Network Working Group
Request for Comments: 1813
Category: Informational

B. Callaghan
B. Pawlowski
P. Staubach
Sun Microsystems, Inc.
June 1995

NFS Version 3 Protocol Specification

Status of this Memo

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

IESG Note

Internet Engineering Steering Group comment: please note that the IETF is not involved in creating or maintaining this specification. This is the significance of the specification not being on the standards track.

Abstract

This paper describes the NFS version 3 protocol. This paper is provided so that people can write compatible implementations.

Table of Contents

1. Introduction

Sun's NFS protocol provides transparent remote access to shared file systems across 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 of the NFS version 2 protocol exist for a variety of machines, from personal computers to supercomputers. The initial version of the NFS protocol is specified in the Network File System Protocol Specification [RFC1094]. A description of the initial implementation can be found in [Sandberg].

The supporting MOUNT protocol performs the operating system-specific functions that allow clients to attach remote directory trees to a point within the local file system. The mount process also allows the server to grant remote access privileges to a restricted set of clients via export control.

The Lock Manager provides support for file locking when used in the NFS environment. The Network Lock Manager (NLM) protocol isolates the inherently stateful aspects of file locking into a separate protocol.

A complete description of the above protocols and their implementation is to be found in [X/OpenNFS].

The purpose of this document is to:

- Specify the NFS version 3 protocol.

- Describe semantics of the protocol through annotation and description of intended implementation.

- Specify the MOUNT version 3 protocol.

- Briefly describe the changes between the NLM version 3 protocol and the NLM version 4 protocol.

The normative text is the description of the RPC procedures and arguments and results, which defines the over-the-wire protocol, and the semantics of those procedures. The material describing implementation practice aids the understanding of the protocol specification and describes some possible implementation issues and solutions. It is not possible to describe all implementations and the UNIX operating system implementation of the NFS version 3 protocol is most often used to provide examples. Given that, the implementation discussion does not bear the authority of the description of the over-the-wire protocol itself.

1.1 Scope of the NFS version 3 protocol

This revision of the NFS protocol addresses new requirements. The need to support larger files and file systems has prompted extensions to allow 64 bit file sizes and offsets. The revision enhances security by adding support for an access check to be done on the server. Performance modifications are of three types:

1. The number of over-the-wire packets for a given set of file operations is reduced by returning file attributes on every operation, thus decreasing the number of calls to get modified attributes.

2. The write throughput bottleneck caused by the synchronous definition of write in the NFS version 2 protocol has been addressed by adding support so that the NFS server can do unsafe writes. Unsafe writes are writes which have not been committed to stable storage before the operation returns. This specification defines a method for committing these unsafe writes to stable storage in a reliable way.

3. Limitations on transfer sizes have been relaxed.

The ability to support multiple versions of a protocol in RPC will allow implementors of the NFS version 3 protocol to define

clients and servers that provide backwards compatibility with the existing installed base of NFS version 2 protocol implementations.

The extensions described here represent an evolution of the existing NFS protocol and most of the design features of the NFS protocol described in [Sandberg] persist. See Changes from the NFS version 2 protocol on page 11 for a more detailed summary of the changes introduced by this revision.

1.2 Useful terms

In this specification, a "server" is a machine that provides resources to the network; a "client" is a machine that accesses resources over the network; a "user" is a person logged in on a client; an "application" is a program that executes on a client.

1.3 Remote Procedure Call

The Sun Remote Procedure Call specification provides a procedure-oriented interface to remote services. Each server supplies a program, which is a set of procedures. The NFS service is one such program. The combination of host address, program number, version number, and procedure number specify one remote service procedure. Servers can support multiple versions of a program by using different protocol version numbers.

The NFS protocol was designed to not require any specific level of reliability from its lower levels so it could potentially be used on many underlying transport protocols. The NFS service is based on RPC which provides the abstraction above lower level network and transport protocols.

The rest of this document assumes the NFS environment is implemented on top of Sun RPC, which is specified in [RFC1057]. A complete discussion is found in [Corbin].

1.4 External Data Representation

The eXternal Data Representation (XDR) specification provides a standard way of representing a set of data types on a network. This solves the problem of different byte orders, structure alignment, and data type representation on different, communicating machines.

In this document, the RPC Data Description Language is used to specify the XDR format parameters and results to each of the RPC service procedures that an NFS server provides. The RPC Data

Description Language is similar to declarations in the C programming language. A few new constructs have been added. The notation:

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

defines name, which is a fixed size block of SIZE bytes, and data, which is a variable sized block of up to DSIZE bytes. This notation indicates fixed-length arrays and arrays with a variable number of elements up to a fixed maximum. A variable-length definition with no size specified means there is no maximum size for the field.

The discriminated union definition:

```
union example switch (enum status) {
     case OK:
        struct {
           filename      file1;
           filename      file2;
           integer       count;
        }
     case ERROR:
        struct {
           errstat       error;
           integer       errno;
        }
     default:
        void;
}
```

defines a structure where the first thing over the network is an enumeration type called status. If the value of status is OK, the next thing on the network will be the structure containing file1, file2, and count. Else, if the value of status is ERROR, the next thing on the network will be a structure containing error and errno. If the value of status is neither OK nor ERROR, then there is no more data in the structure.

The XDR type, hyper, is an 8 byte (64 bit) quantity. It is used in the same way as the integer type. For example:

```
hyper          foo;
unsigned hyper bar;
```

foo is an 8 byte signed value, while bar is an 8 byte unsigned value.

Although RPC/XDR compilers exist to generate client and server stubs from RPC Data Description Language input, NFS implementations do not require their use. Any software that provides equivalent encoding and decoding to the canonical network order of data defined by XDR can be used to interoperate with other NFS implementations.

XDR is described in [RFC1014].

1.5 Authentication and Permission Checking

The RPC protocol includes a slot for authentication parameters on every call. The contents of the authentication parameters are determined by the type of authentication used by the server and client. A server may support several different flavors of authentication at once. The AUTH_NONE flavor provides null authentication, that is, no authentication information is passed. The AUTH_UNIX flavor provides UNIX-style user ID, group ID, and groups with each call. The AUTH_DES flavor provides DES-encrypted authentication parameters based on a network-wide name, with session keys exchanged via a public key scheme. The AUTH_KERB flavor provides DES encrypted authentication parameters based on a network-wide name with session keys exchanged via Kerberos secret keys.

The NFS server checks permissions by taking the credentials from the RPC authentication information in each remote request. For example, using the AUTH_UNIX flavor of authentication, the server gets the user's effective user ID, effective group ID and groups on each call, and uses them to check access. Using user ids and group ids implies that the client and server either share the same ID list or do local user and group ID mapping. Servers and clients must agree on the mapping from user to uid and from group to gid, for those sites that do not implement a consistent user ID and group ID space. In practice, such mapping is typically performed on the server, following a static mapping scheme or a mapping established by the user from a client at mount time.

The AUTH_DES and AUTH_KERB style of authentication is based on a network-wide name. It provides 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. Again, 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 users 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.

The NULL procedure typically requires no authentication.

1.6 Philosophy

This specification defines the NFS version 3 protocol, that is the over-the-wire protocol by which a client accesses a server. The protocol provides a well-defined interface to a server's file resources. A client or server implements the protocol and provides a mapping of the local file system semantics and actions into those defined in the NFS version 3 protocol. Implementations may differ to varying degrees, depending on the extent to which a given environment can support all the operations and semantics defined in the NFS version 3 protocol. Although implementations exist and are used to illustrate various aspects of the NFS version 3 protocol, the protocol specification itself is the final description of how clients access server resources.

Because the NFS version 3 protocol is designed to be operating-system independent, it does not necessarily match the semantics of any existing system. Server implementations are expected to make a best effort at supporting the protocol. If a server cannot support a particular protocol procedure, it may return the error, NFS3ERR_NOTSUP, that indicates that the operation is not supported. For example, many operating systems do not support the notion of a hard link. A server that cannot support hard links should return NFS3ERR_NOTSUP in response to a LINK request. FSINFO describes the most commonly unsupported procedures in the properties bit map. Alternatively, a server may not natively support a given operation, but can emulate it in the NFS version 3 protocol implementation to provide greater functionality.

In some cases, a server can support most of the semantics described by the protocol but not all. For example, the ctime field in the fattr structure gives the time that a file's attributes were last modified. Many systems do not keep this information. In this case, rather than not support the GETATTR operation, a server could simulate it by returning the last modified time in place of ctime. Servers must be careful when simulating attribute information because of possible side effects on clients. For example, many clients use file modification times as a basis for their cache consistency

scheme.

NFS servers are dumb and NFS clients are smart. It is the clients that do the work required to convert the generalized file access that servers provide into a file access method that is useful to applications and users. In the LINK example given above, a UNIX client that received an NFS3ERR_NOTSUP error from a server would do the recovery necessary to either make it look to the application like the link request had succeeded or return a reasonable error. In general, it is the burden of the client to recover.

The NFS version 3 protocol assumes a stateless server implementation. Statelessness means that the server does not need to maintain state about any of its clients in order to function correctly. Stateless servers have a distinct advantage over stateful servers in the event of a crash. With stateless servers, a client need only retry a request until the server responds; the client does not even need to know that the server has crashed. See additional comments in Duplicate request cache on page 99.

For a server to be useful, it holds nonvolatile state: data stored in the file system. Design assumptions in the NFS version 3 protocol regarding flushing of modified data to stable storage reduce the number of failure modes in which data loss can occur. In this way, NFS version 3 protocol implementations can tolerate transient failures, including transient failures of the network. In general, server implementations of the NFS version 3 protocol cannot tolerate a non-transient failure of the stable storage itself. However, there exist fault tolerant implementations which attempt to address such problems.

That is not to say that an NFS version 3 protocol server can't maintain noncritical state. In many cases, servers will maintain state (cache) about previous operations to increase performance. For example, a client READ request might trigger a read-ahead of the next block of the file into the server's data cache in the anticipation that the client is doing a sequential read and the next client READ request will be satisfied from the server's data cache instead of from the disk. Read-ahead on the server increases performance by overlapping server disk I/O with client requests. The important point here is that the read-ahead block is not necessary for correct server behavior. If the server crashes and loses its memory cache of read buffers, recovery is simple on reboot - clients will continue read operations retrieving data from the server disk.

Most data-modifying operations in the NFS protocol are synchronous. That is, when a data modifying procedure returns to the client, the client can assume that the operation has completed and any modified data associated with the request is now on stable storage. For example, a synchronous client WRITE request may cause the server to update data blocks, file system information blocks, and file attribute information - the latter information is usually referred to as metadata. When the WRITE operation completes, the client can assume that the write data is safe and discard it. This is a very important part of the stateless nature of the server. If the server did not flush dirty data to stable storage before returning to the client, the client would have no way of knowing when it was safe to discard modified data. The following data modifying procedures are synchronous: WRITE (with stable flag set to FILE_SYNC), CREATE, MKDIR, SYMLINK, MKNOD, REMOVE, RMDIR, RENAME, LINK, and COMMIT.

The NFS version 3 protocol introduces safe asynchronous writes on the server, when the WRITE procedure is used in conjunction with the COMMIT procedure. The COMMIT procedure provides a way for the client to flush data from previous asynchronous WRITE requests on the server to stable storage and to detect whether it is necessary to retransmit the data. See the procedure descriptions of WRITE on page 49 and COMMIT on page 92.

The LOOKUP procedure is used by the client to traverse multicomponent file names (pathnames). Each call to LOOKUP is used to resolve one segment of a pathname. There are two reasons for restricting LOOKUP to a single segment: it is hard to standardize a common format for hierarchical file names and the client and server may have different mappings of pathnames to file systems. This would imply that either the client must break the path name at file system attachment points, or the server must know about the client's file system attachment points. In NFS version 3 protocol implementations, it is the client that constructs the hierarchical file name space using mounts to build a hierarchy. Support utilities, such as the Automounter, provide a way to manage a shared, consistent image of the file name space while still being driven by the client mount process.

Clients can perform caching in varied manner. The general practice with the NFS version 2 protocol was to implement a time-based client-server cache consistency mechanism. It is expected NFS version 3 protocol implementations will use a similar mechanism. The NFS version 3 protocol has some explicit support, in the form of additional attribute information to eliminate explicit attribute checks. However, caching is not

required, nor is any caching policy defined by the protocol. Neither the NFS version 2 protocol nor the NFS version 3 protocol provide a means of maintaining strict client-server consistency (and, by implication, consistency across client caches).

1.7 Changes from the NFS Version 2 Protocol

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 optimize 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 POSIX pathconf information about a file.

Below 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 which do not utilize 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 error, NFS3ERR_NAMETOOLONG, 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). nfsstat3 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 UNIX major and minor devices numbers. 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 eight-byte unsigned integers from four-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 eight-byte unsigned integer.

Set file attributes
In the NFS version 2 protocol, the settable attributes were represented by a subset of the file attributes structure; the client indicated those attributes which 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 superuser 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 which was made in the NFS version 2 protocol that access to files was based solely on UNIX 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 synchronization 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 which 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 the Unix system df command. 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 synchronization mechanism to be used with asynchronous WRITE operations.

2. RPC Information

2.1 Authentication

The NFS service uses AUTH_NONE in the NULL procedure. AUTH_UNIX, AUTH_DES, or AUTH_KERB are used for all other procedures. Other authentication types may be supported in the future.

2.2 Constants

These are the RPC constants needed to call the NFS Version 3 service. They are given in decimal.

```
PROGRAM  100003
VERSION  3
```

2.3 Transport address

The NFS protocol is normally supported over the TCP and UDP protocols. It uses port 2049, the same as the NFS version 2 protocol.

2.4 Sizes

These are the sizes, given in decimal bytes, of various XDR structures used in the NFS version 3 protocol:

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.

2.5 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. Note that the precise numeric encoding must be followed. 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 specification. 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.

2.6 Defined Error Numbers

A description of each defined error follows:

NFS3_OK
: Indicates the call completed successfully.

NFS3ERR_PERM
: Not owner. The operation was not allowed because the caller is either not a privileged user (root) or not the owner of the target of the operation.

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

NFS3ERR_IO
: I/O error. A hard error (for example, a disk error) occurred while processing the requested operation.

NFS3ERR_NXIO
: I/O error. 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 or privileged user permission failures.

NFS3ERR_EXIST
File exists. The file specified already exists.

NFS3ERR_XDEV
Attempt 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
Illegal NFS file handle. The file handle failed internal consistency checks.

NFS3ERR_NOT_SYNC
Update synchronization mismatch was detected during a SETATTR operation.

NFS3ERR_BAD_COOKIE
READDIR or READDIRPLUS cookie is stale.

NFS3ERR_NOTSUPP
Operation is not supported.

NFS3ERR_TOOSMALL
Buffer or request is too small.

NFS3ERR_SERVERFAULT
An error occurred on the server which does not map to any of the legal NFS version 3 protocol error values. The client should translate this into an appropriate error. UNIX clients 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. The type, NF3REG, is a regular file, NF3DIR is a directory, NF3BLK is a block special device file, NF3CHR is a character special device file, NF3LNK is a symbolic link, NF3SOCK is a socket, and NF3FIFO is a named pipe. Note that the precise enum encoding must be followed.

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. (This is obviously a UNIX-specific interpretation.) For all other file types, these two elements should either be set to 0 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 0. 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 which are not devices, out of band

information can be passed from the server to the client. However, once again, both the server and the client must agree on the values passed.

nfs_fh3

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

The nfs_fh3 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 behavior.

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 error, NFS3ERR_STALE, should be returned.

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 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;
};
```

This 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 main changes from the NFS version 2 protocol are that many of the fields have been widened and the major/minor device information is now presented in a distinct structure rather than being packed into a word.

The fattr3 structure contains the basic attributes of a file. All servers should support this set of attributes even if they have to simulate some of the fields. Type is the type of the file. Mode is the protection mode bits. Nlink is the number of hard links to the file - that is, the number of different names for the same file. Uid is the user ID of the owner of the file. Gid is the group ID of the group of the file. Size is the size of the file in bytes. Used is 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 describes the device file if the file type is NF3CHR or NF3BLK - see specdata3 on page 20. Fsid is the file system identifier for the file system. Fileid is a number which uniquely identifies the file within its file system (on UNIX

this would be the inumber). Atime is the time when the file data was last accessed. Mtime is the time when the file data was last modified. Ctime is 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:

```
0x00800 Set user ID on execution.
0x00400 Set group ID on execution.
0x00200 Save swapped text (not defined in POSIX).
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;
};
```

This 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;
};
```

This is the subset of pre-operation attributes needed to better support the weak cache consistency semantics. Size is the file size in bytes of the object before the operation. Mtime is the time of last modification of the object before the operation. Ctime is the time of last change to the attributes of the object before the operation. See discussion in wcc_attr on page 24.

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 or 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 will need to 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 optimization so that the client is not forced to immediately issue a LOOKUP request 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 settable attributes 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 on page 86, 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 File name component handling on page 101.

3. Server Procedures

The following sections define the RPC procedures that are supplied by an NFS version 3 protocol server. The RPC procedure number is given at the top of the page with the name. The SYNOPSIS provides the name of the procedure, the list of the names of the arguments, the list of the names of the results, followed by the XDR argument declarations and results declarations. The information in the SYNOPSIS is specified in RPC Data Description Language as defined in [RFC1014]. The DESCRIPTION section tells what the procedure

is expected to do and how its arguments and results are used. The ERRORS section lists the errors returned for specific types of failures. These lists are not intended to be the definitive statement of all of the errors which can be returned by any specific procedure, but as a guide for the more common errors which may be returned. Client implementations should be prepared to deal with unexpected errors coming from a server. The IMPLEMENTATION field gives information about how the procedure is expected to work and how it should be used by clients.

```
   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;
```

Out of range (undefined) procedure numbers result in RPC errors. Refer to [RFC1057] for more detail.

3.1 General comments on 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.

3.2 General comments on filenames

The following comments apply to all NFS version 3 protocol procedures in which the client provides one or more filenames in the arguments: LOOKUP, CREATE, MKDIR, SYMLINK, MKNOD, REMOVE, RMDIR, RENAME, and LINK.

1. The filename must not be null nor may it be the null string. The server should return the error, NFS3ERR_ACCES, if it receives such a filename. On some clients, the filename, '''' or a null string, is assumed to be an alias for the current directory. Clients which 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 "." is assumed to be an alias for the current directory. Clients which 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 ".." is assumed to be an alias for the parent of the current directory, i.e. the directory which contains the current directory. The server should be prepared to handle this semantic, if it supports directories, even if those directories do not contain UNIX-style "." or ".." entries.

4. If the filename is longer than the maximum for the file system (see PATHCONF on page 90, 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 error, NFS3ERR_NAMETOOLONG.

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 which will control the client's view of the file system. If the server receives a filename containing characters that it can not handle, the error, NFS3ERR_EACCES, should be returned. Client implementations should be prepared to handle this side affect of heterogeneity.

See also comments in File name component handling on page 101.

3.3.0 Procedure 0: NULL - Do nothing

SYNOPSIS

```
void NFSPROC3_NULL(void) = 0;
```

DESCRIPTION

Procedure NULL does not do any work. It is made available to allow server response testing and timing.

IMPLEMENTATION

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.

ERRORS

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.

3.3.1 Procedure 1: GETATTR - Get file attributes

SYNOPSIS

```
GETATTR3res NFSPROC3_GETATTR(GETATTR3args) = 1;

struct GETATTR3args {
   nfs_fh3  object;
};

struct GETATTR3resok {
   fattr3   obj_attributes;
};

union GETATTR3res switch (nfsstat3 status) {
case NFS3_OK:
   GETATTR3resok  resok;
default:
   void;
};
```

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 elsewhere). On entry, the arguments in GETATTR3args are:

object
The file handle of an object whose attributes are to be retrieved.

On 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

The attributes of file system objects is a point of major disagreement between different operating systems. Servers

should 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.

ERRORS

NFS3ERR_IO
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_SERVERFAULT

SEE ALSO

ACCESS.

3.3.2 Procedure 2: SETATTR - Set file attributes

SYNOPSIS

```
SETATTR3res NFSPROC3_SETATTR(SETATTR3args) = 2;

union sattrguard3 switch (bool check) {
case TRUE:
   nfstime3  obj_ctime;
case FALSE:
   void;
};

struct SETATTR3args {
   nfs_fh3      object;
   sattr3       new_attributes;
   sattrguard3  guard;
};

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;
};
```

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.

On 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

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 error, NFS3ERR_NOT_SYNC, 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 0 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 should 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 quite possible that the server will not be able to support the full range of SETATTR requests. The error, NFS3ERR_INVAL, may be returned if the server can not store a uid or gid in its own representation of uids or gids, 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.

ERRORS

NFS3ERR_PERM
NFS3ERR_IO
NFS3ERR_ACCES
NFS3ERR_INVAL
NFS3ERR_NOSPC
NFS3ERR_ROFS
NFS3ERR_DQUOT
NFS3ERR_NOT_SYNC
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_SERVERFAULT

SEE ALSO

CREATE, MKDIR, SYMLINK, and MKNOD.

3.3.3 Procedure 3: LOOKUP - Lookup filename

SYNOPSIS

```
LOOKUP3res NFSPROC3_LOOKUP(LOOKUP3args) = 3;

struct LOOKUP3args {
     diropargs3  what;
};

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;
};
```

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. Refer to General comments on filenames on page 30.

On 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

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 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 mountpoint to the root of a different filesystem, even if the filesystem is exported. This does not prevent a client from accessing a hierarchy of filesystems exported by a server, but the client must mount each of the filesystems individually so that the mountpoint crossing takes place on the client. A given server implementation may refine these rules given capabilities or limitations particular to that implementation. Refer to [X/OpenNFS] for a discussion on exporting file systems.

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 the NFS protocol assumes a hierarchical organization, organized as a single-rooted tree.

Note that this procedure does not follow symbolic links. The client is responsible for all parsing of filenames including filenames that are modified by symbolic links encountered during the lookup process.

ERRORS

NFS3ERR_IO
NFS3ERR_NOENT
NFS3ERR_ACCES
NFS3ERR_NOTDIR
NFS3ERR_NAMETOOLONG
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_SERVERFAULT

SEE ALSO

CREATE, MKDIR, SYMLINK, MKNOD, READDIRPLUS, and PATHCONF.

3.3.4 Procedure 4: ACCESS - Check Access Permission

SYNOPSIS

```
ACCESS3res NFSPROC3_ACCESS(ACCESS3args) = 4;

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;
};

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;
};
```

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, as 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.

ACCESS3_EXECUTE
Execute file (no meaning for a directory).

On 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

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 uid or gid 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 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 NFS client access procedure (called possibly through access(2)). The intent is to make the behavior of opening a remote file more consistent with the behavior 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 Permission issues on page 98 for more information on this topic.

Many implementations do not directly support the ACCESS3_DELETE permission. Operating systems like UNIX 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 0, indicating that the client does not have this permission.

ERRORS

NFS3ERR_IO
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_SERVERFAULT

SEE ALSO

GETATTR.

3.3.5 Procedure 5: READLINK - Read from symbolic link

SYNOPSIS

```
READLINK3res NFSPROC3_READLINK(READLINK3args) = 5;

struct READLINK3args {
     nfs_fh3  symlink;
};

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;
};
```

DESCRIPTION

Procedure READLINK reads the data associated with a symbolic link. The data is an ASCII 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).

On 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

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 path name 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 error, NFS3ERR_INVAL, if the object is not of type, NF3LNK. (Note: The X/Open XNFS Specification for the NFS version 2 protocol defined the error status in this case as NFSERR_NXIO. This is inconsistent with existing server practice.)

ERRORS

NFS3ERR_IO
NFS3ERR_INVAL
NFS3ERR_ACCES
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_NOTSUPP
NFS3ERR_SERVERFAULT

SEE ALSO

READLINK, SYMLINK.

3.3.6 Procedure 6: READ - Read From file

SYNOPSIS

```
READ3res NFSPROC3_READ(READ3args) = 6;

struct READ3args {
     nfs_fh3  file;
     offset3  offset;
     count3   count;
};

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;
};
```

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 0 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 0 and eof set to TRUE, subject to access permissions checking.

count
The number of bytes of data that are to be read. If count is 0, the READ will succeed and return 0 bytes of data, subject to access permissions checking. 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.

On 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

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 EOF. The addition of the eof flag in the NFS version 3 protocol provides a correct way of handling EOF.

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.

ERRORS

NFS3ERR_IO
NFS3ERR_NXIO
NFS3ERR_ACCES
NFS3ERR_INVAL
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_SERVERFAULT

SEE ALSO

READLINK.

3.3.7 Procedure 7: WRITE - Write to file

SYNOPSIS

```
WRITE3res NFSPROC3_WRITE(WRITE3args) = 7;

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<>;
};

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;
};
```

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 0 means to write data starting at the beginning of the file.

count
The number of bytes of data to be written. If count is 0, the WRITE will succeed and return a count of 0, barring errors due to permissions 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 behavior 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 is free to 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 the discussion on COMMIT on page 92 for more information on if and when data is committed to stable storage.

data
The data to be written to the file.

On 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 service 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 synchronize 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

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 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 0 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:

1. Repeated power failures.

2. Hardware failures (of any board, power supply, and so on.).

3. 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.

ERRORS

NFS3ERR_IO
NFS3ERR_ACCES
NFS3ERR_FBIG
NFS3ERR_DQUOT
NFS3ERR_NOSPC
NFS3ERR_ROFS
NFS3ERR_INVAL
NFS3ERR_STALE
NFS3ERR_BADHANDLE

NFS3ERR_SERVERFAULT

SEE ALSO

COMMIT.

3.3.8 Procedure 8: CREATE - Create a file

SYNOPSIS

```
CREATE3res NFSPROC3_CREATE(CREATE3args) = 8;

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;
};

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;
};
```

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. Refer to General comments on filenames on page 30.

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 file should be created 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 should check for the presence of a duplicate file before performing the create and should 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.

On 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

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 error, NFS3ERR_NOTSUPP.

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.

Refer to General comments on filenames on page 30.

ERRORS

NFS3ERR_IO
NFS3ERR_ACCES
NFS3ERR_EXIST
NFS3ERR_NOTDIR
NFS3ERR_NOSPC
NFS3ERR_ROFS
NFS3ERR_NAMETOOLONG
NFS3ERR_DQUOT
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_NOTSUPP
NFS3ERR_SERVERFAULT

SEE ALSO

MKDIR, SYMLINK, MKNOD, and PATHCONF.

3.3.9 Procedure 9: MKDIR - Create a directory

SYNOPSIS

```
MKDIR3res NFSPROC3_MKDIR(MKDIR3args) = 9;

struct MKDIR3args {
     diropargs3   where;
     sattr3       attributes;
};

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;
};
```

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. Refer to General comments on filenames on page 30.

attributes
The initial attributes for the subdirectory.

On 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

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. Refer to General comments on filenames on page 30.

ERRORS

NFS3ERR_IO
NFS3ERR_ACCES
NFS3ERR_EXIST
NFS3ERR_NOTDIR
NFS3ERR_NOSPC
NFS3ERR_ROFS
NFS3ERR_NAMETOOLONG
NFS3ERR_DQUOT
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_NOTSUPP
NFS3ERR_SERVERFAULT

SEE ALSO

CREATE, SYMLINK, MKNOD, and PATHCONF.

3.3.10 Procedure 10: SYMLINK - Create a symbolic link

SYNOPSIS

```
SYMLINK3res NFSPROC3_SYMLINK(SYMLINK3args) = 10;

struct symlinkdata3 {
     sattr3    symlink_attributes;
     nfspath3  symlink_data;
};

struct SYMLINK3args {
     diropargs3    where;
     symlinkdata3  symlink;
};

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;
};
```

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. Refer to General comments on filenames on page 30.

symlink
The symbolic link to create:

symlink_attributes
The initial attributes for the symbolic link.

symlink_data
The string containing the symbolic link data.

On 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

Refer to General comments on filenames on page 30.

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.

ERRORS

NFS3ERR_IO
NFS3ERR_ACCES
NFS3ERR_EXIST
NFS3ERR_NOTDIR
NFS3ERR_NOSPC
NFS3ERR_ROFS
NFS3ERR_NAMETOOLONG
NFS3ERR_DQUOT
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_NOTSUPP
NFS3ERR_SERVERFAULT

SEE ALSO

READLINK, CREATE, MKDIR, MKNOD, FSINFO, and PATHCONF.

3.3.11 Procedure 11: MKNOD - Create a special device

SYNOPSIS

```
MKNOD3res NFSPROC3_MKNOD(MKNOD3args) = 11;

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;
};
```

```
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;
};
```

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. Refer to General comments on filenames on page 30.

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 structure devicedata3 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.

On 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

Refer to General comments on filenames on page 30.

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 error, NFS3ERR_NOTSUPP, should be returned. Otherwise, if the server does not support the target type or the target type is illegal, the error, NFS3ERR_BADTYPE, should be returned. Note that NF3REG, NF3DIR, and NF3LNK are illegal types for MKNOD. The procedures, CREATE, MKDIR, and SYMLINK should be used to create these file types, respectively, instead of MKNOD.

ERRORS

NFS3ERR_IO
NFS3ERR_ACCES
NFS3ERR_EXIST
NFS3ERR_NOTDIR
NFS3ERR_NOSPC
NFS3ERR_ROFS
NFS3ERR_NAMETOOLONG
NFS3ERR_DQUOT
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_NOTSUPP
NFS3ERR_SERVERFAULT
NFS3ERR_BADTYPE

SEE ALSO

CREATE, MKDIR, SYMLINK, and PATHCONF.

3.3.12 Procedure 12: REMOVE - Remove a File

SYNOPSIS

```
REMOVE3res NFSPROC3_REMOVE(REMOVE3args) = 12;

struct REMOVE3args {
     diropargs3  object;
};

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;
};
```

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. Refer to General comments on filenames on page 30.

On 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

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 should 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.

Refer to General comments on filenames on page 30.

ERRORS

NFS3ERR_NOENT
NFS3ERR_IO
NFS3ERR_ACCES
NFS3ERR_NOTDIR
NFS3ERR_NAMETOOLONG

```
      NFS3ERR_ROFS
      NFS3ERR_STALE
      NFS3ERR_BADHANDLE
      NFS3ERR_SERVERFAULT
```

SEE ALSO

RMDIR and RENAME.

3.3.13 Procedure 13: RMDIR - Remove a Directory

SYNOPSIS

```
RMDIR3res NFSPROC3_RMDIR(RMDIR3args) = 13;

struct RMDIR3args {
     diropargs3  object;
};

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;
};
```

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. Refer to General comments on filenames on page 30.

On 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

Note that on some servers, removal of a non-empty directory is disallowed.

On some servers, the filename, ".", is illegal. These servers will return the error, NFS3ERR_INVAL. On some servers, the filename, "..", is illegal. These servers will return the error, NFS3ERR_EXIST. This would seem inconsistent, but allows these servers to comply with their own specific interface definitions. Clients should 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.

ERRORS

NFS3ERR_NOENT
NFS3ERR_IO
NFS3ERR_ACCES
NFS3ERR_INVAL
NFS3ERR_EXIST
NFS3ERR_NOTDIR
NFS3ERR_NAMETOOLONG
NFS3ERR_ROFS
NFS3ERR_NOTEMPTY
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_NOTSUPP
NFS3ERR_SERVERFAULT

SEE ALSO

REMOVE.

3.3.14 Procedure 14: RENAME - Rename a File or Directory

SYNOPSIS

```
RENAME3res NFSPROC3_RENAME(RENAME3args) = 14;

struct RENAME3args {
     diropargs3   from;
     diropargs3   to;
};

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;
};
```

DESCRIPTION

Procedure RENAME renames the file identified by from.name in the directory, from.dir, to to.name in the di- rectory, to.dir. The operation is required to be atomic to the client. To.dir and from.dir must reside on the same file system and server. On entry, the arguments in RENAME3args are:

from
A diropargs3 structure identifying the source (the file system object to be re-named):

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. Refer to General comments on filenames on page 30.

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. Refer to General comments on filenames on page 30.

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 error, NFS3ERR_EXIST.

On 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

The RENAME operation must be atomic to the client. The message "to.dir and from.dir must reside on the same file system on the server, [or the operation will fail]" means that the fsid fields in the attributes for the directories are the same. If they reside on different file systems, the error, NFS3ERR_XDEV, 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 illegal 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 error, NFS3ERR_INVAL, in these cases.

If from and to both refer to the same file (they might be hard links of each other), then RENAME should perform no action and return NFS3_OK.

Refer to General comments on filenames on page 30.

ERRORS

```
NFS3ERR_NOENT
NFS3ERR_IO
NFS3ERR_ACCES
NFS3ERR_EXIST
NFS3ERR_XDEV
NFS3ERR_NOTDIR
NFS3ERR_ISDIR
NFS3ERR_INVAL
NFS3ERR_NOSPC
NFS3ERR_ROFS
NFS3ERR_MLINK
NFS3ERR_NAMETOOLONG
NFS3ERR_NOTEMPTY
NFS3ERR_DQUOT
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_NOTSUPP
NFS3ERR_SERVERFAULT
```

SEE ALSO

REMOVE and LINK.

3.3.15 Procedure 15: LINK - Create Link to an object

SYNOPSIS

```
LINK3res NFSPROC3_LINK(LINK3args) = 15;

struct LINK3args {
     nfs_fh3     file;
     diropargs3  link;
};

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;
      };
```

DESCRIPTION

Procedure LINK creates a hard link from file to link.name, in the directory, link.dir. 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. Refer to General comments on filenames on page 17.

On 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

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. Refer to General comments on filenames on page 30.

ERRORS

NFS3ERR_IO
NFS3ERR_ACCES
NFS3ERR_EXIST
NFS3ERR_XDEV
NFS3ERR_NOTDIR
NFS3ERR_INVAL
NFS3ERR_NOSPC
NFS3ERR_ROFS
NFS3ERR_MLINK
NFS3ERR_NAMETOOLONG
NFS3ERR_DQUOT
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_NOTSUPP
NFS3ERR_SERVERFAULT

SEE ALSO

SYMLINK, RENAME and FSINFO.

3.3.16 Procedure 16: READDIR - Read From Directory

SYNOPSIS

```
READDIR3res NFSPROC3_READDIR(READDIR3args) = 16;

struct READDIR3args {
     nfs_fh3      dir;
     cookie3      cookie;
     cookieverf3  cookieverf;
     count3       count;
};
```

```
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;
};
```

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
This should be set to 0 in the first request to read the directory. On subsequent requests, it should be a cookie as returned by the server.

cookieverf
This should be set to 0 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 is free to return less than count bytes of data.

On 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

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 READDIRs 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 0. The response includes a new cookie verifier, with a cookie per entry. For subsequent READDIRs, 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. This might be overly restrictive, however. A better approach would be to record the time of the last directory modification that changed the directory organization in a way that would make it impossible to reliably interpret a cookie. 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.

ERRORS

NFS3ERR_IO
NFS3ERR_ACCES
NFS3ERR_NOTDIR
NFS3ERR_BAD_COOKIE
NFS3ERR_TOOSMALL
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_SERVERFAULT

SEE ALSO

READDIRPLUS and FSINFO.

3.3.17 Procedure 17: READDIRPLUS - Extended read from directory

SYNOPSIS

```
READDIRPLUS3res NFSPROC3_READDIRPLUS(READDIRPLUS3args) = 17;

struct READDIRPLUS3args {
     nfs_fh3      dir;
     cookie3      cookie;
     cookieverf3  cookieverf;
     count3       dircount;
     count3       maxcount;
};

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;
};
```

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
This should be set to 0 on the first request to read a directory. On subsequent requests, it should be a cookie as returned by the server.

cookieverf
This should be set to 0 on the first request to read a 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 returned. This number should 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 is free to return fewer than maxcount bytes of data.

On 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

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 which are entered into caches) and over-the-wire overhead versus expected subsequent LOOKUP elimination. It is thought that this procedure may improve performance for directory browsing where attributes are always required as on the Apple Macintosh operating system and for MS-DOS.

The dircount and maxcount fields are included as an optimization. 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.

ERRORS

NFS3ERR_IO
NFS3ERR_ACCES
NFS3ERR_NOTDIR
NFS3ERR_BAD_COOKIE
NFS3ERR_TOOSMALL
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_NOTSUPP
NFS3ERR_SERVERFAULT

SEE ALSO

READDIR.

3.3.18 Procedure 18: FSSTAT - Get dynamic file system information

SYNOPSIS

```
FSSTAT3res NFSPROC3_FSSTAT(FSSTAT3args) = 18;

struct FSSTAT3args {
     nfs_fh3   fsroot;
};

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;
};
```

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.

On 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 inodes 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: this is the number of seconds for which the file system is not expected to change. For a volatile, frequently updated file system, this will be 0. 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, 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

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.

ERRORS

NFS3ERR_IO
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_SERVERFAULT

SEE ALSO

FSINFO.

3.3.19 Procedure 19: FSINFO - Get static file system Information

SYNOPSIS

```
FSINFO3res NFSPROC3_FSINFO(FSINFO3args) = 19;

const FSF3_LINK        = 0x0001;
const FSF3_SYMLINK     = 0x0002;
const FSF3_HOMOGENEOUS = 0x0008;
const FSF3_CANSETTIME  = 0x0010;

struct FSINFOargs {
     nfs_fh3   fsroot;
};

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;
      };
```

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.

On 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 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 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 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 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 0 (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 0 (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

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.

ERRORS

NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_SERVERFAULT

SEE ALSO

READLINK, WRITE, READDIR, FSSTAT and PATHCONF.

3.3.20 Procedure 20: PATHCONF - Retrieve POSIX information

SYNOPSIS

```
PATHCONF3res NFSPROC3_PATHCONF(PATHCONF3args) = 20;

struct PATHCONF3args {
     nfs_fh3   object;
};

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;
};
```

DESCRIPTION

Procedure PATHCONF retrieves the 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.

On 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 of a component of a filename.

no_trunc
If TRUE, the server will reject any request that includes a name longer than name_max with the error, NFS3ERR_NAMETOOLONG. If FALSE, any length 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 is not the privileged user. (Uid 0.)

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

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.

ERRORS

NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_SERVERFAULT

SEE ALSO

LOOKUP, CREATE, MKDIR, SYMLINK, MKNOD, RENAME, LINK and FSINFO.

3.3.21 Procedure 21: COMMIT - Commit cached data on a server to stable storage

SYNOPSIS

```
COMMIT3res NFSPROC3_COMMIT(COMMIT3args) = 21;

struct COMMIT3args {
     nfs_fh3    file;
     offset3    offset;
     count3     count;
};

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;
};
```

DESCRIPTION

Procedure COMMIT forces or flushes data to stable storage 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 0 means to flush data starting at the beginning of the file.

count
The number of bytes of data to flush. If count is 0, a flush from offset to the end of file is done.

On 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

Procedure COMMIT is similar in operation and semantics to the POSIX fsync(2) system call that synchronizes 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 unsynchronized data and metadata on the server to the server's disk for the specified file. Like fsync(2), it may be that there is some modified data or no modified data to synchronize. The data may have been synchronized by the server's normal periodic buffer synchronization activity. COMMIT will always return NFS3_OK, unless there has been an unexpected error.

COMMIT differs from fsync(2) 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 0 and count 0, it should do the equivalent of fsync()'ing the file. Otherwise, it should 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 0 and count of 0, 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 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 on WRITE on page 49.

ERRORS

NFS3ERR_IO
NFS3ERR_STALE
NFS3ERR_BADHANDLE
NFS3ERR_SERVERFAULT

SEE ALSO

WRITE.

4. 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 comments specific to that procedure.

A number of papers have been written describing issues encountered when constructing an NFS version 2 protocol implementation. The best overview paper is still [Sandberg]. [Israel], [Macklem], and [Pawlowski] describe other implementations. [X/OpenNFS] provides a complete description of the NFS version 2 protocol and supporting protocols, as well as a discussion on implementation issues and procedure and error semantics. Many of the issues encountered when constructing an NFS version 2 protocol implementation will be encountered when constructing an NFS version 3 protocol implementation.

4.1 Multiple version support

The RPC protocol provides explicit support for versioning of a service. Client and server implementations of NFS version 3 protocol should support both versions, for full backwards compatibility, when possible. Default behavior of the RPC binding protocol is the client and server bind using the highest version number they both support. Client or server implementations that cannot easily support both versions (for example, because of memory restrictions) will have to choose what version to support. The NFS version 2 protocol would be a safe choice since fully capable clients and servers should support both versions. However, this choice would need to be made keeping all requirements in mind.

4.2 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.

4.3 Path name interpretation

There are a few complications to the rule that path names 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 specification.

Another common problem for non-UNIX 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 as many working non-UNIX implementations exist.

4.4 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 uid, effective gid, 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 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. 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 specification.

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 (pagein). 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 pagein 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 uid 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 most operating systems, a particular user (on UNIX, the uid 0) has access to all files, no matter what permission and ownership they have. This superuser permission may not be allowed on the server, since anyone who can become superuser on their client could gain access to all remote files. A UNIX server by default maps uid 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 superuser access cannot be avoided. Export options are used, on the server, to restrict the set of clients allowed superuser access.

4.5 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.

A good description of the implementation and use of a duplicate request cache can be found in [Juszczak].

4.6 File name component handling

Server implementations of NFS version 3 protocol will frequently impose restrictions on the names which 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, the UFS file system will reject a name which contains "/", while "." and ".." are distinguished in UFS, and may 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.

4.7 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.

4.8 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, nonvolatile RAM).

Some examples of stable storage that are allowable for an NFS server include:

1. Media commit of data, that is, the modified data has been successfully written to the disk media, for example, the disk platter.

2. An immediate reply disk drive with battery-backed on-drive intermediate storage or uninterruptible power system (UPS).

3. Server commit of data with battery-backed intermediate storage and recovery software.

4. Cache commit with uninterruptible power system (UPS) and recovery software.

Conversely, the following are not examples of stable storage:

1. An immediate reply disk drive without battery-backed on-drive intermediate storage or uninterruptible power system (UPS).

2. Cache commit without both uninterruptible power system (UPS) and recovery software.

The only exception to this (introduced in this protocol revision) 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.

4.9 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.

4.10 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. Congestion avoidance schemes in NFS request retransmission are modelled on the work presented in [Jacobson]. [Nowicki] and [Macklem] describe congestion avoidance schemes to be applied to the NFS protocol over UDP.

4.11 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. See [Kazar] for a discussion of the issues of cache synchronization and mechanisms in several distributed file systems.

4.12 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 TRUE. If the request is marked as being asynchronous, the RPC should be generated with stable set to FALSE. If the response comes back with the committed field set to TRUE, the client should just mark the write request as done and no further action is required. If committed is set to FALSE, indicating that the buffer was not synchronized with the server's disk, the client will need to mark the buffer in some way which 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.

One proposal that was rejected was the addition of a boolean commit argument to the WRITE operation. It would be used to indicate whether the server should do a full file commit after doing the write. This seems as if it could be useful if the client knew that it was doing the last write on the file. It is difficult to see how this could be used, given existing client architectures though.

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 synchronization, the behaviour will be indeterminate.

In a high availability (redundant system) server implementation, two cases exist which relate to the verf changing. If the high availability server implementation does not use a shared-memory scheme, then the verf should change on failover, since the unsynchronized data is not available to the second processor and there is no guarantee that the system which 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.

4.13 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. Effectively, it is lying to the application program. 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.

5.0 Appendix I: Mount protocol

The changes from the NFS version 2 protocol to the NFS version 3 protocol have required some changes to be made in the MOUNT protocol. To meet the needs of the NFS version 3 protocol, a new version of the MOUNT protocol has been defined. This new protocol satisfies the requirements of the NFS version 3 protocol and addresses several other current market requirements.

5.1 RPC Information

5.1.1 Authentication

The MOUNT service uses AUTH_NONE in the NULL procedure. AUTH_UNIX, AUTH_SHORT, AUTH_DES, or AUTH_KERB are used for all other procedures. Other authentication types may be supported in the future.

5.1.2 Constants

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

```
PROGRAM  100005
VERSION  3
```

5.1.3 Transport address

The MOUNT service is normally supported over the TCP and UDP protocols. The rpcbind daemon should be queried for the correct transport address.

5.1.4 Sizes

```
const MNTPATHLEN = 1024;  /* Maximum bytes in a path name */
const MNTNAMLEN  = 255;   /* Maximum bytes in a name */
const FHSIZE3    = 64;    /* Maximum bytes in a V3 file handle */
```

5.1.5 Basic Data Types

```
typedef opaque fhandle3<FHSIZE3>;
typedef string dirpath<MNTPATHLEN>;
typedef string name<MNTNAMLEN>;
```

```
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 */
};
```

5.2 Server Procedures

The following sections define the RPC procedures supplied by a MOUNT version 3 protocol server. The RPC procedure number is given at the top of the page with the name and version. The SYNOPSIS provides the name of the procedure, the list of the names of the arguments, the list of the names of the results, followed by the XDR argument declarations and results declarations. The information in the SYNOPSIS is specified in RPC Data Description Language as defined in [RFC1014]. The DESCRIPTION section tells what the procedure is expected to do and how its arguments and results are used. The ERRORS section lists the errors returned for specific types of failures. The IMPLEMENTATION field describes how the procedure is expected to work and how it should be used by clients.

```
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;
```

5.2.0 Procedure 0: Null - Do nothing

SYNOPSIS

```
void MOUNTPROC3_NULL(void) = 0;
```

DESCRIPTION

Procedure NULL does not do any work. It is made available to allow server response testing and timing.

IMPLEMENTATION

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.

ERRORS

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.

5.2.1 Procedure 1: MNT - Add mount entry

SYNOPSIS

```
mountres3 MOUNTPROC3_MNT(dirpath) = 1;

struct mountres3_ok {
     fhandle3   fhandle;
     int        auth_flavors<>;
};

union mountres3 switch (mountstat3 fhs_status) {
case MNT_OK:
     mountres3_ok  mountinfo;
default:
     void;
};
```

DESCRIPTION

Procedure MNT maps a pathname on the server to a file handle. The pathname is an ASCII 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 flavors that are supported with the client's use of the file handle (or any file handles derived from it). The authentication flavors are defined in Section 7.2 and section 9 of [RFC1057].

IMPLEMENTATION

If mountres3.fhs_status is MNT3_OK, then mountres3.mountinfo contains the file handle for the directory and a list of acceptable authentication flavors. 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.

ERRORS

MNT3ERR_NOENT
MNT3ERR_IO
MNT3ERR_ACCES
MNT3ERR_NOTDIR
MNT3ERR_NAMETOOLONG

5.2.2 Procedure 2: DUMP - Return mount entries

SYNOPSIS

```
mountlist MOUNTPROC3_DUMP(void) = 2;


typedef struct mountbody *mountlist;

struct mountbody {
     name       ml_hostname;
     dirpath    ml_directory;
     mountlist  ml_next;
};
```

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

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.

ERRORS

There are no MOUNT protocol errors which can be returned from this procedure. However, RPC errors may be returned for authentication or other RPC failures.

5.2.3 Procedure 3: UMNT - Remove mount entry

SYNOPSIS

```
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

Typically, server implementations have maintained a list of clients which have file systems mounted. In the past, this list has been used to inform clients that the server was going to be shutdown.

ERRORS

There are no MOUNT protocol errors which can be returned from this procedure. However, RPC errors may be returned for authentication or other RPC failures.

5.2.4 Procedure 4: UMNTALL - Remove all mount entries

SYNOPSIS

```
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

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 which 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 usefullness.

ERRORS

There are no MOUNT protocol errors which can be returned from this procedure. However, RPC errors may be returned for authentication or other RPC failures.

5.2.5 Procedure 5: EXPORT - Return export list

SYNOPSIS

```
exports MOUNTPROC3_EXPORT(void) = 5;

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;
};
```

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

This procedure generally returns the contents of a list of shared or exported file systems. These are the file systems which are made available to NFS version 3 protocol clients.

ERRORS

There are no MOUNT protocol errors which can be returned from this procedure. However, RPC errors may be returned for authentication or other RPC failures.

6.0 Appendix II: Lock manager protocol

Because the NFS version 2 protocol as well as the NFS version 3 protocol is stateless, an additional Network Lock Manager (NLM) protocol is required to support locking of NFS-mounted files. The NLM version 3 protocol, which is used with the NFS version 2 protocol, is documented in [X/OpenNFS].

Some of the changes in the NFS version 3 protocol require a new version of the NLM protocol. This new protocol is the NLM version 4 protocol. The following table summarizes the correspondence between versions of the NFS protocol and NLM protocol.

NFS and NLM protocol compatibility

NFS Version	NLM Version
2	1,3
3	4

This appendix only discusses the differences between the NLM version 3 protocol and the NLM version 4 protocol. As in the NFS version 3 protocol, almost all the names in the NLM version 4 protocol have been changed to include a version number. This appendix does not discuss changes that consist solely of a name change.

6.1 RPC Information

6.1.1 Authentication

The NLM service uses AUTH_NONE in the NULL procedure. AUTH_UNIX, AUTH_SHORT, AUTH_DES, and AUTH_KERB are used for all other procedures. Other authentication types may be supported in the future.

6.1.2 Constants

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

```
PROGRAM    100021
VERSION    4
```

6.1.3 Transport Address

The NLM service is normally supported over the TCP and UDP protocols. The rpcbind daemon should be queried for the correct transport address.

6.1.4 Basic Data Types

uint64

```
typedef unsigned hyper uint64;
```

int64

```
typedef hyper int64;
```

uint32

```
typedef unsigned long uint32;
```

int32

```
typedef long int32;
```

These types are new for the NLM version 4 protocol. They are the same as in the NFS version 3 protocol.

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
};
```

Nlm4_stats 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;
};
```

This 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 process 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;
};
```

This 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 process 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 0 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;
};
```

This 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 or process 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.

6.2 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. The procedure names with _MSG and _RES suffixes denote asynchronous messages; for these the void response implies no reply. 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;
```

6.2.0 Procedure 0: NULL - Do nothing

SYNOPSIS

```
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

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.

ERRORS

It is possible that some server implementations may return RPC errors based on security and authentication requirements.

6.3 Implementation issues

6.3.1 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 error, NLM4_FBIG.

6.3.2 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 is assuming 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.

7.0 Appendix III: Bibliography

[Corbin] Corbin, John, "The Art of Distributed Programming-Programming Techniques for Remote Procedure Calls." Springer-Verlag, New York, New York. 1991. Basic description of RPC and XDR and how to program distributed applications using them.

[Glover] Glover, Fred, "TNFS Protocol Specification," Trusted System Interest Group, Work in Progress.

[Israel] Israel, Robert K., Sandra Jett, James Pownell, George M. Ericson, "Eliminating Data Copies in UNIX-based NFS Servers," Uniforum Conference Proceedings, San Francisco, CA, February 27 - March 2, 1989. Describes two methods for reducing data copies in NFS server code.

[Jacobson] Jacobson, V., "Congestion Control and Avoidance," Proc. ACM SIGCOMM '88, Stanford, CA, August 1988. The paper describing improvements to TCP to allow use over Wide Area Networks and through gateways connecting networks of varying capacity. This work was a starting point for the NFS Dynamic Retransmission work.

[Juszczak] Juszczak, Chet, "Improving the Performance and Correctness of an NFS Server," USENIX Conference Proceedings, USENIX Association, Berkeley, CA, June 1990, pages 53-63. Describes reply cache implementation that avoids work in the server by handling duplicate requests. More important, though listed as a side-effect, the reply cache aids in the avoidance of destructive non-idempotent operation re-application -- improving correctness.

[Kazar] Kazar, Michael Leon, "Synchronization and Caching Issues in the Andrew File System," USENIX Conference Proceedings, USENIX Association, Berkeley, CA, Dallas Winter 1988, pages 27-36. A description of the cache consistency scheme in AFS. Contrasted with other distributed file systems.

[Macklem] Macklem, Rick, "Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol," Winter USENIX Conference Proceedings, USENIX Association, Berkeley, CA, January 1991. Describes performance work in tuning the 4.3BSD Reno NFS implementation. Describes performance improvement (reduced CPU loading) through elimination of data copies.

[Mogul] Mogul, Jeffrey C., "A Recovery Protocol for Spritely NFS," USENIX File System Workshop Proceedings, Ann Arbor, MI, USENIX Association, Berkeley, CA, May 1992. Second paper on Spritely NFS proposes a lease-based scheme for recovering state of consistency protocol.

[Nowicki] Nowicki, Bill, "Transport Issues in the Network File System," ACM SIGCOMM newsletter Computer Communication Review, April 1989. A brief description of the basis for the dynamic retransmission work.

[Pawlowski] Pawlowski, Brian, Ron Hixon, Mark Stein, Joseph Tumminaro, "Network Computing in the UNIX and IBM Mainframe Environment," Uniforum ‘89 Conf. Proc., (1989) Description of an NFS server implementation for IBM’s MVS operating system.

[RFC1014] Sun Microsystems, Inc., "XDR: External Data Representation Standard", RFC 1014, Sun Microsystems, Inc., June 1987. Specification for canonical format for data exchange, used with RPC.

[RFC1057] Sun Microsystems, Inc., "RPC: Remote Procedure Call Protocol Specification", RFC 1057, Sun Microsystems, Inc., June 1988. Remote procedure protocol specification.

[RFC1094] Sun Microsystems, Inc., "Network Filesystem Specification", RFC 1094, Sun Microsystems, Inc., March 1989. NFS version 2 protocol specification.

[Sandberg] Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem," USENIX Conference Proceedings, USENIX Association, Berkeley, CA, Summer 1985. The basic paper describing the SunOS implementation of the NFS version 2 protocol, and discusses the goals, protocol specification and trade-offs.

[Srinivasan] Srinivasan, V., Jeffrey C. Mogul, "Spritely NFS: Implementation and Performance of Cache Consistency Protocols", WRL Research Report 89/5, Digital Equipment Corporation Western Research Laboratory, 100 Hamilton Ave., Palo Alto, CA, 94301, May 1989. This paper analyzes the effect of applying a Sprite-like consistency protocol applied to standard NFS. The issues of recovery in a stateful environment are covered in [Mogul].

[X/OpenNFS] X/Open Company, Ltd., X/Open CAE Specification: Protocols for X/Open Internetworking: XNFS, X/Open Company, Ltd., Apex Plaza, Forbury Road, Reading Berkshire, RG1 1AX, United Kingdom, 1991. This is an indispensable reference for NFS version 2 protocol and accompanying protocols, including the Lock Manager and the Portmapper.

[X/OpenPCNFS] X/Open Company, Ltd., X/Open CAE Specification: Protocols for X/Open Internetworking: (PC)NFS, Developer’s Specification, X/Open Company, Ltd., Apex Plaza, Forbury Road, Reading Berkshire, RG1 1AX, United Kingdom, 1991. This is an indispensable reference for NFS version 2 protocol and accompanying protocols, including the Lock Manager and the Portmapper.

8. Security Considerations

Since sensitive file data may be transmitted or received from a server by the NFS protocol, authentication, privacy, and data integrity issues should be addressed by implementations of this protocol.

As with the previous protocol revision (version 2), NFS version 3 defers to the authentication provisions of the supporting RPC protocol [RFC1057], and assumes that data privacy and integrity are provided by underlying transport layers as available in each implementation of the protocol. See section 4.4 for a discussion relating to file access permissions.

9. Acknowledgements

This description of the protocol is derived from an original document written by Brian Pawlowski and revised by Peter Staubach. This protocol is the result of a co-operative effort that comprises the contributions of Geoff Arnold, Brent Callaghan, John Corbin, Fred Glover, Chet Juszczak, Mike Eisler, John Gillono, Dave Hitz, Mike Kupfer, Rick Macklem, Ron Minnich, Brian Pawlowski, David Robinson, Rusty Sandberg, Craig Schamp, Spencer Shepler, Carl Smith, Mark Stein, Peter Staubach, Tom Talpey, Rob Thurlow, and Mark Wittle.

10. Authors' Addresses

Address comments related to this protocol to:

nfs3@eng.sun.com

Brent Callaghan
Sun Microsystems, Inc.
2550 Garcia Avenue
Mailstop UMTV05-44
Mountain View, CA 94043-1100

Phone: 1-415-336-1051
Fax: 1-415-336-6015
EMail: brent.callaghan@eng.sun.com

Brian Pawlowski
Network Appliance Corp.
319 North Bernardo Ave.
Mountain View, CA 94043

Phone: 1-415-428-5136
Fax: 1-415-428-5151
EMail: beepy@netapp.com

Peter Staubach
Sun Microsystems, Inc.
2550 Garcia Avenue
Mailstop UMTV05-44
Mountain View, CA 94043-1100

Phone: 1-415-336-5615
Fax: 1-415-336-6015
EMail: peter.staubach@eng.sun.com