object is marked as local or remote. For each active remote object the file handle and *MountedFileSystem* must be recorded. Whenever a process executing on the client invokes an XSI function which refers to a remote file or directory, the XNFS client implementation must interpret this request in terms of NFS remote procedure calls. Appendix D indicates, but does not completely specify, how this may be achieved. Chapter 3, Chapter 4, Chapter 5 and Chapter 7 define how NFS requests are constructed and transmitted to the server. The XNFS server implementation is responsible for performing UID mapping according to the values of the *AnonMapping* and *Root* attributes of the *ExportedFileSystem*, interpreting the resulting request in terms of local file system operations and transmitting the response to the client. Appendix A, Appendix B and Appendix C indicate, but do not fully specify, the deviations from standard XSI semantics which may occur.

File locking operations are discussed separately in Chapter 9, Chapter 10 and Chapter 11.

2.8 Operation in an International Environment

XNFS can operate in a situation where more than one natural language and cultural environment is in use. There may be different environments on client and server and there may be different environments used within the client or the server.

The internationalisation facilities that are available and the considerations that apply are discussed in the X/Open Internationalisation Guide. When the client is implemented in accordance with its provisions, internationalisation facilities are available, transparently, on the resulting distributed file system.

Two particular considerations are discussed below.

2.8.1 Internationalized XNFS Operations

It is desirable that the XNFS client and server administrative operations should be internationalized. This means that each user can invoke these operations via a user interface that recognizes the character set, collating sequence, date format and other conventions of their cultural environment and that accepts commands and displays text expressed in their natural language. It is recommended that implementations include internationalized administrative operations implemented in accordance with the recommendations of the X/Open Internationalisation Guide.

2.8.2 Remote File Systems Created in Different Locales

It is possible to access a file system that has been created on the server in a locale that is not in use, or even not available, on the client. This may give rise to unpredictable results. For example, filenames created using character sets that are not supported on the client may be displayed incorrectly. This problem can arise within a single system on which several locales are available, but is more likely to occur when XNFS is used to provide transparent file access across several different systems, and users should beware.

Chapter 3

XDR Protocol Specification

This chapter specifies a protocol that is used by many implementors of XNFS. It is derived from a document designated RFC 1014 by the ARPA Network Information Centre (see Section 1.7 on page 7).

This chapter includes only the subset of XDR that is required to define the XNFS protocols.

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 as defined in the C Programming Language, just as Courier, as defined in Courier: The Remote Procedure Call Protocol 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 (see Section 3.1.2 on page 28). 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.

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, as described in On Holy Wars and a Plea for Peace, 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.

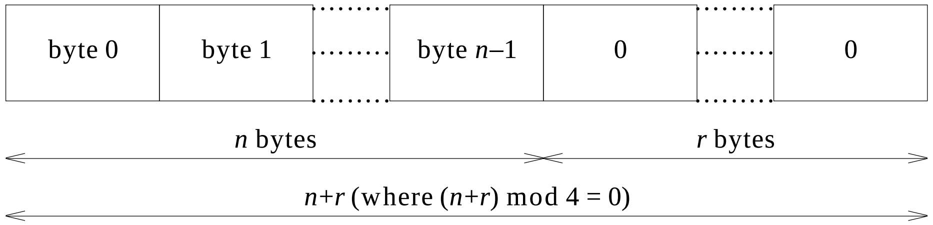
No data-typing is provided in the XDR language as it has a relatively high cost (encoding and interpreting the type fields) and most protocols already know what data types they are expecting. However, one can still get the benefits of data-typing using XDR. One way is to encode two things; first, a string 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 must 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 are 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.



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.

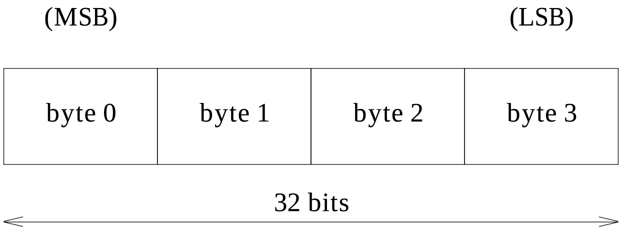
For each data type in the language, a general paradigm declaration is shown. Note that angle brackets (< and >) denote variable-length sequences of data, and square brackets ([and]) denote fixed-length sequences of data. *n*, *m* and *r* denote integers. For the full language specification and more formal definitions of terms such as ‘‘identifier’’ and ‘‘declaration’’, refer to Section 3.3 on page 37.

For some data types, more specific examples are included. A more extensive example of a data description is in Section 3.4 on page 40.

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:

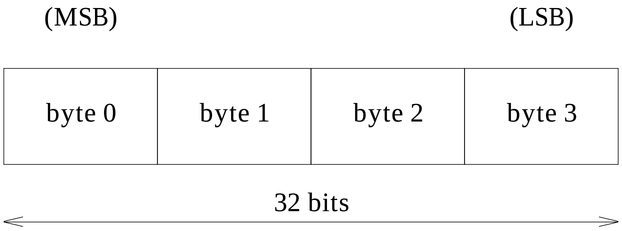
```
int identifier;
```



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:

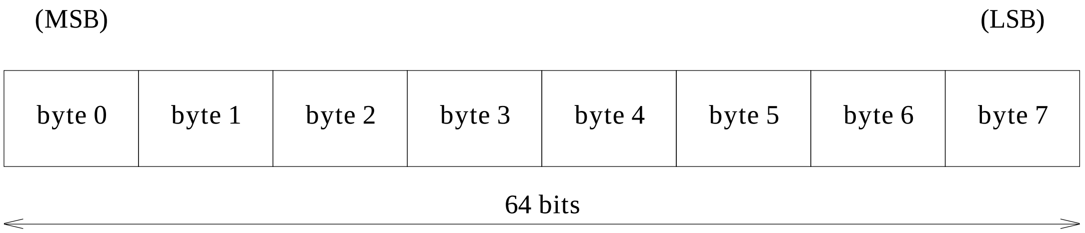
```
unsigned int identifier;
```



3.2.3 Hyper Integer and Unsigned Hyper Integer

Two extensions of the integer and unsigned integer types defined previously are the 64-bit (8-byte) numbers called hyper integer and unsigned hyper integer. They are represented in two's complement notation. The most and least significant bytes are 0 and 7, respectively. Their declarations are as follows:

```
hyper identifier;
unsigned hyper identifier;
```



3.2.4 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.

3.2.5 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:

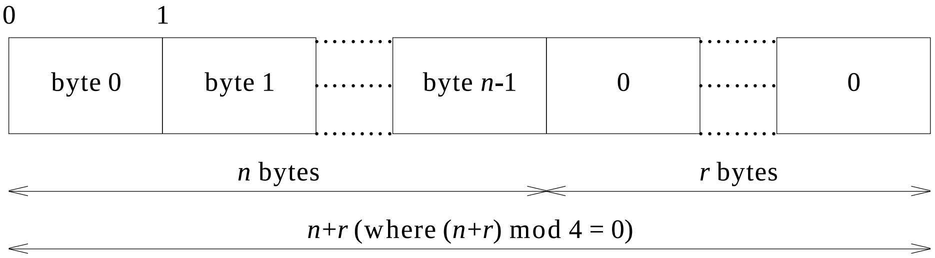
```
enum { FALSE = 0, TRUE = 1 } identifier;
```

3.2.6 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.



3.2.7 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>;
```

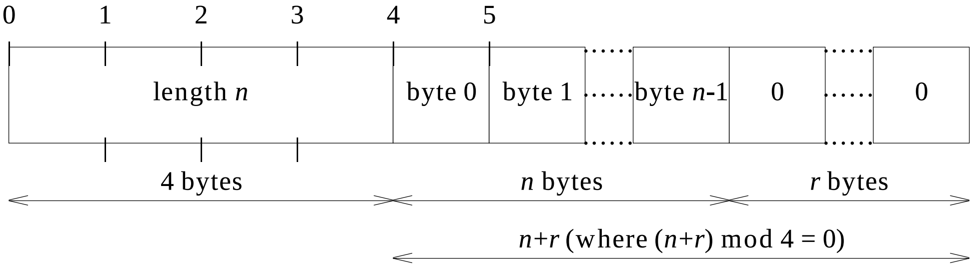
or

```
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.

3.2.8 String

The standard defines a string of n (numbered 0 to n −1) bytes to be the number n encoded as an unsigned integer (as described above), and followed by the n bytes of the string. Each byte must be regarded by the implementation as being 8-bit transparent data. This allows use of arbitrary character set encodings. 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>;
```

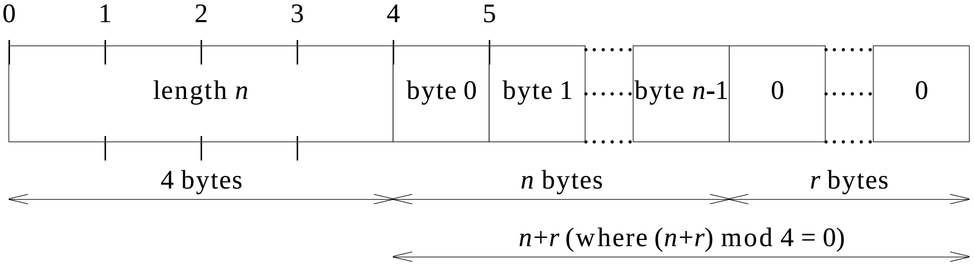
or

```
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 filename can be no longer than 255 bytes, as follows:

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

This can be illustrated as:



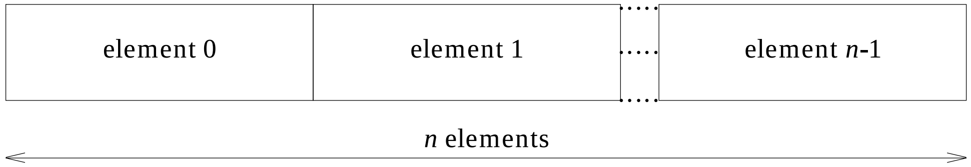
It is an error to encode a length greater than the maximum defined in the declaration.

3.2.9 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.



3.2.10 Variable-Length Array

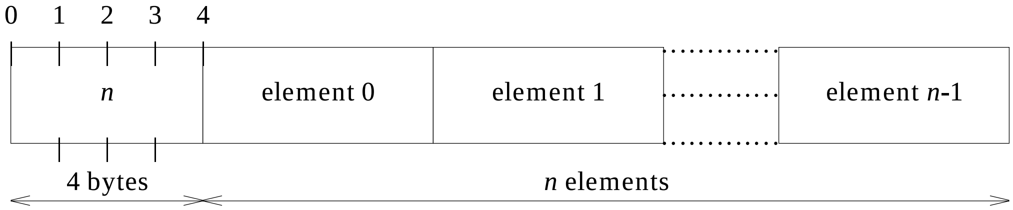
Counted arrays provide the ability to encode variable-length arrays of homogeneous elements. The array is encoded as the element count *n* (an unsigned integer) followed by the encoding of each of the array's elements, starting with element 0 and progressing through element $n - 1$. The declaration for 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 $2^{32} - 1$.



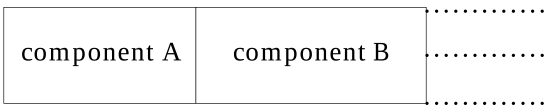
It is an error to encode a value of *n* that is greater than the maximum described in the declaration.

3.2.11 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.



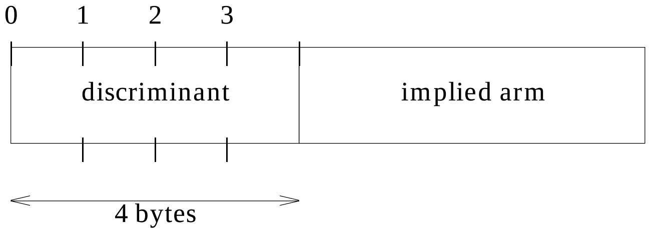
3.2.12 Discriminated Union

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.

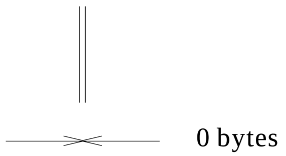


3.2.13 Void

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:



3.2.14 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;
```

3.2.15 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.

3.2.16 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 ‘‘–’’.

The character set is consistent with ISO 8859-1: 1987.

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"
    | [ "unsigned" ] "hyper"
    | "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 *
```

3.3.4 Syntax Notes

- The following are keywords and cannot be used as identifiers: **bool**, **case**, **const**, **default**, **enum**, **hyper**, **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.

3.3.5 Use of XDR

Although XDR is used by many implementations of XNFS, 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. Furthermore, an implementation of the XDR language is not constrained to use the lexical and syntactical conventions defined in this specification; in particular, other codesets and reserved words may be used in implementations that are not based on the English language.

3.4 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

Chapter 4

Remote Procedure Calls : Protocol Specification

This chapter specifies a protocol that is used by many implementors of XNFS. It is derived from a document designated RFC 1057 by the ARPA Network Information Centre (see Section 1.7 on page 7).

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 on page 27). 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, Implementing Remote Procedure Calls 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, as described in Transmission Control Protocol — DARPA Internet Program Protocol Specification, 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, as described in User Datagram Protocol, 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 reuse its previous transaction ID when retransmitting a request. The server application, knowing this fact, may choose to remember this ID after granting a request and not 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, the XNFS specification defines only the use of connectionless 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 on page 61.)

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
- 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
- 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, 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, Inc. 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 on page 52.

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 four ‘‘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,
    AUTH_DES     = 3,
    AUTH_KERB    = 4,
    /*  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 is 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<16>;
};
```

The *stamp* is an arbitrary ID which the caller machine may generate. The *machinename* is the name of the caller’s machine (like ‘‘krypton’’). Implementations and applications must be able to handle machine names as 8-bit transparent data (allowing use of arbitrary character set encodings). For maximum portability and interworking, it is recommended that applications and users define machine names containing only the characters of the Portable Filename Character Set defined in ISO / IEC 9945-1: 1990. 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 XNFS specification and is supported by (PC)NFS clients, it is only optional in the X/Open **(PC)NFS** Specification. It is unspecified whether a server uses all the entries in the *gids* array or only [NGROUPS_MAX] entries, which may be zero. [NGROUPS_MAX] is defined in the X/Open Commands and Utilities Specification (see reference **XCU**). The verifier accompanying the credentials (the *verf* field in *call_body* and *accept_reply*) should be of *AUTH_NULL* (defined above).

4.4.3 DES and Kerberos Authentication

The *AUTH_DES* flavour is defined in RFC 1057. It provides DES-encrypted authentication parameters based on a network-wide name, with session keys exchanged via a public key scheme. The *AUTH_KERB* flavour provides DES encrypted authentication parameters based on a network-wide name with session keys exchanged via Kerberos, Version 4 secret keys.

The *AUTH_DES* and *AUTH_KERB* styles of authentication are based on a network-wide name. They provide greater security through the use of DES encryption and public keys in the case of *AUTH_DES*, and DES encryption and Kerberos secret keys (and tickets) in the *AUTH_KERB* case. The server and client must agree on the identity of a particular name on the network, but the name to identity mapping is more operating system independent than the *uid* and *gid* mapping in *AUTH_UNIX*. Also, because the authentication parameters are encrypted, a malicious user must know another user's network password or private key to masquerade as that user. Similarly, the server returns a verifier that is also encrypted so that masquerading as a server requires knowing a network password.

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. 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.

Chapter 5

RPC Interface to UDP Transport Services

5.1 Introduction

The purpose of this chapter is to describe how protocols defined as part of XNFS interface with the underlying transport. These protocols are 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 RPC 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.

5.2 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.

5.3 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 are contained in RFC 768 User Datagram Protocol and RFC 791 Internet Protocol (see Section 1.7 on page 7).

5.4 RPC Interface

5.4.1 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 RPC client is using.

Destination Port — The 16-bit port number on the destination host at which the RPC server is listening for requests. This must be specified by the protocol layer above UDP. The NFS and Portmap servers use fixed UDP ports; all others request a free port from the transport provider and register this port with the Portmap service; RPC clients will interrogate the Portmap service to determine the port to be used to reach the intended RPC server.

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 must 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.

5.4.2 The RPC Reply

Once the RPC request has been received and processed by the server program, the server must construct a reply packet and send it to the client. The only exception is for *asynchronous* calls, such as those used by the Network Lock Manager (see Section 10.3 on page 134 NLM Procedures, for more details). In that case the server need not send an RPC reply packet. Instead, the server sends the response, if there is one, as a new RPC from the server to the client. (The client need not send an RPC reply to this second RPC). New protocols are strongly discouraged from using asynchronous calls.

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 the 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 RPC server program.

5.4.3 Receiving a UDP Reply Packet

After sending an RPC request, the client program waits for the reply. This may be achieved in several ways; by issuing a blocking request to receive a packet, or by waiting for an asynchronous notification from the transport layer. The program may also do other processing while waiting for the reply. In any case, the application must be able to control how long it waits for a reply before it times-out the RPC.

In either case the application must be able to control how long it waits for a reply before it times-out the RPC. An implementation must consider all of the ways in which the wait may be terminated. If it is impossible to send the request to the server for some reason (if the server program has failed, or the server system is inaccessible), it is usually the case that a notification of some kind is returned to the client system, using the ICMP protocol. It is desirable that this condition can be signalled to the client application so that it need not wait for the request to time out. In addition, the reception of duplicate or delayed packets may mean that a packet is received which has the correct transport addressing but is rejected at the RPC layer (due, for example, to an invalid transaction ID). The transport layer must therefore ensure that packets can be queued in some fashion so that valid replies are not lost.

5.4.4 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 and so on.

Chapter 6

Port Mapper Protocol

6.1 Introduction

This chapter describes the port mapper protocol which is related to, but separate from, the RPC protocol. The port mapper protocol is not specified as a part of the RPC protocol to allow the implementor flexibility and to facilitate the development of new mechanisms without requiring the revision of related protocols.

6.2 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 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 (decimal).

6.3 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<>;
};
```

6.4 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 reference pages define each of the port mapper procedures.

Name

PMAPPROC_NULL Specification — Do Nothing

Call Arguments

None.

Return Arguments

None.

RPC Procedure Descriptions

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

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.

Name

PMAPPROC_SET Specification — Set Mapping

Call Arguments

```
mapping    mapping;
```

Return Arguments

```
bool    ret_value;
```

RPC Procedure Descriptions

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

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*)’’.

Name

PMAPPROC_UNSET Specification — Unset Mapping

Call Arguments

```
mapping    mapping;
```

Return Arguments

```
bool    ret_val;
```

RPC Procedure Descriptions

```
bool
PMAPPROC_UNSET(mapping) = 2;
```

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.

Name

PMAPPROC_GETPORT Specification — Get Port

Call Arguments

```
mapping    mapping;
```

Return Arguments

```
unsigned int    port;
```

RPC Procedure Descriptions

```
unsigned int
PMAPPROC_GETPORT(mapping) = 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.

Name

PMAPPROC_DUMP Specification — Dump Mappings

Call Arguments

None.

Return Arguments

```
pmaplist        mappings;
```

RPC Procedure Descriptions

```
pmaplist
PMAPPROC_DUMP (void) = 4;
```

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.

Chapter 7

XNFS: Protocol Specification, Version 2

This chapter specifies a protocol that Sun Microsystems, Inc. and others are using. It is derived from a document designated RFC 1094 by the ARPA Network Information Center (see Section 1.7 on page 7).

7.1 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 8 on page 107) 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.

7.1.1 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 on page 43). 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.

7.1.2 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 on page 27. Implementations of XDR and RPC are available in the public domain, but XNFS 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 XNFS.

7.1.3 Stateless Servers and Idempotency

The NFS protocol is stateless, in that a server need not maintain any state about the clients which it serves. It may in fact store state to improve performance, but this state is not necessary for correct operation. This means that the protocol does not include any mechanisms for managing server or client failure and restart. 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 usually still be valid and can be used to reconstruct the state needed to perform the requested operation. If the server detects that the state is invalid, it responds with an indication of the problem. In some cases the client may pass the response to the calling application. In other cases the client may take some corrective action and retry the operation.

With a few exceptions, rebooting the server must not invalidate distributed state information. One exception is that the state associated with unstable writes (see *NFSPROC3_WRITE* on page 212) may be invalidated when the server reboots. Another exception is that the state associated with temporary file systems, that is, those that are recreated from scratch by the reboot may be invalidated. This implies that distributed state will usually refer to objects held on stable server storage, though servers may 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 and their applications must be tolerant of such occurrences.

7.2 XNFS 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.

7.2.1 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, and so on) 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 filenames) 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. Ordinary 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 7.4 on page 78.

An exception to the single component lookup policy can be made in the case of a multi-component lookup relative to a public filehandle (see Appendix E). In this case the pathname is required to be slash (/) separated and evaluated by the server. The server must evaluate any symbolic links that occur in intermediate components of the path, but not a link that occurs as the final component.

Although files and directories are similar objects in many ways, different procedures are used to read directories and files. This enforces a common network representation of directory contents and places the XDR encoding of this information directly in the NFS protocol, rather than overloading the interpretation of file access operations. It also enforces an access model in which it is important to retrieve partial directory information or to start a directory search at an invalid point. 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 must 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 *NFSPROC_READLINK* if a *NFSPROC_LOOKUP* returns an entry typed as an **NFLNK**, and should be prepared to handle failures of any symbolic link operations.

7.3 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 uses the UDP portnumber 2049 decimal. Since this is not an officially assigned port, it is possible that it may change in the future. For maximum interoperability it is recommended (but not required) that NFS servers use UDP port 2049 if possible, and that NFS clients use the portmap mechanism to locate the NFS program on a server.

WebNFS servers must use UDP and TCP port 2049.

7.3.1 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 filename 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;
```

7.3.2 Basic Data Types

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

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,
};
```

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 filename 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.

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.

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.

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.

A filehandle that consists of 32 zero bytes is called the *public* filehandle. It is used by WebNFS clients to identify an associated public directory on the server. See Appendix E for further information.

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.

diropok

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

The **diropok** structure is used by the server to return the file handle and attributes of a file after a successful *NFSPROC_LOOKUP*, *NFSPROC_CREATE* or *NFSPROC_MKDIR* operation.

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 preferred block size in bytes for the file; *rdev* is the device number of the file if it is type **NFCHR** or **NFBLK**; *blocks* is the number of 512-byte blocks the file takes up on the server; *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 must be **NFDIR**.
0020000	This is a character special file; *type* field must be **NFCHR**.
0060000	This is a block special file; *type* field must be **NFBLK**.
0100000	This is a regular file; *type* field must be **NFREG**.
0120000	This is a symbolic link file; *type* field must be **NFLNK**.
0140000	This is a named socket; *type* field must 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:

1. The bits correspond to 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.
2. The *rdev* field in the attributes structure is an operating system-specific device specifier.

sattr

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

The **sattr** structure contains the file attributes which can be set from the client. The fields are the same as for **fattr** above. A value of 0xffffffff indicates a field that must be ignored. A *size* of zero means the file must be truncated to zero length.

filename

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

The type **filename** is used for passing filenames or pathname components. A string length of zero is invalid.

Implementations and applications must be able to handle file names as 8-bit transparent data (allowing use of arbitrary character set encodings). For maximum portability and interworking, it is recommended that applications and users define file names containing only the characters of the Portable Filename Character Set defined in ISO / IEC 9945-1: 1990.

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. A string length of zero is invalid.

For maximum portability and interworking, it is recommended that applications and users define path names containing only the slash character (if required) plus the characters of the Portable Filename Character Set defined in ISO / IEC 9945-1: 1990.

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 performed.

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.

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*.

7.4 XNFS 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 has access to all files no matter what permission and ownership they have, an NFS client request on behalf of such a user will be made with the user ID of zero. This ‘‘super-user’’ permission might not be allowed on the server, since anyone who can gain that privilege on their client system could gain access to all remote files. An XNFS 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.

7.5 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;
```

The following reference pages define each of the server mapper procedures.

Name

NFSPROC_NULL — Do Nothing

Call Arguments

None.

Return Arguments

None.

RPC Procedure Description

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

Description

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

Return Codes

None.

Name

NFSPROC_GETATTR — Get File Attributes

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 7.3.2 on page 73.

RPC Procedure Description

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

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 7.3.2 on page 73.

Return Codes

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

Name

NFSPROC_SETATTR — Set File Attributes

Call Arguments

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

fhandle and **sattr** are defined in Section 7.3.2 on page 73.

Return Arguments

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

fattr and **sattr** are defined in Section 7.3.2 on page 73.

RPC Procedure Description

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

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 server will truncate the file. This operation should only be permitted on regular files.

Return Codes

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

Name

NFSPROC_ROOT — Get File System Root

Call Arguments

None.

Return Arguments

None.

RPC Procedure Description

```
void
NFSPROC_ROOT(void) = 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 Chapter 8 on page 107.)

Return Codes

None.

Name

NFSPROC_LOOKUP — Look Up File Name

Call Arguments

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

fhandle and **filename** are defined in Section 7.3.2 on page 73.

Return Arguments

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

diropok is defined in Section 7.3.2 on page 73.

RPC Procedure Description

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

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.

Return Codes

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

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 filename 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.

Name

NFSPROC_READLINK — Read From Symbolic Link

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 7.3.2 on page 73.

RPC Procedure Description

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

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 will return a *PROC_UNAVAIL* error. An NFS client should only issue an *NFSPROC_READLINK* if a lookup returns an entry that is typed as *NFLNK*, and must 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.

Return Codes

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

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

Name

NFSPROC_READ — Read From File

Call Arguments

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

fhandle is defined in Section 7.3.2 on page 73.

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 7.3.2 on page 73.

RPC Procedure Description

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

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 7.5.0 on page 102).

Note that the argument *totalcount* is unused.

Return Codes

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

Name

NFSPROC_WRITECACHE — Write to Cache

Call Arguments

None.

Return Arguments

None.

RPC Procedure Description

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

Description

Function not used.

Return Codes

None.

Name

NFSPROC_WRITE — Write to File

Call Arguments

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

fhandle is defined in Section 7.3.2 on page 73.

Return Arguments

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

fattr and **sattr** are defined in Section 7.3.2 on page 73.

RPC Procedure Description

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

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.

Return Codes

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

Name

NFSPROC_CREATE — Create File

Call Arguments

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

diropargs and **sattr** are defined in Section 7.3.2 on page 73.

Return Arguments

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

fhandle, **fattr** and **stat** are defined in Section 7.3.2 on page 73.

RPC Procedure Description

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

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 7.5.0 on page 99).

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

Return Codes

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

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 filename 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.

Name

NFSPROC_REMOVE — Remove File

Call Arguments

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

fhandle and **filename** are defined in Section 7.3.2 on page 73.

Return Arguments

```
stat status;
```

stat is defined in Section 7.3.2 on page 73.

RPC Procedure Description

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

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 must be performed using the *NFSPROC_RMDIR* procedure (see Section 7.5.0 on page 101).

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 file system. It is impossible for a stateless server to implement these semantics.

Return Codes

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

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 filename 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.

Name

NFSPROC_RENAME — Rename File

Call Arguments

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

diropargs is defined in Section 7.3.2 on page 73.

Return Arguments

```
stat status;
```

stat is defined in Section 7.3.2 on page 73.

RPC Procedure Description

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

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; that is, 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.

Return Codes

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

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 filename 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.

Name

NFSPROC_LINK — Create Link to File

Call Arguments

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

fhandle and **diropargs** are defined in Section 7.3.2 on page 73.

Return Arguments

```
stat status;
```

stat is defined in Section 7.3.2 on page 73.

RPC Procedure Description

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

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.

Return Codes

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

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 filename 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.

Name

NFSPROC_SYMLINK — Create Symbolic Link

Call Arguments

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

diropargs, **path** and **sattr** are defined in Section 7.3.2 on page 73.

Return Arguments

```
stat status;
```

stat is defined in Section 7.3.2 on page 73.

RPC Procedure Description

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

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 will return an *PROC_UNAVAIL* error. An NFS client must 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.

Return Codes

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

Name

NFSPROC_MKDIR — Create Directory

Call Arguments

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

diropargs and **sattr** are defined in Section 7.3.2 on page 73.

Return Arguments

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

diropok is defined in Section 7.3.2 on page 73.

RPC Procedure Description

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

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.

Return Codes

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

Name

NFSPROC_RMDIR — Remove Directory

Call Arguments

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

fhandle and **filename** are defined in Section 7.3.2 on page 73

Return Arguments

```
stat status;
```

stat is defined in Section 7.3.2 on page 73.

RPC Procedure Description

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

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.

Return Codes

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

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 filename 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.

Name

NFSPROC_READDIR — Read From Directory

Call Arguments

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

fhandle and **nfscookie** are defined in Section 7.3.2 on page 73.

Return Arguments

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

filename and **nfscookie** are defined in Section 7.3.2 on page 73.

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

RPC Procedure Description

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

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 must 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 7.3.2 on page 73).

A cookie encodes (opaquely) the notion of a pointer into a directory. The length of time for which a cookie is valid is not defined by this specification. It is possible that directory operations on the server may mean that when a cookie is presented by a client, it is no longer possible to seek to the corresponding position in the directory, or it may be that seeking to that position might cause directory entries which had already been returned to be repeated. In these cases the server should return an **eof** indication even if this means that not all directory entries are returned.

Return Codes

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

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.

Name

NFSPROC_STATFS — Get File System Attributes

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;
};
```

RPC Procedure Description

```
statfsres
NFSPROC_STATFS(fhandle) = 17;
```

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.

Return Codes

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

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 8

Mount Protocol

8.1 Introduction

The mount protocol is separate from, but related to, the NFS protocol. It provides operating system-specific services to get 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.

8.2 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 on page 61, to find the port number on which the mount service is registered.

8.2.1 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;
```

8.2.2 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.

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 7.3.2 on page 73.

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. In case of an error, status will be set to one of the following values:

```
enum stat {
    MNT_OK = 0,
    MNT_EPERM = 1,
    MNT_ENOENT = 2,
    MNT_EACCESS = 13,
    MNT_EINVAL = 22
    };
```

For a detailed description of the error conditions see Section 2.3, Error Numbers of the X/Open System Interfaces and Headers Specification (see reference **XSH**), [EPERM], [ENOENT], [EACCESS] and [EINVAL].

dirpath

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

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

Implementations and applications must be able to handle pathnames as 8-bit transparent data (allowing use of arbitrary character set encodings). For maximum portability and interworking, it is recommended that applications and users define pathnames containing only the slash character (if required) plus the characters of the Portable Filename Character Set defined in ISO/IEC 9945-1:1990.

name

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

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

Implementations and applications must be able to handle names as 8-bit transparent data (allowing use of arbitrary character set encodings). For maximum portability and interworking, it is recommended that applications and users define names containing only the characters of the Portable Filename Character Set defined in ISO / IEC 9945-1: 1990.

8.3 Server Procedures

The following reference pages 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;
```

Name

MNTPROC_NULL — Do Nothing

Call Arguments

None.

Return Arguments

None.

RPC Procedure Description

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

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.

Return Codes

None.

Name

MNTPROC_MNT — Add Mount Entry

Call Arguments

```
dirpath dirname;
```

Return Arguments

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

fhandle is defined in Section 7.3.2 on page 73.

RPC Procedure Description

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

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*.

Return Codes

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

MNT_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 runs with UID 0.) This generally indicates a server configuration error.

MNT_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.

MNT_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*.

MNT_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.

Name

MNTPROC_DUMP — Return Mount Entries

Call Arguments

None.

Return Arguments

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

name and **dirpath** are defined in Section 7.3.2 on page 73.

RPC Procedure Description

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

Description

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

Return Codes

None.

Name

MNTPROC_UMNT — Remove Mount Entries

Call Arguments

```
dirpath dirname;
```

Return Arguments

None.

RPC Procedure Description

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

Description

Removes the mount list entry for the input *dirname* that records the fact that *dirname* has been mounted by the client. *dirname* must be identical to the argument used in the corresponding *MNTPROC_MNT* call. It is not sufficient that the path identified the same file system object after processing of links, and so on. It must be textually identical.

Return Codes

None.

Name

MNTPROC_UMNTALL — Remove all Mount Entries

Call Arguments

None.

Return Arguments

None.

RPC Procedure Descriptions

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

Removes all of the mount list entries for this client.

Return Codes

None.

Name

MNTPROC_EXPORT — Return Export List

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 7.3.2 on page 73.

RPC Procedure Description

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

Description

Returns a variable number of export list entries. Each entry contains a file system 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 XNFS 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 file system, or options describing access control or UID mapping.

Return Codes

None.

Chapter 9

File Locking over XNFS

9.1 Introduction

Because NFS is a stateless service, it cannot provide inherently stateful services such as file locking and access control synchronisation. Instead these services are provided by two cooperating processes: the Network Lock Manager (NLM) and the Network Status Monitor (NSM). The NLM and NSM are RPC-based servers which normally execute as autonomous ‘‘daemon’’ servers on XNFS client and server systems. They work together to provide stateful file locking and access control capability over XNFS. This chapter describes the RPC protocols which the NLM and NSM implement, and defines how they interact. Full specifications of the NLM and NSM protocols are in Chapter 10 on page 127 and Chapter 11 on page 161.

9.1.1 NLM Protocol

The NLM is a service that provides advisory X/Open CAE file and record locking, and DOS compatible file sharing and locking in an XNFS environment. Its use is strongly encouraged but not mandatory. XNFS 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 are honoured by processes running on an X/Open host and *vice versa*.

The NLM provides two types of locks, monitored and non-monitored.

Monitored Locks

Monitored locks are reliable. A client process which establishes monitored locks can be assured that if the server host, on which the locks are established, crashes and recovers, the locks will be reinstated without any action on the client process’ part. Likewise, locks that are held by a client process will be discarded by the NLM on the server host if the client host crashes before the locks are released.

Monitored locks require both the client and server hosts to implement the NSM protocol.

Monitored locks are preferred over the non-monitored locks.

Non-monitored Locks

Non-monitored locks are provided to support single-tasking personal computers that cannot run an NSM due to memory or speed constraints. A client that is able to run an NSM should use monitored locks.

Non-monitored locks provide the same functionality as monitored locks except if the server host, on which the locks are established, crashes and recovers, the locks will not be re-established. The personal computer client is responsible for detecting a server host failure and re-establishing the locks. Additionally, the personal computer client must inform the server NLM when it has been rebooted so it can discard all locks and file shares held for the client.

9.1.2 NSM Protocol

The NSM is a service that provides applications with information on the status of network hosts. It is included in this document as it is heavily used by the NLM to track hosts that have established locks and the hosts that are holding those locks. Although the NSM is a general service, this document will only describe the NSM as it is used by the NLM.

Each NSM keeps track of its own ‘‘state’’ and notifies any interested party of a change in this state. The state is merely a number which increases monotonically each time the condition of the host changes: an even number indicates the host is down, while an odd number indicates the host is up.

The NSM does not actively ‘‘probe’’ hosts it has been asked to monitor; instead it waits for the monitored host to notify it that the monitored host’s status has changed (that is, crashed and rebooted).

When it receives an SM_MON request an NSM adds the information in the SM_MON parameter to a notify list. If the host has a status change (crashes and recovers), the NSM will notify each host on the notify list via the SM_NOTIFY call. If the NSM receives notification of a status change from another host it will search the notify list for that host and call the RPC supplied in the SM_MON call.

For obvious reasons, the NSM maintains copies of its current state and of the notify list on stable storage.

For correct operation of the NLM, the client and server hosts are required to monitor each other. When a lock request is issued by a process running on the client host, the NLM on the client host requests the NSM on the client host to monitor the server host. The client NLM then transmits the lock request to the NLM on the server. On reception of the lock request the NLM on the server host will request the NSM on the server host to monitor the client host. In this way each host is monitored by the NSM on the other host.

9.2 Interaction

It is assumed that the user process requests locks and file shares via a user-level API or system call such as the XSI *fcntl*( ). The NLM protocol provides both synchronous and asynchronous procedures. An implementor may choose to use either the synchronous or asynchronous procedures to implement client functionality, but must be able to accept and process both types of requests.

This section will describe the interaction between the NLM and the NSM for the synchronous procedures. For simplicity it is assumed that all network lock requests are passed to the client NLM for handling. This is an implementation dependency; the protocol does not require this.

9.2.1 Monitored Locks

Monitored locks require both the client and server hosts to support the NSM protocol.

Locking

NLM_LOCK requests may be blocking or non-blocking. When the server NLM receives the NLM_LOCK request, it must make a call to the SM_MON procedure on its local NSM to monitor the calling host. The SM_MON call includes the name of the host to be monitored and an RPC to be called if the NSM is notified of a state change for the monitored host. The RPC information includes a transport end-point, program number, program version, procedure number and an opaque argument. The RPC information is of significance only to an NLM implementation, and is not defined by this specification.

If the lock can be granted immediately, or the call was non-blocking, the RPC returns immediately with the appropriate status (granted or denied).

If the lock cannot be granted immediately (it conflicts with an existing lock) and the call was a blocking call, the RPC will return with a blocked status, thus allowing the client NLM to continue processing. At this point the client NLM can choose to cancel the outstanding lock request by calling the NLM_CANCEL procedure. Upon reception of an NLM_CANCEL request, the server NLM will then delete the outstanding lock request and may request its local NSM to stop monitoring the calling host by calling the SM_UNMON procedure.

When the blocked lock request can be processed, the server NLM makes an NLM_GRANTED call-back to the client NLM indicating success or failure.

Once the lock has been granted, the client NLM instructs the local NSM to monitor the server via the SM_MON RPC, as described above; once again, the RPC used for notification is not defined by this specification. At this point the NSMs on both the client and server hosts are monitoring each other.

Crash Recovery

When the server host crashes and is restarted, its NSM will go through the notify list and will call the SM_NOTIFY procedure for each of these hosts to inform them of the state change. Each local NSM that receives this SM_NOTIFY call will search their notify list and make the corresponding RPC supplied in the previous SM_MON call, to the interested parties. One of the interested parties will be the client NLM protocol implementation which will have supplied an RPC which can go through the steps necessary to re-establish the lost locks during the server NLM server’s grace period.

The grace period is an implementation-dependent time during which the NLM implementation will only accept requests to re-establish locks or shares that were in effect at the time of the crash. During this period any other lock or share requests will be returned with a status indicating that the NLM is in the grace period and is not accepting new requests.

If the client host crashes, upon reboot, the NSM will go through the same process notifying the NSMs on hosts in the notify list via the SM_NOTIFY procedure call that there was a change in state. The server NSM will receive this notification call and in turn notify the server NLM, via the provided RPC, that the client host had crashed. The server NLM can then dispose of all locks and shares held by the crashed host.

Unlocking

A monitored lock is unlocked by making a call to the NLM_UNLOCK procedure. The server NLM will process the request and release the lock and return status. The server NLM can then ask its local NSM to stop monitoring the calling host via the SM_UNMON procedure call. At this point the server NLM will check existing blocked lock requests and service them if possible.

9.2.2 Non-Monitored Locks

Non-monitored locks do not require the client or server host to support the NSM protocol. All non-monitored locks calls are synchronous.

Locking

A client host establishes a non-monitored lock by calling the NLM_NM_LOCK procedure on the server NLM. The server NLM will process the lock and return status to the client host indicating whether the lock was granted or denied. The NLM_NM_LOCK procedure call cannot block and so cannot result in a call-back.

Crash Recovery

If the client host crashes while it has established locks or file shares, it must generate an NLM_FREE_ALL RPC upon reboot. When it receives an NLM_FREE_ALL request the server NLM must free all locks and files shares held by the requesting host. The client host has no way of determining whether the server host crashes, and therefore no way to re-establish the locks during the server NLM grace period. The client must be prepared to handle errors when a previously requested lock is lost.

Unlocking

The unlock operation is the same as for monitored locks. The NLM_UNLOCK procedure is called. The server NLM will process the request and release the lock as requested, and return a status.

9.3 Transport Issues

NLM and NSM implementations are required to support both UDP and TCP transports. Personal computer NFS clients will always use UDP when issuing locking and sharing requests to an NLM. Most implementations of the NSM use TCP when interacting with the local NLM and any remote NSMs, but both NLM and NSM must accept any request over either transport.

9.4 Examples of Locking

This section outlines the behaviour of the NLM and NSM daemons during an X/Open-compliant system boot, crash and reboot. It describes the following two cases:

- a client locks a file, and the server crashes and is restarted while the file is locked
- a client locks a file, then crashes and is restarted without releasing the lock.

9.4.1 Server Crash Example

Server NSM Initialisation

The server NSM is started. (Note that NSM and NLM initialisation proceed in parallel.) The server NSM retrieves a copy of the last *server state* from stable storage, increments it to the next odd value, and saves it on stable storage. It then processes the notify list on stable storage, which is initially empty.

Server NLM Initialisation

The server NLM is started. It issues an SM_UNMON_ALL to the server NSM, from which it obtains a copy of the *server state*. It then enters grace period recovery state, waits for the grace period, and then enters normal service state.

Client NSM Initialisation

The client NSM is started. (Note that NSM and NLM initialisation proceed in parallel.) The client NSM retrieves a copy of the last *client state* from stable storage, increments it to the next odd value, and saves it on stable storage. It then processes the notify list, which is initially empty.

Client NLM Initialisation

The client NLM is started. It issues an SM_UNMON_ALL to the client NSM, from which it obtains a copy of the *client state*. It then enters grace period recovery state, waits for the grace period, and then enters normal service state.

Client Lock Request

A process on the client system requests a byte range lock on a file stored on the server. The client NFS passes this request, including the file handle, to the local NLM using a private protocol.

The client NLM issues an NLM_LOCK RPC to the server NLM. This request includes a copy of the *client state*.

The server NLM determines that the lock can be granted. It verifies that this is the first lock held for the client, and issues an SM_MON call to the server NSM instructing it to monitor the client.

The server NSM saves the client name and RPC information in the notify list, committing the client name to stable storage, and reports success to the server NLM.

The server NLM records the fact that it is holding a lock for the named client which is in *client state*. It then sends a success response to the client NLM.

The client NLM verifies that this is the first lock which it is holding on the server system, and issues an SM_MON call to the client NSM instructing it to monitor the server.

The client NSM saves the server name and RPC information in the notify list, committing the server name to stable storage, and reports success to the client NLM.

The client NLM records the fact that the server NLM is holding a lock for it, and returns to the local application.

Server Failure and Restart

The server system fails and is restarted.

Server NSM Restart

The server NSM is restarted. (Note that NSM and NLM initialisation proceed in parallel.) The server NSM retrieves a copy of the last *server state* from stable storage, increments it to the next odd value, and saves it on stable storage. It then processes the notify list on stable storage, adding each name to a recovery list.

For each name in the recovery list, the server NSM issues an SM_NOTIFY RPC to the NSM on the named host. In this example it will issue an SM_NOTIFY to the client NSM, including the server name and the new *server state*.

The client NSM receives the SM_NOTIFY RPC and compares the hostname against each entry in the notify list. When it encounters a match, it calls back the client NLM using the RPC information provided with the original SM_MON RPC.

The callback procedure in the client NLM notes that the server state has changed and schedules lock recovery (see below). It then acknowledges the RPC callback from the client NSM.

After comparing the name against all entries in the notify list, the client NSM acknowledges the SM_NOTIFY RPC from the server NSM.

After processing all entries in the recovery list, the server NSM enters normal service state with an empty notify list.

Server NLM Restart

The server NLM is started. It issues an SM_UNMON_ALL to the server NSM, from which it obtains a copy of the new *server state*. It then enters grace period recovery state.

Client Lock Recovery

The client NLM now attempts to recover all locks which it was holding on the server. For each lock, it issues an NLM_LOCK request with *reclaim* set to true. This NLM_LOCK is processed as described above; the server NLM and NSM register the lock and (re)initiate monitoring of the client, and the client NLM confirms the lock and arranges for the client NSM to monitor the server.

Server NLM Restart Completion

At the conclusion of the grace period, the server NLM enters normal service mode.

Client Unlock Request

Eventually the client application releases the lock on the file. The client NLM issues an NLM_UNLOCK RPC to the server NLM.

The server NLM releases the lock, and notices that this is the last lock which was being held on behalf of the client. It issues an SM_UNMON RPC to the server NSM.

The server NSM removes the client from the notify list, and returns to the server NLM, which completes the NLM_UNLOCK request.

The client NLM deletes the lock record, and notices that this is the last lock which it was holding on the server. It issues an SM_UNMON RPC to the client NSM.

The client NSM removes the server from the notify list, and returns to the client NLM, which completes the application's unlock request.

9.4.2 Client Crash Example

Initialisation

The server and client NSM and NLM are initialised as described in the previous example.

Client Lock Request

The client lock request is processed as described in the previous example.

Client Failure and Restart

The client system fails and is restarted.

Client NSM Restart

The client NSM is restarted. (Note that NSM and NLM initialisation proceed in parallel.) The client NSM retrieves a copy of the last *client state* from stable storage, increments it to the next odd value, and saves it on stable storage. It then processes the notify list on stable storage, adding each name to a recovery list.

For each name in the recovery list, the client NSM issues an SM_NOTIFY RPC to the NSM on the named host. In this example it will issue an SM_NOTIFY to the server NSM, including the client name and the new *client state*.

The server NSM receives the SM_NOTIFY RPC and compares the hostname against each entry in the notify list. When it encounters a match, it calls back the server NLM using the RPC information provided with the original SM_MON RPC.

The callback procedure in the server NLM notes that the client state has changed and releases all locks held on behalf of the client. It then acknowledges the RPC callback from the server NSM.

After comparing the name against all entries in the notify list, the server NSM acknowledges the SM_NOTIFY RPC from the client NSM.

After processing all entries in the recovery list, the client NSM enters normal service state with an empty notify list.

Client NLM Restart

The client NLM is now restarted as normal.

Chapter 10

Network Lock Manager Protocol

10.1 Introduction

The Network Lock Manager (NLM) is a service that provides advisory X/Open CAE file and record locking, and DOS compatible file sharing and locking in an XNFS environment. Here, DOS refers to MS-DOS or PC DOS, and DOS file sharing and record locking is as defined in Disk Operating System Technical Reference, IBM part no. 6138536.

10.1.1 Versions

There are multiple versions of the NLM. This document describes version 3 which is backward compatible with versions 1 and 2.

10.1.2 Synchronization of NLMs

Due to the stateless nature of XNFS servers it is difficult to incorporate a stateful service. The NLM relies on the server holding the locks as the keeper of the state and on the NSM for information on host status (monitored locks only). When an XNFS server crashes and is rebooted, locks which it had granted may be recreated by the lock holders (clients) during a grace period. During this grace period no new locks are accepted although NFS requests are accepted. The duration of this grace period is implementation-dependent; 45 seconds is common.

10.1.3 DOS-Compatible File-Sharing Support

Version 3 of the protocol supports DOS compatible file locking and sharing. 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 is opened for reading and writing, and that subsequent users may only open it for reading. To use a DOS sharing mode 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. Because the sharing requests were intended to be used by a single-tasking client host, they are non-monitored.

10.2 RPC Information

Authentication

The NLM service uses *AUTH_UNIX* style authentication only.

Transport Protocols

The NLM Protocol supports both UDP/IP and TCP/IP transports. However, a client implementation may choose to only generate requests over the UDP/IP protocol.

Port Number

Consult the server's port mapper, described in Chapter 6 on page 61, to find the port number on which the NLM service is registered.

10.2.1 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 LM_MAXNAMELEN = LM_MAXSTRLEN+1;

const MAXNETOBJ_SZ = 1024;
```

10.2.2 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 NLM.

netobj

```
opaque netobj<MAXNETOBJ_SZ>
```

Netobj is used to identify an object, generally a transaction, owner or file. The contents and form of the netobj are defined by the client.

nlm_stats

```
enum nlm_stats {
    LCK_GRANTED = 0,
    LCK_DENIED = 1,
    LCK_DENIED_NOLOCKS = 2,
    LCK_BLOCKED = 3,
    LCK_DENIED_GRACE_PERIOD = 4
};
```

Nlm_stats are returned whenever the NLM is called upon to create or test a lock on a file.

LCK_GRANTED Indicates that the procedure call completed successfully.

LCK_DENIED Indicates that the request failed.

LCK_DENIED_NOLOCKS
Indicates that the procedure call failed because the server NLM could not

allocate the resources needed to process the request.

LCK_BLOCKED Indicates the blocking request cannot be granted immediately. The server NLM will make a call-back to the client with an NLM_GRANTED procedure call when the lock can be granted.

LCK_DENIED_GRACE_PERIOD
Indicates that the procedure call failed because the server has recently been rebooted and the server NLM is re-establishing existing locks, and is not yet ready to accept normal service requests.

Note that some versions of NFS source may use mixed or lower-case names for the enumeration constants in ‘‘nlm_stats’’.

nlm_stat

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

This structure returns lock status. It is used in many of the other data structures.

nlm_res

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

The **nlm_res** structure is returned by all of the main lock routines except for NLM_TEST which has a separate return structure defined below. Note that clients must *not* rely upon the ‘‘cookie’’ being the same as that passed in the corresponding request.

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. It is used as part of the return value from the NLM_TEST procedure. The boolean ‘‘exclusive’’ indicates whether the lock is exclusively held by the current holder. 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. The ‘‘oh’’ field is an opaque object that identifies the host, or a process on the host, that is holding the lock.

nlm_testrply

```
union nlm_testrply switch (nlm_stats stat) {
    case LCK_DENIED:
        struct nlm_holder holder;    /*  holder of the lock */
    default:
        void;
};
```

The **nlm_testrply** is used as part of the return value from the NLM_TEST procedure. If the lock specified in the NLM_TEST procedure call would conflict with a previously granted lock, information on the holder of the lock is returned in ‘‘holder’’, otherwise just the status is returned.

nlm_testres

```
struct nlm_testres {
    netobj cookie;
    nlm_testrply test_stat;
};
```

This structure is the return value from the NLM_TEST procedure. The other main lock routines return the nlm_res structure.

nlm_lock

```
struct nlm_lock {
    string caller_name<LM_MAXSTRLEN>;
    netobj fh;          /*  identify a file  */
    netobj oh;          /*  identify owner of a lock  */
    int uppid;          /*  Unique process identifier  */
    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 ‘‘caller_name’’ uniquely identifies the host making the call. The ‘‘fh’’ field identifies the file to lock. The ‘‘oh’’ field is an opaque object that identifies the host, or a process on the host, that is making the request. ‘‘uppid’’ uniquely describes the process owning the file on the calling host. The ‘‘uppid’’ may be generated in any system-dependent fashion. On an X/Open-compliant system it is generally the process ID. On a DOS system it may be generated from the program segment prefix (PSP). The ‘‘l_offset’’ and ‘‘l_len’’ 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 NSM state  */
};
```

The **nlm_lockargs** structure defines the information needed to request a lock on a server. The ‘‘block’’ field must be set to *true* if the client wishes the procedure call to block until the lock can be granted (see NLM_LOCK). A *false* value will cause the procedure call to return immediately if the lock cannot be granted. The ‘‘reclaim’’ field must only be set to *true* if the client is attempting to reclaim a lock held by an NLM which has been restarted (due to a server crash, and so on). The ‘‘state’’ field is used with the monitored lock procedure call (NLM_LOCK). It is the state value supplied by the local NSM, see Chapter 11 on page 161.

nlm_cancargs

```
struct nlm_cancargs {
    netobj cookie;
    bool block;
    bool exclusive;
    struct nlm_lock alock;
};
```

The **nlm_cancargs** structure defines the information needed to cancel an outstanding lock request. The data in the nlm_cancargs structure must exactly match the corresponding information in the nlm_lockargs structure of the outstanding lock request to be cancelled.

nlm_testargs

```
struct nlm_testargs {
    netobj cookie;
    bool exclusive;
    struct nlm_lock alock;
};
```

The **nlm_testargs** structure defines the information needed to test a lock. The information in this structure is the same as the corresponding fields in the nlm_lockargs structure.

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.

10.2.3 DOS File-Sharing Data Types

The following data types are used in version 3 of the NLM to support DOS 3.1 and above compatible file-sharing control. All file-sharing procedure calls are non-monitored.

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.

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.

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.

nlm_shareargs

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

This structure encodes the arguments for an NLM_SHARE or NLM_UNSHARE procedure call. The boolean ‘‘reclaim’’ must be *true* if the client is attempting to reclaim a previously-granted sharing request, and *false* otherwise.

nlm_shareres

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

This structure encodes the results of an NLM_SHARE or NLM_UNSHARE procedure call. The ‘‘cookie’’ and ‘‘sequence’’ fields should be ignored; they are required only for compatibility reasons. The result of the request is given by ‘‘stat’’.

nlm_notify

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

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

10.3 NLM Procedures

The following reference pages define the protocol used by the NLM using RPC Language. Version 3 of the protocol is the same as version 1 and 2 with the addition of the non-monitored locking procedures and the DOS compatible sharing procedures.

```
/*
 * NLM procedures
 */

program NLM_PROG {
    version NLM_VERSX {
        /*
         *  synchronous procedures
         */
        void         NLM_NULL(void) = 0;
        nlm_testres  NLM_TEST(struct nlm_testargs) = 1;
        nlm_res      NLM_LOCK(struct nlm_lockargs) = 2;
        nlm_res      NLM_CANCEL(struct nlm_cancargs) = 3;
        nlm_res      NLM_UNLOCK(struct nlm_unlockargs) = 4;

        /*
         *  server   NLM call-back procedure to grant lock
         */
        nlm_res      NLM_GRANTED(struct nlm_testargs) = 5;

        /*
         *  asynchronous requests and responses
         */
        void         NLM_TEST_MSG(struct nlm_testargs) = 6;
        void         NLM_LOCK_MSG(struct nlm_lockargs) = 7;
        void         NLM_CANCEL_MSG(struct nlm_cancargs) =8;
        void         NLM_UNLOCK_MSG(struct nlm_unlockargs) = 9;
        void         NLM_GRANTED_MSG(struct nlm_testargs) = 10;
        void         NLM_TEST_RES(nlm_testres) = 11;
        void         NLM_LOCK_RES(nlm_res) = 12;
        void         NLM_CANCEL_RES(nlm_res) = 13;
        void         NLM_UNLOCK_RES(nlm_res) = 14;
        void         NLM_GRANTED_RES(nlm_res) = 15;

        /*
         *  synchronous non-monitored lock and DOS file-sharing
         *  procedures (not defined for version 1 and 2)
         */
        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;
```

The NLM provides synchronous and asynchronous procedures which provide the same functionality. The client portion of an NLM may choose to implement locking and file-sharing functionality by using either set of procedure calls.

The server portion of an NLM implementation must support both the synchronous and asynchronous procedures.

The asynchronous procedures implement a message passing scheme to facilitate asynchronous handling of locking and unlocking. Each of the functions **Test**, **Lock**, **Unlock** and **Grant** is broken up into a message part, and a result part. An NLM will send a message to another NLM to perform some action. The receiving NLM will queue the request, and when it is dequeued and completed, will send the appropriate result via the result procedure. For example an NLM may send an NLM_LOCK_MSG and will expect an NLM_LOCK_RES in return. These functions have the same functionality and parameters as the synchronous procedures.

Note that most NLM implementations do not send RPC-layer replies to asynchronous procedures. When a client sends an NLM_LOCK_MSG call, for example, it should not expect an RPC reply with the corresponding xid. Instead, it must expect an NLM_LOCK_RES call from the server. The server should not expect an RPC reply to the NLM_LOCK_RES call.

Name

NLM_NULL — Do Nothing

Call Arguments

None.

Return Arguments

None.

RPC Procedure Description

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

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.

Return Codes

None.

Name

NLM_TEST — Test Lock

Call Arguments

```
struct nlm_testargs {
    netobj cookie;
    bool exclusive;
    struct nlm_lock alock;
};
```

Return Arguments

```
struct nlm_testres {
    netobj cookie;
    nlm_testrply test_stat;
};
```

RPC Procedure Description

```
nlm_testres
NLM_TEST(nlm_testargs) = 1;
```

Description

This procedure tests to see whether the monitored lock specified by ‘‘alock’’ is available to this client.

Return Codes

When the procedure returns, ‘‘test_stat.stat’’ will be set to one of the following values:

LCK_GRANTED
: Indicates that the procedure call completed successfully. The server would be able to grant the lock in question.

LCK_DENIED
: Indicates that the test failed as it conflicted with existing lock reservations for the file. ‘‘test_stat.holder’’ describes the current holder of the lock as follows. The boolean ‘‘exclusive’’ indicates whether the lock is exclusively held by the current holder or whether other locks are permitted. The integer ‘‘uppid’’ provides a unique per-process identifier for lock differentiation. The unsigned values ‘‘l_offset’’ and ‘‘l_len’’ define the region of the file locked by this holder.

LCK_DENIED_NOLOCKS
: Indicates that the procedure call failed because the server NLM could not allocate the resources needed to process the request.

LCK_DENIED_GRACE_PERIOD
: Indicates that the procedure call failed because the server host has recently been rebooted and the server NLM is re-establishing existing locks, and is not yet ready to accept normal service requests.

Name

NLM_LOCK — Establish a Lock

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;
};
```

RPC Procedure Description

```
nlm_res
NLM_LOCK(nlm_lockargs) = 2;
```

Description

This procedure attempts to establish a monitored lock described in ‘‘alock’’.

If ‘‘block’’ is *true*, then if the lock request cannot be granted immediately the server will return a status of ‘‘LCK_BLOCKED’’ for this procedure call. When the request can be granted, the server will make a call-back to the client with the NLM_GRANTED procedure call. If ‘‘block’’ is set to *false*, and the lock cannot be granted immediately, the procedure will return with a status of LCK_DENIED, and no NLM_GRANTED call-back will be made.

If ‘‘reclaim’’ is *true*, then the server will assume this is a request to re-establish a previous lock (for example, after the server has crashed and rebooted). During the grace period the server will only accept locks with ‘‘reclaim’’ set to *true*.

‘‘state’’ contains the state of the client’s NSM. This information is kept by the server implementation, so if the client crashes, the server can determine which locks to discard by checking this state against the state in the crash notification (SM_NOTIFY) sent by the NSM. See Chapter 9 on page 117.

Return Codes

When the procedure returns, ‘‘stat’’ will be set to one of the following values:

LCK_GRANTED Indicates that the procedure completed successfully, the lock was granted.

LCK_DENIED Indicates that the procedure failed because the request conflicted with existing lock reservations for the file.

LCK_DENIED_NOLOCKS
Indicates that the procedure failed because the server NLM could not allocate the resources needed to process the request.

LCK_BLOCKED Indicates the blocking request cannot be granted immediately. The server NLM will make a call-back to the client with an NLM_GRANTED procedure when the lock can be granted.

LCK_DENIED_GRACE_PERIOD
Indicates that the procedure failed because the server host has recently been rebooted and the server NLM is re-establishing existing locks, and is not yet ready to accept normal service requests.

Name

NLM_CANCEL — Cancel Lock

Call Arguments

```
struct nlm_cancargs {
    netobj cookie;
    bool block;
    bool exclusive;
    struct nlm_lock alock;
};
```

Return Arguments

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

RPC Procedure Description

```
nlm_res
NLM_CANCEL(nlm_cancargs) = 3;
```

Description

This procedure cancels an outstanding blocked lock request.

If the client made an NLM_LOCK procedure with ‘‘nlm_lockargs.block’’ set to *true*, and the procedure was blocked by the server (that is, the procedure returned a status of ‘‘LCK_BLOCKED’’), the client can choose to cancel this outstanding lock request by using this procedure.

The ‘‘block’’, ‘‘exclusive’’ and ‘‘alock’’ arguments must exactly match the corresponding arguments to the NLM_LOCK procedure.

Return Codes

When the procedure returns, ‘‘stat’’ will be set to one of the following values:

LCK_GRANTED
Indicates that the procedure completed successfully. The NLM may also return this code even if the ‘‘alock.oh’’ of the NLM_CANCEL procedure call does not match that of the outstanding lock request, or if there is no matching outstanding lock request.

LCK_DENIED
Indicates that the procedure failed possibly because there was no lock to cancel.

LCK_DENIED_GRACE_PERIOD
Indicates that the procedure failed because the server host has recently been rebooted and the server NLM is re-establishing existing locks, and is not yet ready to accept normal service requests.

Name

NLM_UNLOCK — Unlock File

Call Arguments

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

Return Arguments

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

RPC Procedure Description

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

Description

This routine will remove the lock specified by ‘‘alock’’. The valuse of the following fields in the ‘‘alock’’ structure must match the corresponding ‘‘alock’’ fields in the call that created the lock (NLM_LOCK, NLM_NM_LOCK or NLM_LOCK_MSG): caller_name, fh, oh, and uppid.

Return Codes

When the procedure returns, ‘‘stat’’ will be set to one of the following values:

LCK_GRANTED Indicates that the procedure completed successfully. The NLM may also return this code even if the ‘‘alock.oh’’ of the NLM_UNLOCK procedure call does not match the holder of the lock, or if there is no matching lock.

LCK_DENIED_GRACE_PERIOD
Indicates that the procedure failed because the server host has recently been rebooted and the server NLM is re-establishing existing locks, and is not yet ready to accept normal service requests.

Name

NLM_GRANTED — Lock Granted

Call Arguments

```
struct nlm_testargs {
    netobj cookie;
    bool exclusive;
    struct nlm_lock alock;
};
```

Return Arguments

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

RPC Procedure Description

```
nlm_res
NLM_GRANTED(nlm_testargs) = 5;
```

Description

This procedure is a call-back procedure from the server NLM running on the host where the file resides to the client.

Note: With this procedure the server is the caller and the client the recipient.

This procedure call is made by the NLM server on the host where the file resides and the procedure is executed, and the return value generated on the client that issued the lock request.

A client issuing an NLM_LOCK procedure that blocks will be returned a status of ‘‘LCK_BLOCKED’’. At a later point, when the lock is granted, the server will issue an NLM_GRANTED procedure call to the client to indicate the lock has been granted. ‘‘exclusive’’ and ‘‘alock’’ will be the values in the original NLM_LOCK procedure. The client must not depend on ‘‘cookie’’ being the same in the NLM_LOCK and NLM_GRANTED procedures.

Return Codes

When the server makes the NLM_GRANTED procedure the lock requested by the client has been granted. The client must now give a return code to the NLM_GRANTED procedure. The client must return the nlm_res structure to the server with ‘‘stat’’ set to one of the following values:

LCK_GRANTED Indicates that the procedure completed successfully.

LCK_DENIED Indicates that the procedure failed, possibly due to internal resource constraints.

LCK_DENIED_GRACE_PERIOD
Indicates that the procedure failed because the client host (the host making the lock request) has recently been rebooted and its NLM is re-establishing existing locks, and is not yet ready to accept normal service requests.

Name

NLM_TEST_MSG — Test Lock Message

Call Arguments

```
struct nlm_testargs {
    netobj cookie;
    bool exclusive;
    struct nlm_lock alock;
};
```

Return Arguments

None. Results are returned asynchronously via the NLM_TEST_RES procedure.

RPC Procedure Description

```
void
NLM_TEST_MSG(nlm_testargs) = 6;
```

Description

This procedure is one of the asynchronous RPCs. It performs the same function as the NLM_TEST procedure.

This procedure tests to see whether the monitored lock specified by ‘‘alock’’ is available to this client.

Name

NLM_LOCK_MSG — Lock Message

Call Arguments

```
struct nlm_lockargs {
    netobj cookie;
    bool block;             /*  Flag to indicate blocking behaviour. */
    bool exclusive;         /*  If exclusive access is required. */
    struct nlm_lock alock; /*  Actual lock data  */
    bool reclaim;           /*  used for recovering locks  */
    int state;              /*  specify local NSM state  */
};
```

Return Arguments

None. Results are returned asynchronously via the NLM_LOCK_RES procedure.

RPC Procedure Description

```
void
NLM_LOCK_MSG(nlm_lockargs) = 7;
```

Description

This procedure is one of the asynchronous RPCs. It performs the same function as the NLM_LOCK procedure.

This procedure attempts to establish a monitored lock described in ‘‘alock’’.

If ‘‘block’’ is *true*, and the lock request cannot be granted immediately, the server will return an NLM_LOCK_RES procedure with a status of ‘‘LCK_BLOCKED’’. When the request can be granted the server will make a call-back to the client with an NLM_GRANTED_MSG procedure. If ‘‘block’’ is set to *false*, and the lock cannot be granted immediately, the server will return an NLM_LOCK_RES procedure with a status of ‘‘LCK_DENIED’’, and no NLM_GRANTED_MSG call-back will be made.

If ‘‘reclaim’’ is *true*, then the server will assume this is a request to re-establish a previous lock (for example, after the server has crashed and rebooted). During the grace period, the server will only accept locks with ‘‘reclaim’’ set to *true*.

‘‘state’’ contains the state of the client’s NSM. This information is kept by the server implementation, so if the client crashes, the server can determine which locks to discard by checking this state against the state in the crash notification (SM_NOTIFY) sent by the NSM. See Chapter 9 on page 117.

The following sequence occurs if the lock request is blocked:

```
Client            Server

NLM_LOCK_MSG --->

        <--- NLM_LOCK_RES  (stat is set to LCK_BLOCKED)

            when the lock can be granted
        <--- NLM_GRANTED_MSG

NLM_GRANTED_RES --->
```

Name

NLM_CANCEL_MSG — Cancel a Lock

Call Arguments

```
struct nlm_cancargs {
    netobj cookie;
    bool block;
    bool exclusive;
    struct nlm_lock alock;
};
```

Return Arguments

None.

RPC Procedure Description

```
void
NLM_CANCEL_MSG(nlm_cancargs) = 8;
```

Description

This procedure is one of the asynchronous RPCs. It performs the same function as the NLM_CANCEL procedure.

If the client makes an NLM_LOCK_MSG procedure with ‘‘nlm_lockargs.block’’ set to *true*, and the procedure is blocked by the server (that is, the procedure returned a status of ‘‘LCK_BLOCKED’’), the client can choose to cancel this outstanding lock request by calling this procedure.

The ‘‘block’’, ‘‘exclusive’’ and ‘‘alock’’ arguments must exactly match the corresponding arguments to the NLM_LOCK_MSG procedure.

Name

NLM_UNLOCK_MSG — Unlock Message

Call Arguments

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

Return Arguments

None. Results are returned asynchronously via the NLM_CANCEL_RES procedure.

RPC Procedure Description

```
void
NLM_UNLOCK_MSG(nlm_unlockargs) = 9;
```

Description

This procedure is one of the asynchronous RPCs. It performs the same function as the NLM_UNLOCK procedure.

This routine will remove the lock specified by ‘‘alock’’.

The vaules of the following fields in the ‘‘alock’’ structure must match the corresponding ‘‘alock’’ fields in the call that created the lock (NLM_LOCK, NLM_NM_LOCK, or NLM_LOCK_MSG): caller_name, fh, oh, and uppid.

Name

NLM_GRANTED_MSG — Lock Grant Message

Call Arguments

```
struct nlm_testargs {
    netobj cookie;
    bool exclusive;
    struct nlm_lock alock;
};
```

Return Arguments

None. Results are returned asynchronously via the NLM_GRANTED_RES procedure.

RPC Procedure Description

```
void
NLM_GRANTED_MSG(nlm_testargs) = 10;
```

Description

This procedure is one of the asynchronous RPCs. It performs the same function as the NLM_GRANTED procedure.

This procedure is a call-back procedure from the server NLM, running on the host where the file resides, to the client. Note that with this procedure the server is the caller and the client the recipient. This procedure is called by the NLM server and the return value is generated by the client via the NLM_GRANTED_RES procedure.

A client issuing an NLM_LOCK_MSG procedure that blocks will be returned an NLM_LOCK_RES procedure with a status of ‘‘LCK_BLOCKED’’. At a later point, when the lock is granted on the server, the server will issue an NLM_GRANTED_MSG procedure to the client to indicate the lock has been granted. ‘‘exclusive’’ and ‘‘alock’’ will be the values in the original NLM_LOCK_MSG procedure. The client must not depend on ‘‘cookie’’ being the same in the NLM_LOCK_MSG and NLM_GRANTED_MSG procedures.

On reception of an NLM_GRANTED_MSG the client should generate an NLM_GRANTED_RES call to the server. See NLM_LOCK_MSG for more information.

Name

NLM_TEST_RES — Test Lock Result

Call Arguments

```
struct    nlm_testres {
   netobj       cookie;
   nlm_testrply test_stat;
};
```

Return Arguments

None.

RPC Procedure Description

```
void
NLM_TEST_RES(nlm_testres) = 11;
```

Description

This procedure is one of the asynchronous RPCs. The server calls this procedure to return results of the NLM_TEST_MSG procedure to the client (the host issuing the NLM_TEST_MSG call).

‘‘test_stat.stat’’ will be set to one of the following values:

LCK_GRANTED Indicates that the ‘‘NLM_TEST_MSG’’ procedure completed successfully. The server would be able to grant the lock in question.

LCK_DENIED Indicates that the ‘‘NLM_TEST_MSG’’ failed as it conflicted with existing lock reservations for the file. ‘‘test_stat.holder’’ describes the current holder of the lock as follows. The boolean ‘‘exclusive’’ indicates whether the lock is exclusively held by the current holder, or whether other locks are permitted. The integer ‘‘uppid’’ provides a unique per-process identifier for lock differentiation. The unsigned values ‘‘l_offset’’ and ‘‘l_len’’ define the region of the file locked by this holder.

LCK_DENIED_NOLOCKS
Indicates that the ‘‘NLM_TEST_MSG’’ failed because the server NLM could not allocate the resources needed to process the request.

LCK_DENIED_GRACE_PERIOD
Indicates that the ‘‘NLM_TEST_MSG’’ failed because the server host has recently been rebooted and its NLM is re-establishing existing locks, and is not yet ready to accept normal service requests.

Name

NLM_LOCK_RES — Establish a Lock Result

Call Arguments

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

Return Arguments

None.

RPC Procedure Description

```
void
NLM_LOCK_RES(nlm_res) = 11;
```

Description

This procedure is one of the asynchronous RPCs. The server calls this procedure to return results of the NLM_LOCK_MSG procedure to the client (the host issuing the NLM_LOCK_MSG call).

‘‘stat’’ will be set to one of the following values:

LCK_GRANTED Indicates that the procedure completed successfully; the lock was granted.

LCK_DENIED Indicates that the procedure failed because the request conflicted with existing lock reservations for the file.

LCK_DENIED_NOLOCKS
Indicates that the procedure failed because the server NLM could not allocate the resources needed to process the request.

LCK_BLOCKED Indicates that the blocking request cannot be granted immediately. The NLM on the server will make a call-back to the client with an NLM_GRANTED procedure when the lock can be granted.

LCK_DENIED_GRACE_PERIOD
Indicates that the procedure failed because the server host (the host making the ‘‘NLM_LOCK_RES’’ call) has recently been rebooted and its NLM is re-establishing existing locks, and is not yet ready to accept normal service requests.

Name

NLM_CANCEL_RES — Cancel Lock Result

Call Arguments

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

Return Arguments

None.

RPC Procedure Description

```
void
NLM_CANCEL_RES(nlm_res) = 13;
```

Description

This procedure is one of the asynchronous RPCs. The server calls this procedure to return results of the NLM_CANCEL_MSG procedure to the client (the host issuing the NLM_CANCEL_MSG call).

‘‘stat’’ will be set to one of the following values:

LCK_GRANTED Indicates that the procedure completed successfully.

LCK_DENIED Indicates that the procedure failed possibly because there was no lock to cancel.

LCK_DENIED_GRACE_PERIOD
Indicates that the procedure failed because the server host has recently been rebooted and its NLM is re-establishing existing locks, and is not yet ready to accept normal service requests.

Name

NLM_UNLOCK_RES — Unlock Result

Call Arguments

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

Return Arguments

None.

RPC Procedure Description

```
void
NLM_UNLOCK_RES(nlm_res) = 14;
```

Description

This procedure is one of the asynchronous RPCs. The server calls this procedure to return results of the NLM_UNLOCK_MSG procedure to the client (the host issuing the NLM_UNLOCK_MSG call).

‘‘stat’’ will be set to one of the following values:

LCK_GRANTED Indicates that the procedure completed successfully.

LCK_DENIED_GRACE_PERIOD
Indicates that the procedure failed because the server host has recently been rebooted and the server NLM is re-establishing existing locks, and is not yet ready to accept normal service requests.

Name

NLM_GRANTED_RES — Lock Granted Result

Call Arguments

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

Return Arguments

None.

RPC Procedure Description

```
void
NLM_GRANTED_RES(nlm_res) = 15;
```

Description

This procedure is one of the asynchronous RPCs. Unlike the other asynchronous calls it is called by the client (not the server). With this call, the client provides the return values for the NLM_GRANTED_MSG procedure, previously called by the server.

See NLM_LOCK_MSG for more information.

The client must send the nlm_res structure to the server with ‘‘stat’’ set to one of the following values:

LCK_GRANTED Indicates that the procedure completed successfully.

LCK_DENIED Indicates that the procedure failed, possibly due to the client not being able to match ‘‘alock’’ from the NLM_GRANTED_MSG call with any outstanding lock requests or lack of internal resources.

LCK_DENIED_GRACE_PERIOD
Indicates that the procedure failed because the client host (the host making the lock request) has recently been rebooted and its NLM is re-establishing existing locks and is not yet ready to accept normal service requests.

Name

NLM_SHARE — Share a File

Call Arguments

```
struct    nlm_shareargs {
    netobj    cookie;
    nlm_share share;
    bool      reclaim;
};
```

Return Arguments

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

RPC Procedure Description

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

Description

This procedure indicates that a client wishes to open a file using the DOS 3.1 and above file-sharing modes.

The file to be opened is ‘‘share.fh’’ for access ‘‘share.access’’ in sharing mode ‘‘share.mode’’. The server will examine any entry sharing reservations for this file to determine whether the share is permitted, that is, it does not conflict with the existing use of the file. If the share can be granted, a sharing reservation is established. If a conflict does exist, the request is rejected immediately. This procedure does not block; it is the responsibility of the client to retry any rejected requests.

If ‘‘reclaim’’ is *true*, then the server will assume this is a request to re-establish a previous share (for example, after the server has crashed and rebooted). During the grace period the server will only accept shares with ‘‘reclaim’’ set to *true*.

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 host crashes and restarts while the reservation is still in effect, the client must call the NLM_FREE_ALL procedure to release all sharing reservations.

Return Codes

When the procedure returns, ‘‘stat’’ will be set to one of the following values:

LCK_GRANTED Indicates that the procedure completed successfully.

LCK_DENIED Indicates that the procedure failed because the request conflicted with existing sharing reservations for the file.

LCK_DENIED_NOLOCKS
Indicates that the procedure failed because the server NLM could not allocate the resources needed to process the request.

LCK_DENIED_GRACE_PERIOD
Indicates that the procedure failed because the server host has recently been rebooted and the server NLM is re-establishing existing locks, and is not yet ready to accept normal service requests.

Name

NLM_UNSHARE — Unshare a File

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;
};
```

RPC Procedure Description

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

Description

This procedure informs the NLM that the client has closed the file ‘‘share.fh’’. The server will release the corresponding share reservation. The ‘‘reclaim’’ field is unused in this procedure and is ignored. It is included for symmetry with the NLM_SHARE procedure.

Return Codes

When the procedure returns, ‘‘stat’’ will be set to one of the following values:

LCK_GRANTED Indicates that the procedure completed successfully.

LCK_DENIED_GRACE_PERIOD
Indicates that the procedure failed because the server host has recently been rebooted and the server NLM is re-establishing existing locks, and is not yet ready to accept normal service requests.

Name

NLM_NM_LOCK — Non-monitored Lock

Call Arguments

```
struct    nlm_lockargs {
    netobj cookie;
    bool   block;          /*  Flag to indicate blocking behaviour. */
    bool   exclusive;      /*  If exclusive access is required. */
    struct nlm_lock alock; /*  Actual lock data  */
    bool   reclaim;        /*  used for recovering locks  */
    int    state;          /*  specify local NSM state  */
};
```

Return Arguments

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

RPC Procedure Description

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

Description

This procedure should only be called by clients that do not run the NSM, for example personal computer clients. If an NSM is run on the client, then the NLM_LOCK (monitored lock) procedure should be used. This procedure has the same functionality as the NLM_LOCK procedure except that there is no monitoring performed via the NSM.

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’’. ‘‘state’’ should be set to 0, ‘‘block’’ should be set to *false.* This procedure does not block; it is the responsibility of the client to retry any rejected requests.

Locks created with this procedure should be released with the NLM_UNLOCK procedure.

The phrase ‘‘non-monitored’’ refers to the fact that the NLM will make no attempt to verify that the lock is still valid; if the client host crashes and restarts while the lock is still in effect, it must call the NLM_FREE_ALL procedure to release all locks held.

Return Codes

When the procedure returns, ‘‘stat’’ will be set to the following values:

LCK_GRANTED Indicates that the procedure completed successfully.

LCK_DENIED Indicates that the procedure failed because the request conflicted with existing locks for the file.

LCK_DENIED_NOLOCKS
Indicates that the procedure failed because the server NLM could not allocate the resources needed to process the request.

LCK_DENIED_GRACE_PERIOD

Indicates that the procedure failed because the server host has recently been rebooted and the server NLM is re-establishing existing locks, and is not yet ready to accept normal service requests.

Name

NLM_FREE_ALL — Free All

Call Arguments

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

Return Arguments

```
void
```

RPC Procedure Description

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

Description

This procedure informs the server that the client ‘‘name’’ has been rebooted. The server will discard all file-sharing reservations and file locks currently being held on behalf of the client. The ‘‘state’’ field is unused and should be set to 0.

This procedure is primarily called by clients that do not implement the NSM protocol and therefore use the non-monitored lock procedure, NLM_NM_LOCK, or the non-monitored file-sharing procedure, NLM_SHARE.

Return Codes

None.

Chapter 11

Network Status Monitor Protocol

11.1 Introduction

This chapter describes the Network Status Monitor (NSM) protocol which is related to, but separate from, the Network Lock Manager (NLM) protocol. The NSM protocol is not specified as a part of the NLM protocol to allow the implementor flexibility and to facilitate the development of new mechanisms without requiring the revision of related protocols.

The NLM uses the NSM protocol to enable it to recover from crashes of either the client or server host. To provide this functionality the NSM and NLM protocols on both the client and server hosts must cooperate.

The NSM is a service that provides applications with information on the status of network hosts. Each NSM keeps track of its own ‘‘state’’ and notifies any interested party of a change in this state to any other NSM upon request. The state is merely a number which increases monotonically each time the state of the host changes; an even number indicates the host is down, while an odd number indicates the host is up.

Applications register the network hosts they are interested in with the local NSM. If one of these hosts crashes, the NSM on the crashed host, after a reboot, will notify the NSM on the local host that the state changed. The local NSM can then, in turn, notify the interested application of this state change.

The NSM is used heavily by the Network Lock Manager (NLM). The local NLM registers with the local NSM all server hosts on which the NLM has currently active locks. In parallel, the NLM on the remote (server) host registers all of its client hosts with its local NSM. If the server host crashes and reboots, the server NSM will inform the NSM on the client hosts of this event. The local NLM can then take steps to re-establish the locks when the server is rebooted. Low-end systems that do not run an NSM, due to memory or speed constraints, are restricted to using non-monitored locks. See Chapter 9 on page 117 and Chapter 10 on page 127.

11.2 RPC Information

Authentication

The NSM service uses AUTH_UNIX style of authentication.

Transport Protocols

The NSM Protocol is required to support the UDP/IP transport protocol to allow the NLM to operate. However, implementors may also choose to support the TCP/IP transport protocol.

Port Number

Consult the server's port mapper, described in Chapter 6 on page 61, to find the port number on which the NSM service is registered.

11.2.1 Sizes of XDR Structures

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

```
/*
 * This defines the maximum length of the string
 * identifying the caller.
 */
const SM_MAXSTRLEN = 1024
```

11.2.2 Basic Data Types

This section presents the data types used by the NSM.

sm_name

```
struct sm_name {
    string mon_name<SM_MAXSTRLEN>;
};
```

sm_name is the name of the host to be monitored by the NSM. It is the parameter to the SM_STAT call.

Implementations and applications must be able to handle host names as 8-bit transparent data (allowing use of arbitrary character set encodings). For maximum portability and interworking, it is recommended that applications and users define host names containing only the characters of the Portable Filename Character Set defined in ISO/IEC 9945-1:1990. (This also applies to the "my_id.my_name" fields in the call arguments of the SM_MON, SM_UNMON and SM_UNMON_ALL procedures.)

res

```
res {
    STAT_SUCC = 0,   /*  NSM agrees to monitor.  */
    STAT_FAIL = 1    /*  NSM cannot monitor.  */
};
```

res is returned when the NSM is asked whether it can monitor the given host or if it has been successful in monitoring the given host.

sm_stat_res

```
struct sm_stat_res {
    res    res_stat;
    int    state;
};
```

sm_stat_res is the return value from SM_STAT and SM_MON procedures. It includes the return status of the call and the state number of the local host.

sm_stat

```
struct sm_stat {
    int    state;    /*  state number of NSM  */
};
```

The state number of the NSM monotonically increases each time state of the host changes; an even number indicates that the host is down, while an odd number indicates that it is up.

my_id

```
struct my_id {
    string my_name<SM_MAXSTRLEN>;  /*  hostname  */
    int    my_prog;                /*  RPC program number  */
    int    my_vers;                /*  program version number  */
    int    my_proc;                /*  procedure number  */
};
```

my_id contains the RPC call-back information. See SM_NOTIFY for more information.

mon_id

```
struct mon_id {
    string mon_name<SM_MAXSTRLEN>; /* name of the host to be monitored */
    struct my_id my_id;
};
```

Contains the name of the host to be monitored and RPC call-back information. See SM_NOTIFY for more information.

mon

```
struct mon {
    struct mon_id mon_id;
    opaque    priv[16];        /*  private information  */
};
```

Parameter to SM_MON call. ‘‘priv’’ is information provided by the client that is returned on notification of a server state change (crash and reboot).

stat_chge

```
struct stat_chge {
    string    mon_name;
    int    state;
};
```

This is the parameter to the SM_NOTIFY call. It contains the name of the host that had a state change (that is, crashed and recovered) and its new state number.

11.3 NSM Procedures

The following reference pages define the RPC procedures supplied by an NSM server.

```
/*
 *  Protocol description for the NSM program.
 */

program SM_PROG {
    version SM_VERS {
        void SM_NULL(void) = 0;
        struct sm_stat_res SM_STAT(struct sm_name) = 1;
        struct sm_stat_res SM_MON(struct mon) = 2;
        struct sm_stat SM_UNMON(struct mon_id) = 3;
        struct sm_stat SM_UNMON_ALL(struct my_id) = 4;
        void SM_SIMU_CRASH(void) = 5;
        void SM_NOTIFY(struct stat_chg) = 6;
    } = 1;
} = 100024;
```

Name

SM_NULL — Do Nothing

RPC Data Descriptions

Call Arguments

None.

Return Arguments

None.

RPC Procedure Description

```
void
SM_NULL(void) = 0;
```

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.

Return Codes

None.

Name

SM_STAT — Check Status

RPC Data Descriptions

Call Arguments

```
struct      sm_name {
    string  mon_name<SM_MAXSTRLEN>;
};
```

Return Arguments

```
struct    sm_stat_res {
    res   res_stat;
    int   state;
};
```

RPC Procedure Description

```
sm_stat_res
SM_STAT(struct sm_name) = 1;
```

Description

This procedure tests to see whether the NSM agrees to monitor the given host.

Implementations should not rely on this procedure being operative. In many current implementations of the NSM it will always return a ‘‘STAT_FAIL’’ status.

Return Codes

When the procedure returns, ‘‘sm_stat_res.sm_stat’’ will be set to one of the following values:

STAT_SUCC The NSM will monitor the given host. ‘‘sm_stat_res.state’’ contains the state of the NSM.

STAT_FAIL The NSM is not be able to monitor the host.

Name

SM_MON — Monitor Host

RPC Data Descriptions

Call Arguments

```
struct     my_id {
    string my_name<SM_MAXSTRLEN>;  /*  hostname  */
    int    my_prog;                /*  RPC program number */
    int    my_vers;                /*  program version number */
    int    my_proc;                /*  procedure number  */
};

struct mon_id {
    string mon_name<SM_MAXSTRLEN>;
    struct    my_id my_id;
};

struct mon {
    struct mon_id mon_id;
    opaque    priv[16];
};
```

Return Arguments

```
struct sm_stat_res {
    res    res_stat;
    int    state;
};
```

RPC Procedure Description

```
sm_stat_res
SM_MON(struct mon) = 2;
```

Description

This procedure initiates the monitoring of the given host. This call enables the NSM to respond to notification of change of state calls (SM_NOTIFY) for the host specified in ‘‘mon_id.mon_name’’, and to notify that host, via the SM_NOTIFY call, when its state (that is, crash and reboot) changes.

‘‘mon_id.mon_name’’ specifies the host to be monitored. ‘‘mon_id.my_id’’ specifies the hostname, RPC program number, version and procedure number in the local application, for example, the NLM, to be called when the NSM receives notification via the SM_NOTIFY call, that the state of host ‘‘mon_id.mon_name’’ has changed. ‘‘priv’’ may contain any private information required by the SM_MON call. This information will be supplied in the SM_NOTIFY call. See SM_NOTIFY for more details.

When an NSM receives an SM_MON call it must save the name of the host, ‘‘mon_id.mon_name’’ in a notify list on stable storage. If the host running the NSM crashes, on reboot it must send out an SM_NOTIFY call to each host in the notify list.

When an NSM receives an SM_NOTIFY call from remote NSM, it must search the notify list for the host specified in the SM_NOTIFY call, if it is found the RPC specified in ‘‘mon_id.my_id’’ is made.

Return Codes

When the procedure returns, ‘‘res_stat’’ will be set to one of the following values:

STAT_SUCC Indicates the procedure completed successfully. The server will be able to monitor the host requested. ‘‘state’’ will be contain the state of the remote NSM.

STAT_FAIL Indicates the procedure failed.

Name

SM_UNMON — Unmonitor Host

RPC Data Descriptions

Call Arguments

```
struct mon_id {
    string mon_name<SM_MAXSTRLEN>;
    struct my_id my_id;
};
```

Return Arguments

```
struct sm_stat {
    int    state;
};
```

RPC Procedure Description

```
sm_stat
SM_UNMON(struct mon_id) = 3;
```

Description

This procedure stops monitoring the host ‘‘mon_name’’. The information in ‘‘my_id’’ must exactly match the information given in the corresponding SM_MON call.

Return Codes

When the procedure returns, ‘‘state’’ . ‘‘state’’ is set to the state of the local NSM.

Name

SM_UNMON_ALL — Unmonitor All Hosts

RPC Data Descriptions

Call Arguments

```
struct my_id {
    string my_name<SM_MAXSTRLEN>;
    int    my_prog;
    int    my_vers;
    int    my_proc;
};
```

Return Arguments

```
struct sm_stat {
    int    state;
};
```

RPC Procedure Description

```
sm_stat
SM_UNMON_ALL(struct my_id) = 4;
```

Description

This procedure stops monitoring all hosts for which monitoring was requested.

Return Codes

When the procedure returns, ‘‘state’’ will be the state of the local NSM.

Name

SM_SIMU_CRASH — Simulate a Crash

RPC Data Descriptions

Call Arguments

None.

Return Arguments

None.

RPC Procedure Description

```
void
SM_SIMU_CRASH(void) = 5;
```

Description

This procedure simulates a crash. The NSM releases all its current state information and reinitialises itself, incrementing its state variable. It reads through its notify list (see SM_MON) and informs the NSM on all hosts on the list that the state of this host has changed, via the SM_NOTIFY procedure.

Return Codes

None.

Name

SM_NOTIFY — Notify

RPC Data Descriptions

Call Arguments

```
struct stat_chge {
    string mon_name;
    int    state;
};
```

Return Arguments

None.

RPC Procedure Description

```
void
SM_NOTIFY(struct stat_chge) = 6;
```

Description

If a host has a state change, either a crash and reboot or the NSM has processed an SM_SIMU_CRASH call, the local NSM must notify each host on its notify list (see SM_MON) of the change in state.

‘‘mon_name’’ is the name of the host that had the state change. ‘‘state’’ is the new state for the host.

When an NSM receives the SM_NOTIFY call it must search its notify list for ‘‘mon_name’’. The host ‘‘mon_name’’ will be found in the notify list if an SM_MON call was made to the NSM to register the host. The NSM must call the RPC program, version, procedure number on the hostname supplied in the ‘‘my_id’’ field of the SM_MON parameter. This RPC will be called with the following parameter:

```
struct    status {
    string mon_name<SM_MAXSTRLEN>;
    int    state;
    opaque priv[16];        /*  for private information  */
};
```

Where ‘‘mon_name’’ and ‘‘state’’ are copied from the SM_NOTIFY parameters. ‘‘priv’’ is the information supplied in the corresponding field of the SM_MON call that registered the host ‘‘mon_name’’.

Return Codes

None.

Chapter 12

XNFS: Protocol Specification, Version 3

This chapter specifies an additional protocol for the Network File System, the Version 3 protocol, which must be supported in addition to the Version 2 protocol specified in Chapter 7. This chapter is written with the assumption that the reader is familiar with the introductory material in Chapter 7.

12.1 Summary of Version 3 Protocol Changes

This section provides an informative summary of changes to the NFS protocol from Version 2 to Version 3. All normative aspects of the protocol are described later in this document.

The *ROOT* and *WRITECACHE* procedures have been removed. A *MKNOD* procedure has been defined to allow the creation of special files, eliminating the overloading of *CREATE*. Caching on the client is not defined nor dictated by the NFS Version 3 protocol, but additional information and hints have been added to the protocol to allow clients that implement caching to manage their caches more effectively. Procedures that affect the attributes of a file or directory may now return the new attributes after the operation has completed to optimise out a subsequent *GETATTR* used in validating attribute caches. In addition, operations that modify the directory in which the target object resides return the old and new attributes of the directory to allow clients to implement more intelligent cache invalidation procedures. The *ACCESS* procedure provides access permission checking on the server, the *FSSTAT* procedure returns dynamic information about a file system, the *FSINFO* procedure returns static information about a file system and server, the *READDIRPLUS* procedure returns file handles and attributes in addition to directory entries, and the *PATHCONF* procedure returns XPG4 *pathconf*() information about a file.

The following is a list of the important changes between the NFS Version 2 protocol and the NFS Version 3 protocol.

File handle size

The file handle has been increased to a variable-length array of 64 bytes maximum from a fixed array of 32 bytes. This addresses some known requirements for a slightly larger file handle size. The file handle was converted from fixed length to variable length to reduce local storage and network bandwidth requirements for systems that do not utilise the full 64 bytes of length.

Maximum data sizes

The maximum size of a data transfer used in the *READ* and *WRITE* procedures is now set by values in the *FSINFO* return structure. In addition, preferred transfer sizes are returned by *FSINFO*. The protocol does not place any artificial limits on the maximum transfer sizes. Filenames and pathnames are now specified as strings of variable length. The actual length restrictions are determined by the client and server implementations as appropriate. The protocol does not place any artificial limits on the length. The *NFS3ERR_NAMETOOLONG* error is provided to allow the server to return an indication to the client that it received a pathname that was too long for it to handle.

Error return

Error returns in some instances now return data (for example, attributes). The **nfsstat3** structure now defines the full set of errors that can be returned by a server. No other values are allowed.

File type The file type now includes *NF3CHR* and *NF3BLK* for special files. Attributes for these types include subfields for major and minor device numbers traditionally found on UNIX systems. *NF3SOCK* and *NF3FIFO* are now defined for sockets and FIFOs in the file system.

File attributes

The *blocksize* (the size in bytes of a block in the file) field has been removed. The *mode* field no longer contains file type information. The *size* and *fileid* fields have been widened to 8 byte unsigned integers from 4 byte integers. Major and minor device information is now presented in a distinct structure. The *blocks* field name has been changed to *used* and now contains the total number of bytes used by the file. It is also an 8 byte unsigned integer.

Set file attributes

In the NFS Version 2 protocol, the attributes that can be set were represented by a subset of the file attributes structure; the client indicated those attributes that were not to be modified by setting the corresponding field to -1, overloading some unsigned fields. The set file attributes structure now uses a discriminated union for each field to tell whether or how to set that field. The *atime* and *mtime* fields can be set to either the server's current time or a time supplied by the client.

LOOKUP The *LOOKUP* return structure now includes the attributes for the directory searched.

ACCESS An *ACCESS* procedure has been added to allow an explicit over-the-wire permissions check. This addresses known problems with the super-user ID mapping feature in many server implementations (where, due to mapping of root user, unexpected permission denied errors could occur while reading from or writing to a file). This also removes the assumption that was made in the NFS Version 2 protocol that access to files was based solely on XPG-style mode bits.

READ The reply structure includes a Boolean that is *TRUE* if the end-of-file was encountered during the *READ*. This allows the client to correctly detect end-of-file.

WRITE The *beginoffset* and *totalcount* fields were removed from the *WRITE* arguments. The reply now includes a count so that the server can write less than the requested amount of data, if required. An indicator was added to the arguments to instruct the server as to the level of cache synchronisation that is required by the client.

CREATE An exclusive flag and a create verifier was added for the exclusive creation of regular files.

MKNOD This procedure was added to support the creation of special files. This avoids overloading fields of *CREATE* as was done in some NFS Version 2 protocol implementations.

READDIR The *READDIR* arguments now include a verifier to allow the server to validate the cookie. The cookie is now a 64 bit unsigned integer instead of the 4 byte array that was used in the NFS Version 2 protocol. This will help to reduce interoperability problems.

READDIRPLUS

This procedure was added to return file handles and attributes in an extended directory list.

FSINFO *FSINFO* was added to provide nonvolatile information about a file system. The reply includes preferred and maximum read transfer size, preferred and maximum write transfer size, and flags stating whether links or symbolic links are supported. Also returned are preferred transfer size for *READDIR* procedure replies, server time granularity and whether times can be set in a *SETATTR* request.

FSSTAT *FSSTAT* was added to provide volatile information about a file system, for use by utilities such as *df*. The reply includes the total size and free space in the file system specified in bytes, the total number of files and number of free file slots in the file system, and an estimate of time between file system modifications (for use in cache consistency checking algorithms).

COMMIT The *COMMIT* procedure provides the synchronisation mechanism to be used with asynchronous *WRITE* operations.

12.2 RPC Information

Authentication

The NFS service uses *AUTH_NONE* in the *NULL* procedure. *AUTH_UNIX*, *AUTH_DES* or *AUTH_KERB* are used for all other procedures.

Transport Protocols

NFS implementations exist for both UDP/IP and TCP/IP protocols.

Port Number

The NFS Version 3 protocol uses the UDP port number 2049 decimal, the same port as the Version 2 protocol. Since this is not an officially assigned port, it is possible that it may change in the future. For maximum interoperability it is recommended (but not required) that NFS servers use UDP port 2049 if possible, and that NFS clients use the portmap mechanism (see Chapter 6) to locate the NFS program on a server.

WebNFS servers must use UDP and TCP port 2049.

12.2.1 Sizes of XDR Structures

The following table specifies the sizes, given in decimal bytes, of various XDR structures used in the protocol:

Structure	Size	Description
NFS3_FHSIZE	64	The maximum size in bytes of the opaque file handle
NFS3_COOKIEVERFSIZE	8	The size in bytes of the opaque cookie verifier passed by *READDIR* and *READDIRPLUS*
NFS3_CREATEVERFSIZE	8	The size in bytes of the opaque verifier used for exclusive *CREATE*
NFS3_WRITEVERFSIZE	8	The size in bytes of the opaque verifier used for asynchronous *WRITE*

12.2.2 Basic Data Types

The following XDR definitions are basic definitions that are used in other structures.

uint64

```
typedef unsigned hyper uint64;
```

int64

```
typedef hyper int64;
```

uint32

```
typedef unsigned long uint32;
```

int32

```
typedef long int32;
```

filename3

```
typedef string filename3<>;
```

nfspath3

```
typedef string nfspath3<>;
```

fileid3

```
typedef uint64 fileid3;
```

cookie3

```
typedef uint64 cookie3;
```

cookieverf3

```
typedef opaque cookieverf3[NFS3_COOKIEVERFSIZE];
```

createverf3

```
typedef opaque createverf3[NFS3_CREATEVERFSIZE];
```

writeverf3

```
typedef opaque writeverf3[NFS3_WRITEVERFSIZE];
```

uid3

```
typedef uint32 uid3;
```

gid3

```
typedef uint32 gid3;
```

size3

```
typedef uint64 size3;
```

offset3

```
typedef uint64 offset3;
```

mode3

```
typedef uint32 mode3;
```

count3

```
typedef uint32 count3;
```

nfsstat3

```
enum nfsstat3 {
    NFS3_OK             = 0,
    NFS3ERR_PERM        = 1,
    NFS3ERR_NOENT       = 2,
    NFS3ERR_IO          = 5,
    NFS3ERR_NXIO        = 6,
    NFS3ERR_ACCES       = 13,
    NFS3ERR_EXIST       = 17,
    NFS3ERR_XDEV        = 18,
    NFS3ERR_NODEV       = 19,
    NFS3ERR_NOTDIR      = 20,
    NFS3ERR_ISDIR       = 21,
    NFS3ERR_INVAL       = 22,
    NFS3ERR_FBIG        = 27,
    NFS3ERR_NOSPC       = 28,
    NFS3ERR_ROFS        = 30,
    NFS3ERR_MLINK       = 31,
    NFS3ERR_NAMETOOLONG = 63,
    NFS3ERR_NOTEMPTY    = 66,
    NFS3ERR_DQUOT       = 69,
    NFS3ERR_STALE       = 70,
    NFS3ERR_REMOTE      = 71,
    NFS3ERR_BADHANDLE   = 10001,
    NFS3ERR_NOT_SYNC    = 10002,
    NFS3ERR_BAD_COOKIE  = 10003,
    NFS3ERR_NOTSUPP     = 10004,
    NFS3ERR_TOOSMALL    = 10005,
    NFS3ERR_SERVERFAULT = 10006,
    NFS3ERR_BADTYPE     = 10007,
    NFS3ERR_JUKEBOX     = 10008
};
```

The **nfsstat3** type is returned with every procedure's results except for the *NULL* procedure. A value of *NFS3_OK* indicates that the call completed successfully. Any other value indicates that some error occurred on the call, as identified by the error code. No other values may be returned by a server. Servers are expected to make a best effort mapping of error conditions to the set of error codes defined. In addition, no error precedences are specified by this document. Error precedences determine the error value that should be returned when more than one error applies in a given situation. The error precedence will be determined by the individual server implementation. If the client requires specific error precedences, it should check for the specific errors for itself.

A description of each defined error follows.

NFS3_OK Indicates the call completed successfully.

NFS3ERR_PERM Not owner. The caller does not have the correct ownership to perform the requested operation.

NFS3ERR_NOENT No such file or directory. The file or directory name specified does not exist.

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_NXIO No such device or address.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_EXIST File exists. The file specified already exists.

NFS3ERR_XDEV The caller attempted to do a cross-device hard link.

NFS3ERR_NODEV No such device.

NFS3ERR_NOTDIR Not a directory. The caller specified a non-directory in a directory operation.

NFS3ERR_ISDIR Is a directory. The caller specified a directory in a non-directory operation.

NFS3ERR_INVAL Invalid argument or unsupported argument for an operation. Two examples are attempting a *READLINK* on an object other than a symbolic link or attempting to *SETATTR* a time field on a server that does not support this operation.

NFS3ERR_FBIG File too large. The operation would have caused a file to grow beyond the server's limit.

NFS3ERR_NOSPC No space left on device. The operation would have caused the server's file system to exceed its limit.

NFS3ERR_ROFS Read-only file system. A modifying operation was attempted on a read-only file system.

NFS3ERR_MLINK Too many hard links.

NFS3ERR_NAMETOOLONG
The filename in an operation was too long.

NFS3ERR_NOTEMPTY
An attempt was made to remove a directory that was not empty.

NFS3ERR_DQUOT Resource (quota) hard limit exceeded. The user's resource limit on the server has been exceeded.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_REMOTE Too many levels of remote in path. The file handle given in the arguments referred to a file on a non-local file system on the server.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_NOT_SYNC
An update synchronisation mismatch was detected during a *SETATTR* operation.

NFS3ERR_BAD_COOKIE
A *READDIR* or *READDIRPLUS* cookie is stale.

NFS3ERR_NOTSUPP
The operation is not supported.

NFS3ERR_TOOSMALL
The buffer or request is too small.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

NFS3ERR_BADTYPE
An attempt was made to create an object of a type not supported by the server.

NFS3ERR_JUKEBOX
The server initiated the request, but was not able to complete it in a timely fashion. The client should wait and then try the request with a new RPC transaction ID. For example, this error should be returned from a server that supports hierarchical storage and receives a request to process a file that has been migrated. In this case, the server should start the immigration process and respond to client with this error.

ftype3

```
enum ftype3 {
    NF3REG  = 1,
    NF3DIR  = 2,
    NF3BLK  = 3,
    NF3CHR  = 4,
    NF3LNK  = 5,
    NF3SOCK = 6,
    NF3FIFO = 7
};
```

The enumeration **ftype3** gives the type of a file, as follows:

NF3REG	Regular file
NF3DIR	Directory
NF3BLK	Block special device file
NF3CHR	Character special device file
NF3LNK	Symbolic link
NF3SOCK	Socket
NF3FIFO	Named pipe

specdata3

```
struct specdata3 {
    uint32 specdata1;
    uint32 specdata2;
};
```

The interpretation of the two words depends on the type of file system object. For a block special (*NF3BLK*) or character special (*NF3CHR*) file, *specdata1* and *specdata2* are the major and minor device numbers, respectively. For all other file types, these two elements should either be set to zero or the values should be agreed upon by the client and server. If the client and server do not agree upon the values, the client should treat these fields as if they are set to zero. This data field is returned as part of the **fattr3** structure and so is available from all replies returning attributes. Since these fields are otherwise unused for objects that are not devices, out of band information can be passed from the server to the client. However, both the server and the client must agree on the values passed.

nfs_fh3

```
struct nfs_fh3 {
    opaque data<NFS3_FHSIZE>;
};
```

The **nfs_fh3** structure is the variable-length opaque object returned by the server on *LOOKUP*, *CREATE*, *SYMLINK*, *MKNOD*, *LINK* or *READDIRPLUS* operations, which is used by the client on subsequent operations to reference the file. The file handle contains all the information the server needs to distinguish an individual file. To the client, the file handle is opaque. The client stores file handles for use in a later request and can compare two file handles from the same server for equality by doing a byte-by-byte comparison, but cannot otherwise interpret the contents of file handles. If two file handles from the same server are equal, they must refer to the same file, but if they are not equal, no conclusions can be drawn. Servers should try to maintain a one-to-one correspondence between file handles and files, but this is not required. Clients should use file handle comparisons only to improve performance, not for correct behaviour.

Servers can revoke the access provided by a file handle at any time. If the file handle passed in a call refers to a file system object that no longer exists on the server or access for that file handle has been revoked, the *NFS3ERR_STALE* error should be returned.

A filehandle with a length of zero is called the *public* filehandle. It is used by WebNFS clients to identify an associated public directory on the server.

nfstime3

```
struct nfstime3 {
    uint32 seconds;
    uint32 nseconds;
};
```

The **nfstime3** structure gives the number of seconds and nanoseconds since midnight January 1, 1970 Greenwich Mean Time. It is used to pass time and date information. The times associated with files are all server times except in the case of a *SETATTR* operation where the client can explicitly set the file time. A server converts to and from local time when processing time values, preserving as much accuracy as possible. If the precision of timestamps stored for a file is less than that defined by the NFS Version 3 protocol, loss of precision can occur. An adjunct time maintenance protocol is recommended to reduce client and server time skew.

fattr3

```
struct fattr3 {
    ftype3 type;
    mode3 mode;
    uint32 nlink;
    uid3 uid;
    gid3 gid;
    size3 size;
    size3 used;
    specdata3 rdev;
    uint64 fsid;
    fileid3 fileid;
    nfstime3 atime;
    nfstime3 mtime;
    nfstime3 ctime;
};
```

The **fattr3** structure defines the attributes of a file system object. It is returned by most operations on an object; in the case of operations that affect two objects (for example, a *MKDIR* that modifies the target directory attributes and defines new attributes for the newly created directory), the attributes for both may be returned. In some cases, the attributes are returned in the structure, **wcc_data**, which is defined below; in other cases the attributes are returned alone.

The **fattr3** structure contains the basic attributes of a file. All servers must support this set of attributes even if they have to simulate some of the fields.

type The type of the file.

mode The protection mode bits.

nlink The number of hard links to the file; that is, the number of different names for the same file.

uid The user ID of the owner of the file.

gid The group ID of the group of the file.

size The size of the file in bytes.

used The number of bytes of disk space that the file actually uses (which can be smaller than the size because the file may have holes or it may be larger due to fragmentation).

rdev The device file, if the file type is *NF3CHR* or *NF3BLK*; see **specdata3**.

fsid The file system identifier for the file system.

fileid A number that uniquely identifies the file within its file system (on traditional UNIX systems, this would be the *i-number*).

atime The time when the file data was last accessed.

mtime The time when the file data was last modified.

ctime The time when the attributes of the file were last changed. Writing to the file changes the *ctime* in addition to the *mtime*.

The mode bits are defined as follows:

Bit	Description
0x00800	Set user ID on execution.
0x00400	Set group ID on execution.
0x00200	Save swapped text (not defined in XPG4).
0x00100	Read permission for owner.
0x00080	Write permission for owner.
0x00040	Execute permission for owner on a file. Or lookup (search) permission for owner in directory.
0x00020	Read permission for group.
0x00010	Write permission for group.
0x00008	Execute permission for group on a file. Or lookup (search) permission for group in directory.
0x00004	Read permission for others.
0x00002	Write permission for others.
0x00001	Execute permission for others on a file. Or lookup (search) permission for others in directory.

post_op_attr

```
union post_op_attr switch (bool attributes_follow){
    case TRUE:
        fattr3 attributes;
    case FALSE:
        void;
};
```

The **post_op_attr** structure is used for returning attributes in those operations that are not directly involved with manipulating attributes. One of the principles of this revision of the NFS protocol is to return the real value from the indicated operation and not an error from an incidental operation. The **post_op_attr** structure was designed to allow the server to recover from errors encountered while getting attributes.

This appears to make returning attributes optional. However, server implementors are strongly encouraged to make best effort to return attributes whenever possible, even when returning an error.

wcc_attr

```
struct wcc_attr {
    size3 size;
    nfstime3 mtime;
    nfstime3 ctime;
};
```

The **wcc_attr** structure is the subset of pre-operation attributes needed to improve support for the weak cache consistency semantics. The *size* argument is the file size in bytes of the object before the operation. The *mtime* argument is the time of last modification of the object before the operation. The *ctime* argument is the time of last change to the attributes of the object before the operation.

The use of *mtime* by clients to detect changes to file system objects residing on a server is dependent on the granularity of the time base on the server.

pre_op_attr

```
union pre_op_attr switch (bool attributes_follow){
    case TRUE:
        wcc_attr attributes;
    case FALSE:
        void;
};
```

wcc_data

```
struct wcc_data {
    pre_op_attr before;
    post_op_attr after;
};
```

When a client performs an operation that modifies the state of a file or directory on the server, it cannot immediately determine from the post-operation attributes whether the operation just performed was the only operation on the object since the last time the client received the attributes for the object. This is important, since if an intervening operation has changed the object, the client will need to invalidate any cached data for the object (except for the data that it just wrote).

To deal with this, the notion of *weak cache consistency data* (**wcc_data**) is introduced. A **wcc_data** structure consists of certain key fields from the object attributes before the operation, together with the object attributes after the operation. This information allows the client to manage its cache more accurately than in NFS Version 2 protocol implementations. The term *weak cache consistency* emphasizes the fact that this mechanism does not provide the strict server-client consistency that a cache consistency protocol would provide.

In order to support the weak cache consistency model, the server must be able to get the pre-operation attributes of the object, perform the intended modify operation, and then get the post-operation attributes atomically. If there is a window for the object to get modified between the operation and either of the get attributes operations, then the client will not be able to determine whether it was the only entity to modify the object. Some information will have been lost, thus weakening the weak cache consistency guarantees.

post_op_fh3

```
union post_op_fh3 switch (bool handle_follows){
    case TRUE:
        nfs_fh3 handle;
    case FALSE:
        void;
};
```

One of the principles of this revision of the NFS protocol is to return the real value from the indicated operation and not an error from an incidental operation. The **post_op_fh3** structure was designed to allow the server to recover from errors encountered while constructing a file handle.

This is the structure used to return a file handle from the *CREATE*, *MKDIR*, *SYMLINK*, *MKNOD* and *READDIRPLUS* requests. In each case, the client can get the file handle by issuing a

LOOKUP request after a successful return from one of the listed operations. Returning the file handle is an optimisation so that the client is not forced to issue a *LOOKUP* request immediately to get the file handle.

sattr3

```
enum time_how {
    DONT_CHANGE        = 0,
    SET_TO_SERVER_TIME = 1,
    SET_TO_CLIENT_TIME = 2
};

union set_mode3 switch (bool set_it) {
    case TRUE:
        mode3 mode;
    default:
        void;
};

union set_uid3 switch (bool set_it) {
    case TRUE:
        uid3 uid;
    default:
        void;
};

union set_gid3 switch (bool set_it) {
    case TRUE:
        gid3 gid;
    default:
        void;
};

union set_size3 switch (bool set_it) {
    case TRUE:
        size3 size;
    default:
        void;
};

union set_atime switch (time_how set_it) {
    case SET_TO_CLIENT_TIME:
        nfstime3 atime;
    default:
        void;
};

union set_mtime switch (time_how set_it) {
    case SET_TO_CLIENT_TIME:
        nfstime3 mtime;
    default:
        void;
};
```

```
struct sattr3 {
    set_mode3 mode;
    set_uid3 uid;
    set_gid3 gid;
    set_size3 size;
    set_atime atime;
    set_mtime mtime;
};
```

The **sattr3** structure contains the file attributes that can be set from the client. The fields are the same as the similarly named fields in the **fattr3** structure. In the NFS Version 3 protocol, the attributes that can be set are described by a structure containing a set of discriminated unions. Each union indicates whether the corresponding attribute is to be updated, and if so, how.

There are two forms of discriminated unions used. In setting the *mode*, *uid*, *gid* or *size*, the discriminated union is switched on a Boolean, *set_it*; if it is *TRUE*, a value of the appropriate type is then encoded.

In setting the *atime* or *mtime*, the union is switched on an enumeration type, *set_it*. If *set_it* has the value *DONT_CHANGE*, the corresponding attribute is unchanged. If it has the value *SET_TO_SERVER_TIME*, the corresponding attribute is set by the server to its local time; no data is provided by the client. Finally, if *set_it* has the value *SET_TO_CLIENT_TIME*, the attribute is set to the time passed by the client in an **nfstime3** structure. (See *FSINFO* in Section 12.4.0 on page 246, which addresses the issue of time granularity).

diropargs3

```
struct diropargs3 {
    nfs_fh3 dir;
    filename3 name;
};
```

The **diropargs3** structure is used in directory operations. The file handle, *dir*, identifies the directory in which to manipulate or access the file, *name*. See additional comments in Section 12.3.5 on page 192.

12.2.3 Attributes and Consistency Data on Failure

For those procedures that return either **post_op_attr** or **wcc_data** structures on failure, the discriminated union may contain the pre-operation attributes of the object or object parent directory. This depends on the error encountered and may also depend on the particular server implementation. Implementors are strongly encouraged to return as much attribute data as possible upon failure, but client implementors need to be aware that their implementation must correctly handle the variant return instance where no attributes or consistency data is returned.

12.2.4 General File Name Requirements

The following requirements apply to all NFS Version 3 protocol procedures in which the client provides one or more file names in the arguments: *LOOKUP*, *CREATE*, *MKDIR*, *SYMLINK*, *MKNOD*, *REMOVE*, *RMDIR*, *RENAME* and *LINK*.

1. The file name must not be null nor may it be the null string. The server should return the *NFS3ERR_ACCES* error if it receives such a file name. On some clients, a null string used as a file name is assumed to be an alias for the current directory. Clients that require this functionality should implement it for themselves and not depend upon the server to support such semantics.

2. A filename having the value of ‘‘.’’ (dot) is assumed to be an alias for the current directory. Clients that require this functionality should implement it for themselves and not depend upon the server to support such semantics. However, the server should be able to handle such a filename correctly.

3. A filename having the value of ‘‘. .’’ (dot-dot) is assumed to be an alias for the parent of the current directory; in other words, the directory that contains the current directory. The server should be prepared to handle this semantic, if it supports directories, even if those directories do not contain XPG-style dot or dot-dot entries.

4. If the filename is longer than the maximum for the file system (see *PATHCONF* in Section 12.4.0 on page 249, specifically *name_max*), the result depends on the value of the *PATHCONF* flag *no_trunc*. If *no_trunc* is *FALSE*, the filename will be silently truncated to *name_max* bytes. If *no_trunc* is *TRUE* and the filename exceeds the server’s file system maximum filename length, the operation will fail with the *NFS3ERR_NAMETOOLONG* error.

5. In general, there will be characters that a server will not be able to handle as part of a filename. This set of characters will vary from server to server and from implementation to implementation. In most cases, it is the server that will control the client’s view of the file system. If the server receives a filename containing characters that it can not handle, the *NFS3ERR_ACCES* error should be returned. Client implementations should be prepared to handle this side affect of heterogeneity.

See additional comments in Section 12.3.5 on page 192.

12.3 XNFS Implementation Issues

The NFS Version 3 protocol was designed to allow different operating systems to share files. However, since it 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 general implementation-specific details and semantic issues. Procedure descriptions have implementation guidance specific to that procedure.

12.3.1 Server/Client Relationship

The NFS Version 3 protocol is designed to allow servers to be as simple and general as possible. Sometimes the simplicity of the server can be a problem, if the client implements complicated file system semantics.

For example, some operating systems allow removal of open files. A process can open a file and, while it is open, remove it from the directory. The file can be read and written as long as the process keeps it open, even though the file has no name in the file system. It is impossible for a stateless server to implement these semantics. The client can do some tricks such as renaming the file on remove (to a hidden name), and only physically deleting it on close. The NFS Version 3 protocol provides sufficient functionality to implement most file system semantics on a client.

Every NFS Version 3 protocol client can also potentially be a server, and remote and local mounted file systems can be freely mixed. This leads to some 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, both NFS Version 2 protocol and NFS Version 3 protocol implementations do not typically let clients cross a server’s mount point. When a client does a *LOOKUP* on a directory on which the server has mounted a file system, the client sees the underlying directory instead of the mounted directory.

For example, if a server has a file system called **/usr** and mounts another file system on **/usr/src**, if a client mounts **/usr**, it does not see the mounted version of **/usr/src**. A client could do remote mounts that match the server's mount points to maintain the server's view. In this example, the client would also have to mount **/usr/src** in addition to **/usr**, even if they are from the same server.

12.3.2 Pathname Interpretation

There are a few complications to the rule that pathnames are always parsed on the client. For example, symbolic links could have different interpretations on different clients. There is no answer to this problem in this document.

Another common problem for non-XPG implementations is the special interpretation of the pathname ''..'' to mean the parent of a given directory. A future revision of the protocol may use an explicit flag to indicate the parent instead; however, it is not a problem because many working non-XPG implementations exist.

12.3.3 Permission Issues

The NFS Version 3 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, or another stronger form of authentication such as *AUTH_DES* or *AUTH_KERB*. With *AUTH_UNIX* authentication, the server gets the client's effective user ID, effective group ID and groups on each call and uses them to check permission. These are the so-called UNIX credentials. *AUTH_DES* and *AUTH_KERB* use a network name, or netname, as the basis for identification (from which a UNIX server derives the necessary standard UNIX credentials). There are problems with this method that have been solved.

Using user ID and group ID implies that the client and server share the same user ID list. Every server and client pair must have the same mapping from user to user ID and from group to group ID. Since every client can also be a server, this tends to imply that the whole network shares the same user/group ID space. If this is not the case, then it usually falls upon the server to perform some custom mapping of credentials from one authentication domain into another. A discussion of techniques for managing a shared user space or for providing mechanisms for user ID mapping is beyond the scope of this document.

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 cannot detect that the file is open and must do permission checking on each read and write call. UNIX client semantics of access permission checking on open can be provided with the *ACCESS* procedure call in this revision, which allows a client to explicitly check access permissions without resorting to trying the operation. 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. This is needed in a practical NFS Version 3 protocol server implementation, but it does depart from correct local file system semantics. This should not affect the return result of access permissions as returned by the *ACCESS* procedure, however.

A similar problem has to do with paging in an executable program 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. In a local UNIX file system, an executable file does not need read permission to execute (page-in). An NFS Version 3 protocol server can not tell the difference between a normal file read (where the read permission bit is meaningful) and a demand page-in read (where the server should allow access to the executable file if the execute bit is set for that user or group or public). To make this work, the server allows reading of files if the user ID given in the call has either execute or read permission on the file, through ownership, group membership or public access. Again, this departs from correct local file system semantics.

In some operating systems, a particular user (on UNIX systems, the user ID 0) has access to all files, no matter what permission and ownership they have. This super-user permission might not be allowed on the server, since anyone who can become super-user on their client could gain access to all remote files. A UNIX server by default maps user ID 0 to a distinguished value (UID_NOBODY), as well as mapping the groups list, before doing its access checking. A server implementation may provide a mechanism to change this mapping. This works except for NFS Version 3 protocol root file systems (required for diskless NFS Version 3 protocol client support), where super-user access cannot be avoided. Export options are used, on the server, to restrict the set of clients allowed super-user access.

12.3.4 Duplicate Request Cache

The typical NFS Version 3 protocol failure recovery model uses client time-out and retry to handle server crashes, network partitions and lost server replies. A retried request is called a duplicate of the original.

When used in a file server context, the term *idempotent* can be used to distinguish between operation types. An idempotent request is one that a server can perform more than once with equivalent results (though it may in fact change, as a side effect, the access time on a file, say for *READ*). Some NFS operations are obviously non-idempotent. They cannot be reprocessed without special attention simply because they may fail if tried a second time. The *CREATE* request, for example, can be used to create a file for which the owner does not have write permission. A duplicate of this request cannot succeed if the original succeeded. Likewise, a file can be removed only once.

The side effects caused by performing a duplicate non-idempotent request can be destructive (for example, a truncate operation causing lost writes). The combination of a stateless design with the common choice of an unreliable network transport (UDP) implies the possibility of destructive replays of non-idempotent requests. Though to be more accurate, it is the inherent stateless design of the NFS Version 3 protocol on top of an unreliable RPC mechanism that yields the possibility of destructive replays of non-idempotent requests, since even in an implementation of the NFS Version 3 protocol over a reliable connection-oriented transport, a connection break with automatic reestablishment requires duplicate request processing (the client will retransmit the request, and the server needs to deal with a potential duplicate non-idempotent request).

Most NFS Version 3 protocol server implementations use a cache of recent requests (called the duplicate request cache) for the processing of duplicate non-idempotent requests. The duplicate request cache provides a short-term memory mechanism in which the original completion status of a request is remembered and the operation attempted only once. If a duplicate copy of this request is received, then the original completion status is returned.

The duplicate-request cache mechanism has been useful in reducing destructive side effects caused by duplicate NFS Version 3 protocol requests. This mechanism, however, does not guarantee against these destructive side effects in all failure modes. Most servers store the duplicate request cache in RAM, so the contents are lost if the server crashes. The exception to this may possibly occur in a redundant server approach to high availability, where the file system itself may be used to share the duplicate request cache state. Even if the cache survives server reboots (or failovers in the high availability case), its effectiveness is a function of its size. A network partition can cause a cache entry to be reused before a client receives a reply for the corresponding request. If this happens, the duplicate request will be processed as a new one, possibly with destructive side effects.

12.3.5 Filename Component Handling

Server implementations of NFS Version 3 protocol will frequently impose restrictions on the names that can be created. Many servers will also forbid the use of names that contain certain characters, such as the path component separator used by the server operating system. For example, an XPG file system will reject a name that contains ''/'', while ''.'' and ''..'' are distinguished in a XPG file system, and must not be specified as the name when creating a file system object. The exact error status values return for these errors is specified in the description of each procedure argument. The values (which conform to NFS Version 2 protocol server practice) are not necessarily obvious, nor are they consistent from one procedure to the next.

12.3.6 Synchronous Modifying Operations

Data-modifying operations in the NFS Version 3 protocol are synchronous. When a procedure returns to the client, the client can assume that the operation has completed and any data associated with the request is now on stable storage.

12.3.7 Stable Storage

NFS Version 3 protocol servers must be able to recover without data loss from multiple power failures (including cascading power failures; that is, several power failures in quick succession), operating system failures and hardware failure of components other than the storage medium itself (for example, disk or non-volatile RAM).

Some examples of stable storage that are allowable for an NFS server include:

- Media commit of data; that is, the modified data has been successfully written to the disk media (for example, the disk platter).
- An immediate reply disk drive with battery-backed on-drive intermediate storage or uninterruptible power system (UPS).
- Server commit of data with battery-backed intermediate storage and recovery software.
- Cache commit with uninterruptible power system and recovery software.

Conversely, the following are not examples of stable storage:

- An immediate reply disk drive without battery-backed on-drive intermediate storage or uninterruptible power system.
- Cache commit without both uninterruptible power system and recovery software.

The only exception to this (introduced in Version 3 protocol) is as described under the *WRITE* procedure on the handling of the *stable* bit, and the use of the *COMMIT* procedure. It is the use of the synchronous *COMMIT* procedure that provides the necessary semantic support in the NFS Version 3 protocol.

12.3.8 Lookups and Name Resolution

A common objection to the NFS Version 3 protocol is the philosophy of component-by-component *LOOKUP* by the client in resolving a name. The objection is that this is inefficient, as latencies for component-by-component *LOOKUP* would be unbearable.

Implementation practice solves this issue. A name cache, providing component to file-handle mapping, is kept on the client to short circuit actual *LOOKUP* invocations over the wire. The cache is subject to cache timeout parameters that bound attributes.

Note that multi-component lookup is allowed relative to the public filehandle (see Appendix E) for use by WebNFS clients as an alternative to the MNTPROC_MNT procedure of the MOUNT protocol. Clients may also cache the results of these multi-component lookups, subject to the same timeout parameters that bound attributes.

12.3.9 Adaptive Retransmission

Most client implementations use either an exponential back-off strategy to some maximum retransmission value, or a more adaptive strategy that attempts congestion avoidance.

12.3.10 Caching Policies

The NFS Version 3 protocol does not define a policy for caching on the client or server. In particular, there is no support for strict cache consistency between a client and server, nor between different clients.

12.3.11 Stable Versus Unstable Writes

The setting of the *stable* field in the *WRITE* arguments (that is, whether or not to do asynchronous *WRITE* requests) is straightforward on a UNIX client. If the NFS Version 3 protocol client receives a write request that is not marked as being asynchronous, it should generate the RPC with *stable* set to *DATA_SYNC* or *FILE_SYNC*. If the request is marked as being asynchronous, the RPC should be generated with *stable* set to *UNSTABLE*. If the response comes back with the *committed* field set to *DATA_SYNC* or *FILE_SYNC*, the client should just mark the write request as done and no further action is required. If *committed* is set to *UNSTABLE*, indicating that the buffer was not synchronised with the server's disk, the client will need to mark the buffer in some way that indicates that a copy of the buffer lives on the server and that a new copy does not need to be sent to the server, but that a commit is required.

Note that this algorithm introduces a new state for buffers, thus there are now three states for buffers. The three states are dirty, done but needs to be committed, and done. This extra state on the client will likely require modifications to the system outside of the NFS Version 3 protocol client.

The asynchronous write opens up the window of problems associated with write sharing. For example: client A writes some data asynchronously. Client A is still holding the buffers cached, waiting to commit them later. Client B reads the modified data and writes it back to the server. The server then crashes. When it comes back up, client A issues a *COMMIT* operation, which returns with a different cookie as well as changed attributes. In this case, the correct action may or may not be to retransmit the cached buffers. Unfortunately, client A can't tell for sure, so it will need to retransmit the buffers, thus overwriting the changes from client B. Fortunately, write sharing is rare and the solution matches the current write sharing situation. Without using locking for synchronisation, the behaviour will be indeterminate.

In a high availability (redundant system) server implementation, two cases exist that relate to the *verf* changing. If the high availability server implementation does not use a shared-memory scheme, then the *verf* must change on failover, since the unsynchronised data is not available to the second processor and there is no guarantee that the system that had the data cached was able to flush it to stable storage before going down. The client will need to retransmit the data to be safe. In a shared-memory high availability server implementation, the *verf* would not need to change because the server would still have the cached data available to it to be flushed. The exact policy regarding the *verf* in a shared memory high availability implementation, however, is up to the server implementor.

12.3.12 32-bit Clients/Servers and 64-bit Clients/Servers

The 64 bit nature of the NFS Version 3 protocol introduces several compatibility problems. The most notable two are mismatched clients and servers; that is, a 32 bit client and a 64 bit server or a 64 bit client and a 32 bit server.

The problems of a 64 bit client and a 32 bit server are easy to handle. The client will never encounter a file that it can not handle. If it sends a request to the server that the server can not handle, the server should reject the request with an appropriate error.

The problems of a 32 bit client and a 64 bit server are much harder to handle. In this situation, the server does not have a problem because it can handle anything that the client can generate. However, the client may encounter a file that it can not handle. The client will not be able to handle a file whose size can not be expressed in 32 bits. Thus, the client will not be able to properly decode the size of the file into its local attributes structure. Also, a file can grow beyond the limit of the client while the client is accessing the file.

The solutions to these problems are left up to the individual implementor. However, there are two common approaches used to resolve this situation. The implementor can choose between them or even can invent a new solution altogether.

The most common solution is for the client to deny access to any file whose size can not be expressed in 32 bits. This is probably the safest, but does introduce some strange semantics when the file grows beyond the limit of the client while it is being access by that client. The file becomes inaccessible even while it is being accessed.

The second solution is for the client to map any size greater than it can handle to the maximum size that it can handle. This allows the application access as much of the file as possible given the 32 bit offset restriction. This eliminates the strange semantic of the file effectively disappearing after it has been accessed, but does introduce other problems. The client will not be able to access the entire file.

Currently, the first solution is the recommended solution. However, client implementors are encouraged to do the best that they can to reduce the effects of this situation.

12.4 Server Procedures

The following reference pages define the additional set of procedures, with arguments and results defined using the RPC language, for the Version 3 protocol.

```
/*
 *  Remote file service routines
 */
program NFS_PROGRAM {
    version NFS_V3 {
        void NFSPROC3_NULL(void) = 0;
        GETATTR3res NFSPROC3_GETATTR(GETATTR3args) = 1;
        SETATTR3res NFSPROC3_SETATTR(SETATTR3args) = 2;
        LOOKUP3res NFSPROC3_LOOKUP(LOOKUP3args) = 3;
        ACCESS3res NFSPROC3_ACCESS(ACCESS3args) = 4;
        READLINK3res NFSPROC3_READLINK(READLINK3args) = 5;
        READ3res NFSPROC3_READ(READ3args) = 6;
        WRITE3res NFSPROC3_WRITE(WRITE3args) = 7;
        CREATE3res NFSPROC3_CREATE(CREATE3args) = 8;
        MKDIR3res NFSPROC3_MKDIR(MKDIR3args) = 9;
        SYMLINK3res NFSPROC3_SYMLINK(SYMLINK3args) = 10;
        MKNOD3res NFSPROC3_MKNOD(MKNOD3args) = 11;
        REMOVE3res NFSPROC3_REMOVE(REMOVE3args) = 12;
        RMDIR3res NFSPROC3_RMDIR(RMDIR3args) = 13;
        RENAME3res NFSPROC3_RENAME(RENAME3args) = 14;
        LINK3res NFSPROC3_LINK(LINK3args) = 15;
        READDIR3res NFSPROC3_READDIR(READDIR3args) = 16;
        READDIRPLUS3res
            NFSPROC3_READDIRPLUS(READDIRPLUS3args) = 17;
        FSSTAT3res NFSPROC3_FSSTAT(FSSTAT3args) = 18;
        FSINFO3res NFSPROC3_FSINFO(FSINFO3args) = 19;
        PATHCONF3res NFSPROC3_PATHCONF(PATHCONF3args) = 20;
        COMMIT3res NFSPROC3_COMMIT(COMMIT3args) = 21;
    } = 3;
} = 100003;
```

Name

NFSPROC3_NULL — Do Nothing

Call Arguments

None.

Return Arguments

None.

RPC Procedure Description

```
void
NFSPROC3_NULL(void) = 0;
```

Description

Procedure *NULL* does no work. It is made available to allow server response testing and timing.

Implementation Guidance

It is important that this procedure do no work at all so that it can be used to measure the overhead of processing a service request. By convention, the *NULL* procedure should never require any authentication. A server may choose to ignore this convention, in a more secure implementation, where responding to the *NULL* procedure call acknowledges the existence of a resource to an unauthenticated client.

Return Codes

Since the *NULL* procedure takes no NFS Version 3 protocol arguments and returns no NFS Version 3 protocol response, it can not return an NFS Version 3 protocol error. However, it is possible that some server implementations may return RPC errors based on security and authentication requirements.

Name

NFSPROC3_GETATTR — Get File Attributes

Call Arguments

```
struct GETATTR3args {
    nfs_fh3 object;
};
```

Return Arguments

```
struct GETATTR3resok {
    fattr3 obj_attributes;
};

union GETATTR3res switch (nfsstat3 status) {
    case NFS3_OK:
        GETATTR3resok resok;
    default:
        void;
};
```

RPC Procedure Description

```
GETATTR3res
NFSPROC3_GETATTR(GETATTR3args) = 1;
```

Description

Procedure *GETATTR* retrieves the attributes for a specified file system object. The object is identified by the file handle that the server returned as part of the response from a *LOOKUP*, *CREATE*, *MKDIR*, *SYMLINK*, *MKNOD* or *READDIRPLUS* procedure (or from the *MOUNT* service, described in Chapter 13 on page 255).

On entry, the arguments in *GETATTR3args* are:

object The file handle of an object whose attributes are to be retrieved.

Upon successful return, *GETATTR3res.status* is *NFS3_OK* and *GETATTR3res.resok* contains:

obj_attributes The attributes for the object.

Otherwise, *GETATTR3res.status* contains the error on failure and no other results are returned.

Implementation Guidance

The attributes of file system objects is a point of major disagreement between different operating systems. Servers must make a best attempt to support all of the attributes in the **fattr3** structure so that clients can count on this as a common ground. Some mapping may be required to map local attributes to those in the **fattr3** structure.

Today, most client NFS Version 3 protocol implementations implement a time-bounded attribute caching scheme to reduce over-the-wire attribute checks.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_SETATTR — Set File Attributes

Call Arguments

```
union sattrguard3 switch (bool check) {
    case TRUE:
        nfstime3 obj_ctime;
    case FALSE:
        void;
};

struct SETATTR3args {
    nfs_fh3 object;
    sattr3 new_attributes;
    sattrguard3 guard;
};
```

Return Arguments

```
struct SETATTR3resok {
    wcc_data obj_wcc;
};

struct SETATTR3resfail {
    wcc_data obj_wcc;
};

union SETATTR3res switch (nfsstat3 status) {
    case NFS3_OK:
        SETATTR3resok resok;
    default:
        SETATTR3resfail resfail;
};
```

RPC Procedure Description

```
SETATTR3res
NFSPROC3_SETATTR(SETATTR3args) = 2;
```

Description

Procedure *SETATTR* changes one or more of the attributes of a file system object on the server. The new attributes are specified by a **sattr3** structure.

On entry, the arguments in *SETATTR3args* are:

object The file handle for the object.

new_attributes A **sattr3** structure containing Booleans and enumerations describing the attributes to be set and the new values for those attributes.

guard A **sattrguard3** union:

check *TRUE* if the server is to verify that *guard.obj_ctime* matches the *ctime* for the object; *FALSE* otherwise.

A client may request that the server check that the object is in an expected state before performing the *SETATTR* operation. To do this, it sets the argument *guard.check* to *TRUE* and the client passes a time value in *guard.obj_ctime*. If *guard.check* is *TRUE*, the server must compare the value of *guard.obj_ctime* to the current *ctime* of the object. If the values are different, the server must preserve the object attributes and must return a status of *NFS3ERR_NOT_SYNC*. If *guard.check* is *FALSE*, the server will not perform this check.

Upon successful return, *SETATTR3res.status* is *NFS3_OK* and *SETATTR3res.resok* contains:

obj_wcc A **wcc_data** structure containing the old and new attributes for the object.

Otherwise, *SETATTR3res.status* contains the error on failure and *SETATTR3res.resfail* contains the following:

obj_wcc A **wcc_data** structure containing the old and new attributes for the object.

Implementation Guidance

The *guard.check* mechanism allows the client to avoid changing the attributes of an object on the basis of stale attributes. It does not guarantee exactly-once semantics. In particular, if a reply is lost and the server does not detect the retransmission of the request, the procedure can fail with the *NFS3ERR_NOT_SYNC* error, even though the attribute setting was previously performed successfully. The client can attempt to recover from this error by getting fresh attributes from the server and sending a new *SETATTR* request using the new *ctime*. The client can optionally check the attributes to avoid the second *SETATTR* request if the new attributes show that the attributes have already been set as desired (though it may not have been the issuing client that set the attributes).

The *new_attributes.size* field is used to request changes to the size of a file. A value of zero causes the file to be truncated, a value less than the current size of the file causes data from new size to the end of the file to be discarded, and a size greater than the current size of the file causes logically zeroed data bytes to be added to the end of the file. Servers are free to implement this using holes or actual zero data bytes. Clients must not make any assumptions regarding a server's implementation of this feature, beyond that the bytes returned will be zeroed. Servers must support extending the file size via *SETATTR*.

SETATTR is not guaranteed atomic. A failed *SETATTR* may partially change a file's attributes.

Changing the size of a file with *SETATTR* indirectly changes the *mtime*. A client must account for this as size changes can result in data deletion.

If server and client times differ, programs that compare client time to file times can break. A time maintenance protocol should be used to limit client/server time skew.

In a heterogeneous environment, it is possible that the server will not be able to support the full range of *SETATTR* requests. The *NFS3ERR_INVAL* error may be returned if the server can not store a user ID or group ID in its own representation of user or group IDs, respectively. If the server can only support 32 bit offsets and sizes, a *SETATTR* request to set the size of a file to larger than can be represented in 32 bits will be rejected with this same error.

Return Codes

NFS3ERR_PERM Not owner. The caller does not have the correct ownership to perform the requested operation.

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_INVAL Invalid argument or unsupported argument for an operation.

NFS3ERR_NOSPC No space left on device. The operation would have caused the server's file system to exceed its limit.

NFS3ERR_ROFS Read-only file system. A modifying operation was attempted on a read-only file system.

NFS3ERR_DQUOT Resource (quota) hard limit exceeded. The user's resource limit on the server has been exceeded.

NFS3ERR_NOT_SYNC
An update synchronisation mismatch was detected during a *SETATTR* operation.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_LOOKUP — Lookup Filename

Call Arguments

```
struct LOOKUP3args {
    diropargs3 what;
};
```

Return Arguments

```
struct LOOKUP3resok {
    nfs_fh3 object;
    post_op_attr obj_attributes;
    post_op_attr dir_attributes;
};

struct LOOKUP3resfail {
    post_op_attr dir_attributes;
};

union LOOKUP3res switch (nfsstat3 status) {
    case NFS3_OK:
        LOOKUP3resok resok;
    default:
        LOOKUP3resfail resfail;
};
```

RPC Procedure Description

```
LOOKUP3res
NFSPROC3_LOOKUP(LOOKUP3args) = 3;
```

Description

Procedure *LOOKUP* searches a directory for a specific name and returns the file handle for the corresponding file system object.

On entry, the arguments in *LOOKUP3args* are:

what Object to look up.

dir The file handle for the directory to search.

name The filename to be searched for. See Section 12.2.4 on page 188 for more information on file names.

Upon successful return, *LOOKUP3res.status* is *NFS3_OK* and *LOOKUP3res.resok* contains:

object The file handle of the object corresponding to *what.name*.

obj_attributes The attributes of the object corresponding to *what.name*.

dir_attributes The post-operation attributes of the directory *what.dir*.

Otherwise, *LOOKUP3res.status* contains the error on failure and *LOOKUP3res.resfail* contains the following:

dir_attributes The post-operation attributes for the directory *what.dir*.

Implementation Guidance

At first glance, in the case where *what.name* refers to a mount point on the server, two different replies seem possible. The server can return either the file handle for the underlying directory that is mounted on it or the file handle of the root of the mounted directory. This ambiguity is simply resolved. A server will not allow a *LOOKUP* operation to cross a mount point to the root of a different file system, even if the file system is exported. This does not prevent a client from accessing a hierarchy of file systems exported by a server, but the client must mount each of the file systems individually so that the mount point crossing takes place on the client. A given server implementation may refine these rules given capabilities or limitations particular to that implementation.

Two filenames are distinguished, as in the NFS Version 2 protocol. The name ‘‘.’’ is an alias for the current directory and the name ‘‘..’’ is an alias for the parent directory; that is, the directory that includes the specified directory as a member. There is no facility for dealing with a multiparented directory and NFS assumes a hierarchical organisation, organised as a single-rooted tree.

Unless the lookup is relative to the public filehandle, this procedure does not follow symbolic links. The client is responsible for all parsing of file names, including file names that are modified by symbolic links encountered during the lookup process.

WebNFS clients may use a pathname in place of the name where the lookup is relative to the public filehandle. In this case, the server must cross mount points when evaluating this pathname and follow any symbolic links that occur in all but the final component of the pathname.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_NOENT No such file or directory. The file or directory name specified does not exist.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_NOTDIR Not a directory. The caller specified a non-directory in a directory operation.

NFS3ERR_NAMETOOLONG
The filename in an operation was too long.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_ACCESS — Check Access Permission

Call Arguments

```
const ACCESS3_READ    = 0x0001;
const ACCESS3_LOOKUP  = 0x0002;
const ACCESS3_MODIFY  = 0x0004;
const ACCESS3_EXTEND  = 0x0008;
const ACCESS3_DELETE  = 0x0010;
const ACCESS3_EXECUTE = 0x0020;

struct ACCESS3args {
    nfs_fh3 object;
    uint32 access;
};
```

Return Arguments

```
struct ACCESS3resok {
    post_op_attr obj_attributes;
    uint32 access;
};

struct ACCESS3resfail {
    post_op_attr obj_attributes;
};

union ACCESS3res switch (nfsstat3 status) {
    case NFS3_OK:
        ACCESS3resok resok;
    default:
        ACCESS3resfail resfail;
};
```

RPC Procedure Description

```
ACCESS3res
NFSPROC3_ACCESS(ACCESS3args) = 4;
```

Description

Procedure *ACCESS* determines the access rights that a user, as identified by the credentials in the request, has with respect to a file system object. The client encodes the set of permissions that are to be checked in a bit mask. The server checks the permissions encoded in the bit mask. A status of *NFS3_OK* is returned along with a bit mask encoded with the permissions that the client is allowed.

The results of this procedure are necessarily advisory in nature. That is, a return status of *NFS3_OK* and the appropriate bit set in the bit mask does not imply that such access will be allowed to the file system object in the future, because access rights can be revoked by the server at any time.

On entry, the arguments in *ACCESS3args* are:

object The file handle for the file system object to which access is to be checked.

access A bit mask of access permissions to check. The following access permissions may be requested:

ACCESS3_READ
Read data from file or read a directory.

ACCESS3_LOOKUP
Look up a name in a directory (no meaning for non-directory objects).

ACCESS3_MODIFY
Rewrite existing file data or modify existing directory entries.

ACCESS3_EXTEND
Write new data or add directory entries.

ACCESS3_DELETE
Delete an existing directory entry (no meaning for non-directory objects).

ACCESS3_EXECUTE
Execute file (no meaning for a directory).

Upon successful return, *ACCESS3res.status* is *NFS3_OK*. The server should return a status of *NFS3_OK* if no errors occurred that prevented the server from making the required access checks. The results in *ACCESS3res.resok* are:

obj_attributes The post-operation attributes of object.

access A bit mask of access permissions indicating access rights for the authentication credentials provided with the request.

Otherwise, *ACCESS3res.status* contains the error on failure and *ACCESS3res.resfail* contains the following:

obj_attributes The attributes of object, if access to attributes is permitted.

Implementation Guidance

In general, it is not sufficient for the client to attempt to deduce access permissions by inspecting the *uid*, *gid* and *mode* fields in the file attributes, since the server may perform user ID or group ID mapping or enforce additional access control restrictions. It is also possible that the NFS Version 3 protocol server may not be in the same ID space as the NFS Version 3 protocol client. In these cases (and perhaps others), the NFS Version 3 protocol client can not reliably perform an access check with only the current file attributes.

In the NFS Version 2 protocol, the only reliable way to determine whether an operation was allowed was to try it and see if it succeeded or failed. Using the *ACCESS* procedure in the NFS Version 3 protocol, the client can ask the server to indicate whether or not one or more classes of operations are permitted. The *ACCESS* operation is provided to allow clients to check before doing a series of operations. This is useful in operating systems (such as UNIX) where permission checking is done only when a file or directory is opened. This procedure is also invoked by the NFS client access procedure (called possibly through *access*( )). The intent is to make the behaviour of opening a remote file more consistent with the behaviour of opening a local file.

The information returned by the server in response to an *ACCESS* call is not permanent. It was correct at the exact time that the server performed the checks, but not necessarily afterwards. The server can revoke access permission at any time.

The NFS Version 3 protocol client should use the effective credentials of the user to build the authentication information in the *ACCESS* request used to determine access rights. It is the effective user and group credentials that are used in subsequent read and write operations. See the comments in Section 12.3.3 on page 190 for more information on this topic.

Many implementations do not directly support the *ACCESS3_DELETE* permission. Operating systems like the UNIX system will ignore the *ACCESS3_DELETE* bit if set on an access request on a non-directory object. In these systems, delete permission on a file is determined by the access permissions on the directory in which the file resides, instead of being determined by the permissions of the file itself. Thus, the bit mask returned for such a request will have the *ACCESS3_DELETE* bit set to zero, indicating that the client does not have this permission.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_READLINK — Read From Symbolic Link

Call Arguments

```
struct READLINK3args {
    nfs_fh3 symlink;
};
```

Return Arguments

```
struct READLINK3resok {
    post_op_attr symlink_attributes;
    nfspath3 data;
};

struct READLINK3resfail {
    post_op_attr symlink_attributes;
};

union READLINK3res switch (nfsstat3 status) {
    case NFS3_OK:
        READLINK3resok resok;
    default:
        READLINK3resfail resfail;
};
```

RPC Procedure Description

```
READLINK3res
NFSPROC3_READLINK(READLINK3args) = 5;
```

Description

Procedure *READLINK* reads the data associated with a symbolic link. The data is a string that is opaque to the server. That is, whether created by the NFS Version 3 protocol software from a client or created locally on the server, the data in a symbolic link is not interpreted when created, but is simply stored.

On entry, the arguments in *READLINK3args* are:

symlink The file handle for a symbolic link (file system object of type *NF3LNK*).

Upon successful return, *READLINK3res.status* is *NFS3_OK* and *READLINK3res.resok* contains:

data The data associated with the symbolic link.

symlink_attributes
The post-operation attributes for the symbolic link.

Otherwise, *READLINK3res.status* contains the error on failure and *READLINK3res.resfail* contains the following:

symlink_attributes
The post-operation attributes for the symbolic link.

Implementation Guidance

A symbolic link is nominally a pointer to another file. The data is not necessarily interpreted by the server, just stored in the file. It is possible for a client implementation to store a pathname that is not meaningful to the server operating system in a symbolic link. A *READLINK* operation returns the data to the client for interpretation. If different implementations want to share access to symbolic links, then they must agree on the interpretation of the data in the symbolic link.

The *READLINK* operation is only allowed on objects of type *NF3LNK*. The server should return the *NFS3ERR_INVAL* error if the object is not of type *NF3LNK*.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_INVAL Invalid argument or unsupported argument for an operation.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_NOTSUPP
The operation is not supported.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_READ — Read From File

Call Arguments

```
struct READ3args {
    nfs_fh3 file;
    offset3 offset;
    count3 count;
};
```

Return Arguments

```
struct READ3resok {
    post_op_attr file_attributes;
    count3 count;
    bool eof;
    opaque data<>;
};

struct READ3resfail {
    post_op_attr file_attributes;
};

union READ3res switch (nfsstat3 status) {
    case NFS3_OK:
        READ3resok resok;
    default:
        READ3resfail resfail;
};
```

RPC Procedure Description

```
READ3res
NFSPROC3_READ(READ3args) = 6;
```

Description

Procedure *READ* reads data from a file.

On entry, the arguments in *READ3args* are:

file The file handle of the file from which data is to be read. This must identify a file system object of type *NF3REG*.

offset The position within the file at which the read is to begin. An *offset* of zero means to read data starting at the beginning of the file. If *offset* is greater than or equal to the size of the file, the status *NFS3_OK* is returned with *count* set to zero and *eof* set to *TRUE*, subject to access permission checking.

count The number of bytes of data that are to be read. If *count* is zero, the *READ* will succeed and return zero bytes of data, subject to access permission checking. The *count* must be less than or equal to the value of the *rtmax* field in the *FSINFO* reply structure for the file system that contains *file*. If greater, the server may return only *rtmax* bytes, resulting in a short read.

Upon successful return, *READ3res.status* is *NFS3_OK* and *READ3res.resok* contains:

file_attributes The attributes of the file on completion of the read.

count The number of bytes of data returned by the read.

eof If the read ended at the end-of-file (formally, in a correctly formed *READ* request, if *READ3args.offset* plus *READ3resok.count* is equal to the size of the file), *eof* is returned as *TRUE*; otherwise it is *FALSE*. A successful *READ* of an empty file will always return *eof* as *TRUE*.

data The counted data read from the file.

Otherwise, *READ3res.status* contains the error on failure and *READ3res.resfail* contains the following:

file_attributes The post-operation attributes of the file.

Implementation Guidance

The **nfsdata** type used for the *READ* and *WRITE* operations in the NFS Version 2 protocol defining the data portion of a request or reply has been changed to a variable-length opaque byte array. The maximum size allowed by the protocol is now limited by what XDR and underlying transports will allow. There are no artificial limits imposed by the NFS Version 3 protocol. Consult the *FSINFO* procedure description for details.

It is possible for the server to return fewer than *count* bytes of data. If the server returns less than the *count* requested and *eof* set to *FALSE*, the client should issue another *READ* to get the remaining data. A server may return less data than requested under several circumstances. The file may have been truncated by another client or perhaps on the server itself, changing the file size from what the requesting client believes to be the case. This would reduce the actual amount of data available to the client. It is possible that the server may back off the transfer size and reduce the read request return. Server resource exhaustion may also occur, necessitating a smaller read return.

Some NFS Version 2 protocol client implementations chose to interpret a short read response as indicating end-of-file. The addition of the *eof* flag in the NFS Version 3 protocol provides a correct way of handling end-of-file.

Some NFS Version 2 protocol server implementations incorrectly returned *NFSERR_ISDIR* if the file system object type was not a regular file. The correct return value for the NFS Version 3 protocol is *NFS3ERR_INVAL*.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_NXIO No such device or address.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_INVAL Invalid argument or unsupported argument for an operation. Two examples are attempting a *READLINK* on an object other than a symbolic link or attempting to *SETATTR* a time field on a server that does not support this operation.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_WRITE — Write to file

Call Arguments

```
enum stable_how {
    UNSTABLE = 0,
    DATA_SYNC = 1,
    FILE_SYNC = 2
};

struct WRITE3args {
    nfs_fh3 file;
    offset3 offset;
    count3 count;
    stable_how stable;
    opaque data<>;
};
```

Return Arguments

```
struct WRITE3resok {
    wcc_data file_wcc;
    count3 count;
    stable_how committed;
    writeverf3 verf;
};

struct WRITE3resfail {
    wcc_data file_wcc;
};

union WRITE3res switch (nfsstat3 status) {
    case NFS3_OK:
        WRITE3resok resok;
    default:
        WRITE3resfail resfail;
};
```

RPC Procedure Description

```
WRITE3res
NFSPROC3_WRITE(WRITE3args) = 7;
```

Description

Procedure *WRITE* writes data to a file.

On entry, the arguments in *WRITE3args* are:

file The file handle for the file to which data is to be written. This must identify a file system object of type *NF3REG*.

offset The position within the file at which the write is to begin. An *offset* of zero means to write data starting at the beginning of the file.

count The number of bytes of data to be written. If *count* is zero, the *WRITE* will succeed and return a *count* of zero, barring errors due to permission checking. The size of data must be less than or equal to the value of the *wtmax* field in the *FSINFO* reply structure for the file system that contains *file*. If greater, the server may write only *wtmax* bytes, resulting in a short write.

stable

If *stable* is *FILE_SYNC*, the server must commit the data written plus all file system metadata to stable storage before returning results. This corresponds to the NFS Version 2 protocol semantics. Any other behaviour constitutes a protocol violation. If *stable* is *DATA_SYNC*, then the server must commit all of the data to stable storage and enough of the metadata to retrieve the data before returning. The server implementor is free to implement *DATA_SYNC* in the same fashion as *FILE_SYNC*, but with a possible performance drop. If *stable* is *UNSTABLE*, the server may commit any part of the data and the metadata to stable storage, including all or none, before returning a reply to the client. There is no guarantee whether or when any uncommitted data will subsequently be committed to stable storage. The only guarantees made by the server are that it will not destroy any data without changing the value of *verf* and that it will not commit the data and metadata at a level less than that requested by the client. See Section 12.4.0 on page 251 for more information on if and when data is committed to stable storage.

data

The data to be written to the file.

Upon successful return, *WRITE3res.status* is *NFS3_OK* and *WRITE3res.resok* contains:

file_wcc

Weak cache consistency data for the file. For a client that requires only the post-write file attributes, these can be found in *file_wcc.after*.

count

The number of bytes of data written to the file. The server may write fewer bytes than requested. If so, the actual number of bytes written starting at location *offset* is returned.

committed

The server should return an indication of the level of commitment of the data and metadata via *committed*. If the server committed all data and metadata to stable storage, *committed* should be set to *FILE_SYNC*. If the level of commitment was at least as strong as *DATA_SYNC*, then *committed* should be set to *DATA_SYNC*. Otherwise, *committed* must be returned as *UNSTABLE*. If *stable* was *FILE_SYNC*, then *committed* must also be *FILE_SYNC*; anything else constitutes a protocol violation. If *stable* was *DATA_SYNC*, then committed may be *FILE_SYNC* or *DATA_SYNC*; anything else constitutes a protocol violation. If *stable* was *UNSTABLE*, then *committed* may be either *FILE_SYNC*, *DATA_SYNC* or *UNSTABLE*.

verf

This is a cookie that the client can use to determine whether the server has changed state between a call to *WRITE* and a subsequent call to either *WRITE* or *COMMIT*. This cookie must be consistent during a single instance of the NFS Version 3 protocol server and must be unique between instances of the NFS Version 3 protocol server, where uncommitted data may be lost.

Otherwise, *WRITE3res.status* contains the error on failure and *WRITE3res.resfail* contains the following:

file_wcc

Weak cache consistency data for the file. For a client that requires only the post-write file attributes, these can be found in *file_wcc.after*. Even though the write failed, full **wcc_data** is returned to allow the client to determine whether the failed write resulted in any change to the file.

If a client writes data to the server with the *stable* argument set to *UNSTABLE* and the reply yields a *committed* response of *DATA_SYNC* or *UNSTABLE*, the client will follow up some time in the future with a *COMMIT* operation to synchronise outstanding asynchronous data and metadata with the server's stable storage, barring client error. It is possible that due to client crash or other error that a subsequent *COMMIT* will not be received by the server.

Implementation Guidance

The **nfsdata** type used for the *READ* and *WRITE* operations in the NFS Version 2 protocol defining the data portion of a request or reply has been changed to a variable-length opaque byte array. The maximum size allowed by the protocol is now limited by what XDR and underlying transports will allow. There are no artificial limits imposed by the NFS Version 3 protocol. Consult the *FSINFO* procedure description in Section 12.4.0 on page 246 for details.

It is possible for the server to write fewer than *count* bytes of data. In this case, the server should not return an error unless no data was written at all. If the server writes less than *count* bytes, the client should issue another *WRITE* to write the remaining data.

It is assumed that the act of writing data to a file will cause the *mtime* of the file to be updated. However, the *mtime* of the file should not be changed unless the contents of the file are changed. Thus, a *WRITE* request with *count* set to zero should not cause the *mtime* of the file to be updated.

The NFS Version 3 protocol introduces safe asynchronous writes. The combination of *WRITE* with *stable* set to *UNSTABLE* followed by a *COMMIT* addresses the performance bottleneck found in the NFS Version 2 protocol, the need to synchronously commit all writes to stable storage.

The definition of stable storage has been historically a point of contention. The following expected properties of stable storage may help in resolving design issues in the implementation. Stable storage is persistent storage that survives:

- Repeated power failures.
- Hardware failures (of any board, power supply and so on).
- Repeated software crashes, including reboot cycle.

This definition does not address failure of the stable storage module itself.

A cookie, *verf*, is defined to allow a client to detect different instances of an NFS Version 3 protocol server over which cached, uncommitted data may be lost. In the most likely case, the *verf* allows the client to detect server reboots. This information is required so that the client can safely determine whether the server could have lost cached data. If the server fails unexpectedly and the client has uncommitted data from previous *WRITE* requests (done with the *stable* argument set to *UNSTABLE* and in which the result *committed* was returned as *UNSTABLE* as well) it may not have flushed cached data to stable storage. The burden of recovery is on the client and the client will need to retransmit the data to the server.

A suggested *verf* cookie would be to use the time that the server was booted or the time the server was last started (if restarting the server without a reboot results in lost buffers).

The *committed* field in the results allows the client to do more effective caching. If the server is committing all *WRITE* requests to stable storage, then it should return with *committed* set to *FILE_SYNC*, regardless of the value of the *stable* field in the arguments. A server that uses an NVRAM accelerator may choose to implement this policy. The client can use this to increase the effectiveness of the cache by discarding cached data that has already been committed on the server.

Some implementations may return *NFS3ERR_NOSPC* instead of *NFS3ERR_DQUOT* when a user's quota is exceeded.

Some NFS Version 2 protocol server implementations incorrectly returned *NFSERR_ISDIR* if the file system object type was not a regular file. The correct return value for the NFS Version 3 protocol is *NFS3ERR_INVAL*.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_FBIG File too large. The operation would have caused a file to grow beyond the server's limit.

NFS3ERR_DQUOT Resource (quota) hard limit exceeded. The user's resource limit on the server has been exceeded.

NFS3ERR_NOSPC No space left on device. The operation would have caused the server's file system to exceed its limit.

NFS3ERR_ROFS Read-only file system. A modifying operation was attempted on a read-only file system.

NFS3ERR_INVAL Invalid argument or unsupported argument for an operation. Two examples are attempting a *READLINK* on an object other than a symbolic link or attempting to *SETATTR* a time field on a server that does not support this operation.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_CREATE — Create a File

Call Arguments

```
enum createmode3 {
    UNCHECKED = 0,
    GUARDED = 1,
    EXCLUSIVE = 2
};

union createhow3 switch (createmode3 mode) {
    case UNCHECKED:
    case GUARDED:
        sattr3 obj_attributes;
    case EXCLUSIVE:
        createverf3 verf;
};

struct CREATE3args {
    diropargs3 where;
    createhow3 how;
};
```

Return Arguments

```
struct CREATE3resok {
    post_op_fh3 obj;
    post_op_attr obj_attributes;
    wcc_data dir_wcc;
};

struct CREATE3resfail {
    wcc_data dir_wcc;
};

union CREATE3res switch (nfsstat3 status) {
    case NFS3_OK:
        CREATE3resok resok;
    default:
        CREATE3resfail resfail;
};
```

RPC Procedure Description

```
CREATE3res
NFSPROC3_CREATE(CREATE3args) = 8;
```

Description

Procedure *CREATE* creates a regular file.

On entry, the arguments in *CREATE3args* are:

where The location of the file to be created:

dir The file handle for the directory in which the file is to be created.

name The name that is to be associated with the created file. See Section 12.2.4 on page 188 for more information on file names.

When creating a regular file, there are three ways to create the file as defined by:

how A discriminated union describing how the server is to handle the file creation along with the appropriate attributes:

mode One of *UNCHECKED*, *GUARDED* and *EXCLUSIVE*. *UNCHECKED* means that the server will create the file without checking for the existence of a duplicate file in the same directory. In this case, *how.obj_attributes* is a **sattr3** describing the initial attributes for the file. *GUARDED* specifies that the server will check for the presence of a duplicate file before performing the create and will fail the request with *NFS3ERR_EXIST* if a duplicate file exists. If the file does not exist, the request is performed as described for *UNCHECKED*. *EXCLUSIVE* specifies that the server is to follow exclusive creation semantics, using the verifier to ensure exclusive creation of the target. No attributes may be provided in this case, since the server may use the target file metadata to store the **createverf3** verifier.

Upon successful return, *CREATE3res.status* is *NFS3_OK* and the results in *CREATE3res.resok* are:

obj The file handle of the newly created regular file.

obj_attributes The attributes of the regular file just created.

dir_wcc Weak cache consistency data for the directory *where.dir*. For a client that requires on the post-*CREATE* directory attributes, these can be found in *dir_wcc.after*.

Otherwise, *CREATE3res.status* contains the error on failure and *CREATE3res.resfail* contains the following:

dir_wcc Weak cache consistency data for the directory *where.dir*. For a client that requires only the post-*CREATE* directory attributes, these can be found in *dir_wcc.after*. Even though the *CREATE* failed, full **wcc_data** is returned to allow the client to determine whether the failing *CREATE* resulted in any change to the directory.

Implementation Guidance

Unlike the NFS Version 2 protocol, in which certain fields in the initial attributes structure were overloaded to indicate creation of devices and FIFOs in addition to regular files, this procedure only supports the creation of regular files. The *MKNOD* procedure was introduced in the NFS Version 3 protocol to handle creation of devices and FIFOs. Implementations should have no reason in the NFS Version 3 protocol to overload *CREATE* semantics.

One aspect of the NFS Version 3 protocol *CREATE* procedure warrants particularly careful consideration: the mechanism introduced to support the reliable exclusive creation of regular files. The mechanism comes into play when *how.mode* is *EXCLUSIVE*. In this case, *how.verf* contains a verifier that can reasonably be expected to be unique. A combination of a client identifier, perhaps the client network address and a unique number generated by the client, perhaps the RPC transaction identifier, may be appropriate.

If the file does not exist, the server creates the file and stores the verifier in stable storage. For file systems that do not provide a mechanism for the storage of arbitrary file attributes, the server may use one or more elements of the file metadata to store the verifier. The verifier must be stored in stable storage to prevent erroneous failure on retransmission of the request. It is assumed that an exclusive create is being performed because exclusive semantics are critical to the application. Because of the expected usage, exclusive *CREATE* does not rely solely on the normally volatile duplicate request cache for storage of the verifier. The duplicate request cache in volatile storage does not survive a crash and may actually flush on a long network partition, opening failure windows. In the UNIX local file system environment, the expected storage location for the verifier on creation is the metadata (time stamps) of the file. For this reason, an exclusive file create may not include initial attributes because the server would have nowhere to store the verifier.

If the server can not support these exclusive create semantics, possibly because of the requirement to commit the verifier to stable storage, it should fail the *CREATE* request with the *NFS3ERR_NOTSUPP* error.

During an exclusive *CREATE* request, if the file already exists, the server reconstructs the file's verifier and compares it with the verifier in the request. If they match, the server treats the request as a success. The request is presumed to be a duplicate of an earlier, successful request for which the reply was lost and that the server duplicate request cache mechanism did not detect. If the verifiers do not match, the request is rejected with the status *NFS3ERR_EXIST*.

Once the client has performed a successful exclusive create, it must issue a *SETATTR* to set the correct file attributes. Until it does so, it should not rely upon any of the file attributes, since the server implementation may need to overload file metadata to store the verifier.

Use of the *GUARDED* attribute does not provide exactly-once semantics. In particular, if a reply is lost and the server does not detect the retransmission of the request, the procedure can fail with *NFS3ERR_EXIST*, even though the create was performed successfully.

See Section 12.2.4 on page 188 for more information on file names.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_EXIST File exists. The file specified already exists.

NFS3ERR_NOTDIR Not a directory. The caller specified a non-directory in a directory operation.

NFS3ERR_NOSPC No space left on device. The operation would have caused the server's file system to exceed its limit.

NFS3ERR_ROFS Read-only file system. A modifying operation was attempted on a read-only file system.

NFS3ERR_NAMETOOLONG
The filename in an operation was too long.

NFS3ERR_DQUOT Resource (quota) hard limit exceeded. The user's resource limit on the server has been exceeded.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_NOTSUPP
The operation is not supported.

NFS3ERR_SERVERFAULT

An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_MKDIR — Create a Directory

Call Arguments

```
struct MKDIR3args {
    diropargs3 where;
    sattr3 attributes;
};
```

Return Arguments

```
struct MKDIR3resok {
    post_op_fh3 obj;
    post_op_attr obj_attributes;
    wcc_data dir_wcc;
};

struct MKDIR3resfail {
    wcc_data dir_wcc;
};

union MKDIR3res switch (nfsstat3 status) {
    case NFS3_OK:
        MKDIR3resok resok;
    default:
        MKDIR3resfail resfail;
};
```

RPC Procedure Description

```
MKDIR3res
NFSPROC3_MKDIR(MKDIR3args) = 9;
```

Description

Procedure *MKDIR* creates a new subdirectory.

On entry, the arguments in *MKDIR3args* are:

where The location of the subdirectory to be created:

dir The file handle for the directory in which the subdirectory is to be created.

name The name that is to be associated with the created subdirectory. See Section 12.2.4 on page 188 for more information on file names.

attributes The initial attributes for the subdirectory.

Upon successful return, *MKDIR3res.status* is *NFS3_OK* and the results in *MKDIR3res.resok* are:

obj The file handle for the newly created directory.

obj_attributes The attributes for the newly created subdirectory.

dir_wcc Weak cache consistency data for the directory *where.dir*. For a client that requires only the post-*MKDIR* directory attributes, these can be found in *dir_wcc.after*.

Otherwise, *MKDIR3res.status* contains the error on failure and *MKDIR3res.resfail* contains the following:

dir_wcc Weak cache consistency data for the directory *where.dir*. For a client that requires only the post-*MKDIR* directory attributes, these can be found in *dir_wcc.after*. Even though the *MKDIR* failed, full **wcc_data** is returned to allow the client to determine whether the failing *MKDIR* resulted in any change to the directory.

Implementation Guidance

Many server implementations will not allow the filenames ''.'' or ''..'' to be used as targets in a *MKDIR* operation. In this case, the server should return *NFS3ERR_EXIST*. See Section 12.2.4 on page 188 for more information on file names.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_EXIST File exists. The file specified already exists.

NFS3ERR_NOTDIR Not a directory. The caller specified a non-directory in a directory operation.

NFS3ERR_NOSPC No space left on device. The operation would have caused the server's file system to exceed its limit.

NFS3ERR_ROFS Read-only file system. A modifying operation was attempted on a read-only file system.

NFS3ERR_NAMETOOLONG
The filename in an operation was too long.

NFS3ERR_DQUOT Resource (quota) hard limit exceeded. The user's resource limit on the server has been exceeded.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_NOTSUPP
The operation is not supported.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_SYMLINK — Create a Symbolic Link

Call Arguments

```
struct symlinkdata3 {
    sattr3 symlink_attributes;
    nfspath3 symlink_data;
};

struct SYMLINK3args {
    diropargs3 where;
    symlinkdata3 symlink;
};
```

Return Arguments

```
struct SYMLINK3resok {
    post_op_fh3 obj;
    post_op_attr obj_attributes;
    wcc_data dir_wcc;
};

struct SYMLINK3resfail {
    wcc_data dir_wcc;
};

union SYMLINK3res switch (nfsstat3 status) {
    case NFS3_OK:
        SYMLINK3resok resok;
    default:
        SYMLINK3resfail resfail;
};
```

RPC Procedure Description

```
SYMLINK3res
NFSPROC3_SYMLINK(SYMLINK3args) = 10;
```

Description

Procedure *SYMLINK* creates a new symbolic link.

On entry, the arguments in *SYMLINK3args* are:

where The location of the symbolic link to be created:

dir The file handle for the directory in which the symbolic link is to be created.

name The name that is to be associated with the created symbolic link. See Section 12.2.4 on page 188 for more information on file names.

symlink The symbolic link to create.

symlink_attributes
The initial attributes for the symbolic link.

symlink_data The string containing the symbolic link data.

Upon successful return, *SYMLINK3res.status* is *NFS3_OK* and *SYMLINK3res.resok* contains:

obj The file handle for the newly created symbolic link.

obj_attributes The attributes for the newly created symbolic link.

dir_wcc Weak cache consistency data for the directory *where.dir*. For a client that requires only the post-*SYMLINK* directory attributes, these can be found in *dir_wcc.after*.

Otherwise, *SYMLINK3res.status* contains the error on failure and *SYMLINK3res.resfail* contains the following:

dir_wcc Weak cache consistency data for the directory *where.dir*. For a client that requires only the post-*SYMLINK* directory attributes, these can be found in *dir_wcc.after*. Even though the *SYMLINK* failed, full **wcc_data** is returned to allow the client to determine whether the failing *SYMLINK* changed the directory.

Implementation Guidance

See Section 12.2.4 on page 188 for more information on file names.

For symbolic links, the actual file system node and its contents are expected to be created in a single atomic operation. That is, once the symbolic link is visible, there must not be a window where a *READLINK* would fail or return incorrect data.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_EXIST File exists. The file specified already exists.

NFS3ERR_NOTDIR Not a directory. The caller specified a non-directory in a directory operation.

NFS3ERR_NOSPC No space left on device. The operation would have caused the server's file system to exceed its limit.

NFS3ERR_ROFS Read-only file system. A modifying operation was attempted on a read-only file system.

NFS3ERR_NAMETOOLONG
The filename in an operation was too long.

NFS3ERR_DQUOT Resource (quota) hard limit exceeded. The user's resource limit on the server has been exceeded.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_NOTSUPP
The operation is not supported.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid

NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_MKNOD — Create a Special Device

Call Arguments

```
struct devicedata3 {
    sattr3 dev_attributes;
    specdata3 spec;
};

union mknoddata3 switch (ftype3 type) {
    case NF3CHR:
    case NF3BLK:
        devicedata3 device;
    case NF3SOCK:
    case NF3FIFO:
        sattr3 pipe_attributes;
    default:
        void;
};

struct MKNOD3args {
    diropargs3 where;
    mknoddata3 what;
};
```

Return Arguments

```
struct MKNOD3resok {
    post_op_fh3 obj;
    post_op_attr obj_attributes;
    wcc_data dir_wcc;
};

struct MKNOD3resfail {
    wcc_data dir_wcc;
};

union MKNOD3res switch (nfsstat3 status) {
    case NFS3_OK:
        MKNOD3resok resok;
    default:
        MKNOD3resfail resfail;
};
```

RPC Procedure Description

```
MKNOD3res
NFSPROC3_MKNOD(MKNOD3args) = 11;
```

Description

Procedure *MKNOD* creates a new special file of the type *what.type*. Special files can be device files or named pipes.

On entry, the arguments in *MKNOD3args* are:

where The location of the special file to be created:

dir The file handle for the directory in which the special file is to be created.

name The name that is to be associated with the created special file. See Section 12.2.4 on page 188 for more information on file names.

what A discriminated union identifying the type of the special file to be created along with the data and attributes appropriate to the type of the special file:

type The type of the object to be created.

When creating a character special file (*what.type* is *NF3CHR*) or a block special file (*what.type* is *NF3BLK*), *what* includes:

device A **devicedata3** structure with the following components:

dev_attributes The initial attributes for the special file.

spec The major number stored in *device.spec.specdata1* and the minor number stored in *device.spec.specdata2*.

When creating a socket (*what.type* is *NF3SOCK*) or a FIFO (*what.type* is *NF3FIFO*), *what* includes:

pipe_attributes The initial attributes for the special file.

Upon successful return, *MKNOD3res.status* is *NFS3_OK* and *MKNOD3res.resok* contains:

obj The file handle for the newly created special file.

obj_attributes The attributes for the newly created special file.

dir_wcc Weak cache consistency data for the directory *where.dir*. For a client that requires only the post-*MKNOD* directory attributes, these can be found in *dir_wcc.after*.

Otherwise, *MKNOD3res.status* contains the error on failure and *MKNOD3res.resfail* contains the following:

dir_wcc Weak cache consistency data for the directory *where.dir*. For a client that requires only the post-*MKNOD* directory attributes, these can be found in *dir_wcc.after*. Even though the *MKNOD* failed, full **wcc_data** is returned to allow the client to determine whether the failing *MKNOD* changed the directory.

Implementation Guidance

See Section 12.2.4 on page 188 for more information on file names.

Without explicit support for special file type creation in the NFS Version 2 protocol, fields in the *CREATE* arguments were overloaded to indicate creation of certain types of objects. This overloading is not necessary in the NFS Version 3 protocol.

If the server does not support any of the defined types, the *NFS3ERR_NOTSUPP* error should be returned. Otherwise, if the server does not support the target type or the target type is invalid, the *NFS3ERR_BADTYPE* error should be returned. Note that *NF3REG*, *NF3DIR* and *NF3LNK* are invalid types for *MKNOD*. The procedures, *CREATE*, *MKDIR* and *SYMLINK* should be used to create these file types, respectively, instead of *MKNOD*.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_EXIST File exists. The file specified already exists.

NFS3ERR_NOTDIR Not a directory. The caller specified a non-directory in a directory operation.

NFS3ERR_NOSPC No space left on device. The operation would have caused the server's file system to exceed its limit.

NFS3ERR_ROFS Read-only file system. A modifying operation was attempted on a read-only file system.

NFS3ERR_NAMETOOLONG
The filename in an operation was too long.

NFS3ERR_DQUOT Resource (quota) hard limit exceeded. The user's resource limit on the server has been exceeded.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_NOTSUPP
The operation is not supported.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

NFS3ERR_BADTYPE
An attempt was made to create an object of a type not supported by the server.

Name

NFSPROC3_REMOVE — Remove a File

Call Arguments

```
struct REMOVE3args {
    diropargs3 object;
};
```

Return Arguments

```
struct REMOVE3resok {
    wcc_data dir_wcc;
};

struct REMOVE3resfail {
    wcc_data dir_wcc;
};

union REMOVE3res switch (nfsstat3 status) {
    case NFS3_OK:
        REMOVE3resok resok;
    default:
        REMOVE3resfail resfail;
};
```

RPC Procedure Description

```
REMOVE3res
NFSPROC3_REMOVE(REMOVE3args) = 12;
```

Description

Procedure *REMOVE* removes (deletes) an entry from a directory. If the entry in the directory was the last reference to the corresponding file system object, the object may be destroyed.

On entry, the arguments in *REMOVE3args* are:

object A *diropargs3* structure identifying the entry to be removed:

dir The file handle for the directory from which the entry is to be removed.

name The name of the entry to be removed. See Section 12.2.4 on page 188 for more information on file names.

Upon successful return, *REMOVE3res.status* is *NFS3_OK* and *REMOVE3res.resok* contains:

dir_wcc Weak cache consistency data for the directory *object.dir*. For a client that requires only the post-*REMOVE* directory attributes, these can be found in *dir_wcc.after*.

Otherwise, *REMOVE3res.status* contains the error on failure and *REMOVE3res.resfail* contains the following:

dir_wcc Weak cache consistency data for the directory *object.dir*. For a client that requires only the post-*REMOVE* directory attributes, these can be found in *dir_wcc.after*.

Even though the *REMOVE* failed, full **wcc_data** is returned to allow the client to determine whether the failing *REMOVE* changed the directory.

Implementation Guidance

In general, *REMOVE* is intended to remove non-directory file objects and *RMDIR* is to be used to remove directories. However, *REMOVE* can be used to remove directories, subject to restrictions imposed by either the client or server interfaces. This had been a source of confusion in the NFS Version 2 protocol.

The concept of last reference is server specific. However, if the *nlink* field in the previous attributes of the object had the value 1, the client should not rely on referring to the object via a file handle. Likewise, the client should not rely on the resources (disk space, directory entry and so on) formerly associated with the object becoming immediately available. Thus, if a client needs to be able to continue to access a file after using *REMOVE* to remove it, the client must take steps to make sure that the file will still be accessible. The usual mechanism used is to use *RENAME* to rename the file from its old name to a new hidden name.

See Section 12.2.4 on page 188 for more information on file names.

Return Codes

NFS3ERR_NOENT No such file or directory. The file or directory name specified does not exist.

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_NOTDIR Not a directory. The caller specified a non-directory in a directory operation.

NFS3ERR_NAMETOOLONG
The filename in an operation was too long.

NFS3ERR_ROFS Read-only file system. A modifying operation was attempted on a read-only file system.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_RMDIR — Remove a Directory

Call Arguments

```
struct RMDIR3args {
    diropargs3 object;
};
```

Return Arguments

```
struct RMDIR3resok {
    wcc_data dir_wcc;
};

struct RMDIR3resfail {
    wcc_data dir_wcc;
};

union RMDIR3res switch (nfsstat3 status) {
    case NFS3_OK:
        RMDIR3resok resok;
    default:
        RMDIR3resfail resfail;
};
```

RPC Procedure Description

```
RMDIR3res
NFSPROC3_RMDIR(RMDIR3args) = 13;
```

Description

Procedure *RMDIR* removes (deletes) a subdirectory from a directory. If the directory entry of the subdirectory is the last reference to the subdirectory, the subdirectory may be destroyed.

On entry, the arguments in *RMDIR3args* are:

object A *diropargs3* structure identifying the directory entry to be removed:

- *dir* The file handle for the directory from which the subdirectory is to be removed.
- *name* The name of the subdirectory to be removed. See Section 12.2.4 on page 188 for more information on file names.

Upon successful return, *RMDIR3res.status* is *NFS3_OK* and *RMDIR3res.resok* contains:

dir_wcc Weak cache consistency data for the directory *object.dir*. For a client that requires only the post-*RMDIR* directory attributes, these can be found in *dir_wcc.after*.

Otherwise, *RMDIR3res.status* contains the error on failure and *RMDIR3res.resfail* contains the following:

dir_wcc Weak cache consistency data for the directory *object.dir*. For a client that requires only the post-*RMDIR* directory attributes, these can be found in *dir_wcc.after*.

Note that even though the *RMDIR* failed, full **wcc_data** is returned to allow the client to determine whether the failing *RMDIR* changed the directory.

Implementation Guidance

Note that on some servers, removal of a non-empty directory is disallowed.

On some servers, the filename ‘‘.’’ is invalid. These servers will return the *NFS3ERR_INVAL* error. On some servers, the filename ‘‘..’’ is invalid. These servers will return the *NFS3ERR_EXIST* error. This would seem inconsistent, but allows these servers to comply with their own specific interface definitions. Clients must be prepared to handle both cases.

The client should not rely on the resources (disk space, directory entry and so on) formerly associated with the directory becoming immediately available.

Return Codes

NFS3ERR_NOENT No such file or directory. The file or directory name specified does not exist.

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_INVAL Invalid argument or unsupported argument for an operation. Two examples are attempting a *READLINK* on an object other than a symbolic link or attempting to *SETATTR* a time field on a server that does not support this operation.

NFS3ERR_EXIST File exists. The file specified already exists.

NFS3ERR_NOTDIR Not a directory. The caller specified a non-directory in a directory operation.

NFS3ERR_NAMETOOLONG
The filename in an operation was too long.

NFS3ERR_ROFS Read-only file system. A modifying operation was attempted on a read-only file system.

NFS3ERR_NOTEMPTY
An attempt was made to remove a directory that was not empty.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_NOTSUPP
The operation is not supported.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_RENAME — Rename a File or Directory

Call Arguments

```
struct RENAME3args {
    diropargs3 from;
    diropargs3 to;
};
```

Return Arguments

```
struct RENAME3resok {
    wcc_data fromdir_wcc;
    wcc_data todir_wcc;
};

struct RENAME3resfail {
    wcc_data fromdir_wcc;
    wcc_data todir_wcc;
};

union RENAME3res switch (nfsstat3 status) {
    case NFS3_OK:
        RENAME3resok resok;
    default:
        RENAME3resfail resfail;
};
```

RPC Procedure Description

```
RENAME3res
NFSPROC3_RENAME(RENAME3args) = 14;
```

Description

Procedure *RENAME* renames the file identified by *from.name* in the directory *from.dir*, to *to.name* in the directory *to.dir*. The operation is required to be atomic to the client. The *to.dir* and *from.dir* must reside on the same file system and server (in other words, the *fsid* fields in the attributes for the directories must be the same).

On entry, the arguments in *RENAME3args* are:

from A *diropargs3* structure identifying the source (the file system object to be renamed):

from.dir The file handle for the directory from which the entry is to be renamed.

from.name The name of the entry that identifies the object to be renamed. See Section 12.2.4 on page 188 for more information on file names.

to A *diropargs3* structure identifying the target (the new name of the object):

to.dir The file handle for the directory to which the object is to be renamed.

to.name The new name for the object. See Section 12.2.4 on page 188 for more information on file names.

If the directory *to.dir* already contains an entry with the name *to.name* the source object must be compatible with the target: either both are non-directories or both are directories and the target must be empty. If compatible, the existing target is removed before the rename occurs. If they are not compatible or if the target is a directory but not empty, the server should return the

NFS3ERR_EXIST error.

Upon successful return, *RENAME3res.status* is *NFS3_OK* and *RENAME3res.resok* contains:

fromdir_wcc Weak cache consistency data for the directory *from.dir*.

todir_wcc Weak cache consistency data for the directory *to.dir*.

Otherwise, *RENAME3res.status* contains the error on failure and *RENAME3res.resfail* contains the following:

fromdir_wcc Weak cache consistency data for the directory *from.dir*.

todir_wcc Weak cache consistency data for the directory *to.dir*.

Implementation Guidance

If *to.dir* and *from.dir* reside on different file systems, the *NFS3ERR_XDEV* error is returned. Even though the operation is atomic, the status *NFS3ERR_MLINK* may be returned if the server used a ‘‘unlink/link/unlink’’ sequence internally.

A file handle may or may not become stale on a rename. However, server implementors are strongly encouraged to attempt to keep file handles from becoming stale in this fashion.

On some servers, the filenames ‘‘.’’ and ‘‘..’’ are invalid as either *from.name* or *to.name*. In addition, neither *from.name* nor *to.name* can be an alias for *from.dir*. These servers will return the *NFS3ERR_INVAL* error in these cases.

See Section 12.2.4 on page 188 for more information on file names.

Return Codes

NFS3ERR_NOENT No such file or directory. The file or directory name specified does not exist.

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_EXIST File exists. The file specified already exists.

NFS3ERR_XDEV The caller attempted to do a cross-device hard link.

NFS3ERR_NOTDIR Not a directory. The caller specified a non-directory in a directory operation.

NFS3ERR_ISDIR Is a directory. The caller specified a directory in a non-directory operation.

NFS3ERR_INVAL Invalid argument or unsupported argument for an operation. Two examples are attempting a *READLINK* on an object other than a symbolic link or attempting to *SETATTR* a time field on a server that does not support this operation.

NFS3ERR_NOSPC No space left on device. The operation would have caused the server’s file system to exceed its limit.

NFS3ERR_ROFS Read-only file system. A modifying operation was attempted on a read-only file system.

NFS3ERR_MLINK Too many hard links.

NFS3ERR_NAMETOOLONG
The filename in an operation was too long.

NFS3ERR_NOTEMPTY
An attempt was made to remove a directory that was not empty.

NFS3ERR_DQUOT Resource (quota) hard limit exceeded. The user's resource limit on the server has been exceeded.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_NOTSUPP
The operation is not supported.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_LINK — Create Link to an Object

Call Arguments

```
struct LINK3args {
    nfs_fh3 file;
    diropargs3 link;
};
```

Return Arguments

```
struct LINK3resok {
    post_op_attr file_attributes;
    wcc_data linkdir_wcc;
};

struct LINK3resfail {
    post_op_attr file_attributes;
    wcc_data linkdir_wcc;
};

union LINK3res switch (nfsstat3 status) {
    case NFS3_OK:
        LINK3resok resok;
    default:
        LINK3resfail resfail;
};
```

RPC Procedure Description

```
LINK3res
NFSPROC3_LINK(LINK3args) = 15;
```

Description

Procedure *LINK* creates a hard link from *file* to *link.name*, in the directory *link.dir*. Both *file* and *link.dir* must reside on the same file system and server.

On entry, the arguments in *LINK3args* are:

file The file handle for the existing file system object.

link The location of the link to be created:

link.dir The file handle for the directory in which the link is to be created.

link.name The name that is to be associated with the created link. See Section 12.2.4 on page 188 for more information on file names.

Upon successful return, *LINK3res.status* is *NFS3_OK* and *LINK3res.resok* contains:

file_attributes
The post-operation attributes of the file system object identified by *file*.

linkdir_wcc Weak cache consistency data for the directory *link.dir*.

Otherwise, *LINK3res.status* contains the error on failure and *LINK3res.resfail* contains the following:

file_attributes
The post-operation attributes of the file system object identified by *file*.

linkdir_wcc Weak cache consistency data for the directory *link.dir*.

Implementation Guidance

Changes to any property of the hard-linked files are reflected in all of the linked files. When a hard link is made to a file, the attributes for the file should have a value for *nlink* that is one greater than the value before the *LINK*.

The comments under *RENAME* regarding object and target residing on the same file system apply here as well. The comments regarding the target name applies as well. See Section 12.2.4 on page 188 for more information on file names.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_EXIST File exists. The file specified already exists.

NFS3ERR_XDEV The caller attempted to do a cross-device hard link.

NFS3ERR_NOTDIR Not a directory. The caller specified a non-directory in a directory operation.

NFS3ERR_INVAL Invalid argument or unsupported argument for an operation. Two examples are attempting a *READLINK* on an object other than a symbolic link or attempting to *SETATTR* a time field on a server that does not support this operation.

NFS3ERR_NOSPC No space left on device. The operation would have caused the server's file system to exceed its limit.

NFS3ERR_ROFS Read-only file system. A modifying operation was attempted on a read-only file system.

NFS3ERR_MLINK Too many hard links.

NFS3ERR_NAMETOOLONG
The filename in an operation was too long.

NFS3ERR_DQUOT Resource (quota) hard limit exceeded. The user's resource limit on the server has been exceeded.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_NOTSUPP
The operation is not supported.

NFS3ERR_SERVERFAULT

An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_READDIR — Read From Directory

Call Arguments

```
struct READDIR3args {
    nfs_fh3 dir;
    cookie3 cookie;
    cookieverf3 cookieverf;
    count3 count;
};
```

Return Arguments

```
struct entry3 {
    fileid3 fileid;
    filename3 name;
    cookie3 cookie;
    entry3 *nextentry;
};

struct dirlist3 {
    entry3 *entries;
    bool eof;
};

struct READDIR3resok {
    post_op_attr dir_attributes;
    cookieverf3 cookieverf;
    dirlist3 reply;
};

struct READDIR3resfail {
    post_op_attr dir_attributes;
};

union READDIR3res switch (nfsstat3 status) {
    case NFS3_OK:
        READDIR3resok resok;
    default:
        READDIR3resfail resfail;
};
```

RPC Procedure Description

```
READDIR3res
NFSPROC3_READDIR(READDIR3args) = 16;
```

Description

Procedure *READDIR* retrieves a variable number of entries, in sequence, from a directory and returns the name and file identifier for each, with information to allow the client to request additional directory entries in a subsequent *READDIR* request.

On entry, the arguments in *READDIR3args* are:

dir The file handle for the directory to be read.

cookie An opaque value identifying a point in a directory The client sets it to zero in the first request to read the directory. On subsequent requests, it should be a *cookie* as returned by the server.

cookieverf An opaque value verifying the value of the *cookie*. The client sets it to zero in the first request to read the directory. On subsequent requests, it should be a *cookieverf*, as returned by the server. The *cookieverf* must match that returned by the *READDIR* in which the cookie was acquired.

count The maximum size of the *READDIR3resok* structure, in bytes. The size must include all XDR overhead. The server may return fewer than *count* bytes of data.

Upon successful return, *READDIR3res.status* is *NFS3_OK* and *READDIR3res.resok* contains:

dir_attributes The attributes of the directory *dir*.

cookieverf The cookie verifier.

reply The directory list:

entries Zero or more directory (*entry3*) entries.

eof *TRUE* if the last member of *reply.entries* is the last entry in the directory or the list *reply.entries* is empty and the cookie corresponded to the end of the directory. If *FALSE*, there may be more entries to read.

Otherwise, *READDIR3res.status* contains the error on failure and *READDIR3res.resfail* contains the following:

dir_attributes The attributes of the directory *dir*.

Implementation Guidance

In the NFS Version 2 protocol, each directory entry returned included a cookie identifying a point in the directory. By including this cookie in a subsequent *READDIR*, the client could resume the directory read at any point in the directory. One problem with this scheme was that there was no easy way for a server to verify that a cookie was valid. If two *READDIR*s were separated by one or more operations that changed the directory in some way (for example, reordering or compressing it), it was possible that the second *READDIR* could miss entries, or process entries more than once. If the cookie was no longer usable, for example, pointing into the middle of a directory entry, the server would have to either round the cookie down to the cookie of the previous entry or round it up to the cookie of the next entry in the directory. Either way would possibly lead to incorrect results and the client would be unaware that any problem existed.

In the NFS Version 3 protocol, each *READDIR* request includes both a cookie and a cookie verifier. For the first call, both are set to zero. The response includes a new cookie verifier, with a cookie per entry. For subsequent *READDIR*s, the client must present both the cookie and the corresponding cookie verifier. If the server detects that the cookie is no longer valid, the server will reject the *READDIR* request with the status *NFS3ERR_BAD_COOKIE*. The client should be careful to avoid holding directory entry cookies across operations that modify the directory contents, such as *REMOVE* and *CREATE*.

One implementation of the cookie-verifier mechanism might be for the server to use the modification time of the directory, but this might be overly restrictive. A better approach would be to record the time of the last directory modification that changed the directory organisation in a way that would make it impossible to interpret a cookie reliably. Servers in which directory cookies are always valid are free to use zero as the verifier always.

The server may return fewer than *count* bytes of XDR-encoded entries. The *count* specified by the client in the request should be greater than or equal to *FSINFO* dtpref.

Since UNIX clients give a special meaning to the *fileid* value zero, UNIX clients should be careful to map zero *fileid* values to some other value and servers should try to avoid sending a zero *fileid*.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_NOTDIR Not a directory. The caller specified a non-directory in a directory operation.

NFS3ERR_BAD_COOKIE
A *READDIR* or *READDIRPLUS* cookie is stale.

NFS3ERR_TOOSMALL
The buffer or request is too small.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_READDIRPLUS — Extended Read From Directory

Call Arguments

```
struct READDIRPLUS3args {
    nfs_fh3 dir;
    cookie3 cookie;
    cookieverf3 cookieverf;
    count3 dircount;
    count3 maxcount;
};
```

Return Arguments

```
struct entryplus3 {
    fileid3 fileid;
    filename3 name;
    cookie3 cookie;
    post_op_attr name_attributes;
    post_op_fh3 name_handle;
    entryplus3 *nextentry;
};

struct dirlistplus3 {
    entryplus3 *entries;
    bool eof;
};

struct READDIRPLUS3resok {
    post_op_attr dir_attributes;
    cookieverf3 cookieverf;
    dirlistplus3 reply;
};

struct READDIRPLUS3resfail {
    post_op_attr dir_attributes;
};

union READDIRPLUS3res switch (nfsstat3 status) {
    case NFS3_OK:
        READDIRPLUS3resok resok;
    default:
        READDIRPLUS3resfail resfail;
};
```

RPC Procedure Description

```
READDIRPLUS3res
NFSPROC3_READDIRPLUS(READDIRPLUS3args) = 17;
```

Description

Procedure *READDIRPLUS* retrieves a variable number of entries from a file system directory and returns complete information about each along with information to allow the client to request additional directory entries in a subsequent *READDIRPLUS*. *READDIRPLUS* differs from *READDIR* only in the amount of information returned for each entry. In *READDIR*, each entry returns the filename and the fileid. In *READDIRPLUS*, each entry returns the name, the *fileid*, attributes (including the *fileid*) and file handle.

On entry, the arguments in *READDIRPLUS3args* are:

dir The file handle for the directory to be read.

cookie An opaque value identifying a point in a directory The client sets it to zero in the first request to read the directory. On subsequent requests, it should be a *cookie* as returned by the server.

cookieverf An opaque value verifying the value of the *cookie*. The client sets it to zero in the first request to read the directory. On subsequent requests, it should be a *cookieverf*, as returned by the server. The *cookieverf* must match that returned by the *READDIRPLUS* call in which the cookie was acquired.

dircount The maximum number of bytes of directory information to be returned. This number does not include the size of the attributes and file handle portions of the result.

maxcount The maximum size of the *READDIRPLUS3resok* structure, in bytes. The size must include all XDR overhead. The server may return fewer than *maxcount* bytes of data.

Upon successful return, *READDIRPLUS3res.status* is *NFS3_OK* and *READDIRPLUS3res.resok* contains:

dir_attributes The attributes of the directory *dir*.

cookieverf The cookie verifier.

reply The directory list:

entries Zero or more directory (*entryplus3*) entries.

eof *TRUE* if the last member of *reply.entries* is the last entry in the directory or the list *reply.entries* is empty and the cookie corresponded to the end of the directory. If *FALSE*, there may be more entries to read.

Otherwise, *READDIRPLUS3res.status* contains the error on failure and *READDIRPLUS3res.resfail* contains the following:

dir_attributes The attributes of the directory *dir*.

Implementation Guidance

Issues that need to be understood for this procedure include increased cache flushing activity on the client (as new file handles are returned with names that are entered into caches) and over-the-wire overhead versus expected subsequent LOOKUP elimination. This procedure may improve performance for directory browsing where attributes are always required (such as for the Apple Macintosh operating system and for MS-DOS).

The *dircount* and *maxcount* fields are included as an optimisation. Consider a *READDIRPLUS* call on a UNIX operating system implementation for 1048 bytes; the reply does not contain many entries because of the overhead due to attributes and file handles. An alternative is to issue a *READDIRPLUS* call for 8192 bytes and then only use the first 1048 bytes of directory information. However, the server doesn't know that all that is needed is 1048 bytes of directory information (as would be returned by *READDIR*). It sees the 8192 byte request and issues a *VOP_READDIR* for 8192 bytes. It then steps through all of those directory entries, obtaining attributes and file handles for each entry. When it encodes the result, the server only encodes until it gets 8192 bytes of results, which include the attributes and file handles. Thus, it has done a larger *VOP_READDIR* and many more attribute fetches than it needed to. The ratio of the directory entry size to the size of the attributes plus the size of the file handle is usually at least 8 to 1. The server has done much more work than it needed to.

The solution to this problem is for the client to provide two counts to the server. The first is the number of bytes of directory information that the client really wants, *dircount*. The second is the maximum number of bytes in the result, including the attributes and file handles, *maxcount*. Thus, the server will issue a *VOP_READDIR* for only the number of bytes that the client really wants to get, not an inflated number. This should help to reduce the size of *VOP_READDIR* requests on the server, thus reducing the amount of work done there, and to reduce the number of *VOP_LOOKUP*, *VOP_GETATTR* and other calls done by the server to construct attributes and file handles.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_ACCES Permission denied. The caller does not have the correct permission to perform the requested operation. Contrast this with *NFS3ERR_PERM*, which restricts itself to owner permission failures.

NFS3ERR_NOTDIR Not a directory. The caller specified a non-directory in a directory operation.

NFS3ERR_BAD_COOKIE
A *READDIR* or *READDIRPLUS* cookie is stale.

NFS3ERR_TOOSMALL
The buffer or request is too small.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_NOTSUPP
The operation is not supported.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_FSSTAT — Get Dynamic File System Information

Call Arguments

```
struct FSSTAT3args {
    nfs_fh3 fsroot;
};
```

Return Arguments

```
struct FSSTAT3resok {
    post_op_attr obj_attributes;
    size3 tbytes;
    size3 fbytes;
    size3 abytes;
    size3 tfiles;
    size3 ffiles;
    size3 afiles;
    uint32 invarsec;
};

struct FSSTAT3resfail {
    post_op_attr obj_attributes;
};

union FSSTAT3res switch (nfsstat3 status) {
    case NFS3_OK:
        FSSTAT3resok resok;
    default:
        FSSTAT3resfail resfail;
};
```

RPC Procedure Description

```
FSSTAT3res
NFSPROC3_FSSTAT(FSSTAT3args) = 18;
```

Description

Procedure *FSSTAT* retrieves volatile file system state information.

On entry, the arguments in *FSSTAT3args* are:

fsroot A file handle identifying a object in the file system. This is normally a file handle for a mount point for a file system, as originally obtained from the *MOUNT* service on the server.

Upon successful return, *FSSTAT3res.status* is *NFS3_OK* and *FSSTAT3res.resok* contains:

obj_attributes The attributes of the file system object specified in *fsroot*.

tbytes The total size, in bytes, of the file system.

fbytes The amount of free space, in bytes, in the file system.

abytes The amount of free space, in bytes, available to the user identified by the authentication information in the RPC. (This reflects space that is reserved by the file system; it does not reflect any quota system implemented by the server.)

tfiles The total number of file slots in the file system. (On a UNIX server, this often corresponds to the number of i-nodes configured.)

ffiles The number of free file slots in the file system.

afiles The number of free file slots that are available to the user corresponding to the authentication information in the RPC. (This reflects slots that are reserved by the file system; it does not reflect any quota system implemented by the server.)

invarsec A measure of file system volatility—the number of seconds for which the file system is not expected to change. For a volatile, frequently updated file system, this will be zero. For an immutable file system, such as a CD-ROM, this would be the largest unsigned integer. For file systems that are infrequently modified (for example, one containing local executable programs and on-line documentation), a value corresponding to a few hours or days might be used. The client may use this as a hint in tuning its cache management. Note, however, that this measure is assumed to be dynamic and may change at any time.

Otherwise, *FSSTAT3res.status* contains the error on failure and *FSSTAT3res.resfail* contains the following:

obj_attributes The attributes of the file system object specified in *fsroot*.

Implementation Guidance

Not all implementations can support the entire list of attributes. It is expected that servers will make a best effort at supporting all the attributes.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_FSINFO — Get Staticfile System Information

Call Arguments

```
struct FSINFO3args {
    nfs_fh3 fsroot;
};
```

Return Arguments

```
const FSF3_LINK = 0x0001;
const FSF3_SYMLINK = 0x0002;
const FSF3_HOMOGENEOUS = 0x0008;
const FSF3_CANSETTIME = 0x0010;

struct FSINFO3resok {
    post_op_attr obj_attributes;
    uint32 rtmax;
    uint32 rtpref;
    uint32 rtmult;
    uint32 wtmax;
    uint32 wtpref;
    uint32 wtmult;
    uint32 dtpref;
    size3 maxfilesize;
    nfstime3 time_delta;
    uint32 properties;
};

struct FSINFO3resfail {
    post_op_attr obj_attributes;
};

union FSINFO3res switch (nfsstat3 status) {
    case NFS3_OK:
        FSINFO3resok resok;
    default:
        FSINFO3resfail resfail;
};
```

RPC Procedure Description

```
FSINFO3res
NFSPROC3_FSINFO(FSINFO3args) = 19;
```

Description

Procedure *FSINFO* retrieves nonvolatile file system state information and general information about the NFS Version 3 protocol server implementation.

On entry, the arguments in *FSINFO3args* are:

fsroot A file handle identifying a file object. Normal usage is to provide a file handle for a mount point for a file system, as originally obtained from the *MOUNT* service on the server.

Upon successful return, *FSINFO3res.status* is *NFS3_OK* and *FSINFO3res.resok* contains:

obj_attributes The attributes of the file system object specified in *fsroot*.

rtmax The maximum size in bytes of a *READ* request supported by the server. Any *READ* with a number greater than *rtmax* will result in a short read of *rtmax* bytes or less.

rtpref The preferred size in bytes of a *READ* request. This should be the same as *rtmax* unless there is a clear benefit in performance or efficiency.

rtmult The suggested multiple for the size of a *READ* request.

wtmax The maximum size in bytes of a *WRITE* request supported by the server. In general, the client is limited by *wtmax* since there is no guarantee that a server can handle a larger write. Any *WRITE* with a *count* greater than *wtmax* will result in a short write of at most *wtmax* bytes.

wtpref The preferred size in bytes of a *WRITE* request. This should be the same as *wtmax* unless there is a clear benefit in performance or efficiency.

wtmult The suggested multiple for the size of a *WRITE* request.

dtpref The preferred size of a *READDIR* request.

maxfilesize The maximum size in bytes of a file on the file system.

time_delta The server time granularity. When setting a file time using *SETATTR*, the server guarantees only to preserve times to this accuracy. If this is { 0, 1 }, the server can support nanosecond times, { 0, 1000000 } denotes millisecond precision and { 1, 0 } indicates that times are accurate only to the nearest second.

properties A bit mask of file system properties. The following values are defined:

FSF_LINK
If this bit is 1 (*TRUE*), the file system supports hard links.

FSF_SYMLINK
If this bit is 1 (*TRUE*), the file system supports symbolic links.

FSF_HOMOGENEOUS
If this bit is 1 (*TRUE*), the information returned by *PATHCONF* is identical for every file and directory in the file system. If it is zero (*FALSE*), the client should retrieve *PATHCONF* information for each file and directory as required.

FSF_CANSETTIME
If this bit is 1 (*TRUE*), the server will set the times for a file via *SETATTR* if requested (to the accuracy indicated by *time_delta*). If it is zero (*FALSE*), the server cannot set times as requested.

Otherwise, *FSINFO3res.status* contains the error on failure and *FSINFO3res.resfail* contains the following:

attributes The attributes of the file system object specified in *fsroot*.

Implementation Guidance

Not all implementations can support the entire list of attributes. It is expected that a server will make a best effort at supporting all the attributes.

The file handle provided is expected to be the file handle of the file system root, as returned to the *MOUNT* operation. Since mounts may occur anywhere within an exported tree, the server should expect *FSINFO* requests specifying file handles within the exported file system. A server may export different types of file systems with different attributes returned to the *FSINFO* call. The client should retrieve *FSINFO* information for each mount completed. Though a server may return different *FSINFO* information for different files within a file system, there is no requirement that a client obtain *FSINFO* information for other than the file handle returned at mount.

The *maxfilesize* field determines whether a server's particular file system uses 32 bit sizes and offsets or 64 bit file sizes and offsets. This may affect a client's processing.

The preferred sizes for requests are nominally tied to an exported file system mounted by a client. A surmountable issue arises in that the transfer size for an NFS Version 3 protocol request is not only dependent on characteristics of the file system but also on characteristics of the network interface, particularly the maximum transfer unit (MTU). A server implementation can advertise different transfer sizes (for the fields *rtmax*, *rtpref*, *wtmax*, *wtpref* and *dtpref*) depending on the interface on which the *FSINFO* request is received. This is an implementation issue.

Return Codes

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_PATHCONF — Retrieve XPG4 Information

Call Arguments

```
struct PATHCONF3args {
    nfs_fh3 object;
};
```

Return Arguments

```
struct PATHCONF3resok {
    post_op_attr obj_attributes;
    uint32 linkmax;
    uint32 name_max;
    bool no_trunc;
    bool chown_restricted;
    bool case_insensitive;
    bool case_preserving;
};

struct PATHCONF3resfail {
    post_op_attr obj_attributes;
};

union PATHCONF3res switch (nfsstat3 status) {
    case NFS3_OK:
        PATHCONF3resok resok;
    default:
        PATHCONF3resfail resfail;
};
```

RPC Procedure Description

```
PATHCONF3res
NFSPROC3_PATHCONF(PATHCONF3args) = 20;
```

Description

Procedure *PATHCONF* retrieves the XPG4 *pathconf*( ) information for a file or directory. If the *FSF_HOMOGENEOUS* bit is set in *FSFINFO3resok.properties*, the *pathconf*( ) information will be the same for all files and directories in the exported file system in which this file or directory resides.

On entry, the arguments in *PATHCONF3args* are:

object The file handle for the file system object.

Upon successful return, *PATHCONF3res.status* is *NFS3_OK* and *PATHCONF3res.resok* contains:

obj_attributes The attributes of the object specified by *object*.

linkmax The maximum number of hard links to an object.

name_max The maximum length in bytes of a filename (pathname component).

no_trunc If *TRUE*, the server will reject any request that includes a name longer than *name_max* with the *NFS3ERR_NAMETOOLONG* error. If *FALSE*, any name over *name_max* bytes will be silently truncated to *name_max* bytes.

chown_restricted
If *TRUE*, the server will reject any request to change either the owner or the

group associated with a file if the caller does not have the appropriate privileges.

case_insensitive
If *TRUE*, the server file system does not distinguish case when interpreting filenames.

case_preserving
If *TRUE*, the server file system will preserve the case of a name during a *CREATE*, *MKDIR*, *MKNOD*, *SYMLINK*, *RENAME* or *LINK* operation.

Otherwise, *PATHCONF3res.status* contains the error on failure and *PATHCONF3res.resfail* contains the following:

obj_attributes The attributes of the object specified by *object*.

Implementation Guidance

In some implementations of the NFS Version 2 protocol, *pathconf*( ) information was obtained at mount time through the *MOUNT* protocol. The proper place to obtain it is as here, in the NFS Version 3 protocol itself.

Return Codes

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Name

NFSPROC3_COMMIT — Commit Cached Server Data to Stable Storage

Call Arguments

```
struct COMMIT3args {
    nfs_fh3 file;
    offset3 offset;
    count3 count;
};
```

Return Arguments

```
struct COMMIT3resok {
    wcc_data file_wcc;
    writeverf3 verf;
};

struct COMMIT3resfail {
    wcc_data file_wcc;
};

union COMMIT3res switch (nfsstat3 status) {
    case NFS3_OK:
        COMMIT3resok resok;
    default:
        COMMIT3resfail resfail;
};
```

RPC Procedure Description

```
COMMIT3res
NFSPROC3_COMMIT(COMMIT3args) = 21;
```

Description

Procedure *COMMIT* forces or flushes to stable storage data that was previously written with a *WRITE* procedure call with the *stable* field set to *UNSTABLE*.

On entry, the arguments in *COMMIT3args* are:

file The file handle for the file to which data is to be flushed (committed). This must identify a file system object of type *NF3REG*.

offset The position within the file at which the flush is to begin. An *offset* of zero means to flush data starting at the beginning of the file.

count The number of bytes of data to flush. If *count* is zero, a flush from *offset* to the end-of-file is done.

Upon successful return, *COMMIT3res.status* is *NFS3_OK* and *COMMIT3res.resok* contains:

file_wcc Weak cache consistency data for the file. For a client that requires only the post-operation file attributes, these can be found in *file_wcc.after*.

verf This is a cookie that the client can use to determine whether the server has rebooted between a call to *WRITE* and a subsequent call to *COMMIT*. This cookie must be consistent during a single boot session and must be unique between instances of the NFS Version 3 protocol server where uncommitted data may be lost.

Otherwise, *COMMIT3res.status* contains the error on failure and *COMMIT3res.resfail* contains the following:

file_wcc Weak cache consistency data for the file. For a client that requires only the post-write file attributes, these can be found in *file_wcc.after*. Even though the *COMMIT* failed, full **wcc_data** is returned to allow the client to determine whether the file changed on the server between calls to *WRITE* and *COMMIT*.

Implementation Guidance

Procedure *COMMIT* is similar in operation and semantics to the XPG4 *fsync*( ) system call that synchronises a file's state with the disk, that is it flushes the file's data and metadata to disk. *COMMIT* performs the same operation for a client, flushing any unsynchronised data and metadata on the server to the server's disk for the specified file. Like *fsync*( ), it may be that there is some modified data or no modified data to synchronise. The data may have been synchronised by the server's normal periodic buffer synchronisation activity. *COMMIT* will always return *NFS3_OK*, unless there has been an unexpected error.

COMMIT differs from *fsync*( ) in that it is possible for the client to flush a range of the file (most likely triggered by a buffer-reclamation scheme on the client before file has been completely written).

The server implementation of *COMMIT* is reasonably simple. If the server receives a full file *COMMIT* request; that is, starting at *offset* zero and *count* zero, it should do the equivalent of performing *fsync*( ) on the file. Otherwise, it will arrange to have the cached data in the range specified by *offset* and *count* to be flushed to stable storage. In both cases, any metadata associated with the file must be flushed to stable storage before returning. It is not an error for there to be nothing to flush on the server. This means that the data and metadata that needed to be flushed have already been flushed or lost during the last server failure.

The client implementation of *COMMIT* is a little more complex. There are two reasons for wanting to commit a client buffer to stable storage. The first is that the client wants to reuse a buffer. In this case, the *offset* and *count* of the buffer are sent to the server in the *COMMIT* request. The server then flushes any cached data based on the *offset* and *count*, and flushes any metadata associated with the file. It then returns the status of the flush and the *verf* verifier. The other reason for the client to generate a *COMMIT* is for a full file flush, such as may be done at close. In this case, the client would gather all of the buffers for this file that contain uncommitted data, do the *COMMIT* operation with an *offset* of zero and *count* of zero, and then free all of those buffers. Any other dirty buffers would be sent to the server in the normal fashion.

This implementation will require some modifications to the buffer cache on the client. After a buffer is written with *stable* set to *UNSTABLE*, it must be considered as dirty by the client system until it is either flushed via a *COMMIT* operation or written via a *WRITE* operation with *stable* set to *FILE_SYNC* or *DATA_SYNC*. This is done to prevent the buffer from being freed and reused before the data can be flushed to stable storage on the server.

When a response comes back from either a *WRITE* or a *COMMIT* operation that contains an unexpected *verf*, the client will need to retransmit all of the buffers containing uncommitted cached data to the server. How this is to be done is up to the implementor. If there is only one buffer of interest, then it should probably be sent back over in a *WRITE* request with the appropriate *stable* flag. If there more than one, it might be worthwhile retransmitting all of the buffers in *WRITE* requests with *stable* set to *UNSTABLE* and then retransmitting the *COMMIT* operation to flush all of the data on the server to stable storage. The timing of these retransmissions is left to the implementor.

The above description applies to page-cache-based systems as well as buffer-cache-based systems. In those systems, the virtual memory system will need to be modified instead of the buffer cache.

See additional comments in *NFSPROC3_WRITE* on page 212.

Return Codes

NFS3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFS3ERR_STALE Invalid file handle. The file handle given in the arguments was invalid. The file referred to by that file handle no longer exists or access to it has been revoked.

NFS3ERR_BADHANDLE
Invalid NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_SERVERFAULT
An error occurred on the server, which does not map to any of the valid NFS Version 3 protocol error values. The client should translate this into an appropriate error. Clients based on an XPG system may choose to translate this to *EIO*.

Chapter 13

Mount Protocol, Version 3

This chapter specifies an additional protocol for the mount service, the V3 protocol, which must be supported in addition to the protocol specified in Chapter 8.

13.1 RPC Information

Authentication

The mount service uses *AUTH_NONE* in the *NULL* procedure. *AUTH_UNIX*, *AUTH_DES* or *AUTH_KERB* are used for all other procedures, although the mount service can decline to use the authentication information that is provided.

Transport Address

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

Port Number

Consult the server's port mapper, described in Chapter 6, to find the port number on which the mount service is registered.

13.1.1 Sizes of XDR Structures

The following table specifies the sizes, given in decimal bytes, of various XDR structures used in the protocol:

Structure	Size	Description
MNTPATHLEN	1024	Maximum bytes in a pathname
MNTNAMLEN	255	Maximum bytes in a name
FHSIZE3	64	Maximum bytes in a V3 file handle

13.1.2 Basic Data Types

fhandle3

```
typedef opaque fhandle3<FHSIZE3>;
```

dirpath

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

name

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

mountstat3

```
enum mountstat3 {
    MNT3_OK              = 0,     /* No error */
    MNT3ERR_PERM         = 1,     /* Not owner */
    MNT3ERR_NOENT        = 2,     /* No such file or directory */
    MNT3ERR_IO           = 5,     /* I/O error */
    MNT3ERR_ACCES        = 13,    /* Permission denied */
    MNT3ERR_NOTDIR       = 20,    /* Not a directory */
    MNT3ERR_INVAL        = 22,    /* Invalid argument */
    MNT3ERR_NAMETOOLONG  = 63,    /* Filename too long */
    MNT3ERR_NOTSUPP      = 10004, /* Operation not supported */
    MNT3ERR_SERVERFAULT  = 10006  /* A failure on the server */
};
```

13.2 Server Procedures

The following reference pages define the RPC procedures supplied by a *MOUNT* Version 3 protocol server.

```
program MOUNT_PROGRAM {
    version MOUNT_V3 {
        void MOUNTPROC3_NULL(void)         = 0;
        mountres3 MOUNTPROC3_MNT(dirpath)  = 1;
        mountlist MOUNTPROC3_DUMP(void)    = 2;
        void MOUNTPROC3_UMNT(dirpath)      = 3;
        void MOUNTPROC3_UMNTALL(void)      = 4;
        exports MOUNTPROC3_EXPORT(void)    = 5;
    } = 3;
} = 100005;
```

Name

MOUNTPROC3_NULL — Do Nothing

Call Arguments

None.

Return Arguments

None.

RPC Procedure Description

```
void
MOUNTPROC3_NULL(void) = 0;
```

Description

Procedure *NULL* does no work. It is made available to allow server response testing and timing.

Implementation Guidance

It is important that this procedure do no work at all so that it can be used to measure the overhead of processing a service request. By convention, the *NULL* procedure should never require any authentication. A server may choose to ignore this convention, in a more secure implementation, where responding to the *NULL* procedure call acknowledges the existence of a resource to an unauthenticated client.

Return Codes

Since the *NULL* procedure takes no *MOUNT* protocol arguments and returns no *MOUNT* protocol response, it can not return a *MOUNT* protocol error. However, it is possible that some server implementations may return RPC errors based on security and authentication requirements.

Name

MOUNTPROC3_MNT — Add Mount Entry

Call Arguments

```
dirpath dirname;
```

Return Arguments

```
struct mountres3_ok {
   fhandle3 fhandle;
   int auth_flavors<>;
};
union mountres3 switch (mountstat3 fhs_status) {
    case MNT_OK:
        mountres3_ok mountinfo;
    default:
        void;
};
```

RPC Procedure Description

```
mountres3
MOUNTPROC3_MNT(dirpath) = 1;
```

Description

Procedure *MNT* maps a pathname on the server to a file handle. The pathname is a string that describes a directory on the server. If the call is successful (*MNT3_OK*), the server returns an NFS Version 3 protocol file handle and a vector of RPC authentication flavours that are supported with the client's use of the file handle (or any file handles derived from it). The authentication flavours are defined in Section 4.4 on page 52.

Implementation Guidance

If *mountres3.fhs_status* is *MNT3_OK*, then *mountres3.mountinfo* contains the file handle for the directory and a list of acceptable authentication flavours. This file handle may only be used in the NFS Version 3 protocol. This procedure also results in the server adding a new entry to its mount list recording that this client has mounted the directory. *AUTH_UNIX* authentication or better is required.

Return Codes

MNT3ERR_NOENT The specified directory does not exist. If the server exports only **/a/b**, an attempt to mount **/a/b/c** will fail with *MNT3ERR_NOENT* if the directory does not exist; on the other hand, an attempt to mount **/a/x** would fail with *MNT3ERR_ACCES*.

MNT3ERR_IO I/O error. Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

MNT3ERR_ACCES 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.

MNT3ERR_NOTDIR The specified file is not a directory.

MNT3ERR_NAMETOOLONG
The filename in an operation was too long.

Name

MOUNTPROC3_DUMP — Return Mount Entries

Call Arguments

None.

Return Arguments

```
typedef struct mountbody *mountlist;

struct mountbody {
    name ml_hostname;
    dirpath ml_directory;
    mount listml_next;
};
```

RPC Procedure Description

```
mountlist
MOUNTPROC3_DUMP(void) = 2;
```

Description

Procedure *DUMP* returns the list of remotely mounted file systems. The *mountlist* contains one entry for each client host name and directory pair.

Implementation Guidance

This list is derived from a list maintained on the server of clients that have requested file handles with the *MNT* procedure. Entries are removed from this list only when a client calls the *UMNT* or *UMNTALL* procedure. Entries may become stale if a client crashes and does not issue either *UMNT* calls for all of the file systems that it had previously mounted or a *UMNTALL* to remove all entries that existed for it on the server.

Return Codes

There are no *MOUNT* protocol errors that can be returned from this procedure. However, RPC errors may be returned for authentication or other RPC failures.

Name

MOUNTPROC3_UMNT — Remove Mount Entry

Call Arguments

```
dirpath dirname;
```

Return Arguments

None.

RPC Procedure Description

```
void
MOUNTPROC3_UMNT(dirpath) = 3;
```

Description

Procedure *UMNT* removes the mount list entry for the directory that was previously the subject of a *MNT* call from this client. *AUTH_UNIX* authentication or better is required.

Implementation Guidance

Typically, server implementations have maintained a list of clients that have file systems mounted. In the past, this list has been used to inform clients that the server was going to be shutdown.

Return Codes

There are no *MOUNT* protocol errors that can be returned from this procedure. However, RPC errors may be returned for authentication or other RPC failures.

Name

MOUNTPROC3_UMNTALL — Remove All Mount Entries

Call Arguments

None.

Return Arguments

None.

RPC Procedure Description

```
void
MOUNTPROC3_UMNTALL(void) = 4;
```

Description

Procedure *UMNTALL* removes all of the mount entries for this client previously recorded by calls to *MNT*. *AUTH_UNIX* authentication or better is required.

Implementation Guidance

This procedure should be used by clients when they are recovering after a system shutdown. If the client could not successfully unmount all of its file systems before being shutdown or the client crashed because of a software or hardware problem, there may be servers that still have mount entries for this client. This is an easy way for the client to inform all servers at once that it does not have any mounted file systems. However, since this procedure is generally implemented using broadcast RPC, it is only of limited usefulness.

Return Codes

There are no *MOUNT* protocol errors that can be returned from this procedure. However, RPC errors may be returned for authentication or other RPC failures.

Name

MOUNTPROC3_EXPORT — Return Export List

Call Arguments

None.

Return Arguments

```
typedef struct groupnode *groups;

struct groupnode {
    name gr_name;
    groups gr_next;
};

typedef struct exportnode *exports;

struct exportnode {
    dirpath ex_dir;
    groups ex_groups;
    exports ex_next;
};
```

RPC Procedure Description

```
exports
MOUNTPROC3_EXPORT(void) = 5;
```

Description

Procedure *EXPORT* returns a list of all the exported file systems and which clients are allowed to mount each one. The names in the group list are implementation-specific and cannot be directly interpreted by clients. These names can represent hosts or groups of hosts.

Implementation Guidance

This procedure generally returns the contents of a list of shared or exported file systems. These are the file systems that are made available to NFS Version 3 protocol clients.

Return Codes

There are no *MOUNT* protocol errors that can be returned from this procedure. However, RPC errors may be returned for authentication or other RPC failures.

Chapter 14

Network Lock Manager Protocol, Version 4

14.1 Introduction

This chapter specifies an additional protocol for the Network Lock Manager (NLM), the Version 4 protocol, which must be supported in addition to the Version 3 protocol specified in Chapter 10. The NLM Version 4 protocol is valid only when used with the Network File System Version 3 protocol; the NLM Version 3 protocol is valid only when used with the NFS Version 2 protocol.

This chapter only discusses the differences between the NLM Version 3 and Version 4 protocols. As with the NFS Version 3 protocol, almost all the names in the NLM Version 4 protocol have been changed to include a version number. This chapter does not discuss changes that consist solely of a name change.

14.2 RPC Information

Authentication

The NLM service uses *AUTH_NONE* in the *NULL* procedure. *AUTH_UNIX*, *AUTH_DES* or *AUTH_KERB* are used for all other procedures.

Transport Address

The NLM service is supported on both TCP/IP and UDP/IP. However, a client implementation may choose to only generate requests over the UDP/IP protocol.

Port Number

Consult the server's port mapper, described in Chapter 6, to find the port number on which the lock manager is registered.

14.2.1 Basic Data Types

uint64

```
typedef unsigned hyper uint64;
```

int64

```
typedef hyper int64;
```

uint32

```
typedef unsigned long uint32;
```

int32

```
typedef long int32;
```

nlm4_stats

```
enum nlm4_stats {
    NLM4_GRANTED             = 0,
    NLM4_DENIED              = 1,
    NLM4_DENIED_NOLOCKS      = 2,
    NLM4_BLOCKED             = 3,
    NLM4_DENIED_GRACE_PERIOD = 4,
    NLM4_DEADLCK             = 5,
    NLM4_ROFS                = 6,
    NLM4_STALE_FH            = 7,
    NLM4_FBIG                = 8,
    NLM4_FAILED              = 9
};
```

The **nlm4_stats** value indicates the success or failure of a call. This version contains several new error codes, so that clients can provide more precise failure information to applications.

NLM4_GRANTED The call completed successfully.

NLM4_DENIED The call failed. For attempts to set a lock, this status implies that if the client retries the call later, it may succeed.

NLM4_DENIED_NOLOCKS
The call failed because the server could not allocate the necessary resources.

NLM4_BLOCKED Indicates that a blocking request cannot be granted immediately. The server will issue an *NLMPROC4_GRANTED* callback to the client when the lock is granted.

NLM4_DENIED_GRACE_PERIOD
The call failed because the server is reestablishing old locks after a reboot and is not yet ready to resume normal service.

NLM4_DEADLCK The request could not be granted and blocking would cause a deadlock.

NLM4_ROFS The call failed because the remote file system is read-only. For example, some server implementations might not support exclusive locks on read-only file systems.

NLM4_STALE_FH The call failed because it uses an invalid file handle. This can happen if the file has been removed or if access to the file has been revoked on the server.

NLM4_FBIG The call failed because it specified a length or offset that exceeds the range supported by the server.

NLM4_FAILED The call failed for some reason not already listed. The client should take this status as a strong hint not to retry the request.

nlm4_holder

```
struct nlm4_holder {
    bool exclusive;
    int32 svid;
    netobj oh;
    uint64 l_offset;
    uint64 l_len;
};
```

Th **nlm4_holder** structure indicates the holder of a lock. The *exclusive* field tells whether the holder has an exclusive lock or a shared lock. The *svid* field identifies the process that is holding the lock. The *oh* field is an opaque object that identifies the host, or a process on the host, that is holding the lock. The *l_len* and *l_offset* fields identify the region that is locked. The only difference between the NLM Version 3 protocol and the NLM Version 4 protocol is that in the NLM Version 3 protocol, the *l_len* and *l_offset* fields are 32 bits wide, while they are 64 bits wide in the NLM Version 4 protocol.

nlm4_lock

```
struct nlm4_lock {
    string caller_name<LM_MAXSTRLEN>;
    netobj fh;
    netobj oh;
    int32 svid;
    uint64 l_offset;
    uint64 l_len;
};
```

The **nlm4_lock** structure describes a lock request. The *caller_name* field identifies the host that is making the request. The *fh* field identifies the file to lock. The *oh* field is an opaque object that identifies the host, or a process on the host, that is making the request, and the *svid* field identifies the process that is making the request. The *l_offset* and *l_len* fields identify the region of the file that the lock controls. A *l_len* of zero means ‘‘to end-of-file.’’

There are two differences between the NLM Version 3 protocol and the NLM Version 4 protocol versions of this structure. First, in the NLM Version 3 protocol, the length and offset are 32 bits wide, while they are 64 bits wide in the NLM Version 4 protocol. Second, in the NLM Version 3 protocol, the file handle is a fixed-length NFS Version 2 protocol file handle, which is encoded as a byte count followed by a byte array. In the NFS Version 3 protocol, the file handle is already variable-length, so it is copied directly into the *fh* field. That is, the first four bytes of the *fh* field are the same as the byte count in an NFS Version 3 protocol **nfs_fh3**. The rest of the *fh* field contains the byte array from the NFS Version 3 protocol **nfs_fh3**.

nlm4_share

```
struct nlm4_share {
    string caller_name<LM_MAXSTRLEN>;
    netobj fh;
    netobj oh;
    fsh4_mode mode;
    fsh4_access access;
};
```

The **nlm4_share** structure is used to support DOS file sharing. The *caller_name* field identifies the host making the request. The *fh* field identifies the file to be operated on. The *oh* field is an opaque object that identifies the host that is making the request. The *mode* and *access* fields specify the file-sharing and access modes. The encoding of *fh* is a byte count, followed by the file handle byte array. See the description of **nlm4_lock** for more details.

14.3 NLM Procedures

The procedures in the NLM Version 4 protocol are semantically the same as those in the NLM Version 3 protocol. The only semantic difference is the addition of a *NULL* procedure that can be used to test for server responsiveness. A syntactic change is that the procedures were renamed to avoid name conflicts with the values of **nlm4_stats**. Thus the procedure definition is as follows.

```
version NLM4_VERS {
    void NLMPROC4_NULL(void) = 0;
    nlm4_testres NLMPROC4_TEST(nlm4_testargs) = 1;
    nlm4_res NLMPROC4_LOCK(nlm4_lockargs) = 2;
    nlm4_res NLMPROC4_CANCEL(nlm4_cancargs) = 3;
    nlm4_res NLMPROC4_UNLOCK(nlm4_unlockargs) = 4;
    nlm4_res NLMPROC4_GRANTED(nlm4_testargs) = 5;
    void NLMPROC4_TEST_MSG(nlm4_testargs) = 6;
    void NLMPROC4_LOCK_MSG(nlm4_lockargs) = 7;
    void NLMPROC4_CANCEL_MSG(nlm4_cancargs) = 8;
    void NLMPROC4_UNLOCK_MSG(nlm4_unlockargs) = 9;
    void NLMPROC4_GRANTED_MSG(nlm4_testargs) = 10;
    void NLMPROC4_TEST_RES(nlm4_testres) = 11;
    void NLMPROC4_LOCK_RES(nlm4_res) = 12;
    void NLMPROC4_CANCEL_RES(nlm4_res) = 13;
    void NLMPROC4_UNLOCK_RES(nlm4_res) = 14;
    void NLMPROC4_GRANTED_RES(nlm4_res) = 15;
    nlm4_shareres NLMPROC4_SHARE(nlm4_shareargs) = 20;
    nlm4_shareres NLMPROC4_UNSHARE(nlm4_shareargs) = 21;
    nlm4_res NLMPROC4_NM_LOCK(nlm4_lockargs) = 22;
    void NLMPROC4_FREE_ALL(nlm4_notify) = 23;
} = 4;
```

The following reference page defines the additional *NULL* procedure.

Name

NLMPROC3_NULL — Do Nothing

Call Arguments

None.

Return Arguments

None.

RPC Procedure Description

```
void
NLMPROC4_NULL(void) = 0;
```

Description

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

Implementation Guidance

It is important that this procedure do no work at all so that it can be used to measure the overhead of processing a service request. By convention, the *NULL* procedure should never require any authentication.

Return Codes

It is possible that some server implementations may return RPC errors based on security and authentication requirements.

14.4 Implementation Guidance

64-bit Offsets and Lengths

Some NFS Version 3 protocol servers can only support requests where the file offset or length fits in 32 or fewer bits. For these servers, the lock manager will have the same restriction. If such a lock manager receives a request that it cannot handle (because the offset or length uses more than 32 bits), it should return the *NLM4_FBIG* error.

File Handles

The change in the file handle format from the NFS Version 2 protocol to the NFS Version 3 protocol complicates the lock manager. First, the lock manager needs some way to tell when an NFS Version 2 protocol file handle refers to the same file as an NFS Version 3 protocol file handle. (This assumes that the lock manager supports both NLM Version 3 protocol clients and NLM Version 4 protocol clients.) Second, if the lock manager runs the file handle through a hashing function, the hashing function may need to be retuned to work with NFS Version 3 protocol file handles as well as NFS Version 2 protocol file handles.

Appendix A

Semantic Difference Summary for File Access

A.1 Introduction

Many of the entries described in the X/Open System Interfaces and Headers Specification (see reference **XSH**) and the X/Open Commands and Utilities Specification (see reference **XCU**) are directly or indirectly concerned with accessing files stored on the system's file system.

When documenting those utilities and functions the assumption was made that files being accessed would reside on the local file system of the system, and that all local file system semantics would be supported. Users or applications using an XCU utility or XSH function could therefore rely on described behaviour and error codes being returned as described in that XPG entry.

When XPG-compliant systems are linked together using XNFS protocols, files on remote systems may be accessible to a process running on a local system, without the process being aware that files are remote. In fact, because XNFS provides transparent access to remote files, it is not possible for a process to distinguish between local and remote files before they are used. Due to the nature of the way XNFS works, there are some semantic differences between operations on local files and equivalent operations on remote files.

This appendix gives a summary of these semantic differences. Together with Appendix B and Appendix C this appendix specifies differences that can occur when using a given utility or function with a file on a remote file system.

XNFS describes a number of new errors which may occur when CAE applications issue XSH function calls which refer to XNFS file systems. One error, which occurs when a file handle is rejected as invalid by the XNFS server, is represented by a new error code, [ESTALE]. All other errors discussed in this specification are represented by existing XSH error codes; thus the set of possible interpretations for each of these codes is extended when XNFS is being used. It should be noted that the X/Open Commands and Utilities Specification (see reference **XCU**) explicitly allows implementors to define additional error codes, and does not define which error should be reported when multiple errors occur during a single operation. Notwithstanding this, the XNFS specification defines only one new error code, [ESTALE], and implementors of XNFS are strongly discouraged from introducing additional error codes which are specific to XNFS. See also the description of *NFSERR_STALE* in **stat** on page 73.

There are not many areas in which semantic differences manifest themselves. The following list summarises differences which are common to many system interface functions or utilities. Specific differences that are not common to several functions or utilities are discussed in the description of that function or utility in Appendix B and Appendix C respectively.

A.2 Special File Access

A process may create, rename, unlink, and get the attributes of a special file that is located on a remote file system. A process may also *fattach*( ) a STREAMS file descriptor to a remote file. If a process opens a remote special file, a local device will be used instead of the (possibly desired) remote device. This includes the FIFO special file type, support for which is not mandated in the XNFS specification.

If a process opens a remote special file, the local device that is used will be the local device that corresponds to the major and minor numbers associated with the remote special file. If there is no local device corresponding to this major and minor number pair, the operation will fail. For example, if *pax* is used to extract files from a tape, *pax* must be run on the host that owns the tape drive.

A process may be unable to create a remote special file, either because the server doesn't support special files at all ([EOPNOTSUPP]), or because the server doesn't support the requested special file type ([EINVAL]).

See also Section A.15.9 on page 282 for semantic differences that can arise when accessing special files with Version 2 of the NFS protocol.

A.3 UID Mapping by Server

Access to a file located on a remote file system can be denied, even in the case that the file permissions do not themselves restrict access. When the server exports a file system, the following attributes can be specified which control access to the file system:

1. The server will treat requests from a client process with an unknown user ID as having the user ID that is specified in the ‘‘AnonMapping=’’ ExportedFileSystem attribute. (See the discussion of the ‘‘AnonMapping=’’ ExportedFileSystem attribute in Section 2.4.1 on page 14.)
2. The server can deny access by privileged user from specified hosts. Should the server receive a request for file access from a process with an effective user ID of 0 on a denied host, the request will be processed. However, the value specified in the ‘‘AnonMapping=’’ ExportedFileSystem attribute will be used instead of the effective user ID of the process. (See the discussion of the ‘‘AnonMapping=’’ and ‘‘Root=’’ ExportedFileSystem attributes in Section 2.4.1 on page 14.)

If such a mapping occurs and thus causes the server to deny access to a file, the error [EACCES] will be returned to the process.

In the case that the server denies access by a privileged user, the semantics of those file access functions that require appropriate privileges may not be available to the calling process, the error [EPERM] will be returned to the process in these cases.

For access to regular files using Version 3 of the NFS protocol, restrictions due to user ID mapping are enforced at the time the file is opened, except when a stale entry in the access cache causes the call to succeed. (See Section A.5 on page 273 for a discussion of the access cache.) With Version 2, the client need not enforce the restrictions at the time the file is opened. If the process is able to open the file, the client will generally enforce the restrictions when the process tries to read or write the file. Because of data caching, though, the client need not enforce the restrictions until it has tried to read from or write to the server. (See Section A.9.1 on page 276 for a discussion of delayed write errors.)

Reads or writes may also fail if, for example, the permissions on the file are changed after the file was opened, as described in Section A.6 on page 274

Operations that require directory updates, such as *mkdir*( ), *link*( ), *remove*( ) and *mknod*( ), will reflect access restrictions immediately for both versions of the NFS protocol. Because of privilege restrictions, it may be difficult to perform system administration tasks from an XNFS client. For example, *cron* jobs may need to be run on the server, even if the relevant files are exported by the server. Other tasks, such as restoring or installing files using *pax*, may only be possible on the server as well.

A.4 Execution of Set-user-ID Programs

Execution semantics of a program having the set-user-ID mode bit are different over NFS (see *exec* in the X/Open System Interfaces and Headers Specification (see reference **XSH**)). When an NFS file system is mounted by a client, the ‘‘SetUID=’’ mount attribute determines whether normal set-user-ID execution semantics are in effect. If the attribute is ‘‘False’’, execution of such a program over NFS will still occur. However, the effective user ID will not be reassigned to the owner of the program, and will remain equal to the user ID of the process. (See the discussion of the ‘‘SetUID=’’ MountedFileSystem attribute in Section 2.4.2 on page 15.)

During execution, such a program may be denied an operation such as *open*( ), *read*( ) or *write*( ), because it is not running with the effective user ID with which it was designed to run. In this case an [EACCES] or an [EPERM] error may be generated depending upon the type of operation being performed.

A.5 Attribute and Access Caching

There are several semantic differences that occur due to attribute and access caching on the client. These differences occur when information in the client’s cache does not match the information that is on the server. Since these caches are updated frequently, there is only a small window of time in which this information can differ. If the attribute and access caches are disabled through the ‘‘AttribCaching=’’ *MountedFileSystem* attribute, or if the client does not implement these caches, these problems no longer exist. (See the ‘‘AttribCaching=’’, ‘‘ACRegMin=’’ and ‘‘ACRegMax=’’ *MountedFileSystem* attributes in Section 2.4.2 on page 15.)

Information in a client’s attribute and access caches becomes inaccurate when the attributes of a file on the server are changed. When such changes are made from a client, that client’s attribute and access caches are updated immediately in order to maintain a consistent view of the server’s file system. However, when a process running on the server or on a client changes the attributes of a file, these results may not be immediately noticeable to other clients.

The attribute cache contains information about files on the server, including:

- the mode of a file
- the user ID and group ID of a file
- the number of bytes in a file
- the access times associated with a file.

The access cache contains the results of previous NFSPROC3_ACCESS calls. On a per-file, per-user basis, it records which access modes are known to succeed, known to fail, or indeterminate.

Functions such as *stat*( ) may return incorrect information if the client's attribute cache is inaccurate. Other operations that are based on information in the client's attribute or access caches may behave incorrectly if the information in the caches is inaccurate. Examples of this are given below.

A.5.1 Denial of Access

Access to a file on the server may be denied because the attributes in the client caches are more restrictive than the attributes on the server. If a privilege has just been allowed on the server, and the client's caches still record this privilege as being ''off'', functions such as *open*( ) may fail on the client.

If the attributes in the client caches are less restrictive than the attributes on the server, functions such as *open*( ) may succeed, but functions like *read*( ) or *write*( ) may fail.

A.5.2 Operations Using File's Byte Count

File operations that rely on the byte count of a file may function incorrectly if the byte count stored in the attribute cache is inaccurate. For example, a *write*( ) to a file that was opened with the O_APPEND flag set may result in the overwriting of data that was appended to that file by a process executing on another XNFS client. Record locking operations (see Section A.14 on page 278) where *l_whence* is set to SEEK_END may also fail to behave as intended.

A.5.3 File Times

Programs that use file access and modification times may behave incorrectly. For example, **make** may fail to rebuild a target because of stale information about the source's modification time.

A.6 File Accessibility Changed after Open

A process may be denied further access to an open file if the file is located on a remote file system. There are two reasons for this given below.

A.6.1 File Attributes Changed after Open

The attributes of the file could be changed after the file was opened.

As part of the stateless behaviour of NFS, the server does not maintain information regarding files that are open by processes on client systems. Therefore, each time a client process issues a request to access a file, the request is validated. It is possible that the attributes of a file can change in between requests from a client process, thus denying further access to a file that had been previously accessible. In this case functions such as *read*( ) and *write*( ), or derived functions such as *fread*( ), *fgets*( ), etc. may fail with [EACCES].

In the XSI specifications, it is stated that the effect of calling *chmod*( ) to change the access permissions of an open file is implementation-dependent. With NFS, the semantics of *chmod*( ) are different, in that effects are immediate and can cause further access to an open file to be denied or allowed in a manner different to access to a local file. The two additional cases where access is allowed by NFS are:

1. Once a file handle has been established, the owner of the file is allowed to access it regardless of the permission settings associated with the file.

2. Once a file handle has been established and a user has permission to execute a file, then that user is also granted permission to read the data from the file.

A.6.2 File Deleted after Open

The file could be deleted by a process on the server, or on another client.

Another function of the stateless behaviour of NFS is that the server cannot prevent the deletion of a file that is open by a process on a client system and is not open by a process on the server. When this occurs, the next client request which refers to the file will be rejected with the XNFS-specific error [ESTALE], indicating that the file handle no longer refers to a valid file system object. An XNFS client implementation may return a code for [ESTALE] as the reason for the failure of the XSI call, or may translate it into some other error code. In addition to file operations such as *remove*( ), *read*( ) and *stat*( ), and operations on directories, such as *rmdir*( ), may fail with [ESTALE]. As described in Section A.14.5 on page 279, record locking operations may fail in a manner that is implementation-dependent.

If a file is deleted while a process on the server has the file open, client requests using that file may fail with [ESTALE]. Alternatively, the server may continue granting access to the file as long as one or more processes on the server have the file open.

As discussed in Section A.22 on page 284, it is possible for a function to fail with [ESTALE] even if it does not take a file handle or filename as an argument. For example, if a system's user database resides on a remote file, *getpwuid*( ) will pass along errors that it receives accessing the database file.

A.7 No Protection for In-Use Executables

If an executable file stored on an XNFS server is being executed on a client system, there is no mechanism that prevents the file from being deleted, truncated, or overwritten (for example, via *remove*( ), *fopen*( ), *truncate*( ) or *write*( )). The execution of the program may be terminated if this occurs.

The error [ETXTBSY] is never returned by a function operating on a a remote file over NFS. Nonetheless, it is possible for calls such as *truncate*( ) or *write*( ) to fail because an executable file is being executed on the server. This XNFS Specification does not specify which error code is returned in this case.

A.8 Transparent Rename or Unlink While Open

If a file on the server is open by a process running on a client, and the file is deleted by the same or a different process on that client (for example, using *remove*( ) or *unlink*( )), the client will rename the file to a temporary file. Any process that has this file open can continue using it. When the last process on the client to have this file open closes it, the client issues a request to the server to delete the temporary file. If a process on the client attempts to remove the temporary file, the client may remove it, ignore the request, or rename the file to a different temporary file. Should the client fail before removing the temporary file, the temporary file may remain indefinitely. It is common practice to have an entry in the XNFS server's *crontab* database to regularly delete these lingering temporary files.

The client's request to delete the temporary file may not be processed on the server if the server fails and the ''RetrySemantics='' attribute of the mounted file system is *Soft*. (See the discussion of ''RetrySemantics='' in Section 2.4.2 on page 15.)

This XNFS specificationdoes not specify how a client generates the name for the temporary file described in this section. Nonetheless, it is common practice for the client to pick a name starting with a period (.) in the same directory as the original file. The presence of this file can cause attempts to remove the directory (for example, *rmdir*( ) or ‘‘rm -rf’’) to fail.

A.9 Data Caching

The following semantic differences may occur due to the buffer cache mechanisms.

A.9.1 Delayed Write Errors

Errors that occur when writing data to the file server will not necessarily be returned through *write*( ). Calls to *write*( ) may put data into the client’s buffer cache and return without error. The error may occur later when the buffer cache is flushed to the server.

The buffer cache may be flushed in response to a call from the process, for example, *sync*( ) or *close*( ), or it may be flushed asynchronously. If the cache is flushed in response to a call from the process, the call will return the error. If the buffer cache is flushed asynchronously, the error will be reflected by a subsequent call to *fsync*( ) or *close*( ). Some systems may reflect the error at a subsequent *write*( ), but application writers must not depend on this behavior.

The *sync*( ) call may fail silently because *sync*( ) only schedules writes. The actual writes may happen after *sync*( ) has returned.

Some routines, such as *fgetpos*( ), may cause a *write*( ) to occur as a side effect. These writes are also potentially subject to delayed write errors.

A.9.2 Read of Old Data

The information in the buffer cache may be inaccurate and not reflect the latest changes to a file. Therefore, the *read*( ) function may not get the latest contents of a file.

Similarly, functions that invoke *read*( ), such as *fscanf*( ) or *fgets*( ), need not return the latest contents of a file, even when the referenced I/O stream is unbuffered.

A.9.3 Atomicity of Transfer

When a file is being read or written by several processes on different systems, the operations of the caches on the server and clients will affect the atomicity of data reading and writing. No assumption can be made that common submultiples for all the cache sizes or alignments will exist. Consequently, it is impossible to guarantee that any arbitrary multi-byte read or write will be atomic.

A.9.4 File Time Updates

If a client is able to satisfy a read or write request without contacting the server, the client need not update one or more of the corresponding file times (st_mtime, st_ctime, and st_atime). This applies to explicit reads and writes and to implicit reads and writes, such as those from mapped files and the *exec* family of functions.

For write requests, programs can force st_mtime and st_ctime to be updated by forcing the writes to the server, for example with *fsync*( ). There is no corresponding mechanism for forcing st_atime to be updated. In particular, the functions *fstat*( ) and *stat*( ) cannot be used to force the update of file times for either reads or writes.

A.10 Directory Caching

There are two semantic differences concerning the client's directory caching mechanism. These differences occur when the information in the directory cache does not reflect the server's directories. As with attribute caching, the information in this cache is refreshed frequently from the server. However, there is a window of time in which the information in this cache can be inaccurate. If this caching mechanism is disabled through the ''AttribCaching=False'' MountedFileSystem attribute, these semantic differences no longer exist. (See the discussions of the ''AttribCaching='', ''ACDirMin='' and ''ACDirMax'' MountedFileSystem attributes in Section 2.4.2 on page 15.)

The following semantic differences can be noted:

1. It is possible that a client may still record an entry of a file that has just been deleted on the server. Attempts on the client to create a new file with the same name, for example, using *creat*( ) or *link*( ), may fail, typically with [EEXIST]. Attempts to remove an otherwise empty directory may fail, typically with [EEXIST] or
2. The result of a process that modifies the contents of a directory may not be immediately noticeable to other clients. For example, calls to *open*( ) or *stat*( ) may fail, even if the file exists on the server. Functions such as *tmpnam*( ) may return a filename that is already in use on the server.

A.11 Time Skew

A process cannot rely on the access times of a remote file to be correct. The access times associated with a remote file will relate to the system clock on the server system rather than to that of the client system which last updated the remote file. A comparison of these access times with the local system clock or the access times of a local file may not be of value. The reasons for this are:

1. The system clocks on the server and client may not be synchronised.
2. The client and server systems may each have a different notion of system date and time; for instance, they may be in different time zones.
3. The attribute cache may be holding inaccurate information concerning the times of a file.

As a result, programs whose behavior depends on file times may behave differently or incorrectly in an XNFS environment. For example, if a file has been modified recently, the ls command may display the file's date and time using the format for a modification time in the future. The make command may mistakenly conclude that a particular target is (or is not) out of date with respect to its sources. The *pax* command may function incorrectly with the -*u* option.

This XNFS specification does not specify whether the routines *utime*( ) and *utimes*( ) use the client time or server time when a null time pointer is used, that is, when setting the file times to the current time.

A.12 Server or Network-Induced Delays

When a client makes a request to the server, the client expects a response within a certain time limit. (See the discussion of the ‘‘NFSTimeOut=’’ MountedFileSystem attribute in Section 2.4.2 on page 15.) Should a response not occur within this time limit, the client may react in two ways, depending upon values of certain mount attributes:

1. The client may reissue the request repeatedly until either the server responds, or the maximum number of retries is reached. Should this maximum number of retries be reached, the client has detected a server failure and returns an error to the process. (See the discussions of the ‘‘RetrySemantics=’’ and ‘‘NFSRetransmissions=’’ MountedFileSystem attributes in Section 2.4.2 on page 15.)

2. The client may reissue the request repeatedly until the the server reponds. (See the discussions of the ‘‘RetrySemantics=’’ and ‘‘NFSRetransmissions=’’ MountedFileSystem attributes in Section 2.4.2 on page 15.)

   This behaviour may cause a function which is waiting for the server’s response to not return to the calling process for an arbitrary duration of time. The effect on the process is that it will sleep until the server responds.

   When a process issues a *stat*( ) of a remote NFS mounted object, or of a directory containing a remote NFS mounted object, a server request is generated. This may cause a delay if the server does not respond.

A.13 Interruption of Function Calls

A signal may be posted to a process that is making a request that involves an NFS file. For example, a user may wish to interrupt a request that is taking a long time to complete. If the file system was mounted with the ‘‘Intr=’’ attribute set to ‘‘True’’ (see Section 2.4.2 on page 15), the pending NFS operation will be cancelled and the signal will be processed. Otherwise the NFS operation will be retried as described in Section A.12, and the signal will not be processed until the NFS operation succeeds or the retry limit is reached. This behavior applies to any operation that must contact the server, even functions that are not documented as potentially failing with [EINTR] (for example, *stat*( ), *readlink*( )).

A.14 File and Record Locking

File and record locking allows all or part of a file to be locked by calling *fcntl*( ) (using F_SETLK or F_SETLKW) or *lockf*( ). There are several semantic differences regarding file and record locking over XNFS.

A.14.1 Availability of Locking

For file and record locking to be available, the client and server must support the NLM protocol, typically by running the XNFS lock server. This XNFS specification does not specify any mechanism whereby a client can determine whether a server provides locking services, nor does it define the effects of issuing locking calls when the server does not provide these services.

A.14.2 F_GETLK l_pid

The structure **flock** that is returned from an F_GETLK command contains the process ID (l_pid) of the process that is holding the lock. When this is a process that is accessing a file on a remote machine, this process ID is provided as a unique identifier for the process holding the lock, but it is not necessarily the same as the process ID of that process. This XNFS specification does not specify a mechanism for identifying which machine the process is running on.

A.14.3 Signals

If a process receives a signal while attempting to acquire or release a remote lock, the call may return with the error [EINTR], even for non-blocking locks (*fcntl*( ) with F_SETLK or *lockf*( ) with F_TLOCK). If this happens, the process cannot determine whether its request succeeded. If the process exits, the client operating system will release the lock if necessary. If the process catches the signal and resumes processing, the process should either resubmit the lock request, or it should explicitly unlock the region it had tried to lock.

A.14.4 Memory-Mapped Files

Under some circumstances it is possible for cooperating processes to inadvertently defeat file and record locking by mapping all or part of the file into memory with *mmap*( ). For this reason, *mmap*( ) may fail if all or part of the file is locked, and attempts to acquire a lock may fail if all or part of the file is mapped into memory. Applications can reduce the chances of running into these restrictions by always locking the entire file, or by locking regions that correspond to whole pages on the client.

A.14.5 Error Handling

Suppose a process tries to acquire a lock. Due to limitations in Version 3 of the NLM protocol, the client is unable to distinguish unrecoverable errors from the case where another process already has a conflicting lock. Examples of unrecoverable errors include:

- after the process had opened the file, the file was deleted by a process on the server or on a different client.
- the process requested an exclusive lock, the server has the file system mounted read-only, and the server implementation requires a read-write file system for an exclusive lock.

This XNFS specification does not define the behavior of the client under these circumstances.

A.15 Network Heterogeneity

There are several differences that occur because XNFS can connect heterogeneous environments.

A.15.1 Local Execution of a Remote Program

It is possible that a binary file residing on the server may be incompatible for execution on a client. For example, it may have been compiled for a different type of machine, or a different X/Open-compliant operating system. An attempt to execute such a program, for example, via *popen*( ) or *system*( ), or from the shell, will usually fail.

A program residing on the server which is compatible for execution with a client may be incompatible in other ways. For example, it may include references to local resources which are not accessible on all systems, or have been compiled for a different version of the operating system which uses a slightly different binary interface. In this case the failure mode cannot be predicted.

This XNFS specification does not specify any mechanism for determining the compatibility of a binary program with a particular system, nor the ways in which incompatibilities may manifest themselves.

A.15.2 Use of Remote Input Files with Varying Formats

Several utilities, for example *tabs*, as well as several functions, for example *getpwuid*( ), process files that are in a predefined format. Since there is no standard defining the format for some of these input files, different implementations may use different formats for the same input file. Incompatibilities may arise when a utility or function is used with a remote input file having a different format than expected.

A.15.3 Architectural Dependencies

Some utilities may use input files that have a format which is dependent upon the underlying architecture of the system. For example, a utility which uses a binary file may not operate correctly with a binary file from a foreign architecture. Any utility which operates on binary object files and executable files such as *ar* and *cc* will not operate correctly with a binary file from a foreign system.

Some algorithms, for example the computation of a file's checksum using *sum*, may be dependent upon the architecture of the machine. Therefore, computing the checksum of two identical files residing on two different machines having different architectures may yield two different values.

A.15.4 Output Displayed in Conventions of Local System

Utilities which report information about files and/or file systems will behave consistently when viewed from a single system. However, the results of some of these utilities that are executed on different systems may differ. For instance, the execution of *df* on two different systems may compute the free space of the same mounted file system differently. The output from the *nm* or *od* utilities on two different systems may be formatted differently when run on the same object file. In many cases, though, the utility provides a mechanism for generating output in a portable fashion, such as the *-P* option to *df* and *nm*, or the *-t* option to *od*.

A.15.5 Filesize Differences

A system need not support native 64-bit file sizes to support the NFS Version 3 protocol. This lets 32-bit clients interoperate with 64-bit servers and vice-versa, with certain restrictions.

If a program is running on a 64-bit client, calls to routines such as *write*( ) or *truncate*( ) will fail if the program tries to create a file that is larger than that which the server supports. Record locking operations (see Section A.14 on page 278) will fail if the program tries to lock a region of the file that the server cannot support. The failure status in these cases is typically [EFBIG].

If a program is running on a 32-bit client, routines such as *stat*( ) may return an incorrect size for a file. If a file is too big for the client to handle, the client may handle requests for the file in one of two ways:

- Deny access to the file. That is, calls to *open*( ) or *fopen*( ) will fail if the file exists and is too big for the client to handle. If the file is small enough initially and then grown too big, for example, by a process on the server, the call to *open*( ) or *fopen*( ) will succeed, but attempts to read or write the file will fail after the client determines that the file has become too big.
- Allow access to the first part of the file. That is, the program will be allowed to read or write the file up to the 32-bit limit, but not beyond.

A.15.6 Characters in File Names

A server may have an implementation-specific set of characters that it does not allow in file names. If a program on the client uses one of these characters in a file name, for example in a *creat*( ) or *mkdir*( ) call, it may get back the error [EACCES]. Functions that expect the name of an existing file, for example, *stat*( ), may fail with [EACCES] or [ENOENT]. Note that the contents of a symbolic link for the *symlink*( ) function are not subject to any character restrictions.

Some server implementations do not preserve character case when creating an object in a file system. That is, they may map all the characters in the name to either lower or upper case. Also, even if they preserve case when creating an object, some server implementations may ignore case distinctions for lookup operations. For example, an attempt to create the file ‘‘Makefile’’ may instead create the file ‘‘MAKEFILE’’. Even if the server creates the file using the name ‘‘Makefile’’, attempts to reference ‘‘makefile’’ or ‘‘MakeFILE’’ may all reference the first file ‘‘Makefile’’.

A.15.7 Server Access Control

The server may use an access model other than the traditional UNIX mode bits, for example, Access Control Lists. In this case the mode bits reported by the client need not accurately represent the permissions on a file. This inaccuracy can cause several problems, as discussed below. Some systems implement an additional protocol to avoid these problems. The details of that protocol are not covered by this XNFS specification and may vary from vendor to vendor.

If the client and server are connected using Version 2 of the NFS protocol, the client relies on the mode bits to determine whether a given process has access to a given file. If the mode bits are sufficiently inaccurate, the client may deny access to a process even though the request would succeed on the server. That is, calls to *open*( ) or *fopen*( ) may fail on the client when they would succeed on the server. Conversely, the client may grant access based on the mode bits, only to have the request denied by the server. That is, the *open*( ) call may succeed, but calls to *read*( ) or *write*( ) may fail, or there may be delayed write errors as described in Section A.9.1 on page 276.

If the client incorrectly grants a process access to a file and can satisfy read requests from its cache, the process may successfully read or execute the file, bypassing the restrictions on the server. It is also possible for an unauthorized process to read a directory in this manner. A similar security breach is possible for writes to a file: if the client incorrectly grants access to an unauthorized process while a second, authorized, process is writing to the file, the first process may successfully update the file on the server. Unauthorized directory updates such as *mkdir*( ), *rmdir*( ), *rename*( ) and *remove*( ) should always fail.

If the client is using Version 3 of the NFS protocol, the mode bit information may still be incorrect. Nonetheless, the client's access decisions should be consistent with the server's, after allowing for caching issues as described in Section A.5 on page 273. (With NFS Version 3, the client can ask the server whether a particular access request should be granted.) This greatly reduces the probability that a process will incorrectly granted or denied an operation by the client.

A.15.8 Server Support for File Times

Not all server implementations let the client set the times for a file (st_atime, st_mtime). Those that do need not support a fine enough clock granularity to fully support interfaces like *utime*( ) or *utimes*( ). For example, they may support a resolution in milliseconds or in minutes. This means that attempts to set a file's times, for example with *touch*( ) or *utimes*( ), need not change the file times at all, or the file times may change to values different than those requested.

A.15.9 Special Files

The encoding of major and minor device numbers is not specified by Version 2 of the NFS protocol, so different implementations may use different encodings. This means that the major and minor device numbers of a special file may have different values, depending on where the file is referenced from. Functions such as *open*( ) may behave in an unintended manner under these circumstances.

Bounds checking on the server can cause *mknod*( ) to fail, particularly if the client and server are communicating using Version 2 of the NFS protocol and they disagree on the encoding of the major and minor device numbers.

A.16 User and Group ID Database Consistency

It is necessary to maintain a consistent user ID to username mapping across the collection of XNFS servers and clients. The group ID to group name mapping should be consistent as well. If these databases are not consistently maintained, programs such as **ls** may report different owning users or groups depending on where the program is run. Also, access restriction mechanisms need not function as intended. Access which should be denied may be allowed, and conversely access which should be allowed may be denied.

In some cases the server may limit the user or group ID values to a subset of the values that are possible on the client. In this case, of course, it is impossible to have consistent user and group ID mappings. Routines such as *chown*( ) will fail with [EINVAL] when an unsupported user or group ID is used.

A.17 Access to Read-Only File Systems

Operations which attempt to write to or modify a remote read-only file system will fail and may return the error [EROFS]. This will occur if the file system was exported with the ExportedFileSystem ‘‘Mode=’’ attribute set to ‘‘ReadOnly’’, or if the the file system was mounted with the MountedFileSystem ‘‘Mode=’’ attribute set to ‘‘ReadOnly’’. (See the discussions of the ‘‘Mode=’’ attribute in both Section 2.4.1 on page 14 and Section 2.4.2 on page 15.) For access using Version 3 of the NFS protocol, the failure will occur when the file is opened for writing, unless a stale entry in the client access cache (see Section A.5 on page 273) causes the call to succeed. With Version 2, the call *open*( ), *fopen*( ), etc. will fail if the client has mounted the file system read-only, but the call may succeed if the client has mounted the file system read-write and the server has exported it read-only. If the process is able to open the file for writing, attempts to write to a regular file (for example, *write*( ), *writev*( )) will generally fail, though data caching can delay these failures, as described in Section A.9.1 on page 276.

Directory operations on a read-only file system, for example, *rmdir*( ), *unlink*( ), *creat*( ), *mknod*( ), will fail immediately for both versions of the NFS protocol.

Writes to special files are not subject to read-only restrictions, as they do not require updates to the file system on the server.

An attempt to obtain an exclusive record lock on a file in a read-only file system may fail. As described in Section A.14.5 on page 279, the request may fail in an implementation-dependent manner.

A.18 Group Ownership of Created Files

When the ‘‘GrpID=’’ attribute of the MountedFileSystem object is set to True, the group owner of a newly created file is always set to the group owner of the directory containing that file. This may differ from the semantics applied to a locally created file where the group owner may be set to the effective group ID of the calling process. This semantic difference affects functions such as *open*( ) and *creat*( ), as well as operations such as I/O redirection from the shell.

A.19 Consistency of Limits

The limit on the maximum number of simultaneous supplementary group IDs per process is allowed to vary between systems and XNFS does not impose any restrictions on the number of group IDs which are used to determine accessibility to a remote file. However, the server may impose a lower maximum number than the client and may reject requests from the client which contain more than the maximum number of supplementary groups allowed by the server.

A similar problem exists with other limits, such as the maximum number of characters in a filename, or the maximum number of hard links to a file. These limits may vary between different file systems, and Version 2 of the NFS protocol does not provide a mechanism for this information to be made available to a client. The limits provided from a call to *pathconf*( ) need not accurately reflect the limit imposed by the server’s file system. This may cause file access requests to be rejected by the server or for files to be inaccessible from the client.

A.20 Symbolic Links

If the server does not support symbolic links, an attempt to create one with *symlink*( ) will fail with [EOPNOTSUPP]. Also, calls to *readlink*( ) may fail with [EOPNOTSUPP] instead of [EINVAL].

The server treats the contents of a symbolic link as an opaque object, and the XNFS specification does not define a format for symbolic links. This means that a *readlink*( ) call may return a string that does not look like a path name. It also means that the client may be unable to resolve the symbolic link, causing routines that involve path names (*open*( ), *remove*( ), *mkdir*( ), *stat*( ), etc.) to fail if the link is part of the given path.

A client resolves a symbolic link using its own namespace, not the server's namespace. For example, a client will interpret a link to ''/bin/sh'' as referring to the client's /bin/sh, even if the link was created on the server. Similarly, a link to ''../new_dir/some_file'' may take the client to a local file or a completely different server if it appears in the top directory of a mounted file system (see Section 2.4.2 on page 15 for an explanation of mounted file systems).

A.21 Interrupted Root File System Service

A client's root file system may be accessed using the NFS protocol. Diskless clients usually operate in this manner, and an individual process may change its root file system to a remote file system using the *chroot*( ) call. If service for the root file system is interrupted, service for other file systems may be interrupted as well. For example, if the server for a diskless client's root file system crashes, the client may be unable to access any file systems, even file systems that are served by a different server, until the crashed server resumes operation.

A.22 Implicit File Access

Several functions in the X/Open System Interfaces and Headers Specification (see reference **XSH**) access files even though they do not take a stream handle, file descriptor or path name as an argument. For example, *getpwnam*( ) and *getgrnam*( ) use the system's user and group databases, respectively. If the databases are implemented as remote files, either through individual mounts or because the client is diskless, the operations performed by these functions are subject to the semantic differences described in this appendix.

Other functions may access files other than those specified in the argument list. For example, *ttyname*( ) and *getcwd*( ) may use calls to *stat*( ) to find a desired file or directory. It is possible that some of these calls to *stat*( ) will inadvertently refer to remote files or directories. Calls to *system*( ) may result in references to remote files, depending on the contents of environment variables such as PATH and ENV. These references to remote files are subject to the semantic differences described in this appendix.

A.23 Multiple Hosts

Software that was written for a local file system environment may make uniqueness assumptions, for example, it may assume that process identifiers are unique. These assumptions need not be true in an XNFS environment.

A.23.1 Process Identifiers

A common approach for generating a unique filename, particularly in shell scripts, is to append the process's process identifier to an application-dependent prefix. This approach can fail if two processes, each with the same identifier on a different host, attempt to create a unique file in a shared directory. The X/Open System Interfaces and Headers Specification (see reference **XSH**) provides several functions, for example, *mkstemp*( ), that can avoid this uniqueness problem. Shell scripts must rely on ad-hoc mechanisms for generating unique files, such as including the host name in the file name.

A.23.2 Unique Daemons

Some system facilities, such as the *crontab* facility or printer spoolers, may assume the existence of a single daemon that is started during system initialization. If the files that the daemon uses are shared using the NFS protocol, the daemon may behave in an unintended manner. For example, if the *crontab* database is accessible from more than one system, the same command may be executed concurrently on different hosts, even if this was not the user's intent.

Appendix B

Open-System Interface Semantics over XNFS

B.1 Introduction

Many of the interfaces described in the X/Open System Interfaces and Headers Specification (see reference **XSH**) are directly or indirectly concerned with accessing files stored on the system's file system. This appendix identifies those interfaces which may operate differently when used with XNFS.

Section B.2 on page 288 lists those functions that show no semantic differences when invoked in an XNFS environment. That is, they behave the same as when running in a local file system environment. Section B.3 on page 292 lists those functions that may show a semantic difference when invoked in an XNFS environment. Appendix A explains the differences that can cause a function to appear in Section B.3 on page 292. There is only one distinct XNFS error, [ESTALE]. This error occurs when a remote open file or directory is deleted and its file handle becomes invalid. (See Section A.6 on page 274.)

Another difference concerning errors is that [EACCES] and [EPERM] may occur in unexpected situations due to the stateless nature of NFS. (See the discussions in Section A.3 on page 272, Section A.4 on page 273, Section A.6.1 on page 274 and Section A.10 on page 277.)

In all operations that modify a file or directory, the error [EROFS] may be returned. (See the discussion in Section A.17 on page 283 for more information.)

For a complete list of the non-NFS related errors, as well as the mechanism by which a process can ascertain an error return code, see Section 2.3, Error Numbers in the X/Open System Interfaces and Headers Specification (see reference **XSH**).

B.2 Functions with no Semantic Differences

The following functions have no change over XNFS:

Functions with no Semantic Differences		
a64l( )	*abs*( )	*acos*( )
acosh( )	*advance*( )	*alarm*( )
asctime( )	*asctime_r*( )	*asin*( )
asinh( )	*assert*( )	*atanh*( )
atexit( )	*atof*( )	*atoi*( )
atol( )	*basename*( )	*bcmp*( )
bcopy( )	*brk*( )	*bsd_signal*( )
bsearch( )	*btowc*( )	*bzero*( )
calloc( )	*catclose*( )	*cbrt*( )
ceil( )	*cfgetispeed*( )	*cfgetospeed*( )
cfsetispeed( )	*cfsetospeed*( )	*clearerr*( )
clock( )	*clock_getres*( )	*clock_gettime*( )
clock_settime( )	*closedir*( )	*closelog*( )
compile( )	*confstr*( )	*cos*( )
cosh( )	*crypt*( )	*ctermid*( )
ctime( )	*ctime_r*( )	*dbm_clearerr*( )
dbm_error( )	*difftime*( )	*dirname*( )
div( )	*drand48*( )	*dup*( )
ecvt( )	*encrypt*( )	*endgrent*( )
endpwent( )	*erand48*( )	*erf*( )
erfc( )	*exp*( )	*expm1*( )
fabs( )	*fdopen*( )	*fileno*( )
floor( )	*fmod*( )	*fnmatch*( )
fork( )	*mblen*( )	*free*( )
frexp( )	*ftime*( )	*gamma*( )
getcontext( )	*gcvt*( )	*getdtablesize*( )
getegid( )	*getenv*( )	*geteuid*( )
getgid( )	*getgroups*( )	*gethostid*( )
getitimer( )	*getmsg*( )	*getopt*( )
getpagesize( )	*getpass*( )	*getpgid*( )
getpgrp( )	*getpid*( )	*getpmsg*( )
getppid( )	*getpriority*( )	*getrlimit*( )
getrusage( )	*getsid*( )	*getsubopt*( )
gettimeofday( )	*getuid*( )	*globfree*( )
gmtime( )	*gmtime_r*( )	*grantpt*( )
hcreate( )	*hdestroy*( )	*hsearch*( )
hypot( )	*iconv*( )	*iconv_close*( )
ilogb( )	*index*( )	*initstate*( )
insque( )	*ioctl*( )	*isalnum*( )
isalpha( )	*isascii*( )	*isastream*( )
iscntrl( )	*isdigit*( )	*isgraph*( )
islower( )	*isnan*( )	*isprint*( )
ispunct( )	*isspace*( )	*isupper*( )
iswalnum( )	*iswcntrl*( )	*iswctype*( )
iswdigit( )	*iswgraph*( )	*iswlower*( )
iswprint( )	*iswpunct*( )	*iswspace*( )
iswupper( )	*iswxdigit*( )	*isxdigit*( )
j0( )	*j1*( )	*jn*( )
jrand48( )	*l64a*( )	*labs*( )

Functions with no Semantic Differences		
ldiv( )	*lcong48*( )	*ldexp*( )
lfind( )	*lgamma*( )	*localeconv*( )
localtime( )	*localtime_r*( )	*log*( )
logb( )	*log1p*( )	*log10*( )
logb( )	*longjump*( )	*lrand48*( )
lsearch( )	*makecontext*( )	*malloc*( )
mblen( )	*mbrlen*( )	*mbrtowc*( )
mbsinit( )	*mbsrtowcs*( )	*mbstowcs*( )
mbtowc( )	*memccpy*( )	*memchr*( )
memcmp( )	*memcpy*( )	*memmove*( )
memprotect( )	*memset*( )	*mktime*( )
mlock( )	*mlockall*( )	*modf*( )
mq_close( )	*mq_getattr*( )	*mq_notify*( )
mq_open( )	*mq_receive*( )	*mq_send*( )
mq_setattr( )	*mq_unlink*( )	*mrand48*( )
msgctl( )	*msgget*( )	*msgrcv*( )
msgsnd( )	*munlock*( )	*munlockall*( )
nanosleep( )	*openlog*( )	*nextafter*( )
nice( )	*nl_langinfo*( )	*nrand48*( )
pause( )	*pclose*( )	*pipe*( )
poll( )	*pow*( )	*pthread_attr_setstacksize*( )
pthread_atfork( )	*pthread_attr_destroy*( )	*pthread_attr_getdetachstate*( )
pthread_attr_getinheritsched( )	*pthread_attr_getschedparam*( )	*pthread_attr_getschedpolicy*( )
pthread_attr_getscope( )	*pthread_attr_getstackaddr*( )	*pthread_attr_getstacksize*( )
pthread_attr_init( )	*pthread_attr_setdetachstate*( )	*pthread_attr_setinheritsched*( )
pthread_attr_setschedparam( )	*pthread_attr_setschedpolicy*( )	*pthread_attr_setscope*( )
pthread_attr_setstackaddr( )	*pthread_cleanup_pop*( )	*pthread_cleanup_push*( )
pthread_cond_broadcast( )	*pthread_cond_destroy*( )	*pthread_cond_init*( )
pthread_cond_signal( )	*pthread_cond_timedwait*( )	*pthread_cond_wait*( )
pthread_condattr_destroy( )	*pthread_condattr_getpshared*( )	*pthread_condattr_init*( )
pthread_condattr_setpshared( )	*pthread_create*( )	*pthread_detach*( )
pthread_equal( )	*pthread_getschedparam*( )	*pthread_getspecific*( )
pthread_key_create( )	*pthread_key_delete*( )	*pthread_kill*( )
pthread_mutex_destroy( )	*pthread_mutex_getprioceiling*( )	*pthread_mutex_init*( )
pthread_mutex_lock( )	*pthread_mutex_setprioceiling*( )	*pthread_mutex_trylock*( )
pthread_mutex_unlock( )	*pthread_mutexattr_destroy*( )	*pthread_mutexattr_getprioceiling*( )
pthread_mutexattr_getprotocol( )	*pthread_mutexattr_getpshared*( )	*pthread_mutexattr_init*( )
pthread_mutexattr_setprioceiling( )	*pthread_mutexattr_setprotocol*( )	*pthread_mutexattr_setpshared*( )
pthread_once( )	*pthread_self*( )	*pthread_setcancelstate*( )
pthread_setcanceltype( )	*pthread_setschedparam*( )	*pthread_setspecific*( )
pthread_sigmask( )	*ptsname*( )	*putenv*( )
putmsg( )	*putpmsg*( )	*qsort*( )
raise( )	*rand*( )	*random*( )
realloc( )	*re_comp*( )	*re_exec*( )
regcmp( )	*regcomp*( )	*regex*( )
regexec( )	*regerror*( )	*regfree*( )
remainder( )	*remque*( )	*rewind*( )
rindex( )	*rint*( )	*sbrk*( )
scalb( )	*sched_get_priority_max*( )	*sched_get_priority_min*( )
sched_getparam( )	*sched_getscheduler*( )	*sched_rr_get_interval*( )
sched_setparam( )	*sched_setscheduler*( )	*sched_yield*( )
seed48( )	*seekdir*( )	*select*( )
sem_close( )	*semctl*( )	*semget*( )

Functions with no Semantic Differences		
sem_getvalue( )	*sem_init*( )	*semop*( )
sem_open( )	*sem_post*( )	*sem_trywait*( )
sem_unlink( )	*sem_wait*( )	*setbuf*( )
setcontext( )	*setgid*( )	*setgrent*( )
setitimer( )	*setjmp*( )	*setkey*( )
setlocale( )	*setlogmask*( )	*setpgid*( )
setpgrp( )	*setpriority*( )	*setpwent*( )
setregid( )	*setreuid*( )	*setrlimit*( )
setsid( )	*setstate*( )	*setuid*( )
setutxent( )	*setvbuf*( )	*shmat*( )
shmctl( )	*shmdt*( )	*shmget*( )
shm_open( )	*shm_unlink*( )	*sigaction*( )
sigaddset( )	*sigdelset*( )	*sigemptyset*( )
sigfillset( )	*sighold*( )	*sigignore*( )
siginterrupt( )	*sigismember*( )	*siglongjmp*( )
signal( )	*signalstack*( )	*sigpause*( )
sigqueue( )	*sigrelse*( )	*sigstack*( )
sigpending( )	*sigprocmask*( )	*sigset*( )
sigsetjmp( )	*sigsuspend*( )	*sigtimedwait*( )
sigwaitinfo( )	*sin*( )	*sinh*( )
sleep( )	*snprintf*( )	*sprintf*( )
sqrt( )	*srand*( )	*srand48*( )
srandom( )	*sscanf*( )	*step*( )
strcasecmp( )	*strcat*( )	*strchr*( )
strcmp( )	*strcoll*( )	*strcpy*( )
strcspn( )	*strdup*( )	*strerror*( )
strfmon( )	*strftime*( )	*strlen*( )
strncasecmp( )	*strncat*( )	*strncmp*( )
strncpy( )	*strpbrk*( )	*strptime*( )
strrchr( )	*strspn*( )	*strstr*( )
strtod( )	*strtok*( )	*strtol*( )
strtoul( )	*strxfrm*( )	*swab*( )
swapcontext( )	*swprintf*( )	*swscanf*( )
sysconf( )	*syslog*( )	*tan*( )
atan( )	*tanh*( )	*atan2*( )
tcdrain( )	*tcflow*( )	*tcflush*( )
tcgetattr( )	*tcgetpgrp*( )	*tcgetsid*( )
tcsendbreak( )	*tcsetattr*( )	*tcsetpgrp*( )
tdelete( )	*telldir*( )	*tfind*( )
time( )	*timer_create*( )	*timer_delete*( )
timer_getoverrun( )	*timer_gettime*( )	*timer_settime*( )
times( )	*toascii*( )	*_tolower*( )
tolower( )	*_toupper*( )	*toupper*( )
towctrans( )	*towlower*( )	*towupper*( )
tsearch( )	*twalk*( )	*tzset*( )
ualarm( )	*ulimit*( )	*ulockpt*( )
umask( )	*uname*( )	*ungetc*( )
ungetwc( )	*usleep*( )	*valloc*( )
vfork( )	*vsnprintf*( )	*vsprintf*( )
vswprintf( )	*vwprintf*( )	*wait*( )
wait3( )	*waitid*( )	*waitpid*( )
wcrtomb( )	*wcscat*( )	*wcschr*( )
wcscmp( )	*wcscoll*( )	*wcscpy*( )

Functions with no Semantic Differences		
wcscspn( )	*wcsftime*( )	*wcslen*( )
wcsncat( )	*wcsncmp*( )	*wcsncpy*( )
wcspbrk( )	*wcsrchr*( )	*wcsrtombs*( )
wcsspn( )	*wcsstr*( )	*wcstod*( )
wcstok( )	*wcstol*( )	*wcstombs*( )
wcstoul( )	*wcswcs*( )	*wcswidth*( )
wcsxfrm( )	*wctob*( )	*wctomb*( )
wctrans( )	*wctype*( )	*wcwidth*( )
wmemchr( )	*wmemcmp*( )	*wmemcpy*( )
wmemmove( )	*wmemset*( )	*wordexp*( )
wordfree( )	*wprintf*( )	*wscanf*( )
y0( )	*y1*( )	*yn*( )

It should be noted that the functions which provide inter-process communications via message queues, shared memory and semaphores all use a *key_t* to generate the corresponding inter-process communication channel. Objects of type *key_t* are not transferable between different systems and inter-process communications associated with these keys are not available across an NFS system.

B.3 Functions with Semantic Differences

The following functions have semantic differences over XNFS:

Functions with Semantic Differences		
abort( )	*access*( )	*aio_cancel*( )
aio_error( )	*aio_fsync*( )	*aio_read*( )
aio_return( )	*aio_suspend*( )	*aio_write*( )
catgets( )	*catopen*( )	*chdir*( )
chmod( )	*chown*( )	*chroot*( )
close( )	*creat*( )	*cuserid*( )
dbm_close( )	*dbm_delete*( )	*dbm_fetch*( )
dbm_firstkey( )	*dbm_nextkey*( )	*dbm_open*( )
dbm_store( )	*dlclose*( )	*dlerror*( )
dlopen( )	*dlsym*( )	*dup2*( )
endutxent( )	*exec*( )	*exit*( )
fattach( )	*fchdir*( )	*fchmod*( )
fchown( )	*fclose*( )	*fcntl*( )
fdatasync( )	*fdetach*( )	*ferror*( )
feof( )	*fflush*( )	*fgetc*( )
getc_unlocked( )	*getchar_unlocked*( )	*getgrgid_r*( )
getgrnam_r( )	*getlogin_r*( )	*fgetpos*( )
getpwnam_r( )	*getpwuid_r*( )	*fgets*( )
fgetwc( )	*flockfile*( )	*fmtmsg*( )
fopen( )	*fpathconf*( )	*fprintf*( )
fputc( )	*fputs*( )	*fputwc*( )
fputws( )	*fread*( )	*freopen*( )
fscanf( )	*fseek*( )	*fsetpos*( )
fstat( )	*fstatvfs*( )	*fsync*( )
ftell( )	*ftrylockfile*( )	*ftw*( )
funlockfile( )	*fwide*( )	*fwprintf*( )
fwrite( )	*fwscanf*( )	*getc*( )
getchar( )	*getcwd*( )	*getdate*( )
getgrent( )	*getgrgid*( )	*getgrnam*( )
getlogin( )	*getpwent*( )	*getpwnam*( )
getpwuid( )	*gets*( )	*getutxent*( )
getutxid( )	*getutxline*( )	*getw*( )
getwc( )	*getwd*( )	*getwchar*( )
fgetws( )	*fseeko*( )	*ftello*( )
ftok( )	*ftruncate*( )	*glob*( )
iconv_open( )	*isatty*( )	*kill*( )
killpg( )	*lchown*( )	*link*( )
lio_listio( )	*lockf*( )	*lseek*( )
lstat( )	*mkdir*( )	*mknod*( )
mkfifo( )	*mkstemp*( )	*mktemp*( )
mmap( )	*msync*( )	*munmap*( )
nftw( )	*open*( )	*opendir*( )
pathconf( )	*perror*( )	*popen*( )
pread( )	*printf*( )	*pthread_cancel*( )
pthread_exit( )	*pthread_join*( )	*pthread_testcancel*( )
putc( )	*putc_unlocked*( )	*putchar*( )
putchar_unlocked( )	*puts*( )	*pututxline*( )
putw( )	*putwc*( )	*putwchar*( )
pwrite( )	*read*( )	*readdir*( )

Functions with Semantic Differences		
readdir_r( )	*readlink*( )	*readv*( )
realpath( )	*remove*( )	*rename*( )
rewinddir( )	*rmdir*( )	*scanf*( )
stat( )	*statvfs*( )	*symlink*( )
sync( )	*system*( )	*tempnam*( )
tmpfile( )	*tmpnam*( )	*truncate*( )
ttyname( )	*ttyname_r*( )	*ttyslot*( )
unlink( )	*utime*( )	*utimes*( )
vfprintf( )	*vfwprintf*( )	*vprintf*( )
write( )	*writev*( )	

Where the semantic difference described only applies to Issue 4 X/Open Specifications this is identified by shading of the appropriate text. Semantic differences which do not contain this code marking are applicable to both Issue 3 and Issue 4 X/Open Specifications.

Appendix C

Open System Utilities Semantics over XNFS

C.1 Introduction

Many of the interfaces described in the X/Open Commands and Utilities Specification (see reference **XCU**) are directly or indirectly concerned with accessing files stored on the system's file system. This appendix identifies those commands and utilities which may operate differently when used with XNFS.

The table ''Utilities with no Semantic Differences'' below lists the commands and utilities that show no semantic difference when invoked in an XNFS environment. That is, they behave the same as when running in a local file system environment.

Section C.3 on page 297 lists the commands and utilities that may show semantic differences when invoked in an XNFS environment.

Appendix A lists the potential differences that can affect those commands and utilities.

Section C.2 on page 296 lists the differences that are most common.

The following utilities and commands have no change over XNFS:

Utilities with no Semantic Differences			
alias	*banner*	*basename*	*bg*
cal	*col*	*dirname*	*echo*
env	*expr*	*false*	*fg*
getopts	*hash*	*ipcrm*	*ipcs*
jobs	*line*	*locale*	*logger*
lpstat	*mesg*	*nice*	*printf*
read	*renice*	*sleep*	*time*
tr	*true*	*type*	*ulimit*
umask	*unalias*	*uname*	*wait*
xargs			

C.2 Common Semantic Differences

The following differences apply to almost all of the commands and utilities that may show semantic differences under an XNFS environment.

C.2.1 Execution of Remote Files

This applies to all of the utilities and commands. If local execution of a remote utility or command occurs (such as executing the remote image of *cmp* locally), then the following semantic difference applies:

- It is possible that the remote program may be incompatible for execution on a local machine. (See Section A.15.1 on page 279.)

C.2.2 Interruption of any XNFS Operation

All NFS operations involve issuing one or more requests from the client to the server. Therefore the following semantic difference applies to all NFS operations:

- As described in Section A.12 on page 278, a system call which refers to an NFS file or directory may take an arbitrary length of time to complete. However, it is possible for the user to interrupt such an operation if the ‘‘Intr=’’ MountedFileSystem attribute has been set to ‘‘True’’. (See Section A.13 on page 278.)

C.2.3 File Access

Every command or utility that opens and then accesses a remote file is subject to the following semantic differences:

- Access to a file located on a remote file system can be denied, even in the case that the file permissions do not themselves restrict access. (See Section A.3 on page 272.)
- Access to a file on the server may be denied because the attributes in the client cache are more restrictive than the attributes on the server. (See Section A.5.1 on page 274.)
- Further access to an open file may be denied if the file is located on a remote file system and the file has either been deleted or the file attributes have changed. (See Section A.6 on page 274.)
- The deletion of a file on a server may not be immediately recorded on the clients due to directory caching. (See Section A.10 on page 277.)
- The creation of a file on a server may not be immediately noticeable to clients due to directory caching. (See Section A.10 on page 277.)
- During client/server operations, there may be delays of arbitrary duration. These delays can be the result of network traffic and server load or availability. (See Section A.12 on page 278 and Section A.21 on page 284.)
- Access protection mechanisms will not function as intended if the user ID databases are not consistently maintained over the network. (See Section A.16 on page 282.)

Remote file access could conceivably occur with every command or utility if its standard input or standard output is redirected to a remote file.

C.3 Utilities with Semantic Differences

The following is a list of those utilities and commands that have semantic differences:

Utilities with Semantic Differences			
admin	*ar*	*asa*	*at*
awk	*batch*	*bc*	*c89*
calendar	*cancel*	*cat*	*cc*
cd	*cflow*	*chgrp*	*chmod*
chown	*chroot*	*cksum*	*cmp*
comm	*command*	*compress*	*cp*
cpio	*crontab*	*csplit*	*ctags*
cu	*cut*	*cxref*	*date*
dd	*delta*	*df*	*diff*
dircmp	*dis*	*du*	*ed*
egrep	*ex*	*expand*	*fc*
fgrep	*file*	*find*	*fold*
fort77	*fuser*	*gencat*	*get*
getconf	*grep*	*head*	*iconv*
id	*join*	*kill*	*lex*
link	*unlink*	*lint*	*ln*
localedef	*logname*	*lp*	*ls*
m4	*mail*	*mailx*	*make*
man	*mkdir*	*mkfifo*	*more*
mv	*newgrp*	*nl*	*nm*
nohup	*od*	*pack*	*paste*
patch	*pathchk*	*pax*	*pcat*
pg	*pr*	*prs*	*ps*
pwd	*red*	*rm*	*rmdel*
rmdir	*sact*	*sccs*	*sdb*
sed	*sh*	*sort*	*spell*
split	*strings*	*strip*	*stty*
sum	*tabs*	*tail*	*talk*
tar	*tee*	*test*	*touch*
tput	*tsort*	*tty*	*uncompress*
unexpand	*unget*	*uniq*	*unpack*
uucp	*uudecode*	*uuencode*	*uulog*
uuname	*uupick*	*uustat*	*uuto*
uux	*val*	*vi*	*wall*
wc	*what*	*who*	*write*
yacc	*zcat*		

Where the semantic difference described only applies to Issue 4 X/Open Specifications this is identified by shading of the appropriate text. Semantic differences which do not contain this code marking are applicable to both Issue 3 and Issue 4 X/Open Specifications.

Appendix D

Open Systems Transmission Analysis

D.1 Introduction

When an XSI System Interface function is applied to an XNFS file system object, the XNFS client implementation executes a sequence of NFS RPCs to the XNFS server in order to perform the requested operation. This appendix, describing transmission analysis, provides a general indication of the sequence of RPCs which is performed for each XSI System Interface function.

In a rudimentary implementation, it might be possible to define the precise sequence of RPCs that would be performed for each XSI function. However, this is unlikely to be feasible for any commercially available implementation of NFS within an X/Open-compliant system. Practical NFS implementations include various cache mechanisms, the purpose of which is to increase performance by not performing possibly redundant remote procedure calls. However the use of local caches can introduce ‘‘windows’’ in which the client’s view of the state of the file system object is incorrect for a short period of time. For this reason, implementations will usually include mechanisms whereby the administrator of the XNFS client can disable some or all of the cache schemes, usually on a per-file system basis. The attributes of the MountedFileSystem which control this are described in Chapter 2 on page 9.

This chapter describes the sequence of NFS RPCs corresponding to each of the most basic XSI System Interface functions. For example, *read*( ) is described, but *fread*( ) is not. This corresponds to the distinction between ‘‘system calls’’ and ‘‘library routines’’ which is made in some X/Open-compliant systems. For each function, the sequence of RPCs which would occur if all attribute and directory caching were disabled is described. This is followed by comments on the way in which the operation of the attribute or directory caches may alter the sequence.

The following general points should be noted:

- Attribute cache consistency.

  If attribute caching is enabled, an NFSPROC_GETATTR may be performed at any time in order to ensure cache consistency.

- Buffer cache operation.

  Data to be read will usually be buffered. This may mean that extra NFSPROC_READ requests may be issued at any time to load the buffers, and that *read*( ) requests may be satisfied from the buffered data, so that there is no obvious correlation between *read*( ) requests and NFSPROC_READ RPCs. Similarly, data which is written by *write*( ) and similar functions may be buffered, with the NFSPROC_WRITE being deferred. This specification does not define any mechanism whereby this data buffering may be disabled, but it is expected that an implementation will disable data buffering on a file for which record locking is in use.

- Looking up a path.

  Many of the XSI functions are invoked with a path which is to be interpreted as identifying a particular file system object. The process of interpreting a path is handled in a broadly similar manner for all functions, and is denoted by LOOKUP_SEQUENCE in the following descriptions:

- — If the path is relative to the current directory, issue an NFSPROC_GETATTR to verify that the directory is still valid.
- — For each component in the path, issue an NFSPROC_LOOKUP to retrieve the file handle and attributes for the corresponding file system object.

If attribute caching is enabled, the initial NFSPROC_GETATTR call may not be needed. If directory caching is enabled, it may not be necessary to perform the NFSPROC_LOOKUP for some components; on the other hand it may be necessary to issue additional NFSPROC_GETATTR calls to verify that the cached state is up-to-date.

- Relationship to Service Model.

Before an application can invoke XSI functions which refer to an NFS mounted file system, the file system must be mounted as described in Chapter 2 on page 9. This creates a MountedFileSystem object which includes the file handle for the remote file system; this is then used during any LOOKUP_SEQUENCE which traverses the corresponding mount point.

- Version 2 versus Version 3.

This appendix is derived from Version 2 of the NFS protocol. The sequence of RPCs using Version 3 of the protocol bears some resemblance to the sequence using Version 2, but it is complicated by features that exist only in Version 3, such as

- — new RPC procedures (for example, ACCESS and READDIRPLUS)
- — pre- and post-operation attributes
- — asynchronous WRITE calls.

These protocol features are discussed in more detail in Chapter 12 on page 175.

D.2 RPC Calls Generated by Basic XSI Functions

ACCESS

The ACCESS( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- An NFSPROC_GETATTR is performed to retrieve the attributes of the object.

If attribute caching is enabled, the NFSPROC_GETATTR may not be performed.

CHDIR

The CHDIR( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- An NFSPROC_GETATTR is performed to retrieve the attributes of the object to verify that it is a directory.

If attribute caching is enabled, the NFSPROC_GETATTR may not be performed.

CHMOD

The CHMOD( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- An NFSPROC_GETATTR is performed to retrieve the attributes of the object.
- An NFSPROC_SETATTR is performed to update the attributes.

If attribute caching is enabled, the NFSPROC_GETATTR may not be performed.

CHOWN

The CHOWN( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- An NFSPROC_GETATTR is performed to retrieve the attributes of the object.
- An NFSPROC_SETATTR is performed to update the attributes.

If attribute caching is enabled, the NFSPROC_GETATTR may not be performed.

CHROOT

The CHROOT( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- An NFSPROC_GETATTR is performed to retrieve the attributes of the object and verify that it is a directory.

If attribute caching is enabled, the NFSPROC_GETATTR may not be performed.

CLOSE

The CLOSE( ) XSI System Interface function is implemented as follows:

- Any unwritten data is written to the file using NFSPROC_WRITE.
- If the file was unlinked while open, and this is the last reference to the file, it is finally unlinked using NFSPROC_REMOVE.

If the XNFS implementation supports the NoCto attribute for MountedFileSystems, and NoCto for the file system in question is ‘‘true’’, step (i) may be deferred.

CREAT

The CREAT( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- If ‘‘exclusive’’ mode is specified, an NFSPROC_LOOKUP will be performed to verify that the file does not already exist.
- An NFSPROC_CREATE is performed to create the file.

FCNTL

Certain subfunctions of the FCNTL( ) XSI System Interface function will lead to client-server interactions. The subfunctions are those concerned with advisory file locking, F_GETLK, F_SETLK and F_SETLKW. As noted in Chapter 9 on page 117, an implementation may elect to use synchronous or asynchronous requests for locking services. In addition, locking operations may cause the lock manager subsystems to initiate or terminate status monitoring. Here we consider only the case of synchronous lock manager interactions.

- For an F_GETLK request, the local NLM performs a synchronous NLM_TEST RPC to the remote NLM. If there are any conflicting locks on the file, the first of these is returned in the RPC reply.
- For an F_SETLK request to lock a file (with ‘‘l_type’’ set to F_RDLCK or F_WRLCK), the local NLM performs an NLM_LOCK RPC to the remote NLM. In addition, data and attribute caching are disabled for the file, in order that lock/update/unlock sequences can be performed synchronously.
- For an F_SETLK (or F_SETLKW) request to unlock a file (with ‘‘l_type’’ set to F_UNLCK), the local NLM performs an NLM_UNLOCK RPC to the remote NLM.
- For an F_SETLKW request to lock a file (with ‘‘l_type’’ set to F_RDLCK or F_WRLCK), the local NLM performs an NLM_LOCK RPC to the remote NLM with ‘‘block’’ set to true. If the request cannot be granted immediately, the server returns the code ‘‘LCK_BLOCKED’’. When the request can subsequently be honoured, the server NLM will perform an NLM_GRANTED RPC to the client NLM. If the client NLM does not wish to wait for the lock to be granted, it may perform an NLM_CANCEL RPC to the server NLM. As for F_SETLK, data and attribute caching are disabled for the file, in order that lock/update/unlock sequences can be performed synchronously.

FSTAT

The FSTAT( ) XSI System Interface function is implemented as follows:

- An NFSPROC_GETATTR is performed to retrieve the attributes of the object.

If attribute caching is enabled, the NFSPROC_GETATTR may not be performed.

FSYNC

The FSYNC( ) XSI System Interface function is implemented as follows:

- Any unwritten data is written to the file using NFSPROC_WRITE.

LINK

The LINK( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- An NFSPROC_GETATTR is performed to retrieve the attributes of the object.
- An NFSPROC_LINK is performed to create the link.

If attribute caching is enabled, the NFSPROC_GETATTR may not be performed.

LSEEK

The LSEEK( ) XSI System Interface function is implemented as follows:

- An NFSPROC_GETATTR is performed to retrieve the attributes of the object.

If attribute caching is enabled, the NFSPROC_GETATTR may not be performed.

MKDIR

The MKDIR( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- An NFSPROC_MKDIR is performed to create the directory.
- An NFSPROC_SETATTR is performed to set the group ID for the newly-created directory. (This step is necessary to accommodate a problem with certain server implementations.)

MKFIFO

The MKFIFO( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- An NFSPROC_CREATE is performed to create the file system object.

OPEN

The OPEN( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- An NFSPROC_GETATTR is performed to retrieve the attributes of the object.
- If ‘‘truncate’’ mode is required, an NFSPROC_CREATE is performed to create a new file.

If attribute caching is enabled, the NFSPROC_GETATTR may not be performed.

OPENDIR

The OPENDIR( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- An NFSPROC_GETATTR is performed to retrieve the attributes of the object to verify that it is a directory.

If attribute caching is enabled, the NFSPROC_GETATTR may not be performed.

PATHCONF

The PATHCONF( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.

READ

The READ( ) XSI System Interface function is implemented as follows:

- An NFSPROC_READ is performed to retrieve the data.

If data caching is enabled, the READ( ) may be satisfied using cached data. Note that the use of advisory file locking will disable caching for a file; see FCNTL( ) for more details.

READDIR

The READDIR( ) XSI System Interface function is implemented as follows:

- An NFSPROC_READDIR is performed to retrieve the next directory entry.

If directory caching is enabled, the NFSPROC_READDIR may be satisfied from the cached results of a previous NFSPROC_READDIR, provided that the directory information has not timed out and is more recent than the last modification time on the directory, which in turn depends on the (possibly cached) directory attributes.

RENAME

The RENAME( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- An NFSPROC_GETATTR is performed to retrieve the attributes of the object.
- An NFSPROC_RENAME is performed to rename the file.

If attribute caching is enabled, the NFSPROC_GETATTR may not be performed.

RMDIR

The RMDIR( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- An NFSPROC_RMDIR is performed to remove the directory.

STAT

The STAT( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- An NFSPROC_GETATTR is performed to retrieve the attributes of the object.

If attribute caching is enabled, the NFSPROC_GETATTR may not be performed.

UNLINK

The UNLINK( ) XSI System Interface function is implemented as follows:

- A LOOKUP sequence is performed to interpret the path.
- If the file is not open on the client system, an NFSPROC_REMOVE is performed to remove the file.
- If the file is open on the client system, an NFSPROC_RENAME is performed to rename the file temporarily.

WRITE

The WRITE( ) XSI System Interface function is implemented as follows:

- An NFSPROC_WRITE is performed to write the data.

If data caching is enabled, the data may be buffered and not written out immediately. In this case, if attribute caching is enabled, the cached attributes may be updated locally to reflect the new file size. Note that the use of advisory file locking will disable caching for a file; see FCNTL( ) for more details.

Appendix E

WebNFS Extensions

E.1 Introduction

The WebNFS extensions are an optional set of extensions to the NFS protocol. They provide additional semantics that can be applied to versions 2 and 3 of the protocol to: eliminate the overhead of PORTMAP and MOUNT protocols, make the protocol easier to use where firewall transit is required, and reduce the number of LOOKUP requests required to identify a particular file on the server.

This Appendix describes the changes to an NFS implementation that are needed to support the WebNFS extensions. It also describes NFS URLs, which can be used by WebNFS clients, including web browsers, to specify files on WebNFS servers.

E.2 TCP versus UDP

The NFS protocol is most well known for its use of UDP, which performs acceptably on local area networks. However, on wide area networks with error prone, high-latency connections and bandwidth contention, TCP is well respected for its congestion control and superior error handling. A growing number of NFS implementations now support the NFS protocol over TCP connections.

Version 3 of the NFS protocol is particularly well matched to the use of TCP as a transport protocol. Version 3 removes the arbitrary 8k transfer size limit of version 2, allowing the READ or WRITE of very large streams of data over a TCP connection. Note that version 2 is also supported on TCP connections, though the benefits of TCP data streaming will not be as great.

A WebNFS client must first attempt to connect to its server with a TCP connection. If the server refuses the connection, the client should attempt to use UDP.

E.3 Well-Known Port

While Internet protocols are generally identified by registered port number assignments, RPC-based protocols register a 32-bit program number and a dynamically assigned port with the portmap service which is registered on the well-known port 111. Since the NFS protocol is RPC-based, NFS servers register their port assignment with the portmap service.

NFS servers are constrained by a requirement to re-register at the same port after a server crash and recovery so that clients can recover simply by retransmitting an RPC request until a response is received. This is simpler than the alternative of having the client repeatedly check with the portmap service for a new port assignment. NFS servers typically achieve this port invariance by registering a constant port assignment, 2049, for both UDP and TCP.

To avoid the overhead of contacting the server's portmap service, and to facilitate transit through packet filtering firewalls, WebNFS clients optimistically assume that NFS servers register on port 2049. Most NFS servers use this port assignment already, so this client optimism is well justified.

E.4 Server Port Monitoring

Some NFS servers accept requests only from reserved UDP or TCP ports, that is, port numbers below 1024. These *privileged* ports are available only to UNIX processes with *Super-user* permissions. Requests that do not originate from the range of reserved ports are rejected. This is an optimistic way of preventing direct access to the server from user processes that may attempt to spoof AUTH_UNIX RPC credentials.

WebNFS clients are not required to use reserved ports. This means that a WebNFS server must not check the originating port for requests to filesystems which are made available to WebNFS clients.

E.5 Public Filehandle

NFS filehandles are normally created by the server and used to identify uniquely a particular file or directory on the server. The client does not normally create filehandles or have any knowledge of the contents of a filehandle.

The public filehandle is an an exception. It is an NFS filehandle with a reserved value and special semantics. A WebNFS client uses the public filehandle as an initial filehandle rather than using the MOUNT protocol. Since versions 2 and 3 of the NFS protocol have different filehandle formats, the public filehandle is defined differently for each.

E.5.1 NFS Version 2 Public Filehandle

A version 2 filehandle is an opaque value occupying 32 octets. A version 2 public filehandle has a zero in each octet (that is, all zeros).

1																															32
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

E.5.2 NFS Version 3 Public Filehandle

A version 3 filehandle is defined in as a variable length opaque value occupying up to 64 octets. The length of the filehandle is indicated by an integer value contained in a 4 octet value which describes the number of valid octets that follow. A version 3 public filehandle has a length of zero.

0	0	0	0

E.6 Multi-Component Lookup

Normally the NFS LOOKUP request (version 2 or 3) takes a directory filehandle along with the name of a directory member, and returns the filehandle of the directory member. If a client needs to evaluate a pathname that contains a sequence of components, then beginning with the directory filehandle of the first component it must issue a series of LOOKUP requests one component at a time. For example, evaluation of the path ‘‘a/b/c’’ will generate separate LOOKUP requests for each component of the pathname ‘‘a’’, ‘‘b’’, and ‘‘c’’.

A LOOKUP request that uses the public filehandle can provide a pathname containing multiple components. The server is expected to evaluate the entire pathname and return a filehandle for the final component.

For example, rather than evaluate the path ‘‘a/b/c’’ as:

```
LOOKUP  FH=0x0  "a"  →
                     ←  FH=0x1
LOOKUP  FH=0x1  "b"  →
                     ←  FH=0x2
LOOKUP  FH=0x2  "c"  →
                     ←  FH=0x3
```

Relative to the public filehandle, these 3 LOOKUP requests can be replaced by a single multi-component lookup:

```
LOOKUP  FH=0x0  "a/b/c"  →
                         ←  FH=0x3
```

Multi-component lookup is supported only for LOOKUP requests relative to the public filehandle.

The *url-path* of the NFS URL (see Section E.7.1 on page 311 must be evaluated as a multi-component lookup. This implies that the path components are delimited by slashes, and the characters that make up the path must be in the printable US-ASCII character set.

If the *url-path* is empty, the client must send a multi-component lookup for the pathname ‘‘.’’ (dot).

E.6.1 Canonical Path versus Native Path

If the pathname in a multi-component LOOKUP request begins with an ASCII character, then it must be a canonical path. A canonical path is a hierarchically-related, slash-separated sequence of components, <directory>/<directory>/.../<name>. Occurrences of the ‘‘/’’ character within a component must be escaped using the escape code %2f. Non-ASCII characters within components must also be escaped using the ‘‘%’’ character to introduce a 2-digit hexadecimal code. Occurrences of the ‘‘%’’ character that do not introduce an encoded character must themselves be encoded with %25.

If the first character of the path is a slash, then the canonical path is evaluated relative to the server’s root directory. If the first character is not a slash, then the path is evaluated relative to the directory with which the public filehandle is associated.

Not all WebNFS servers can support arbitrary use of absolute paths. Clearly, the server cannot return a filehandle if the path identifies a file or directory that is not exported by the server. In addition, servers need not return a filehandle if the path names a file or directory in an exported filesystem different from the one that is associated with the public filehandle.

If the first character of the path is 0x80 (non-ASCII), then the following character is the first in a native path. A native path conforms to the normal pathname syntax of the server. For example:

```
Lookup for Canonical Path:
       LOOKUP FH=0x0 "/a/b/c"
Lookup for Native Path:
       LOOKUP FH=0x0  0x80 "a:b:c"
```

E.7 NFS URL

An NFS URL is based on the Common Internet Scheme Syntax described in Section 3.1 of RFC 1738. It has the general form:

```
nfs://<host>:<port>/<url-path>
```

The :<port> part is optional. If omitted then port 2049 is assumed. The <url-path> is also optional. If it is omitted, then the ‘‘/’’ between <host>:<port> and <url-path> may also be omitted.

The <url-path> is a hierarchical directory path of the form

```
<directory>/<directory>/<directory>/.../<name>
```

The <url-path> must consist only of characters within the US-ASCII character set. Within a <directory> or <name> component, the character ‘‘/’’ is reserved and must be encoded as described in Section 2.2 of RFC 1738. If <url-path> is omitted, it must default to the path "." (dot).

When presented with an NFS URL, a client must use the WebNFS technique described in this document to bind to the server, and evaluate the pathname using the public filehandle and multi-component lookup.

E.7.1 Absolute versus Relative Pathname

A pathname that begins with a slash character is considered *absolute* and will be evaluated relative to the server’s root. A pathname that does not begin with a slash is *relative* and will be evaluated relative to the directory with which the public filehandle is associated.

Note that the ‘‘/’’ in an NFS URL that delimits the <host>:<port> from the <url-path> is not considered part of the pathname. For example, if the public filehandle is associated with the server’s directory ‘‘/a/b/c’’ then the URL:

```
nfs://server/d/e/f
```

will be evaluated with a relative multi-component lookup of the path ‘‘d/e/f’’ relative to the server’s directory ‘‘/a/b/c’’, while the URL:

```
nfs://server//a/b/c/d/e/f
```

will locate the same file with an absolute multi-component lookup of the path ‘‘/a/b/c/d/e/f’’ relative to the server’s filesystem root. Notice that a double slash is required at the beginning of the path; the first slash is the URL delimiter between the <host>:<port> and the <url-path>, and the second slash is the first character of <url-path>.

Not all WebNFS servers can support arbitrary use of absolute paths. Clearly, the server must not return a filehandle if the path identifies a file or directory that is not exported by the server. In addition, servers need not return a filehandle if the path names a file or directory in an exported filesystem different from the one that is associated with the public filehandle.

E.7.2 Symbolic Links

The NFS protocol supports symbolic links, which are the filesystem equivalent of a relative URL. If a WebNFS client retrieves a filehandle for a symbolic link (as indicated by the file type attribute) then it should send a READLINK request to the server to retrieve the path comprising the symbolic link.

This path should then be combined with the URL which referenced the symbolic link according to the rules described in RFC 1808. If the relative URL in the symbolic link text is to be resolved successfully then it must contain only ASCII characters and conform to the syntax described in RFC 1808. Note that this allows a symbolic link to contain an entire URL and it may specify a scheme that is not necessarily an NFS URL (for example, HTTP).

An exception to RFC 1808 rules applies in the case of an absolute symbolic link, where the path begins with a ‘‘/’’. RFC 1808 describes a method for resolving relative URLs with respect to the base URL. Given a base URL of ‘‘nfs://s/a/b/c’’ which references a symbolic link with contents ‘‘/a/b/c/d’’, the method would yield a URL ‘‘nfs://s/a/b/c/’’ which would be correct only if the public filehandle were co-located with the server’s filesystem root.

If the symbolic link begins with a slash, then after resolving a relative URL derived from the symbolic link contents according to the method in RFC 1808, the client must insert an additional slash in front of the path so that the server will evaluate the path relative to the server’s root, rather than the public filehandle directory. This variation from the normal method of resolving a relative URL applies only to handling of symbolic links. The additional slash must not be inserted if the relative URL was embedded in a document or other encapsulating entity.

For example, if the symbolic link is named by the URL:

```
nfs://server/a/b
```

then the the following examples show how a new URL can be formed from the symbolic link text:

```
c                    = nfs://server/a/c

c/d                  = nfs://server/a/c/d

../c                 = nfs://server/c

/c/d                 = nfs://server//c/d

nfs://server2/a/b    = nfs://server2/a/b
```

E.7.3 Export Spanning Pathnames

The server may evaluate a pathname, either through a multi-component LOOKUP or as a symbolic link embedded in a pathname, that references a file or directory outside of the exported hierarchy.

Clearly, if the destination of the path is not in an exported filesystem, then the server must return an error to the client.

Many NFS server implementations rely on the MOUNT protocol for checking access to exported filesystems, and their NFS server does no access checking. The NFS server assumes that the filehandle does double duty: identifying a file as well as being a security token. Since WebNFS clients do not normally use the MOUNT protocol, a server that relies on MOUNT checking cannot automatically grant access to another exported filesystem at the destination of a spanning path. These servers must return an error.

For example: suppose the server exports two filesystems. One is associated with the public filehandle:

```
/export/this   (public filehandle)

/export/that
```

The server receives a LOOKUP request with the public filehandle that identifies a file or directory in the other exported filesystem:

```
LOOKUP 0x0  "../that/file"
```

or

```
LOOKUP 0x0  "/export/that/file"
```

Even though the pathname destination is in an exported filesystem, the server cannot return a filehandle without an assurance that the client's use of this filehandle will be authorized.

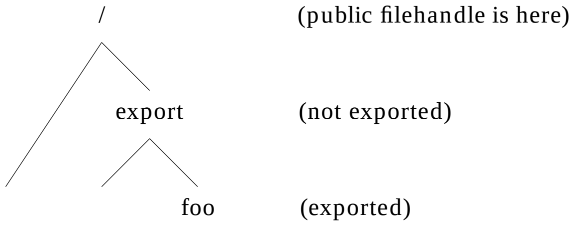
Servers that check client access to an export on every NFS request have more flexibility. These servers can return filehandles for paths that span exports since the client's use of the filehandle for the destination filesystem will be checked by the NFS server.

E.8 Location of Public Filehandle

A server administrator can associate the public filehandle with any file or directory. For instance, a WebNFS server administrator could attach the public filehandle to the root of an anonymous FTP archive, so that anonymous FTP pathnames could be used to identify files in the FTP hierarchy.

On servers that support spanning paths, the public filehandle need not necessarily be attached to an exported directory, though a successful LOOKUP relative to the public filehandle must identify a file or directory that is exported.

For instance, if an NFS server exports a directory ‘‘/export/foo’’ and the public filehandle is attached to the server’s root directory, then a LOOKUP of ‘‘export/foo’’ relative to the public filehandle will return a valid file handle but a LOOKUP of ‘‘export’’ will return an access error since the server’s ‘‘/export’’ directory is not exported.



LOOKUP	0x0	"export"	(returns an error)
LOOKUP	0x0	"export/foo"	(returns a filehandle)

E.9 Contacting the Server

WebNFS clients should be optimistic in assuming that the server supports the WebNFS extensions, but must be capable of fallback to conventional methods for server access if the server does not support them.

The client should start with the assumption that the server supports:

- Version 3 of the NFS protocol
- NFS TCP connections
- Public Filehandles.

If these assumptions are not met, the client should fall back gracefully with a minimum number of messages. The following steps are recommended:

1. Attempt to create a TCP connection to the server's port 2049.

   If the connection fails then assume that a request sent over UDP will work. Use UDP port 2049.

   Do not use the PORTMAP protocol to determine the server's port unless the server does not respond to port 2049 for both TCP and UDP.

2. Assume WebNFS and V3 are supported. Send an NFS version 3 LOOKUP, with the public filehandle for the requested pathname.

   If the server returns an RPC PROG_MISMATCH error then assume that version 3 is not supported. Retry the LOOKUP with a version 2 public filehandle.

   Note: The first call may not necessarily be a LOOKUP if the operation is directed at the public filehandle itself, for example, a READDIR or READDIRPLUS of the directory that is associated with the public filehandle.

   If the server returns an NFS3ERR_STALE, NFS3ERR_INVAL, or NFS3ERR_BADHANDLE error, then assume that the server does not support the WebNFS extensions because it does not recognize the public filehandle. The client must use the server's portmap service to locate and use the MOUNT protocol to obtain an initial filehandle for the requested path.

WebNFS clients can benefit by caching information about the server:

- Whether the server supports TCP connections (if TCP is supported then the client should cache the TCP connection as well)
- Which protocol the server supports
- Whether the server supports public filehandles

If the server does not support public filehandles, the client may choose to cache the port assignment of the MOUNT service as well as previously used pathnames and their filehandles.

E.10 Mount Protocol

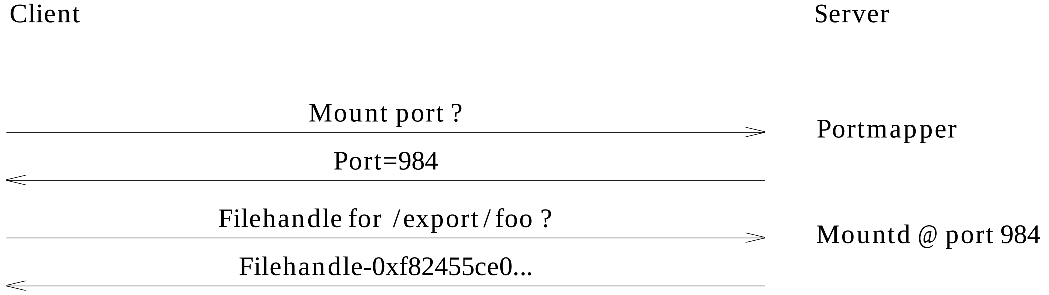
The NFS URL may have limited use for naming files on servers that do not support the public filehandle and multi-component lookup.

If the server returns an NFS3ERR_STALE, NFS3ERR_INVAL, or NFS3ERR_BADHANDLE error in response to the client's use of a public filehandle, then the client should attempt to resolve the <url-path> to a filehandle using the MOUNT protocol.

Note that the pathname sent to the server in the MOUNTPROC_MNT request is assumed to be a server native path, rather than a slash-separated path described by RFC 1738. Hence, the MOUNT protocol can reasonably be expected to map a <url-path> to a filehandle only on servers that support slash-separated ASCII native paths. In general, servers that do not support WebNFS access or slash-separated ASCII native paths should not advertise NFS URLs.

At this point the client must already have some indication as to which version of the NFS protocol is supported on the server. Since the filehandle format differs between NFS versions 2 and 3, the client must select the appropriate version of the MOUNT protocol. MOUNT versions 1 and 2 return only NFS version 2 filehandles, whereas MOUNT version 3 returns NFS version 3 filehandles.

Unlike the NFS service, the MOUNT service is not registered on a well-known port. The client must use the PORTMAP service to locate the server's MOUNT port before it can transmit a MOUNTPROC_MNT request to retrieve the filehandle corresponding to the requested path. This is described in the following diagram:



NFS servers commonly use a client's successful MOUNTPROC_MNT request as an indication that the client has *mounted* the filesystem and may maintain this information in a file that lists the filesystems that clients currently have mounted. This information is removed from the file when the client transmits a MOUNTPROC_UMNT request. Upon receiving a successful reply to a MOUNTPROC_MNT request, a WebNFS client should send a MOUNTPROC_UMNT request to prevent an accumulation of *mounted* records on the server.

Glossary

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. They are 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 largescale 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.

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.

broadcast

To broadcast a packet is 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.

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.

connectionless service

In a connectionless 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 connectionless 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 recalculated 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; for example, 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.

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.

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.

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

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

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.

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.

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 and military 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; for example, formatted data streams or shared memory.

LAN
(Local Area Network) A physical network that operates at a high speed over short distances; for example, Ethernet.

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.

Mount Protocol
This protocol obtains a file handle from the server for the root of a file system which will then be available through NFS.

Multi-component Lookup
The filename used in a lookup request that uses a public filehandle may contain a slash-separated pathname. The server is required to evaluate this pathname (crossing intermediate mountpoints and evaluating symbolic links if necessary) and return the filehandle for the final component.

NFS
(Network File System) A protocol which allows a set of computers access to each others file systems. NFS was developed by Sun Microsystems, Inc. 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.

NSFnet
(National Science Foundation NETwork) The collection of networks across the United States sponsored by NSF.

NSM
Network Status Monitor.

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.

PID
(Process ID) The number assigned to a process so that it can be uniquely identified.

port mapper
The port mapper is a program that maps RPC program and version numbers to transport-specific port numbers thus providing a dynamic binding capability for remote programs.

Public Filehandle
A filehandle with a known value associated with a specific directory on the server. A "version 2" public filehandle has 32 zero bytes. A "version 3" public filehandle has zero length.

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.

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 file system)
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) A mechanism allowing a client to call a procedure that a remote server executes.

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

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.

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 connectionless protocol. Part of the Internet Protocol Suite.

umask
The XSI process's 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.

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, Inc.

XNFS
The name given to the X/Open Specification for file-sharing services based upon the NFS architecture developed by Sun Microsystems, Inc.

Index