Network File System: Version 2 Protocol Specification

1. Status of this Standard

Note: This document specifies a protocol that Sun Microsystems, Inc., and others are using. It specifies it in standard ARPA RFC form.

2. 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). Implementations exist for a variety of machines, from personal computers to supercomputers.

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

2.1. Remote Procedure Call

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

2.2. External Data Representation

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

2.3. Stateless Servers

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

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

On the other hand, NFS deals with objects such as files and directories that inherently have state -- what good would a file be if it did not keep its contents intact? The goal is to not introduce any extra state in the protocol itself. Another way to simplify recovery is by making operations "idempotent" whenever possible (so that they can potentially be repeated).

3. NFS Protocol Definition

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. This RFC describes version two of the NFS protocol. Even in the second version, there are various obsolete procedures and parameters, which will be removed in later versions. An RFC for version three of the NFS protocol is currently under preparation.

3.1. File System Model

NFS assumes a file system that is hierarchical, with directories as all but the bottom-level files. Each entry in a directory (file, directory, device, etc.) has a string name. Different operating systems may have restrictions on the depth of the tree or the names used, as well as using different syntax to represent the "pathname", which is the concatenation of all the "components" (directory and file names) in the name. A "file system" is a tree on a single server (usually a single disk or physical partition) with a specified "root". Some operating systems provide a "mount" operation to make all file systems appear as a single tree, while others maintain a "forest" of file systems. Files are unstructured streams of uninterpreted bytes. Version 3 of NFS uses a slightly more general file system model.

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

Although files and directories are similar objects in many ways, different procedures are used to read directories and files. This provides a network standard format for representing directories. The same argument as above could have been used to justify a procedure that returns only one directory entry per call. The problem is efficiency. Directories can contain many entries, and a remote call to return each would be just too slow.

3.2. RPC Information

Authentication

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

Transport Protocols

NFS currently is supported on UDP/IP only.

Port Number

The NFS protocol currently uses the UDP port number 2049. This is not an officially assigned port, so later versions of the protocol use the “Portmapping” facility of RPC.

3.3. Sizes of XDR Structures

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

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

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

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

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

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

3.4. Basic Data Types

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

3.4.1. stat

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

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

3.4.2. ftype

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

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

3.4.3. fhandle

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

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

3.4.4. timeval

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

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

3.4.5. fattr

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

The *fattr* structure contains the attributes of a file; "type" is the type of the file; "nlink" is the number of hard links to the file (the number of different names for the same file); "uid" is the user identification number of the owner of the file; "gid" is the group identification number of the group of the file; "size" is the size in bytes of the file; "blocksize" is the size in bytes of a block of the file; "rdev" is the device number of the file if it is type *NFCHR* or *NFBLK*; "blocks" is the number of blocks the file takes up on disk; "fsid" is the file system identifier for the filesystem containing 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. 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 will be fixed in future versions. The descriptions given below specify the bit positions using octal numbers.

Bit	*Description*
0040000	This is a directory; "type" field should be NFDIR.
0020000	This is a character special file; "type" field should be NFCHR.
0060000	This is a block special file; "type" field should be NFBLK.
0100000	This is a regular file; "type" field should be NFREG.
0120000	This is a symbolic link file; "type" field should be NFLNK.
0140000	This is a named socket; "type" field should be NFNON.
0004000	Set user id on execution.
0002000	Set group id on execution.
0001000	Save swapped text even after use.
0000400	Read permission for owner.
0000200	Write permission for owner.
0000100	Execute and search permission for owner.
0000040	Read permission for group.
0000020	Write permission for group.
0000010	Execute and search permission for group.
0000004	Read permission for others.
0000002	Write permission for others.
0000001	Execute and search permission for others.

Notes:

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

The "rdev" field in the attributes structure is an operating system specific device specifier. It will be removed and generalized in the next revision of the protocol.

3.4.6. sattr

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

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

3.4.7. filename

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

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

3.4.8. 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.

3.4.9. attrstat

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

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

3.4.10. diropargs

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

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

3.4.11. diropres

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

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

3.5. Server Procedures

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

All of the procedures in the NFS protocol are 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), 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.

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

3.5.1. Do Nothing

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

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

3.5.2. Get File Attributes

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

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

3.5.3. Set File Attributes

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

attrstat
NFSPROC_SETATTR (sattrargs) = 2;
```

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

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

3.5.4. Get Filesystem Root

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

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 the *Mount Protocol Definition* later in this chapter for details).

3.5.5. Look Up File Name

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

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

3.5.6. Read From Symbolic Link

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

readlinkres
NFSPROC_READLINK(fhandle) = 5;
```

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

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

3.5.7. Read From File

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

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

readres
NFSPROC_READ(readargs) = 6;
```

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

Note: The argument "totalcount" is unused, and is removed in the next protocol revision.

3.5.8. Write to Cache

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

To be used in the next protocol revision.

3.5.9. Write to File

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

attrstat
NFSPROC_WRITE(writeargs) = 8;
```

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

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

3.5.10. Create File

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

diropres
NFSPROC_CREATE(createargs) = 9;
```

The file "name" is created in the directory given by "dir". The initial attributes of the new file are given by "attributes". A reply "status" of NFS_OK indicates that the file was created, and 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.

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

3.5.11. Remove File

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

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

Note: possibly non-idempotent operation.

3.5.12. Rename File

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

stat
NFSPROC_RENAME(renameargs) = 11;
```

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

Note: possibly non-idempotent operation.

3.5.13. Create Link to File

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

stat
NFSPROC_LINK(linkargs) = 12;
```

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

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

Note: possibly non-idempotent operation.

3.5.14. Create Symbolic Link

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

stat
NFSPROC_SYMLINK(symlinkargs) = 13;
```

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

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

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

3.5.15. Create Directory

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

The new directory "where.name" is created in the directory given by "where.dir". The initial attributes of the new directory are given by "attributes". A reply "status" of NFS_OK indicates that the new directory was created, and 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.

Note: possibly non-idempotent operation.

3.5.16. Remove Directory

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

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

Note: possibly non-idempotent operation.

3.5.17. Read From Directory

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

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

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

readdirres
NFSPROC_READDIR (readdirargs) = 16;
```

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

3.5.18. Get Filesystem Attributes

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

statfsres
NFSPROC_STATFS(fhandle) = 17;
```

If the reply "status" is *NFS_OK*, then the reply "info" gives the attributes for the filesystem that contains file referred to by the input fhandle. The attribute fields contain the following values:

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

bsize:
: The block size in bytes of the 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.

Note: This call does not work well if a filesystem has variable size blocks.

4. NFS Implementation Issues

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

4.1. 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, 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 filesystem. It is impossible for a stateless server to implement these semantics. The client can do some tricks such as renaming the file on remove, and only removing it on close. We believe that the server provides enough functionality to implement most file system semantics on the client.

Every NFS client can also potentially be a server, and remote and local mounted 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 follow the second remote mount would require 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 on which the server has mounted a filesystem, 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.

4.2. Pathname Interpretation

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

4.3. Permission Issues

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

Using "uid" and "gid" implies that the client and server share the same "uid" list. Every server and client pair must have the same mapping from user to "uid" and from group to "gid". Since every client can also be a server, this tends to imply that the whole network shares the same "uid/gid" space. *AUTH_DES* (and the next revision of the NFS protocol) uses string names instead of numbers, but there are still complex problems to be solved.

Another problem arises due to the usually stateful open operation. Most operating systems check permission at open time, and then check that the file is open on each read and write request. With stateless servers, the server has no idea that the file is open and must do permission checking on each read and write call. On a local filesystem, 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 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 regardless of the permission setting.

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

In most operating systems, a particular user (on the user ID zero) has access to all files no matter what permission and ownership they have. This "super-user" permission may not be allowed on the server, since anyone who can become super-user on their workstation could gain access to all remote files. The UNIX server by default maps user id 0 to -2 before doing its access checking. This works except for NFS root filesystems, where super-user access cannot be avoided.

4.4. Setting RPC Parameters

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

5. Mount Protocol Definition

5.1. Introduction

The mount protocol is separate from, but related to, the NFS protocol. It provides operating system specific services to get the NFS off the ground -- looking up server 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 intended for advisory use only, for example, to warn possible clients when a server is going down.

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

5.2. RPC Information

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

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

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

5.3. Sizes of XDR Structures

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

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

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

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

5.4. Basic Data Types

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

5.4.1. fhandle

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

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

This is the same as the "fhandle" XDR definition in version 2 of the NFS protocol; see *Basic Data Types* in the definition of the NFS protocol, above.

5.4.2. fhstatus

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

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

5.4.3. dirpath

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

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

5.4.4. name

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

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

5.5. Server Procedures

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

```
/*
* Protocol description for the mount program
*/

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

5.5.1. Do Nothing

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

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

5.5.2. Add Mount Entry

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

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

5.5.3. Return Mount Entries

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

mountlist
MNTPROC_DUMP(void) = 2;
```

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

5.5.4. Remove Mount Entry

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

Removes the mount list entry for the input "dirpath".

5.5.5. Remove All Mount Entries

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

Removes all of the mount list entries for this client.

5.5.6. Return Export List

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

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

exportlist
MNTPROC_EXPORT(void) = 5;
```

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

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