Network File System
Protocol Specification

Network File System
Protocol Specification

1. Introduction

The Sun Network Filesystem (NFS) protocol provides transparent remote access to shared filesystems over local area networks. The NFS protocol is designed to be machine, operating system, network architecture, and transport protocol independent. This independence is achieved through the use of Remote Procedure Call (RPC) primitives built on top of an eXternal Data Representation (XDR).

The supporting mount protocol allows the server to hand out remote access privileges to a restricted set of clients. Thus, it allows clients to attach a remote directory tree at any point on some local filesystem.

1.1. Remote Procedure Call

Sun's remote procedure call specification, described in the *RPC Programming Guide*, provides a clean, procedure-oriented interface to remote services. Each server supplies a program that is a set of procedures. The combination of host address, program number, and procedure number specifies one remote service procedure.

RPC is a high-level protocol built on top of low-level transport protocols. It does not depend on services provided by specific protocols, so it can be used easily with any underlying transport protocol. Currently the only supported transport protocol is UDP/IP.

The RPC protocol includes a slot for authentication parameters on every call. The contents of the authentication parameters are determined by the ''flavor'' (type) of authentication used by the server and client. A server may support several different flavors of authentication at once: `AUTH_NONE` passes no authentication information (this is called null authentication); `AUTH_UNIX` passes the UNIX uid, gid, and groups with each call.

Servers have been known to change over time, and so can the protocol that they use. So RPC provides a version number with each RPC request. Thus, one server can service requests for several different versions of the protocol at the same time.

1.2. External Data Representation

Sun's external data representation specification, described in the *XDR Protocol Specification*, provides a common way of representing a set of data types over a network. This takes care of problems such as different byte ordering on different communicating machines. It also defines the size of each data type so that machines with different structure alignment algorithms can share a common format over the network.

In this document we use the XDR data definition language to specify the parameters and results of each RPC service procedure that a NFS server provides. The XDR data definition language reads a lot like C, although a few new constructs have been added. The notation

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

defines `name`, which is a fixed size block of `SIZE` bytes, and `data`, which is a variable size block of up to `DSIZE` bytes. This same notation is used to indicate fixed length arrays, and arrays with a variable number of elements up to some maximum.

The discriminated union definition

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

means the first thing over the network is an enumeration type called `status`; if its value is `NFS_OK`, the next thing on the network will be the structure containing `file1`, `file2`, and `count`. If the value of `status` is neither `NFS_OK` nor `NFS_ERROR`, then there is no more data to look at.

1.3. Stateless Servers

The NFS protocol is stateless. That is, a server does not need to maintain state about any of its clients in order to function correctly. Stateless servers have a distinct advantage over stateful servers in the event of a crash. With stateless servers, a client need only retry a request until the server responds; it does not even need to know that the server has crashed. The client of a stateful server, on the other hand, needs to detect a server crash and rebuild the server's state when it comes back up.

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

2. NFS Protocol Definition

The NFS protocol is designed to be operating system independent, but let's face it, it was designed in a UNIX environment. As such, it has some features which are very UNIX-ish. When in doubt about how something should work, a quick look at how it is done on UNIX will probably put you on the right track.

The protocol definition is given as a set of procedures with arguments and results defined using XDR. A brief description of the function of each procedure should provide enough information to allow implementation on most machines. There is a different section provided for each supported version of the

protocol. Most of the procedures, and their parameters and results, are self-explanatory. A few do not fit into the normal UNIX mold, however.

The LOOKUP procedure looks up one component of a pathname at a time. It is not obvious at first why it does not just take the whole pathname, traipse down the directories, and return a file handle when it is done. There are two good reasons not to do this. First, pathnames need separators between the directory components, and different operating systems use different separators. We could define a Network Standard Pathname Representation, but then every pathname would have to be parsed and converted at each end. Second, if pathnames were passed, the server would have to keep track of the mounted filesystems for all of its clients, so that it could break the pathname at the right point and pass the remainder on to the correct server.

Another procedure which might seem strange to UNIX people is the READDIR procedure. What READDIR does is provide a network standard format for representing directories. The same argument as above could have been used to justify a READDIR procedure that returns only one directory entry per call. The problem is efficiency. Directories can contain many entries, and a remote call to return each would just be too slow.

2.1. Version 2

The released version of the NFS protocol is actually the second. Even in the second version, there are various obsolete procedures and parameters, which will probably be removed in later versions.

2.1.1. Server/Client Relationship

The NFS protocol is designed to allow servers to be as simple and general as possible. Sometimes the simplicity of the server can be a problem, if the client wants to implement complicated filesystem semantics.

For example, UNIX allows removal of open files. A process can open a file and, while it is open, remove it from the directory. The file can be read and written as long as the process keeps it open, even though the file has no name in the filesystem. It is impossible for a stateless server to implement these semantics. The client can do some tricks like renaming the file on remove, and only removing it on close. We believe that the server provides enough functionality to implement most filesystem semantics on the client.

Every NFS client can also be a server, and remote and local mounted filesystems can be freely intermixed. This leads to some interesting problems when a client travels down the directory tree of a remote filesystem and reaches the mount point on the server for another remote filesystem. Allowing the server to following the second remote mount means it must do loop detection, server lookup, and user revalidation. Instead, we decided not to let clients cross a server's mount point. When a client does a LOOKUP on a directory that the server has mounted a filesystem on, the client sees the underlying directory instead of the mounted directory. A client can do remote mounts that match the server's mount points to maintain the server's view.

2.1.2. Permission Issues

The NFS protocol, strictly speaking, does not define the permission checking used by servers. However, it is expected that a server will do normal UNIX permission checking using AUTH_UNIX style authentication as the basis of its protection mechanism. The server gets the client's effective *uid*, effective *gid* and groups on each call, and uses them to check permission. There are various problems with this method that can been resolved in interesting ways.

Using *uid* and *gid* implies that the client and server share the same *uid* list. Every server and client pair must have the same mapping from user to *uid* and from group to *gid*. Since every client can also be a server this tends to imply that the whole network shares the same *uid/gid* space. This is acceptable for the short term, but a more workable network authentication method will be necessary before long.

Another problem arises due to the semantics of open. UNIX does its permission checking at open time and then that the file is open, and has been checked on later read and write requests. With stateless servers this breaks down, because the server has no idea that the file is open and it must do permission checking on each read and write call. On a local filesystem, a user can open a file then change the permissions so that no one is allowed to touch it, but will still be able to write to the file because it is open. On a remote filesystem, 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 no matter what the permissions are set to.

A similar problem has to do with paging in from a file over the network. The UNIX kernel checks for execute permission before opening a file for demand paging, then reads blocks from the open file. The file may not have read permission but after it is opened it doesn't matter. An NFS server can't tell the difference between a normal file read and a demand page-in read. To make this work the server allows reading of files if the *uid* given in the call has execute or read permission on the file.

In UNIX , the user ID zero has access to all files no matter what permission and ownership they have. This super-user permission is not allowed on the server since anyone who can become super-user on their workstation could gain access to all remote files. Instead, the server maps *uid* 0 to –2 before doing its access checking. This works as long as the NFS is not used to supply root filesystems, where super-user access cannot be avoided. Eventually servers will have to allow some kind of limited super-user access.

2.1.3. RPC Information

Authentication

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

Protocols

NFS currently is supported on UDP/IP only.

Constants

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

PROGRAM	100003
VERSION	2

Port Number

The NFS protocol currently uses the UDP port number 2049. This is a bug in the protocol and will be changed very shortly.

2.1.4. Sizes

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

MAXDATA 8192

The maximum number of bytes of data in a `READ` or `WRITE` request.

MAXPATHLEN 1024

The maximum number of bytes in a pathname argument.

MAXNAMLEN 255

The maximum number of bytes in a file name argument.

COOKIESIZE 4

The size in bytes of the opaque ‘‘cookie’’ passed by `READDIR`.

FHSIZE 32

The size in bytes of the opaque file handle.

2.1.5. Basic Data Types

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

2.1.5.1. stat

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

The `stat` type is returned with every procedure's results. A value of `NFS_OK` indicates that the call completed successfully and the results are valid. The other values indicate some kind of error occurred on the server side during the servicing of the procedure. The error values are derived from UNIX error numbers.

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

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

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

NFSERR_NXIO
: No such device or address.

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

NFSERR_EXIST
: File exists. The file specified already exists.

NFSERR_NODEV
: No such device.

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

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

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

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

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

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

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

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

NFSERR_STALE
The `fhandle` given in the arguments was invalid. That is, the file referred to by that file handle no longer exists, or access to it has been revoked.

NFSERR_WFLUSH
The server's write cache used in the `WRITECACHE` call got flushed to disk.

2.1.5.2. ftype

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

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

2.1.5.3. fhandle

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

The `fhandle` is the file handle that the server passes to the client. All file operations are done using file handles to refer to a file or directory. The file handle can contain whatever information the server needs to distinguish an individual file.

2.1.5.4. timeval

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

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

2.1.5.5. fattr

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

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

`Mode` is the access mode encoded as a set of bits. The bits are the same as the mode bits returned by the *stat* (2) system call in UNIX . Notice that the file type is specified both in the mode bits and in the file type. This is really a bug in the protocol and should be fixed in future versions. The descriptions given below specify the bit positions using octal numbers.

0040000 This is a directory. The `type` field should be `NFDIR`.

0020000 This is a character special file. The `type` field should be `NFCHR`.

0060000 This is a block special file. The `type` field should be `NFBLK`.

0100000 This is a regular file. The `type` field should be `NFREG`.

0120000 This is a symbolic link file. The `type` field should be `NFLNK`.

0140000 This is a named socket. The `type` field should be `NFNON`.

0004000 Set user id on execution.

0002000 Set group id on execution.

0001000 Save swapped text even after use.

0000400 Read permission for owner.

0000200 Write permission for owner.

0000100 Execute and search permission for owner.

0000040 Read permission for group.

0000020 Write permission for group.

0000010 Execute and search permission for group.

0000004 Read permission for others.

0000002 Write permission for others.

0000001 Execute and search permission for others.

2.1.5.6. sattr

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

The `sattr` structure contains the file attributes which can be set from the client. The fields are the same as for `fattr` above. A `size` of zero means the file should be truncated. A value of –1 indicates a field that should be ignored.

2.1.5.7. filename

```
typedef string  filename<MAXNAMLEN>;
```

The type `filename` is used for passing file names or pathname components.

2.1.5.8. path

```
typedef string  path<MAXPATHLEN>;
```

The type `path` is a pathname. The server considers it as a string with no internal structure, but to the client it is the name of a node in a filesystem tree.

2.1.5.9. attrstat

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

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

2.1.5.10. diropargs

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

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

2.1.5.11. dirop res

```
typedef union switch (stat status) {
    NFS_OK:
        struct {
                fhandle file;
                fattr   attributes;
        }
    default:
        struct {}
} diropres;
```

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

2.1.6. Server Procedures

The following sections define the RPC procedures supplied by a NFS server. The RPC procedure number and version are given in the header, along with the name of the prodedure. The synopsis of prodecures has this format:

```
<proc #>. <proc name> ( <arguments> ) returns ( <results> )
        <argument declarations>
        <results declarations>
```

In the first line, *proc name* is the name of the procedure, *arguments* is a list of the names of the arguments, and *results* is a list of the names of the results. The second and third lines give the XDR *argument declarations* and *results declarations*. Afterwards, there is a description of what the procedure is expected to do, and how its arguments and results are used. If there are bugs or problems with the procedure, they are listed at the end.

All of the procedures in the NFS protocol are assumed to be synchronous. When a procedure returns to the client, the client can assume that the operation has completed and any data associated with the request is now on stable storage. For example, a client WRITE request may cause the server to update data blocks, filesystem information blocks (such as indirect blocks in UNIX), and file attribute information (size and modify times). When the WRITE returns to the client, it can assume that the write is safe, even in case of a server crash, and it can discard the data written. This is a very important part of the statelessness of the server. If the server waited to flush data from remote requests the client would have to save those requests so that it could resend them in case of a server crash.

2.1.6.1. Do Nothing (Procedure 0, Version 2)

```
0. NFSPROC_NULL ( ) returns ( )
```

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

2.1.6.2. Get File Attributes (Procedure 1, Version 2)

```
1. NFSPROC_GETATTR (file) returns (reply)
        fhandle  file;
        attrstat reply;
```

If `reply.status` is `NFS_OK` then `reply.attributes` contains the attributes for the file given by `file`.

Bugs: the `rdev` field in the attributes structure is a UNIX device specifier. It should be removed or generalized.

2.1.6.3. Set File Attributes (Procedure 2, Version 2)

```
2. NFSPROC_SETATTR (file, attributes) returns (reply)
        fhandle  file;
        sattr    attributes;
        attrstat reply;
```

The `attributes` argument contains fields which are either –1 or are the new value for the attributes of `file`. If `reply.status` is `NFS_OK` then `reply.attributes` has the attributes of the file after the `setattr` operation has completed.

Bugs: the use of –1 to indicate an unused field in `attributes` is wrong.

2.1.6.4. Get Filesystem Root (Procedure 3, Version 2)

```
3. NFSPROC_ROOT ( ) returns ( )
```

Obsolete. This procedure is no longer used because finding the root file handle of a filesystem requires moving pathnames between client and server. To do this right we would have to define a network standard representation of pathnames. Instead, the function of looking up the root file handle is done by the `MNTPROC_MNT` procedure (see section entitled *Mount Protocol Definition* for details).

2.1.6.5. Look Up File Name (Procedure 4, Version 2)

```
4. NFSPROC_LOOKUP (which) returns (reply)
        diropargs which;
        diropres  reply;
```

If `reply.status` is `NFS_OK` then `reply.file` and `reply.attributes` are the file handle and attributes for the file `which.name` in the directory given by `which.dir`.

Bugs: there is some question as to what is the correct reply to a LOOKUP request when `which.name` is a mount point on the server for a remote mounted filesystem. Currently, we return the `fhandle` of the underlying directory. This is not completely acceptable, as the clients see a different view of the filesystem than the server does.

2.1.6.6. Read From Symbolic Link (Procedure 5, Version 2)

```
5. NFSPROC_READLINK (file) returns (reply)
        fhandle file;
        union switch (stat status) {
            NFS_OK:
                path   data;
            default:
                struct {}
        } reply;
```

If `status` has the value NFS_OK then `reply.data` is the data in the symbolic link given by `file`.

2.1.6.7. Read From File (Procedure 6, Version 2)

```
6. NFSPROC_READ (file, offset, count, totalcount) returns (reply)
        fhandle  file;
        unsigned offset;
        unsigned count;
        unsigned totalcount;
        union switch (stat status) {
            NFS_OK:
                fattr   attributes;
                string  data<MAXDATA>;
            default:
                struct {}
        } reply;
```

Returns up to `count` bytes of `data` from the file given by `file`, starting at `offset` bytes from the beginning of the file. The first byte of the file is at offset zero. The file attributes after the read takes place are returned in `attributes`.

Bugs: the argument `totalcount` is unused, and should be removed.

2.1.6.8. Write to Cache (Procedure 7, Version 2)

```
7. NFSPROC_WRITECACHE ( ) returns ( )
```

Obsolete.

2.1.6.9. Write to File (Procedure 8, Version 2)

```
8. NFSPROC_WRITE (file,beginoffset,offset,totalcount,data) returns (reply)
        fhandle  file;
        unsigned beginoffset;
        unsigned offset;
        unsigned totalcount;
        string   data<MAXDATA>;
        attrstat reply;
```

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

Bugs: the arguments `beginoffset` and `totalcount` are ignored and should be removed.

2.1.6.10. Create File (Procedure 9, Version 2)

```
9. NFSPROC_CREATE (where, attributes) returns (dir)
        diropargs where;
        sattr     attributes;
        diropres  dir;
```

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

Bugs: this routine should pass an exclusive create flag meaning, create the file only if it is not already there.

2.1.6.11. Remove File (Procedure 10, Version 2)

```
10. NFSPROC_REMOTE (which) returns (status)
        diropargs which;
        stat      status;
```

The file `which.name` is removed from the directory given by `which.dir`. A `status` of `NFS_OK` means the directory entry was removed.

2.1.6.12. Rename File (Procedure 11, Version 2)

```
11. NFSPROC_RENAME (from, to) returns (status)
        diropargs from;
        diropargs to;
        stat      status;
```

The existing file `from.name` in the directory given by `from.dir` is renamed to `to.name` in the directory given by `to.dir`. If `status` is `NFS_OK` the file was renamed. The `RENAME` operation is atomic on the server; it cannot be interrupted in the middle.

2.1.6.13. Create Link to File (Procedure 12, Version 2)

```
12. NFSPROC_LINK (from, to) returns (status)
        fhandle   from;
        diropargs to;
        stat      status;
```

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

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

2.1.6.14. Create Symbolic Link (Procedure 13, Version 2)

```
13. NFSPROC_SYMLINK (from, to, attributes) returns (status)
        diropargs from;
        path      to;
        sattr     attributes;
        stat      status;
```

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

A symbolic link is a pointer to another file. The name given in `to` is not interpreted by the server, just stored in the newly created file. A `READLINK` operation returns the data to the client for interpretation.

Bugs: on UNIX servers the attributes are never used, since symbolic links always have mode 0777.

2.1.6.15. Create Directory (Procedure 14, Version 2)

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

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

2.1.6.16. Remove Directory (Procedure 15, Version 2)

```
15. NFSPROC_RMDIR (which) returns (status)
        diropargs       which;
        stat            status;
```

The existing, empty directory `which.name` in the directory given by `which.dir` is removed. If `status` is `NFS_OK` the directory was removed.

2.1.6.17. Read From Directory (Procedure 16, Version 2)

```
16. NFSPROC_READDIR (dir, cookie, count) returns (entries)
        fhandle  dir;
        opaque   cookie[COOKIESIZE];
        unsigned count;
        union switch (stat status) {
            NFS_OK:
                typedef union switch (boolean valid) {
                    TRUE:
                        struct {
                            unsigned    fileid;
                            filename    name;
                            opaque      cookie[COOKIESIZE];
                            entry       nextentry;
                        }
                    FALSE:
                        struct {}
                } entry;
                boolean eof;
            default:
        } entries;
```

Returns a variable number of directory entries, with a total size of up to `count` bytes, from the directory given by `dir`. Each `entry` contains a `fileid` which is a unique number to identify the file within a filesystem, the `name` of the file, and a `cookie` which is an opaque pointer to the next entry in the directory. The cookie is used in the next READDIR call to get more entries starting at a given point in the directory. The special cookie zero (all bits zero) can be used to get the entries starting at the beginning of the directory. The `fileid` field should be the same number as the `fileid` in the the attributes of the file (see the section entitled *fattr* under *Basic Data Types*). The `eof` flag has a value of `TRUE` if there are no more entries in the directory; `valid` is used to mark the end of the entries. If the returned value of `status` is `NFS_OK` then it is followed by a variable number of `entries`.

2.1.6.18. Get Filesystem Attributes (Procedure 17, Version 2)

```
17. NFSPROC_STATFS (file) returns (reply)
        fhandle  file;
        union switch (stat status) {
            NFS_OK:
                struct {
                    unsigned    tsize;
                    unsigned    bsize;
                    unsigned    blocks;
                    unsigned    bfree;
                    unsigned    bavail;
                } fsattr;
            default:
                struct {}
        } reply;
```

If `reply.status` is `NFS_OK` then `reply.fsattr` gives the attributes for the filesystem that contains `file`. The attribute fields contain the following values:

tsize The optimum transfer size of the server in bytes. This is the number of bytes the server would like to have in the data part of *READ* and `WRITE` requests.

bsize The block size in bytes of the filesystem.

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

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

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

Bugs: this call does not work well if a filesystem has variable size blocks.

3. Mount Protocol Definition

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

The mount protocol is kept separate from the NFS protocol to make it easy to plug in new access checking and validation methods without changing the NFS server protocol.

Notice that the protocol definition implies stateful servers because the server maintains a list of client's mount requests. The mount list information is not critical for the correct functioning of either the client or the server. It is intented for advisory use only, for example, to warn possible clients when a server is going down.

3.1. Version 1

Version one of the mount protocol communicates with the version two of the NFS protocol. The only connecting point is the `fhandle` structure, which is the same for both protocols.

3.1.1. RPC Information

Authentication
: The mount service uses `AUTH_UNIX` style authentication only.

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

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

PROGRAM	100005
VERSION	1

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

3.1.2. Sizes

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

MNTPATHLEN 1024
: The maximum number of bytes in a pathname argument.

MNTNAMLEN 255
: The maximum number of bytes in a name argument.

FHSIZE 32
: The size in bytes of the opaque file handle.

3.1.3. Basic Data Types

3.1.3.1. fhandle

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

The `fhandle` is the file handle that the server passes to the client. All file operations are done using file handles to refer to a file or directory. The file handle can contain whatever information the server needs to distinguish an individual file.

This is the same as the `fhandle` XDR definition in version 2 of the NFS protocol; see the section on `fhandle` under *Basic Data Types*.

3.1.3.2. fhstatus

```
typedef union switch (unsigned status) {
    0:
        fhandle directory;
    default:
        struct {}
}
```

If a `status` of zero is returned, the call completed successfully, and a file handle for the `directory` follows. A non-zero status indicates some sort of error. In this case the status is a UNIX error number.

3.1.3.3. dirpath

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

The type `dirpath` is a normal UNIX pathname of a directory.

3.1.3.4. name

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

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

3.1.4. Server Procedures

The following sections define the RPC procedures supplied by a mount server. The RPC procedure number and version are given in the header, along with the name of the procedure. The synopsis of procedures has this format:

```
<proc #>. <proc name> ( <arguments> ) returns ( <results> )
        <argument declarations>
        <results declarations>
```

In the first line, *proc name* is the name of the procedure, *arguments* is a list of the names of the arguments, and *results* is a list of the names of the results. The second and third lines give the XDR *argument declarations* and *results declarations*. Afterwards, there is a description of what the procedure is expected to do, and how its arguments and results are used. If there are bugs or problems with the procedure, they are listed at the end.

3.1.4.1. Do Nothing (Procedure 0, Version 1)

```
0. MNTPROC_NULL ( ) returns ( )
```

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

3.1.4.2. Add Mount Entry (Procedure 1, Version 1)

```
1. MNTPROC_MNT (directory) returns (reply)
        dirpath  dirname;
        fhstatus reply;
```

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

3.1.4.3. Return Mount Entries (Procedure 2, Version 1)

```
2. MNTPROC_DUMP ( ) returns (mountlist)
        union switch (boolean more_entries) {
            TRUE:
                struct {
                        name      hostname;
                        dirpath   directory;
                        mountlist nextentry;
                }
            FALSE:
                struct {}
        } mountlist;
```

Returns the list of remote mounted filesystems. The `mountlist` contains one entry for each `hostname` and `directory` pair.

3.1.4.4. Remove Mount Entry (Procedure 3, Version 1)

```
3. MNTPROC_UMNT (directory) returns ( )
        dirpath directory;
```

Removes the mount list entry for `directory`.

3.1.4.5. Remove All Mount Entries (Procedure 4, Version 1)

```
4. MNTPROC_UMNTALL ( ) returns ( )
```

Removes all of the mount list entries for this client.

3.1.4.6. Return Export List (Procedure 5, Version 1)

```
5. MNTPROC_EXPORT ( ) returns (exportlist)
        union switch (boolean more_entries) {
            TRUE:
                struct {
                    dirpath     filesys;
                    typedef union switch (boolean more_groups) {
                        TRUE:
                            struct {
                                name    grname;
                                groups  nextgroup;
                            }
                        FALSE:
                            struct {}
                    } groups;
                    mountlist   nextentry;
                }
            FALSE:
                struct {}
        } exportlist;
```

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

Bugs: the exportlist should contain more information about the status of the filesystem, such as a read-only flag.