Network Extensible File System Protocol Specification

Comments to:

sun!nfs3
nfs3@SUN.COM

Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA 94043

1.0 Introduction

The Network Extensible File System protocol(NeFS) provides transparent remote access to shared file systems over networks. The NeFS protocol is designed to be machine, operating system, network architecture, and transport protocol independent. This document is the draft specification for the protocol. It will remain in draft form during a period of public review. Italicized comments in the document are intended to present the rationale behind elements of the design and to raise questions where there are doubts. Comments and suggestions on this draft specification are most welcome.

1.1 The Network File System

The Network File System (NFS™[*]) has become a de facto standard distributed file system. Since it was first made generally available in 1985 it has been licensed by more than 120 companies. If the NFS protocol has been so successful why does there need to be NeFS ? Because the NFS protocol has deficiencies and limitations that become more apparent and troublesome as it grows older.

1. Size limitations. The NFS version 2 protocol limits filehandles to 32 bytes, file sizes to the magnitude of a signed 32 bit integer, timestamp accuracy to 1 second. These and other limits need to be extended to cope with current and future demands.
2. Non-idempotent procedures. A significant number of the NFS procedures are not idempotent. In certain circumstances these procedures can fail unexpectedly if retried by the client. It is not always clear how the client should recover from such a failure.
3. Unix®[†] bias. The NFS protocol was designed and first implemented in a Unix environment. This bias is reflected in the protocol: there is no support for record-oriented files, file versions or non-Unix file attributes. This bias must be removed if NFS is to be truly machine and operating system independent.
4. No access procedure. Numerous security problems and program anomalies are attributable to the fact that clients have no facility to ask a server whether they have permission to carry out certain operations.
5. No facility to support atomic filesystem operations. For instance the POSIX O_EXCL flag makes a requirement for exclusive file creation. This cannot be guaranteed to work via the NFS protocol without the support of an auxiliary locking service. Similarly there is no way for a client to guarantee that data written to a file is appended to the current end of the file.

*. NFS is a registered trademark of Sun Microsystems, Inc.
†. Unix is a registered trademark of AT&T.

6. Performance. The NFS version 2 protocol provides a fixed set of operations between client and server. While a degree of client caching can significantly reduce the amount of client-server interaction, a level of interaction is required just to maintain cache consistency and there yet remain many examples of high client-server interaction that cannot be reduced by caching. The problem becomes more acute when a client's set of filesystem operations does not map cleanly into the set of NFS procedures.

1.2 The Network Extensible File System

NeFS addresses the problems just described. Although a draft specification for a revised version of the NFS protocol has addressed many of the deficiencies of NFS version 2, it has not made non-Unix implementations easier, not does it provide opportunities for performance improvements. Indeed, the extra complexity introduced by modifications to the NFS protocol makes all implementations more difficult. A revised NFS protocol does not appear to be an attractive alternative to the existing protocol.

Although it has features in common with NFS, NeFS is a radical departure from NFS. The NFS protocol is built according to a *Remote Procedure Call* model (RPC) where filesystem operations are mapped across the network as remote procedure calls. The NeFS protocol abandons this model in favor of an *interpretive* model in which the filesystem operations become operators in an interpreted language. Clients send their requests to the server as programs to be interpreted. Execution of the request by the server's interpreter results in the filesystem operations being invoked and results returned to the client. Using the interpretive model, filesystem operations can be defined more simply. Clients can build arbitrarily complex requests from these simple operations.

2.0 Philosophy

Firstly some terminology: a *client* is a computer or network node that requires access to filesystems held on a server. A *server* is another computer that provides access to one or more filesystems; it may also be a client itself. A client sends a *request* to the server and will receive zero or more *responses* from the server. Typically a request will generate one response.

2.1 Filesystem Model

The NeFS protocol does not conform to a specific filesystem model. It is designed to permit access to a wide variety of filesystems from a diverse set of operating systems with little or no loss of semantics. The protocol attempts to resolve a conflict between what appear to be conflicting requirements: while allowing different clients and servers to interoperate, the protocol must maintain the access semantics and performance expected by clients and servers that share the same architecture. By defining a small, yet complete set of filesystem objects and operations, it is hoped that each implementation of the protocol will perceive the protocol as a natural network extension of the filesystems it supports.

A *filesystem* is assumed to be an independent collection of data organized into named files. In practical terms a filesystem can be regarded as the files on a magnetic disk or in a disk partition. The protocol assumes that the files are accessed via *directories* and the directories may be organized into a hierarchical tree structure. A *flat* filesystem that has no hierarchical structure is considered to be a degenerate instance of a tree structure with a single directory. The NeFS protocol requires every filesystem object to be represented by a *filehandle* (see Filehandle on page 8).

Although this draft of the protocol specification does not define it, it is important that a *common set* of filesystem objects and operations be specified as the *lingua franca* of filesystem operations. Server implementations should make a best effort at making their own filesystems appear to be just another instance of the common set. Similarly, clients should attempt to map their file access onto the common set.

2.2 Statelessness

In common with the NFS protocol, NeFS 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.

This is not to say that servers are not allowed to maintain state about client operations, but that the state held by servers is not required for correct operation. In many cases servers will maintain state 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 hope that the client is doing a sequential read and the next client **read** request will be satisfied from the cache instead of from the disk. The read–ahead block is not necessary for correct server behavior, but it increases the server's performance.

If the client chooses to maintain state on the server it has the responsibility to restore that state in the event of a server crash. For instance, a client may choose to perform a series of writes to the server without committing the writes to disk. The client must retain the uncommitted data in case the server crashes before the client is able to confirm that the data is safe on the server's disk. The client must be able to replay the writes when the server has recovered.

Clients can do as much or as little caching as they want. The protocol does not have any explicit support for cache consistency.

2.3 Idempotency

An idempotent request is one that can be successfully repeated by the client. An example is a read request; since it specifies both an offset and length the request will always successfully return the same data no matter how many times it is repeated. This attribute is important because a client that does not receive a response within a timeout period will retry the request until a response is received. This behavior could result in the same request being executed several times on a server that is having difficulty getting a response back to the client within its timeout period.

The NFS version 2 protocol did not attain the goal of completely idempotent operations. For instance, a **create** request would fail when executed a second time because the file already existed from the first attempt. Implementations of the version 2 protocol not only required code in the client implementation that would recover from such retry errors, but also a request cache on the server that could be used to detect duplicate requests of non-idempotent operations and avoid re-doing them.

The NeFS protocol does not guarantee idempotent operations per se, but it does allow requests to be assembled that can detect retries of themselves and take account of changes made to the filesystem by previous incomplete or unacknowledged executions. Though network transports with *at-most-once* semantics and/or duplicate request caches can still be usefully employed, they are not required by the protocol to implement idempotency on all requests.

3.0 Interpreter

The NeFS interpreter resides in the server. It interprets a tokenized form of the POSTSCRIPT®* language originally developed by Adobe Systems, Incorporated. Although POSTSCRIPT is better known as a page description language, it is also a powerful, general-purpose programming language. The version of the language as implemented by the NeFS interpreter replaces the imaging model of

*. POSTSCRIPT is a trademark of Adobe Systems, Inc.

the page description language with a filesystem model. The result is a language that allows complex and varied filesystem operations to be expressed in a simple, compact form.

Given the flexibility and generality of a programming language the NeFS protocol is no longer rigidly tied to a single filesystem model and a strictly limited set of client interactions. The new language-based protocol allows clients with diverse operating systems and requirements to tailor their requests to the server to suit their unique needs. A goal of the protocol is that it will appear to be a natural extension of a client's native operating system operations to the network.

3.1 The Language

The PostScript language employs a postfix notation in which operators are preceded by their operands. While this notation is counter-intuitive for human readers, it simplifies interpreter design; operands are simply pushed onto an operand stack and operators take their operands from this stack. For instance, the PostScript notation to add two numbers is:

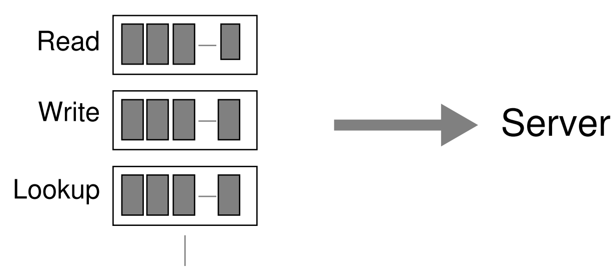
```
87 22 add
```

As the interpreter encounters the `87` and `22` it recognizes them as operands and pushes them onto its operand stack. The `add` operator just adds removes the two values from the stack, adds them, and pushes their sum onto the stack. The value left on the stack can be used as an operand by another operator. The language data model provides integers, booleans, strings and arrays (Fundamental Objects on page 7). The execution model provides conditional evaluation and looping operators (Operators on page 12).

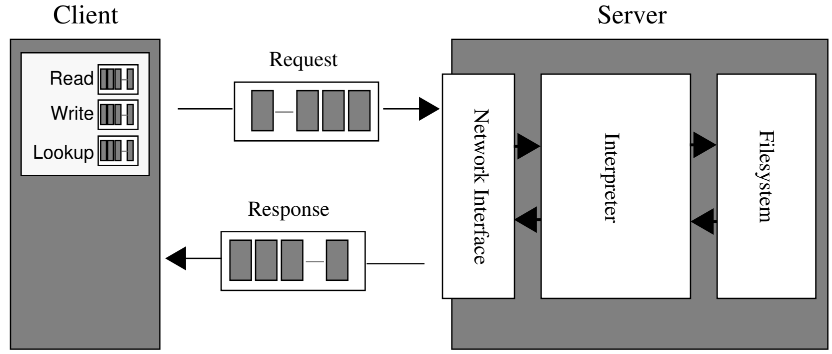
Since the NeFS interpreter receives a program as a tokenized sequence of objects it is beyond the scope of this document to specify a human-readable form for the language but a useful medium for communication is the ASCII representation of the PostScript language as described in the "Red Book" (*PostScript Language Reference Manual*, Adobe Systems Incorporated).

3.2 Request and Response

A client's request to the server is a small program to be interpreted. The program is a tokenized form of PostScript represented as a sequence of *objects*. These objects can be roughly classified into operators and operands. It is assumed that each client will maintain a set of tokenized requests ready for transmission to the server. The client will choose the appropriate request for the operation it wishes to perform, substitute the required data objects, and transmit the request to the server.



The client receives a response to its request as a sequence of data objects. Within the client's request are operators that assemble data objects into a response and transmit them back to the client. The client not only has complete control over the composition of the response but it can determine whether a response is transmitted at all. A single request could also result in multiple responses.

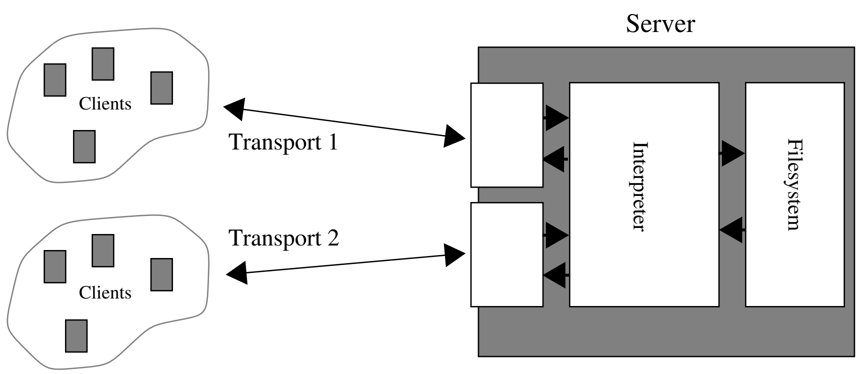


3.3 Interpreter Network Interface

The NeFS protocol describes the operand and data objects interpreted by a NeFS server. It does not define a standard network representation of the protocol. Standard network representations for the protocol must be defined separately and are implemented by a network interface at the client and at the server.

The interpreter network interface module has a dual function. Firstly, it is a transport layer that hides the underlying means of conveying objects between the client and server. Secondly, it is a NeFS client's agent on the server for the duration of its request.

As a transport layer it moves client requests and responses between the network and the NeFS interpreter. Both the request and the response packets comprise a sequence of objects. The network interface module translates objects between their network representation and their representation as known to the interpreter. The protocol is not tied to a single transport, nor even a single network representation for objects. A single server could make several network interfaces available to diverse sets of clients.



This document describes only the NeFS language - not a network standard. The NFS protocol is inextricably bound to Sun RPC. NeFS does not specify how objects comprising requests and responses are conveyed between client and server. Sun RPC will be just one of potentially many network standards. These network standards must be described if implementations of the NeFS protocol are to interoperate. These standards should be described separately.

3.3.1 Interpreter Context

On successful receipt of a client request, the network interface module invokes an instance of the NeFS interpreter. The network interface presents an executable object to the interpreter as a single argument. This object may be viewed as the interpreter's handle to the network interface - the *interface handle*. As the interpreter executes the client's request the interface handle is used by the interpreter to make callbacks into the network interface module to request objects from the network. The network interface also implements the operations to enqueue objects in a response packet (**encodereply** on page 18), transmit a response (**sendreply** on page 34), or flush enqueued objects (**flushreply** on page 20).

When the interpreter execution of the client's request is complete the network interface module deletes the request context. It then proceeds to service a new request.

3.3.2 Object Representation

Objects are the data items manipulated by the interpreter. All objects have a *type* attribute. The composition of an object varies depending on its type. A simple object like an integer contains just a value. A composite object like an array contains a length attribute and a reference to an external vector of elements. The interpreter sees objects as structures containing binary data and memory references to data. This representation is not suitable for transport across a network. The network interface module translates objects back and forth between their network and internal representations.

To convert an object to its network representation, memory addresses must be dereferenced to real data values and the data values must be represented in a portable format that takes account of byte ordering and data alignment differences between different computer architectures.

3.3.3 Duplicate Requests

The network interface may receive duplicate client requests, usually as the result of a client that retries a request for which no response has been received. It is strongly recommended that the network interface be able to detect duplicate requests and avoid unnecessary work by withholding them from the interpreter. Assuming that every request contains a unique identifier in its header, a cache of recent requests indexed by a unique request identifier can be used to detect duplicates.

3.3.4 Authentication

The network interface must convey the client's authentication information where present into the interpreter environment and verify that it is correct. It may also be required to restrict access to the interpreter based on the client's identity. Authentication information in the environment is conveyed by the interpreter to the NeFS operators to be used in checking access permissions for filesystem objects.

3.3.5 Interface Errors

The network interface may detect errors or conditions that preclude invocation of the interpreter. These may be transport or protocol errors that require a response via the appropriate protocol layer e.g. a packet that fails a checksum verification. Service may also be denied based on the requestor's network address or credentials.

3.4 Resource Limits

The server must establish resource limits to prevent malicious or ill-behaved requests from monopolizing all of the server's resources. These limits are server implementation dependent. Clients must interrogate the servers in which they interact and adjust their behavior accordingly. The NeFS model will define a minimum to which all servers will adhere. Server implementations are free to provide resources in excess of the established minimum. It is intended that client requests be written to scale easily to the varying resource limits imposed by servers.

Need to establish minimum resource limits. Already have memstatus and timestatus operators to query memory and time usage and limits, but need operators to query stack limits.

4.0 Data Objects

4.1 Fundamental Objects

The following object types are fundamental to all implementations. Additional object types may need to be supported.

4.1.1 Integer

A simple object that contains a 32 bit signed integer value in the range [-2147483648, 214748367].

4.1.2 Hyper

A simple object that contains a 64 bit signed integer value.

4.1.3 Boolean

A simple object that contains a boolean value **true** or **false**.

4.1.4 Array

A composite object that represents an array of objects. The first element in an array has an index of 0. Arrays have a length attribute that is fixed when the array is created. The objects in an array may have different types. Multidimensional arrays may be created by including arrays as elements of arrays.

4.1.5 String

Similar to an array with the restriction that it can hold only small integers in the range [0, 255] as elements. A string may be used to hold ASCII strings or binary data.

4.1.6 Name

An atomic identifier within the interpreter. Externally a name can be represented as a string object.

4.1.7 Dictionary

A dictionary is a table that contains pairs of key-value objects. A key can be any kind of object though it is usually a name object. The corresponding value can be any kind of object. A dictionary has two attributes: **maxlen** which is that maximum number of key-value pairs that it can hold, and **length** which is the number of key-value pairs currently in the dictionary.

4.1.8 Timeval

A composite object that represents a time and date. This object comprises two unsigned 32 bit integer values: the first is the number of seconds since midnight January 1, 1970 GMT, the second is a fraction of a second counted in nanoseconds.

4.1.9 Null

A simple object that is used to fill uninitialized positions in arrays or dictionaries. The null object has no value.

4.1.10 Mark

A simple object that is used to mark a position in the operand stack. The mark object has no value.

4.1.11 Error

A composite object that is created by the interpreter to report error information (see section 6.2 on page 46).

4.1.12 Transport

A composite object that represents the underlying transport medium between the client and server. The composition of this object needs to be known only to the Interpreter Network Interface (page 5).

4.2 Filesystem Objects

The following objects are filesystem objects defined in terms of fundamental objects.

4.2.1 Filehandle

A filehandle is a string that represents a filesystem object on the server. The filehandle is created by the server must be used by the client to uniquely identify filesystem objects. A client can store filehandles for use in later requests, 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 a filehandle. If two filehandles 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 can revoke the access provided by a filehandle at any time. If a filehandle is used for an object that no longer exists on the server or access for that filehandle has been revoked the **nefs_stale** error will result.

Should the client should be able to find out the length of the server's filehandles a priori ? It might make life easier on the client when it comes to memory allocation.

4.2.2 Filename

A filename is a string that represents the name of a file in a directory. The NeFS protocol does not attempt to define the characteristics of a filename, but it is generally understood to be a human-readable string of characters. Although the server may choose to map an invalid filename to a valid representation in its filesystem, the client is responsible for presenting filenames to the server that are valid for the filesystem. The client can query the server's filesystem attributes (section 4.2.4 on page 10) to determine filename characteristics on the server. The client should use the filesystem **nameset** attribute when interpreting the filename data.

The NFS protocol required all implementations to support ASCII names up to 255 characters in length. The onus was on the server to either attempt to map the name into a valid representation on the server or to return an error. The intention with NeFS is to shift the responsibility for creating valid names to the client. For instance, if the client uses ASCII names and the server uses EBCDIC names, the client is responsible for translation. The advantage of this approach is that the protocol appears to be more natural to clients and servers that share a common OS and architecture.

Is it necessary to have an filesystem attribute that describes the syntax of a name ? An example is IBM and MSDOS that allow the use of a dot separated extension and VMS which allows an appended version number extension.

4.2.3 File Attributes

The attributes of a filesystem object are represented by a dictionary. Each attribute is represented by a key value in the dictionary (a name) and a corresponding value. The value may be null if the value is boolean and just the appearance of the name implies true.

Servers must make an attempt to map file attributes into the following set. The server should not attempt to fudge attributes that have no satisfactory representation. For instance, servers that do not support record oriented files should not provide a maxrecsize attribute.

ftype A name or integer. The file type. This attribute must be supported by all implementations. The file type determines the set of file attributes. The **ftype** is represented by a name. Commonly used values may also have an integer representation also. The following are some initial values for **ftype** with integer representations included:

dir	1	A directory
file.byte	2	A byte-oriented data file
file.rec.fix	3	A record-oriented data file with fixed length records
file.rec.var	4	A record oriented data file with variable length records
file.key	5	A keyed access file
unix.sym	6	A Unix symbolic link

The filetype is represented by an open-ended namespace. Servers may choose to represent "common" attributes by a well-known integer but they should continue to accept name representations of filetype from clients.

There is no structuring implied by the format of the names; should there be one ? Is there a better one ?

A mechanism for extending the filetype namespace in an organized way needs to be developed in order to avoid name collisions.

fsize Integer or Hyper. The size of a file. For a byte-oriented file this is the size of the file in bytes. For a record-oriented file this is a record count.

fbytes Integer or Hyper. The size of a file in bytes. for a byte-oriented file this is the same as fsize. For a record-oriented file this is the number of bytes used by all the records in a file. For a variable length record file it includes per-record overhead for record delimiters.

permbits Integer. A bit field that contains bit encodings for the types of access permitted on the filesystem object.

0x40000	Hidden
0x20000	Read permission for system
0x10000	Write permission for system
0x08000	Execute permission for system on a file
0x08000	Lookup permission for system on a directory
0x04000	Remove permission for owner
0x02000	Remove permission for group
0x01000	Remove permission for others
0x00800	Set user id on execution
0x00400	Set group id on execution
0x00100	Read permission for owner
0x00080	Write permission for owner
0x00040	Execute permission for owner on file
0x00040	Lookup permission for owner on a directory
0x00020	Read permission for group
0x00010	Write permission for group
0x00008	Execute permission for group on file
0x00008	Lookup permission for group on directory
0x00004	Read permission for others
0x00002	Write permission for others
0x00001	Execute permission for others on file
0x00001	Lookup permission for others on a directory

Is this set of permission bit sufficient. Does there need to be a mechanism for extending this set ?

owner String or integer. The owner of a file.

group String or integer. The group of a file.

fileno Integer. A number that uniquely identifies the file in the filesystem.

atime Timeval. The time when the file data was last accessed.

mtime Timeval. The time when the file data was last modified (written).

ctime Timeval. The time when the file attributes were changed.

btime Timeval. The time when the file was created (born).

nlinks Integer. The number of names that reference the object. Must be non-zero.

recminsize Integer. The minimum number of bytes per record in a record oriented file.

recmaxsize Integer. The maximum number of bytes per record in a record oriented file.

symlink String. The pathname that represents a Unix symbolic link

4.2.4 Filesystem Attributes

Like file attributes, the attributes of a filesystem are represented by a dictionary. Each attribute is represented by a key value in the dictionary and a corresponding value. The value may be null if the value is boolean and just the appearance of the name implies true.

fstype String. The filesystem type. This attribute determines the set of names that define the attribute set. The subtype attribute may be used to extend this attribute set.

fssubtype String. The filesystem subtype. This optional attribute may be used to extend the set of attributes defined by **fstype**.

services Integer. A bitfield that encodes the set of services that the filesystem makes available. A bit set to a binary 1 means that the service is available.
0x0001 Read only access allowed.
0x0002 Supports version numbers
0x0004 Hard links supported.

ftypes Array. An array of **ftype** values (see ftype page 8). These are the file types that the filesystem supports. For any filetype included in this array the filesystem guarantees to support all operations that relate to that filetype.

namemaxlen Integer. The maximum number of characters in a name

namebadch String. The set of characters that are not permitted in file names

nameset Integer. The character set used for names. An encoding based on ISO codes seems to be appropriate here. This is yet to be defined.

maxreadsize Integer. The maximum number of bytes that can be read from a file.byte file in a single read operation.

where **fstype** = “unix”:

bytes Integer. The total number of bytes in the filesystem.

bfree Integer. The number of free bytes in the filesystem.

bavail Integer. The number of bytes available to non-privileged users.

files Integer. The total number of file slots in the filesystem.

ffree Integer. The number of free file slots on the filesystem.

favail Integer. The number of file slots available to non-privileged users.

where fstype = (whatever): need to define attribute sets for other filesystem types.

This list of object types is intended to represent the "core set" that all implementations will attempt to support. The list is not complete nor precisely defined - more types will likely be added in successive drafts.

There needs to be an extension mechanism defined to cope with new types of fundamental objects and new types of filesystem objects.

5.0 Operators

5.1 Introduction

The operators are presented here in alphabetic order. Each operator description begins first with the name of the opcode followed by a format that describes the stack before and after execution of the operator. The sequence of tokens appearing to the left of the arrow → is the required operand stack contents before execution of the operator. The object at the top of the stack is the rightmost object. The objects to the right of the arrow are the stack contents following execution of the operator; again, the top of stack is the rightmost object. The words *first* or *second* in a description apply from the left of a sequence i.e. the first operand is the first pushed onto the stack. A null sequence of objects is marked by a "–" where "– → –" means that the operator requires no arguments and leaves none. The special token "|-" represents the bottom of the stack. Then follows a detailed explanation of what the operator does. An IMPLEMENTATION entry may be included to describe issues relating to the implementation of the operator on different servers. The ERRORS entry lists the errors that may be executed by the operator. The SEE ALSO entry lists related operators. A COMMENT section is included in this draft of the protocol to present issues or arguments relating to the operator.

5.2 Operator Details

This list of operators is intended to represent the "core set" that all implementations will attempt to support. The list is not complete nor precisely defined - more operators will likely be added in successive drafts.

There are no operators to support versioned file systems. The lookup operator is defined to return just the "most recent" version of a file. There needs to be an equivalent operator that allows a particular version of a file to be selected. Similarly, there needs to be support for versioned files either in the create and readdir operators or, perhaps more suitably, new operators that are designed specifically to support versioned files.

There needs to be an extension mechanism defined to cope with new operators and the objects that they operate on.

[

– → mark

Pushes a **mark** object onto the operand stack. This is used to mark the beginning of an arbitrary number of objects that will be put on the stack to be formed into a new array with the **]** operator.

ERRORS:
stackoverflow

SEE ALSO:
], mark, array, astore

]

mark obj_0 ... obj_{n-1} → array

Pops *n* elements from the stack down to the mark and forms them into a new array with *n* elements. The new array is pushed onto the stack.

ERRORS:
unmatchedmark

SEE ALSO:
], mark, array, astore

{ — $\rightarrow$ —

Marks the beginning of a sequence of objects that define an executable array. The following objects in the input stream are scanned by the interpreter in *deferred execution* mode.

ERRORS:
stackoverflow

SEE ALSO:
}, mark, array

} — $\rightarrow$ proc

Marks the end of a sequence of objects that define an executable array. The executable array is pushed onto the operand stack.

ERRORS:
unmatchedmark

SEE ALSO:
{, mark, array

= obj $\rightarrow$ —

Print a text representation of the object on the server's standard output.

IMPLEMENTATION:
This operator is intended for debugging. Servers may choose to support it selectively.

ERRORS:
stackunderflow

SEE ALSO:
print, pstack

abs $int_1 \rightarrow int_2$

Pushes the absolute value of int_1 onto the stack

ERRORS:
stackunderflow, typecheck

SEE ALSO:
typecheck

access

fh → accessbits

Determine what type of access is allowed on the filesystem object represented by *fh*. A*ccessbits* is an integer that represents a bit field containing bit encodings of the types of access permitted. Each kind of access is represented by a bit. A bit set to a 1 means that the corresponding access type is allowed. Bit encodings are as follows:

0x01	READ	The data in the object *fh* can be accessed
0x02	WRITE	The data in the object *fh* can be changed
0x04	CHANGE	The attributes of the object *fh* can be changed
0x08	LOOKUP	Where *fh* is a directory a **lookup** may be performed
0x10	REMOVE	Where *fh* is a directory a **remove** may be performed

Is this set of access types sufficient ? What others are there ?

IMPLEMENTATION:

The **access** operation is provided to allow clients to do access checking 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. The permissions returned by **access** are not permanent. Access permissions can change at any time.

The credentials of the requestor are assumed to be in the execution environment and accessible to the filesystem operations. Credentials are conveyed into the environment by the Interpreter Network Interface (see section 3.3.4 on page 6).

ERRORS:

stackunderflow, typecheck, nefs_stale

SEE ALSO:

getattr, setattr

COMMENT:

Remove permission assumes that the filesystem object referenced by the directory entry may itself be removed even if write permission is not granted.

Write permission may allow attributes to change as a side-effect even if change permission is not granted.

add

int_1 int_2 → sum

The sum of two integers.

ERRORS:

stackunderflow, typecheck, undefinedresult

SEE ALSO:

div, mul, sub, div, mod

aload

array → obj_0 ... obj_{n-1} array

Pushes all the elements of *array* onto the operand stack followed by the array itself.

ERRORS:

invalidaccess, stackoverflow, stackunderflow, typecheck

SEE ALSO:

astore, get, getinterval

and

$bool_1$ $bool_2$ → $bool_3$
int_1 int_2 → int_3

If the objects are booleans it pushes a boolean object that represents their *logical* and. If the objects are integers the result is their *bitwise* and.

ERRORS:
stackunderflow, typecheck

SEE ALSO:
or, xor, not, true, false

array

int → array

Returns an *array* of size *int*. The array is initialized with null objects.

ERRORS:
rangecheck, stackunderflow, typecheck, nomem

SEE ALSO:
[,], aload, astore

astore

obj_n ... obj_{n-1} array → array

Stores *n* objects from the stack into the array, where *n* is the length of the array. This is the inverse of **load**.

ERRORS:
invalidaccess, stackunderflow, typecheck

SEE ALSO:
aload, put, putinterval

begin

dict → –

Push *dict* onto the dictionary stack making it the current dictionary.

ERRORS:
dictstackoverflow, invalidaccess, stackunderflow, typecheck

SEE ALSO:
end, countdictstack, dictstack

bitshift

int_1 shift → int_2

Shifts the binary representation of int_1 left by *shift* bits. Bits shifted out are lost. Bits shifted in are zero. If *shift* is negative then the shift is a rightward shift by –*shift* bits.

ERRORS:
stackunderflow, typecheck

SEE ALSO:
and, or, xor, not

clear |- obj_1 ... obj_n → |-

Pops all objects from the operand stack.

ERRORS: (none)

SEE ALSO:
count, cleartomark, pop

cleartomark mark obj_1 ... obj_n → –

Pops all objects down to a *mark* and pops the mark itself.

ERRORS:
unmatchedmark

SEE ALSO:
clear, mark, counttomark, pop

copy obj_1 ... obj_n n → obj_1 ... obj_n obj_1 ... obj_n
$dict_1$ $dict_2$ → $dict_2$
$array_1$ $array_2$ → subarray
str_1 str_2 → substring

If the topmost object is an integer then the topmost *n* objects on the stack are duplicated. If the topmost objects are arrays then all the objects of $array_1$ are copied into $array_2$. $Array_2$ must be at least as big as $array_1$. If the objects are dictionaries, the contents of $dict_1$ are copied into $dict_2$. The **length** of $dict_2$ must be zero and it must have a **maxlength** as least as big as $dict_1$. If the objects are strings then all the bytes of $string_1$ are copied into $string_2$. $String_2$ must be at least as big as $string_1$.

ERRORS:
invalidaccess, rangecheck, stackunderflow, stackoverflow, typecheck

SEE ALSO:
dup, get, put, putinterval

count |- obj_1 ... obj_n → |- obj_1 ... obj_n n

Counts the number of objects on the operand stack and pushes the count on the operand stack.

ERRORS:
stackoverflow

SEE ALSO:
counttomark

countexecstack – → int

Count the number of objects on the execution stack.

ERRORS:
stackoverflow

SEE ALSO:
execstack

counttomark mark obj_1 ... obj_n → mark obj_1 ... obj_n n

Counts the objects on the operand stack down to the first mark and pushes this count on the operand stack.

ERRORS:
stackoverflow, unmatchedmark

SEE ALSO:
mark, count

create fh_1 filename attrs → fh_2

fh_1 null attrs → fh_2 filename

Create a filesystem object with *filename* in the directory represented by fh_1. If a *null* object is provided as a filename the server will create a assign a unique filename to the new object and return the filename. The new object represented by fh_2 will have the attributes contained in *attrs*. The *filetype* file attribute must be set. Any previously existing object with the same filename will be destroyed.

IMPLEMENTATION:
Clients may set the **size** attribute as a hint to the server of the final size of the file. This may be useful to servers that allocate space for large files differently from small files. Space allocated in this way need not be physically allocated but it must appear to contain binary zeroes.

ERRORS:
stackunderflow, typecheck, nefs_stale, nefs_notdir, nefs_nametoolong, nefs_rofs, nefs_nospace, nefs_badname

SEE ALSO:
remove

COMMENT:
Both the valid and lock ops may be used in conjunction with create to duplicate the semantics of a non-exclusive create e.g.

```
dfh true null {
      dfh (foo) valid
      {
            oldfh ne { - return error - } if
      } if
      attrs create
} lock
```

means "create a new file (foo). If an old (foo) exists it must have the filehandle oldfh."

cvn string → name

Converts a string object to a name object.

ERRORS:
invalidaccess, rangecheck, stackunderflow, typecheck

SEE ALSO:
string

def key value → –

Stores *key* and *value* in the current dictionary (at the top of the dictionary stack). If *key* is already in the dictionary then *value* replaces a previous value.

ERRORS:
dictfull, invalidaccess, limitcheck, stackunderflow, typecheck

SEE ALSO:
store, put

dict int → dict

Creates a new dictionary *dict* with a capacity for *int* key/value pairs.

ERRORS:
rangecheck, stackunderflow, typecheck, nomem

SEE ALSO:
begin, end, length, maxlength

div int_1 int_2 → quotient

Divides int_1 by int_2 and leaves the integer quotient on the stack.

ERRORS:
stackunderflow, typecheck, undefinedresult

SEE ALSO:
div, add, mul, sub, mod

dup obj → obj obj

Duplicates the object on the top of the operand stack. **Dup** duplicates only the object itself. The value of a composite object is not copied – it is shared.

ERRORS:
stackoverflow, stackunderflow

SEE ALSO:
copy, index

encodereply obj_1 ... obj_n n → –

Encodes the top *n* objects on the stack into the output stream. The encoded objects are appended to previously encoded data. The encoding used and the method of enqueuing the data are determined by the underlying network interface.

ERRORS:
stackunderflow, typecheck

SEE ALSO:
sendreply, flushreply

COMMENT:
There must be a limit to the number of objects that can be encoded in the reply buffer. Should this be just part of the overall memory allocation (memstatus on page 28) or should there be a separate limit for the response buffer and a unique error when it is exceeded ?

end $- \rightarrow -$

Pops the current dictionary from the dictionary stack making the one below it the current dictionary.

ERRORS:
dictstackunderflow

SEE ALSO:
begin, dictstack, countdictstack

eq $obj_1\ obj_2 \rightarrow bool$

Pops the two objects from the top of the operand stack and pushes a boolean value *true* if they are equal, *false* if not. Simple objects are equal if their types and values are the same. Strings are equal if they have the same length and sequence of bytes. Arrays and dictionaries are considered equal only if they share the same value; separate values are considered unequal even if their component values are equal.

ERRORS:
invalidaccess, stackunderflow

SEE ALSO:
ne, le, lt, ge, gt

exch $obj_1\ obj_2 \rightarrow obj_2\ obj_1$

Exchange the order of the two objects on top of the stack.

ERRORS:
stackunderflow

SEE ALSO:
dup, roll, index, pop

exec $obj \rightarrow -$

Pops the object from the operand stack and pushes it on the execution stack, thereby executing it.

ERRORS:
stackunderflow

SEE ALSO:
xcheck, cvx

execstack array $\rightarrow$ subarray

Copies the objects from the execution stack into *array* returning *subarray*. *Array* must be big enough to hold all the objects in the execution stack.

ERRORS:
invalidaccess, rangecheck, stackunderflow, typecheck

SEE ALSO:
countexecstack, exec

exit

– → –

Terminates execution of the innermost looping context e.g. loop, for, forall, repeat.

ERRORS:
invalidexit

SEE ALSO:
for, forall, loop, repeat, stop

flushreply

– → –

Discard the objects enqueued for transmission to the client with encodereply.

SEE ALSO:
encodereply, sendreply

for

initial incr limit proc → –

Executes *proc* repeatedly passing it a sequence of values starting with initial and increasing in steps of incr to limit. The control variable is pushed onto the stack prior to each invocation of *proc*. If *incr* is negative the control variable is decremented until it is smaller than *limit*.

ERRORS:
stackoverflow, stackunderflow, typecheck

SEE ALSO:
repeat, loop, forall, exit

forall

array proc → –
dict proc → –
string proc → –

Executes *proc* once for every element in the first object. If it is an array or string it begins with the element whose index is 0. The element in each case is pushed onto the stack prior to each invocation of the procedure proc. For an array the element is an object; for a string an integer in the range 0 – 255, not a one character string. If the first object is a dictionary both the key and value for each entry are pushed onto the stack. Dictionary entries are not presented in any particular order.

ERRORS:
invalidaccess, stackunderflow, typecheck

SEE ALSO:
for, repeat, loop, exit

freeze

attrdict$_1$ keyarray → attrdict$_2$
attrdict$_1$ null → attrdict$_2$

Freeze values in the attribute dictionary *attrdict$_1$*. *Keyarray* is an array of dictionary keys to be frozen. A *null* in place of a *keyarray* implies a **freeze** of all the attributes. Since some file attributes may be expensive to obtain, a server may choose to *lazy-evaluate* file attributes; some attribute values may not evaluated until a **get** following the **getattr** that obtains the attribute dictionary. The **freeze** operator forces the server to evaluate a set of attributes atomically in order that their values are mutually consistent.

IMPLEMENTATION:

The values of attributes named in the *keyarray* are guaranteed not to change even if the real attributes of the file are changed either directly or indirectly as a side-effect. Implementations that **freeze** attribute values with getattr may treat this operator as a null operation. The **freeze** operator should be used immediately after a **getattr** since it cannot be assumed that a **freeze** will obtain current values for the attributes to be frozen.

EXAMPLE:

```
dfh (foo) lookup getattr [ /fsize /ctime ] freeze
```

ERRORS:

stackunderflow, typecheck, undefined

SEE ALSO:

getattr, getfsattr

ge

int$_1$ int$_2$ → bool
str$_1$ str$_2$ → bool

If integers at the top of the stack then it returns *true* if *int$_1$* is greater or equal to *int$_2$*, *false* otherwise. If strings then it assumes that the shorter string is padded with zeros then does a byte by byte comparison assuming that the string bytes are integers in the range 0 – 255.

ERRORS:

invalidaccess, stackunderflow, typecheck

SEE ALSO:

gt, eq, ne, le, lt

get

array index → obj
dict key → obj
string index → int

If the first object is an *array* or *string* **get** returns a single element from the location defined by *index*. The first element has an index of 0. If the first object is a dictionary, **get** returns the value associated with the *key*.

ERRORS:

invalidaccess, rangecheck, stackunderflow, typecheck, undefined

SEE ALSO:

put, getinterval

getattr fh → attrdict

Returns the attributes of the filesystem object represented by *fh*. Each file attribute is represented by a name and value in *attrdict*. Attribute values that may not be modified will have the **readonly** access attribute. The set of file attributes will vary depending on the value of the **filetype** attribute.

IMPLEMENTATION:

The attributes of filesystem objects is the point of most disagreement between different operating systems. Servers should make an effort to map file attributes into the core set. Attributes that have no equivalent in the core set should be mapped into the set of names defined for the filesystem flavor.

All native attributes must be represented in this dictionary. The set of native attributes comprises the attributes that are known to the underlying filesystem. Servers may choose to support foreign attributes as well. Foreign attributes may be set and read by clients but no interpretation will be done by the server.

The set of attributes in attrdict is not required to be consistent. The implementation may choose just to create references to the real file attributes and defer their evaluation until a **get** operator is used. The **freeze** operator may be used to guarantee consistency for any set of attributes.

ERRORS:

stackunderflow, typecheck, nefs_stale, nefs_access

SEE ALSO:

setattr, freeze

COMMENT:

The client must have a way of knowing the accuracy of the time-valued attributes supplied by the server. A time granularity factor may be a good candidate for a filesystem attribute obtainable with getfsattr.

getfsattr fh → fsattr

Return a dictionary *fsattr* that contains the attributes of the filesystem in which the filesystem object represented by the filehandle *fh* is a member.

IMPLEMENTATION:

Servers should make an effort to map filesystem attributes into the core set. Attributes that have no equivalent in the core set should be mapped into the set of names defined for the filesystem flavor.

All native attributes must be represented in this dictionary. The set of native attributes comprises the attributes that are known to the underlying filesystem. Servers may choose to support foreign filesystem attributes as well. Foreign attributes may be set and read by clients but no interpretation will be done by the server.

The set of attributes in attrdict is not required to be consistent. The implementation may choose just to create references to the real filesystem attributes and defer their evaluation until a **get** operator is used. The **freeze** operator may be used to guarantee consistency for any set of attributes.

ERRORS:

stackunderflow, nefs_stale

SEE ALSO:

getattr, freeze

getinterval

array index count → subarray
string index count → substring

Gets a *subarray* or *substring* object that shares some sequence of elements from the original *array* or *string*.

ERRORS:
invalidaccess, rangecheck, stackunderflow, typecheck

SEE ALSO:
get, putinterval

gt

int_1 int_2 → bool
str_1 str_2 → bool

If integers at the top of the stack then it returns *true* if int_1 is greater than int_2, *false* otherwise. If strings then it assumes that a shorter string is padded with zeros then does an element by element comparison assuming that the string elements are integers in the range 0 – 255.

ERRORS:
invalidaccess, stackunderflow, typecheck

SEE ALSO:
ge, eq, ne, le, lt

if

bool proc → –

Pops both objects from the top of the stack. If *bool* is **true** then *proc* is pushed onto the execution stack and executed.

ERRORS:
stackunderflow, typecheck

SEE ALSO:
ifelse

ifelse

bool $proc_1$ $proc_2$ → –

Pops all three objects from the top of the stack. If *bool* is **true** then $proc_1$ is pushed onto the execution stack and executed; if *false* then $proc_2$ is executed.

ERRORS:
stackunderflow, typecheck

SEE ALSO:
if

inactive

fh → –

The client no longer requires the filehandle *fh*. The server may choose to release resources associated with the filehandle itself.

IMPLEMENTATION:

The inactive operation is advisory only. The server cannot rely on the client to do an **inactive** for every filehandle since filehandles may be lost due to client crashes. Clients should make a best effort to invoke **inactive** whenever a filehandle is thrown away.

ERRORS:

stackunderflow, typecheck, nefs_stale

SEE ALSO:

create, remove

index

obj_n ... obj_0 n → obj_n ... obj_0 obj_n

Pops the integer *n* from the stack, fetches the *nth* element from the top of the stack and pushes it onto the stack. “0 index” has the same effect as **dup**.

ERRORS:

rangecheck, stackunderflow, typecheck

SEE ALSO:

copy, dup, roll, sget

instance

– → instance

Return the server’s current instance. This is an integer quantity that the client may use to compare with previous values of instance to detect a server crash and possible loss of state.

ERRORS:

stackoverflow

SEE ALSO:

setattr, write, sync

COMMENT:

A server under heavy load may choose to arbitrarily discard client state e.g. cached file modifications and signal its loss of state to clients by changing the instance.

known

dict key → bool

Returns **true** if *key* has an entry in *dict*.

ERRORS:

invalidaccess, stackunderflow, typecheck

SEE ALSO:

where, load, get

le

int_1 int_2 → bool
str_1 str_2 → bool

If integers at the top of the stack then it returns **true** if *int_1* is less than or equal to *int_2*, **false** otherwise. If strings then it assumes that a shorter string is padded with zeros then does an element by element comparison assuming that the string elements are integers in the range 0 – 255.

ERRORS:
invalidaccess, stackunderflow, typecheck

SEE ALSO:
lt, eq, ne, ge, gt

length

array → int
dict → int
string → int

If the object is an *array* or *string* **length** returns the number of elements in its value. If the object is a *dictionary* it returns the current number of key/value pairs that it contains.

ERRORS:
invalidaccess, stackunderflow, typecheck

SEE ALSO:
maxlength, array, dict, string

link

dfh_1 filename dfh_2 → fh

Create an additional filename in directory *dfh_2* for the file represented by *dfh_1*. Any previous instance of *filename* in *dfh_2* will be destroyed.

ERRORS:
stackunderflow, typecheck, nefs_stale, nefs_access, nefs_notdir, nefs_nametoolong, nefs_badname, nefs_rofs, nefs_nospace, nefs_xfsop

SEE ALSO:
rename, remove

COMMENT:
This operator is very Unix-specific. Should all the OS-specific operators and filetypes by relegated to appropriate appendices? This operator list should be confined to only those operators that all implementations can reasonably expect to implement.

Note that there are no operators for Unix symbolic links. A symbolic link can be represented by a special file type - the link itself can be represented as an attribute of the file. This file type will be described in a Unix "flavor" of the protocol.

load

key → value

Searches the dictionary stack beginning with the current dictionary for a dictionary that contains an entry for *key*. When located the *value* is returned.

ERRORS:
invalidaccess, stackunderflow, typecheck, undefined

SEE ALSO:
where, get

lock

fh excl timeval proc → bool
fh excl null proc → –

Execute *proc* with the filesystem object represented by *fh* locked. While executing proc with the filehandle locked the server guarantees consistency with a *shared* lock or atomic update with an *exclusive* lock. If *excl* is **true** the lock is exclusive. An exclusive lock will not be granted unless the client has write access to the object and there are no other locks held on the object (exclusive or shared). If *excl* is **false** the lock is shared. A shared lock will not be granted unless the client has read access to the object and there are no exclusive locks being held. Multiple requests may concurrently hold shared locks but only a single request can hold an exclusive lock.

If the lock cannot be granted because of currently held locks the **lock** operator will block until the lock can be granted or until the time specified by *timeval* has elapsed. A zero timeval may be used to avoid blocking. If the lock is acquired and *proc* is executed then a boolean value **true** is left on the stack. If the lock cannot be acquired within the time specified by *timeval* then a **false** is left on the stack. If a null object is used (to represent an infinite timeval) - no boolean is left on the stack.

Beware of deadlocks when using the **lock** operator within *proc*.

EXAMPLE:

```
dfh true null {
      dfh (foo) valid { - exist error - } { attrs create } ifelse }
lock

fh true null { fh dup getattr /fsize get data write } lock
```

IMPLEMENTATION:

Mandatory locking is required for all NeFS clients. Mandatory locking for access by local processes on the server is preferable but not required. Locking may be implemented at the filehandle level.

ERRORS:

stackunderflow, nefs_stale, nefs_access

COMMENT:

Should it be an error to nest locks? Deadlock is very dangerous.

Should there be a limit on the number of locks that can be held simultaneously by a single request ?

SEE ALSO:

exec

loop

proc → –

Repeatedly executes *proc* until *proc* executes an **exit**.

ERRORS:

stackunderflow, typecheck

SEE ALSO:

for, repeat, forall, exit

lookup

dfh filename → fh

Returns the filehandle *fh* that corresponds to *filename* in the directory *dfh*.

IMPLEMENTATION:

Where the filename refers to a mount point on the server two different replies are possible. The server can return either the filehandle for the underlying directory that is mounted on, or the filehandle of the root of the mounted directory. Returning the filehandle of the underlying directory forces the client to use the MOUNT service to access the mounted directory, which insures that MOUNT access checking can be done on the server. Such a directory has the attribute name **mount**. Alternatively, servers that don't care about MOUNT access checking can return the filehandle of the mounted directory and automatically provide access to the filesystem. Such a directory does not have **mount** as an attribute name.

A **lookup** in a filesystem that supports multiple file versions returns the most recent version of the file.

ERRORS:

stackunderflow, typecheck, nefs_stale, nefs_access, nefs_noent, nefs_notdir, nefs_nametoolong

SEE ALSO:

create, remove, inactive, getattr, valid

lt

int_1 int_2 → bool

str_1 str_2 → bool

If integers at the top of the stack then it returns *true* if int_1 is less than int_2, *false* otherwise. If strings then it assumes that a shorter string is padded with zeros then does an element by element comparison assuming that the string elements are integers in the range 0 – 255.

ERRORS:

invalidaccess, stackunderflow, typecheck

SEE ALSO:

le, eq, ne, ge, gt

mark

– → mark

Pushes a mark object onto the operand stack.

ERRORS:

stackoverflow

SEE ALSO:

counttomark, cleartomark, pop

maxlength

dict → int

Returns the maximum number of key-value pairs that *dict* can hold.

ERRORS:

invalidaccess, stackunderflow, typecheck

SEE ALSO:

length, dict

memstatus – → used limit

Return the number of bytes of dynamically allocated memory currently being *used* and the memory *limit*. If *used* exceeds *limit* then a **nomem** error will be raised.

IMPLEMENTATION:

Some statically allocated memory will be assigned to the request for the duration of its execution. This memory includes that allocated to the operand, execution and dictionary stacks. Statically allocated memory is not included in *used* or *limit*

Dynamically allocated memory is consumed by new composite objects: strings, arrays and dictionaries. These objects can be explicitly allocated with the **string**, **array** and **dict** operators. They may also be allocated as side-effects of other operators: **getattr** creates a new dictionary, **read** creates a new string. The interpreter frees dynamic memory implicitly when there are no objects referencing it; for instance a **pop** operator that removes a string object from the stack will free the memory assigned to hold the string value if the stack object is the only reference.

ERRORS:

stackoverflow

COMMENT:

Should the request be able to change the limit ? The client may wish to make the limit smaller.

mod int_1 int_2 → remainder

Returns the remainder that results from dividing int_1 by int_2.

ERRORS:

stackunderflow, typecheck, undefinedresult

SEE ALSO:

div

mul int_1 int_2 → product

Returns the product of int_1 and int_2.

ERRORS:

stackunderflow, typecheck, undefinedresult

SEE ALSO:

div, add, sub, mod

ne obj_1 obj_2 → bool

Compare two objects and push a boolean value *false* if they are equal, *true* if not. See **eq** for what it means to be equal.

ERRORS:

invalidaccess, stackunderflow

SEE ALSO:

eq, ge, gt, le, lt

neg

$\text{int}_1 \rightarrow \text{int}_2$

Returns the negative of *int*.

ERRORS:
stackunderflow, typecheck

SEE ALSO:
abs

not

$\text{bool}_1 \rightarrow \text{bool}_2$
$\text{int}_1 \rightarrow \text{int}_2$

If the operand is a *boolean*, **not** returns its logical negation. If the operand is an *integer*, **not** returns its *bitwise* (one's) complement.

ERRORS:
stackunderflow, typecheck

SEE ALSO:
and, or, xor, if

null

$- \rightarrow \text{null}$

Push a null object onto the operand stack.

ERRORS:
stackoverflow

SEE ALSO:
array, type

or

$\text{bool}_1\ \text{bool}_2 \rightarrow \text{bool}_3$
$\text{int}_1\ \text{int}_2 \rightarrow \text{int}_3$

If the operands are booleans, **or** returns the *logical* or of their values. If they are integers, **or** returns their *bitwise* inclusive or.

ERRORS:
stackunderflow, typecheck

SEE ALSO:
and, not, xor

pop

$\text{obj} \rightarrow -$

Discard the object at the top of the stack.

ERRORS:
stackunderflow

SEE ALSO:
clear, dup

print str → –

Write the string *str* on the server's standard output.

IMPLEMENTATION:

The string is assumed to be printable ASCII. A newline character (0x0a) must be appended if a line-feed is required.

ERRORS:

stackunderflow, typecheck

SEE ALSO:

=, pstack

COMMENT:

This operator is intended for debugging. Is it otherwise useful ? It could be used by a client to present messages on the server's console. Is a more specific operator needed e.g. something akin to the Unix syslog facility - it allows messages to be prioritized.

pstack |- obj_0 ... obj_n → |- obj_0 ... obj_n

Write an ASCII representation of every object in the stack to the interpreter's *stderr*.

ERRORS:

stackoverflow

SEE ALSO:

=, count

COMMENT:

This operator is intended for debugging. Servers could choose to support it selectively.

put array index obj → –

dict key obj → –

string index int → –

Replace the single element of the *array*, dict or *string*. If an array or string he index must be in the range 0 – n-1 where n is the length of the array or string. If a dictionary the *key* and *obj* are stored in the dictionary. If an entry already exists for *key* then its value is replaced.

ERRORS:

dictfull, invalidaccess, rangecheck, stackunderflow, typecheck,

SEE ALSO:

get, putinterval

putinterval $array_1$ index $array_2$ → –

str_1 index str_2 → –

Replace a sequence of elements in the first array or string by the entire contents of the second array or string. The sequence replaced begins at *index* in the first array or string. The first array or string must be at least as long as the second.

ERRORS:

invalidaccess, rangecheck, stackunderflow, typecheck

SEE ALSO:

getinterval, put

quit

– → –

Terminate the interpreter for this context.

ERRORS: (none)

SEE ALSO:
stop

rcheck

obj → bool

Return **true** if the value of the object can be read, **false** otherwise.

ERRORS:
stackunderflow, typecheck

SEE ALSO:
readonly, wcheck

read

fh offset length → data

Read the string *data* from the byte-oriented file represented by *fh*. The first byte of data in the file has an offset of 0. *Length* is the number of bytes of data to be read. The server may return less data than requested if the length exceeds the filesystem **maxreadsize** attribute. The *data* is returned as a string. This operator should be supported only for filesystem objects that allow access to arbitrary ranges of bytes. The *offset* value may be represented by either an integer or a hyper. Fewer than *length* bytes of data will be returned if *offset* plus *length* extends beyond the end of the file.

IMPLEMENTATION:

The server has the option of returning less than *length* bytes of data if it is unable to manage the amount of data requested. The maximum read length supported by the server is available as a filesystem attribute.

ERRORS:
stackunderflow, typecheck, nefs_access, nefs_stale, nefs_wrongtype, nefs_badoffset, nefs_io

SEE ALSO:
readrec, write

COMMENT:

Does there need to be an explicit indication of end of file?

Should data that consists of length binary zeroes be represented by a null object? In the case of a sparse file the server can avoid allocating, copying, and transmitting blocks of binary zeroes. Where an implementation allows files to have "holes", a read inside a hole could easily return a null without having to do a byte-by-byte check for non-null data.

readonly

obj → obj

Changes the access attribute of the object to read-only. Once the read-only attribute has been set it cannot be reset. Any attempt to modify the value of a read-only object will result in an **invalidaccess** error. For an array or string object the only the access to the object itself is affected, other objects that share the same value will retain their access attributes. There is an exception for dictionary objects; readonly affects the value of the object. Other objects sharing the same dictionary value will be affected.

ERRORS:
stackunderflow, invalidaccess, typecheck

SEE ALSO:
rcheck, wcheck

readrec

fh $offset_1 \rightarrow$ data $offset_2$
fh key $\rightarrow$ data

Read a record. If *fh* represents a file where records are indexed by an integer then $offset_1$ is an integer that identifies the record to be read. The first record always has an $offset_1$ of 0. If the records are fixed length then the offset represents the ordinal number of the record in the file.

If the records have variable length then the $offset_1$ value may be interpreted only by the server. The first record has an offset of 0. Only sequential access is allowed - each **readrec** returns an $offset_2$ value that may be used to read the next record in the sequence.

Whether the file has fixed or variable length records, the returned $offset_2$ is always that of the next record in sequence. The $offset_2$ returned by the last record in the file is 0.

If *fh* represents a keyed access file then *key* is a string used to locate the record to be read. No offset value is returned.

ERRORS:
stackunderflow, typecheck, nefs_access, nefs_stale, nefs_wrongtype, nefs_badoffset, nefs_io

SEE ALSO:
read, writerec, setattr, sync

COMMENT:
Does this operator provide sufficient support for record-oriented files? Can clients and servers that mutually support record oriented files use this operator? Can Unix clients make use of this to access record oriented files? Should a Unix server attempt to emulate record oriented files?

Can all implementations of record oriented files support random access to fixed length records efficiently?

readdir

dfh offset proc → –

For each entry in a directory represented by *dfh*, **readdir** pushes the offset of the next entry followed by the filename of the entry onto the operand stack and invokes *proc*. Every entry in a directory is assumed to have a unique offset. The *offset* of the first entry is 0. The units of the offset value may vary depending on the implementation. Invocation of the **exit** or **stop** operator will terminate the **readdir** operation. The offset value pushed onto the stack may be used to continue reading a directory from an arbitrary position in a subsequent **readdir** invocation.

A **readdir** in a filesystem that supports multiple file versions will return only the most recent version of a file.

EXAMPLE:

```
dfh 0 { pop dfh exch lookup getattr /size get total add /total exch def }  readdir
```

IMPLEMENTATION:

The *proc* should not contain operations which may change the directory contents. Such changes may invalidate the *offset* value and/or result in failure of the **readdir** operator. Concurrent changes to the directory by other clients may also cause **readdir** to fail. After failure, recovery may be attempted by restarting the **readdir** with an *offset* of 0.

ERRORS:

stackunderflow, typecheck, nefs_access, nefs_stale, nefs_notdir, nefs_badoffset

SEE ALSO:

create, lookup, valid, remove

COMMENT:

Changes to the directory during execution of a readdir could definitely cause problems if the changes cause the offsets to become invalid and prevent continued execution. For Unix implementations this does not appear to be a problem since the offset cannot become invalid through concurrent directory modification.

An extreme solution would be to have the readdir op apply an implicit shared lock on the directory to prevent modification.

remove

dfh filename → –

Remove the entry *filename* for a filesystem object from the directory represented by *dfh*. This operation will succeed if entry has already been removed.

IMPLEMENTATION:

This operator may not necessarily remove the filesystem object that the directory entry references if the object has an **nlinks** attribute and it has a value greater than 1.

ERRORS:

stackunderflow, typecheck, nefs_stale, nefs_noent, nefs_notdir, nefs_access, nefs_notempty, nefs_rofs

SEE ALSO:

create, inactive

rename

dfh_1 $filename_1$ dfh_2 $filename_2$ → fh

Change the filename of a filesystem object from *filename*$_1$ in directory *dfh*$_1$ to *filename*$_2$ in directory *dfh*$_2$. Both *dfh*$_1$ and *dfh*$_2$ may be the same. As well as a new filename the filesystem object gets a new filehandle *fh*. **Rename** must be atomic on the server - it cannot be interrupted in the middle. This operation will destroy any previously existing reference to a file with *filename*$_2$ in directory *dfh*$_2$.

IMPLEMENTATION:
Some implementations may require the directories *dfh*$_1$ and *dfh*$_2$ to be in the same filesystem.

ERRORS:
stackunderflow, typecheck, nefs_stale, nefs_notdir, nefs_access, nefs_noent, nefs_nametoolong, nefs_badname, nefs_rofs, nefs_xfsop

SEE ALSO:
link

repeat

int proc → –

Removes both operands from the stack and executes *proc int* times. If *proc* executes an **exit** then the **repeat** will terminate prematurely.

ERRORS:
rangecheck, stackunderflow, typecheck

SEE ALSO:
for, loop, forall, exit

roll

obj_{n-1} ... obj_0 size count → *(rolled objs)*

Performs a circular shift of size objects on the operand stack count times. If *count* is positive the movement is up the stack. A negative count gives movement down the stack.

```
EXAMPLE:
(a) (b) (c) 3 -1 roll → (b) (c) (a)
(a) (b) (c) 3  1 roll → (c) (a) (b)
(a) (b) (c) 3  0 roll → (a) (b) (c)
```

ERRORS:
rangecheck, stackoverflow, stackunderflow, typecheck

SEE ALSO:
exch, index, copy, pop

sendreply

– → –

Transmits the objects encoded by **encodereply** to the client.

ERRORS: (none)

SEE ALSO:
encodereply, flushreply

setattr

fh attrdict → –

Sets the file attributes of *fh* to the values specified by the entries in the dictionary *attrdict*. Attributes of *fh* that do not appear in *attrdict* will be unchanged. The implementation may restrict the set of attributes that may be set.

IMPLEMENTATION:

When setting time values the client may choose either to supply a time value from its own clock or use the **tod** operator to set the time according to the server's clock.

ERRORS:

stackunderflow, typecheck, nefs_access

SEE ALSO:

getattr

sget

|- obj_0 ... obj_n i → |- obj_0 ... obj_n obj_i

Gets the object at position *i* in the operand stack and pushes it. The object at the base of the stack has an index of 0.

ERRORS:

rangecheck, stackunderflow, typecheck, stackoverflow

SEE ALSO:

index, sput

sput

|- obj_0 ... obj_n i obj → |- obj_0 ... obj_n

Replace the object at position *i* relative to the base of the operand stack by *obj*. The object at the base of the stack has an index of zero.

ERRORS:

rangecheck, stackunderflow, typecheck

SEE ALSO:

index, sget

stop

– → –

Terminates the innermost instance of a stopped context and leaves the boolean value *true* on the operand stack. A stopped context is a procedure invoked by the **stopped** operator.

ERRORS: (none)

SEE ALSO:

stopped, exit

stopped

proc → –

Pushes *proc* onto the execution stack and executes it. Like **exec** except if *proc* runs to completion it leaves a boolean *false* on top of the stack. If proc terminates prematurely as the result of a **stop** then a boolean *true* will be left on the operand stack.

ERRORS:

stackunderflow

SEE ALSO:

stop

store

key value → –

Searches the dictionary stack for a dictionary with an entry for *key* and replaces its value with *value*. If the *key* is not found in any dictionary then a new entry for *key* is made in the current dictionary.

ERRORS:

dictfull, invalidaccess, limitcheck, stackunderflow

SEE ALSO:

def, put, where

string

int → string

Creates a string object of length *int* and pushes it on the stack. The string is initialized with zeros.

ERRORS:

limitcheck, rangecheck, stackunderflow, typecheck, nomem

SEE ALSO:

length, array, dict, type

sub

int_1 int_2 → difference

Returns the result of subtracting int_2 from int_1.

ERRORS:

stackunderflow, typecheck, undefinedresult

SEE ALSO:

add, div, mul, div, mod

sync

fh → –

Commit any changes made to fh to stable storage. This operator must not complete until all changes are on stable storage.

IMPLEMENTATION:

A server that utilizes an I/O cache will not arbitrarily delete client data from the cache without first committing it to the disk or by changing the instance value. The client can detect a loss of cached data by comparing the server's current instance (see **instance** on page 24) with a previous value obtained at the time the data was cached. The client must not delete data from its own cache until it receives confirmation of a successful sync since it must be able to restore the cached data if the server loses it.

Stable storage must be immune to server crash and recovery and temporary power outages. Suitable stable media include magnetic or optical disk, ferrite core, and battery backed-up RAM. A server that lacks an I/O cache must commit all file changes to stable storage immediately. For such a server **sync** will do nothing.

Although a server may choose to commit all cached changes to the file (even those pending from other clients) it is not required to. The work required of the **sync** may be reduced by committing only the changes made by the client requesting the **sync**.

ERRORS:

stackunderflow, typecheck, nefs_stale, nefs_access, nefs_rofs

SEE ALSO:

setattr, write

timestatus

– → used limit

Return the number of milliseconds of clock time that the request has been running. Limit is the maximum time for execution. If *used* exceeds *limit* then a **timeout** error will be raised.

ERRORS:

stackoverflow

COMMENT:

Should the request be able to change the limit ? The client may wish to make the limit smaller.

Should there be a limit on opcodes executed too ?

tod

– → timedate

Return the server's current time and date.

ERRORS:

stackoverflow

SEE ALSO:

setattr

valid

dfh filename → dfh filename fh true

dfh filename → dfh filename false

If *filename* has an entry in directory *dfh* then its filehandle and **true** are pushed onto the stack. If there is no entry for *filename* in *dfh* then **false** is returned. This operator is useful where a subsequent action will be based on the presence or absence of a file.

EXAMPLE:

```
dfh (foo) valid { - exist error - } { attrs create } ifelse
dfh (bar) valid { oldfh ne { - duplicate error - } if } attrs create
```

ERRORS:

stackunderflow, typecheck, nefs_stale, nefs_access, nefs_noent, nefs_notdir, nefs_nametoolong

SEE ALSO:

lookup

wcheck

obj → bool

Return **true** if the value of the object can be changed, **false** otherwise.

ERRORS:

stackunderflow, typecheck

SEE ALSO:

readonly, rcheck

where

key → dict true

key → false

Searches the dictionary stack for a dictionary that contains an entry for *key*. If found it returns the dictionary and *true* on the stack, otherwise it returns *false*.

ERRORS:

invalidaccess, stackoverflow, stackunderflow

SEE ALSO:

known, load, get

write fh offset data → –
fh offset length → –

Write to the byte-oriented file represented by *fh*. *Offset* is a byte offset - the first byte in the file has an offset of 0. If the string *data* is provided it is written to the file. If *length* is provided, it represents the number of zero bytes to be written to the file beginning at *offset*.

IMPLEMENTATION:

The server may or may not commit the data to stable storage. The client must be careful to **sync** the filehandle and confirm that the data is safe before destroying the data at its end in case a server crash loses the data from the server's cache. The client must be able to replay the **write** after server recovery. The **instance** operator may be used by the client to detect loss of uncommitted data from the server's cache.

ERRORS:

stackunderflow, typecheck, nefs_access, nefs_stale, nefs_wrongtype, nefs_fbig, nefs_rofs, nefs_nospace, nefs_badoffset

SEE ALSO:

read, setattr, sync

writerec fh offset data → –
fh key data → –

Write *data* to a record oriented file. *Data* represents the full record to be written. If *fh* represents a file where records are indexed by an integer then *offset* is an integer that indicates the record to be written. The first record has an *offset* of 0. If *fh* represents a keyed access file then *key* is a string used to locate the record to be written. A null object in place of d*ata* implies that an existing record is to be deleted.

IMPLEMENTATION:

The server may or may not commit the data to stable storage. The client must be careful to sync the filehandle and confirm that the data is safe before destroying the data at its end in case a server crash loses the data from the server's buffers. The client must be able to replay the **writerec** after server recovery. The **instance** operator may be used by the client to detect a server crash and possible loss of uncommitted data from the server's cache.

ERRORS:

stackunderflow, typecheck, nefs_access, nefs_stale, nefs_wrongtype, nefs_fbig, nefs_rofs, nefs_nospace, nefs_badoffset

SEE ALSO:

read, setattr, sync

COMMENT:

Does this operator provide sufficient support for record-oriented files? Can clients and servers that mutually support record oriented files use this operator? Can Unix clients make use of this to access record oriented files? Should a Unix server attempt to emulate record oriented files?

xcheck obj → bool

Return **true** if the object is executable, **false** if it is literal.

ERRORS:

stackunderflow

xor $bool_1$ $bool_2$ $\rightarrow$ $bool_3$

int_1 int_2 $\rightarrow$ int_3

If the operands are booleans, **xor** pushes their *logical* exclusive or. If they are integers, **xor** pushes their *bitwise* exclusive or.

ERRORS:

stackunderflow, typecheck

SEE ALSO:

or, and, not

5.3 Operator Summary

5.3.1 Operand Stack Manipulation Operators

pop	obj $\longrightarrow$ —	Discard top element 29
exch	obj_1 obj_2 $\longrightarrow$ obj_2 obj_1	Exchange top two elements 19
dup	obj $\longrightarrow$ obj obj	Duplicate top element 18
copy	obj_1 ... obj_n n $\longrightarrow$ obj_1 ... obj_n obj_1 ... obj_n	Duplicate top n elements 16
index	obj_n ... obj_0 n $\longrightarrow$ obj_n ... obj_0 obj_n	Duplicate arbitrary element 24
roll	obj_{n-1} ... obj_0 size count $\longrightarrow$ (rolled objs)	Roll size elements up count times 34
clear	|- obj_1 ... obj_n $\longrightarrow$ |-	Discard all elements 16
count	|- obj_1 ... obj_n $\longrightarrow$ |- obj_1 ... obj_n n	Count elements on stack 16
mark	— $\longrightarrow$ mark	Push mark on the stack 27
sget	|- obj_0 ... obj_n i $\longrightarrow$ |- obj_0 ... obj_n obj_i	Duplicate arbitrary element in stack 35
sput	|- obj_0 ... obj_n i obj $\longrightarrow$ |- obj_0 ... obj_n	Replace element in stack 35
cleartomark	mark obj_1 ... obj_n $\longrightarrow$ —	Discard elements down through mark 16
counttomark	mark obj_1 ... obj_n $\longrightarrow$ mark obj_1 ... obj_n n	Count elements down to mark 17

5.3.2 Arithmetic Operators

add	int_1 int_2 $\longrightarrow$ sum	int_1 plus int_2 14
div	int_1 int_2 $\longrightarrow$ quotient	int_1 divided by int_2 18
mod	int_1 int_2 $\longrightarrow$ remainder	int_1 mod int_2 28
mul	int_1 int_2 $\longrightarrow$ product	int_1 multiplied by int_2 28
sub	int_1 int_2 $\longrightarrow$ difference	int_1 minus int_2 36
abs	int_1 $\longrightarrow$ int_2	Absolute value of int_1 13
neg	int $\longrightarrow$ int	Negative of int 29

5.3.3 Array Operators

array	int → array	Create an array of length int 15
[	— → mark	Begin an array 12
]	mark obj_0 ... obj_{n-1} → array	End an array 12
{	— → —	Begin an executable array 13
}	— → proc	End an executable array 13
length	array → int	The length of array 25
get	array index → obj	Get the element from array at offset index 21
put	array index obj → —	Replace the element in array at offset index 30
getinterval	array index count → subarray	Copy count elements from array from index 23
putinterval	$array_1$ index $array_2$ → —	Replace elements of $array_1$ by $array_2$ at index 30
aload	array → obj_0 ... obj_{n-1} array	Copy array elements onto the stack 14
astore	obj_n ... obj_{n-1} array → array	Store objects from the stack into array 15
copy	$array_1$ $array_2$ → subarray	Copy initial elements of $array_1$ into $array_2$ 16
forall	array proc → —	Execute proc for each element of array 20

5.3.4 Dictionary Operators

dict	int → dict	Create a dictionary of size int 18
length	dict → int	The number of key-value pairs in dict 25
maxlength	dict → int	The maximum number of key-value pairs in dict 27
begin	dict → —	Push dict on the dictionary stack 15
end	— → —	Pop an entry off the dictionary stack 19
def	key obj → —	Assign obj to key in the current dictionary 18
load	key → value	Find key in dictionary stack and return its value 25
store	key obj → —	Replace topmost entry for key in dictionary stack 36
get	dict key → obj	Get the value for key in dict 21
put	dict key obj → —	Assign obj to key in dict 30
known	dict key → bool	Test if key is in dict 24
where	key → dict true	
	key → false	Find dict in which key exists 37
copy	$dict_1$ $dict_2$ → $dict_3$	Copy elements of $dict_1$ to $dict_2$ 16
forall	dict proc → —	Execute proc for each entry in dict 20

5.3.5 String Operators

string	int → string	Create a string with length int 36
length	str → int	The number of bytes in str 25
get	str index → obj	Get the byte with offset index in str 21
put	str index int → —	Put int into str at offset index 30
getinterval	str index count → substr	Get a substring of str starting at index for count bytes 23
putinterval	str_1 index str_2 → —	Replace bytes in str_1 by str_2 beginning at index 30
copy	str_1 str_2 → substring	Replace initial bytes of str_2 by str_1 16
forall	str proc → —	Invoke proc for each element of str 20

5.3.6 Relational Boolean and Bitwise Operators

eq	obj_1 obj_2 $\longrightarrow$ bool	Equal 19
ne	obj_1 obj_2 $\longrightarrow$ bool	Not equal 28
ge	int_1 | str_1 int_2 | str_2 $\longrightarrow$ bool	Greater than or equal 21
gt	int_1 | str_1 int_2 | str_2 $\longrightarrow$ bool	Greater than 23
le	int_1 | str_1 int_2 | str_2 $\longrightarrow$ bool	Less than or equal 25
lt	int_1 | str_1 int_2 | str_2 $\longrightarrow$ bool	Less than 27
and	$bool_1$ | int_1 $bool_2$ | int_2 $\longrightarrow$ $bool_3$ | int_3	Logical or bitwise and 15
not	$bool_1$ | int_1 $\longrightarrow$ $bool_2$ | int_2	Logical or bitwise not 29
or	$bool_1$ | int_1 $bool_2$ | int_2 $\longrightarrow$ $bool_3$ | int_3	Logical or bitwise or 29
xor	$bool_1$ | int_1 $bool_2$ | int_2 $\longrightarrow$ $bool_3$ | int_3	Logical or bitwise exclusive or 39
bitshift	int_1 shift $\longrightarrow$ int_2	Bitwise shift of int_1 15

5.3.7 Control Operators

exec	obj $\longrightarrow$ –	Execute an object 19
if	bool proc $\longrightarrow$ –	Execute proc if bool is true 23
ifelse	bool $proc_1$ $proc_2$ $\longrightarrow$ –	Execute $proc_1$ if bool is true, otherwise $proc_2$ 23
for	initial incr limit proc $\longrightarrow$ –	Execute proc in steps of incr from initial to limit 20
repeat	int proc $\longrightarrow$ –	Execute proc int times 34
loop	proc $\longrightarrow$ –	Execute proc indefinitely 26
exit	– $\longrightarrow$ –	Exit innermost looping construct 20
stop	– $\longrightarrow$ –	Exit a stopped context 35
stopped	proc $\longrightarrow$ –	Establish proc as a stopped context 35
countexecstack	– $\longrightarrow$ int	The number of elements on the execution stack 16
execstack	array $\longrightarrow$ subarray	Copy exec stack into array 19
quit	– $\longrightarrow$ –	Exit the interpreter immediately 31

5.3.8 NeFS Operators

create	dfh filename attrdict $\longrightarrow$ fh	Create a file 17
	dfh null attrdict $\longrightarrow$ fh filename	Create a file with a unique filename 17
remove	dfh filename $\longrightarrow$ —	Remove a file 33
rename	dfh_1 $filename_1$ dfh_2 $filename_2$ $\longrightarrow$ fh	Rename a file 34
access	fh $\longrightarrow$ accessbits	Get file accessibility 14
readdir	dfh offset proc $\longrightarrow$ —	Read a directory 33
lookup	dfh filename $\longrightarrow$ fh	Lookup a filename in a directory 27
valid	dfh filename $\longrightarrow$ dfh filename fh true	
	dfh filename $\longrightarrow$ dfh filename false	Validate directory entry 37
lock	fh excl timeval proc $\longrightarrow$ bool	
	fh excl null proc $\longrightarrow$ —	Lock a filesystem object 26
freeze	$attrdict_1$ keyarray $\longrightarrow$ $attrdict_2$	
	$attrdict_1$ null $\longrightarrow$ $attrdict_2$	Freeze attribute values 21
getattr	fh $\longrightarrow$ attrdict	Get file attributes 22
setattr	fh attrdict $\longrightarrow$ —	Set file attributes 35
read	fh offset length $\longrightarrow$ data	Read data from a byte oriented file 31
readrec	fh offset $\longrightarrow$ data	Read a record 32
	fh key $\longrightarrow$ data	Read a keyed access record 32
write	fh offset data $\longrightarrow$ —	Write data to a byte oriented file 38
	fh offset length $\longrightarrow$ —	Write nulls to a byte oriented file 38
writerec	fh offset data $\longrightarrow$ —	Write a record 38
	fh key data $\longrightarrow$ —	Write a keyed access record 38
link	dfh_1 filename dfh_2 $\longrightarrow$ fh	Create a link to a file 25
sync	fh $\longrightarrow$ —	Flush changes to stable storage 36
inactive	fh $\longrightarrow$ —	Filehandle no longer needed 24
getfsattr	fh $\longrightarrow$ fsattr	Get filesystem attributes 22
tod	— $\longrightarrow$ timedate	Get the server's time of day 37
instance	— $\longrightarrow$ instance	Get the server's instance cookie 24

5.3.9 I/O Operators

=	obj $\longrightarrow$ —	Print the value of obj at the server 13
encodereply	obj_1 ... obj_n n $\longrightarrow$ —	Enqueue objects for transmission to client 18
print	str $\longrightarrow$ —	Print the string at the server 30
pstack	|- obj_1 ... obj_n $\longrightarrow$ |- obj_1 ... obj_n	Print the values of all objects on the opstack 30
sendreply	— $\longrightarrow$ —	Transmit enqueued objects to the client 34
flushreply	— $\longrightarrow$ —	Discard objects enqueued for transmission 20

5.3.10 Miscellaneous Operators

timestatus	– $\rightarrow$ used limit	Execution time used and limit 37
memstatus	– $\rightarrow$ used limit	Memory used and limit 28
readonly	obj $\rightarrow$ obj	Make obj readonly 31
xcheck	obj $\rightarrow$ bool	Query executable object attribute 38
rcheck	obj $\rightarrow$ bool	Query read access to obj 31
wcheck	obj $\rightarrow$ bool	Query write access to objs 37
cvn	string $\rightarrow$ name	Convert a string to a name 17

6.0 Errors

Error handling is conducted according to the PostScript model. When an operator raises an error it references a system-wide dictionary called **errordict** and looks up a name that corresponds to the error. These error names are listed in section 6.3 on page 46 and section 6.4 on page 47. The object corresponding to the name is usually a procedure that just collects error information (including the error name) into a dictionary called **$error**. It then executes a **stop** operator.

Execution of the **stop** causes the request to exit the innermost enclosing **stopped** context. Assuming that the request has not invoked **stopped** itself, the **stop** is caught by a **stopped** context that has been set up by the interpreter to enclose the entire request. An exit of this context via a **stop** results in the execution of the default error handler in **errordict** called h**andleerror** which performs a **flushreply** to remove objects enqueued for transmission, followed by an **encodereply** and **sendreply** for an error object that describes the error to the client.

```
{  - request - } stopped { handleerror } if
```

6.1 Error Handling

The client's request can modify the default error handling scheme in several ways.

1. By substituting alternative procedures in **errordict** the request can change the way errors are reported.
   ```
   errordict /nefs_access { - substituted reporter - } put
   ```
2. By substituting an alternative **handleerror** procedure in **errordict** the request could return to the client information in addition to the error object.
   ```
   /handleerror { - new error handler - } def
   ```
3. A client request may enclose any sequence of operators within a **stopped** context for the purpose of catching and recovering from errors.
   ```
   { fh lookup getattr } stopped
         { $error /errname get /nefs_stale ne { stop } if null }
   if
   ```
   The previous example uses a **stopped** context to catch an error from either the **lookup** or the **getattr** operator. If it catches an error it checks for an **nefs_stale** error - if not then it invokes **stop** to proceed to whatever error handling is defined in the next enclosing context. Otherwise it pushes a **null** object onto the stack in lieu of a file attribute dict.

*Do we need warnings ? A warning could be raised if the operator succeeded (required results on the stack) but warning information needs to be communicated. A possible implementation would be to have warnings ignored by default. Warnings would have names in **errordict** but their associated value would be a null object that implies no action be taken. To register interest in warnings the client's request would merely replace the null object by a procedure to handle the error.*

Clients must take care in formulating requests that may leave state in the server's filesystem following an error abort. Client requests should be idempotent - a request should clean up any remaining state following an error.

6.2 The Error Object

The error object conveys error information back to the client. It is intended to be a concise description of an error. It comprises:

1. **Error Number**
   A well-defined range of numbers that can be used to index into a table on the client. This can be used instead of (2). A zero here means "look at the short string".
2. **Error Name**
   The error name. A short string of ASCII characters e.g. **typecheck**.
3. **Error Description**
   Optional. This can be used to expand on the error described in (1) and (2). For instance if the short string is **nefs_ioerr** then the error description could be "IDCAMS error #87".
4. **Location**
   An integer that identifies the offset of the failed operator in the clients request.

The client may choose to return extra context information following the error object.

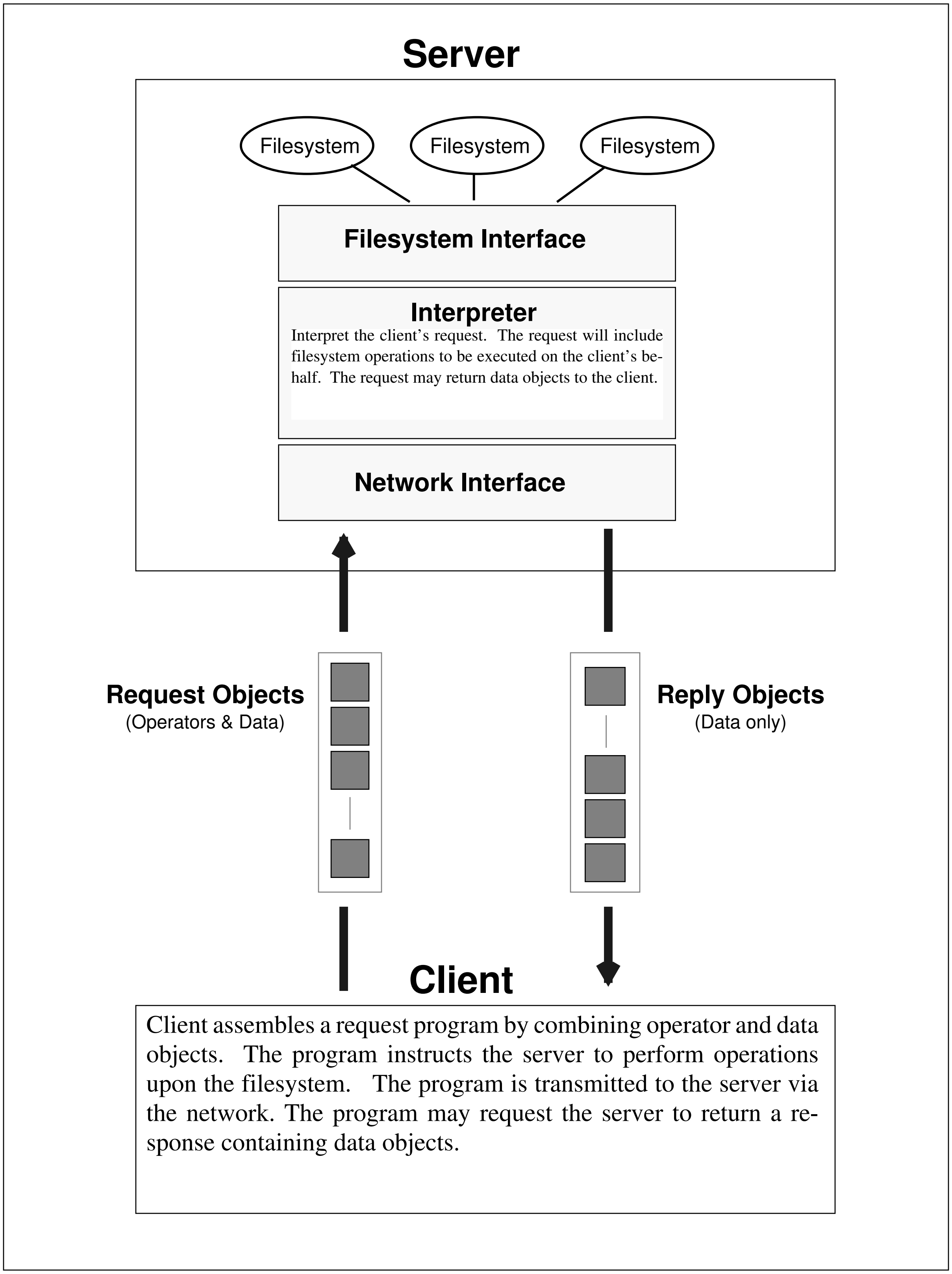
6.3 Interpreter Errors

dictfull	No more room in dictionary
dictstackoverflow	Too many begins
dictstackunderflow	Too many ends
execstackoverflow	Exec nesting too deep
invalidaccess	Attempt to violate access attribute
limitcheck	Implementation limit exceeded
nomem	Memory resources exhausted
rangecheck	Operand out of bounds
stackoverflow	Operand stack overflow
stackunderflow	Operand stack underflow
timeout	Time limit exceeded
typecheck	Operand of wrong type
undefined	Name not known
undefinedresult	Overflow, underflow or meaningless result
unmatchedmark	Expected mark not on stack
unregistered	Internal error

6.4 Filesystem Errors

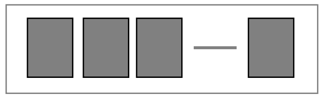
nefs_unknown	Unknown error. Use error string.
nefs_noent	No such file or directory. The filename specified does not exist.
nefs_io	I/O error. A hard error (e.g. disk error) occurred when the operation was in progress.
nefs_badtype	File type not supported. The file type is not supported by the server.
nefs_wrongtype	Operation inappropriate for this filetype.
nefs_notdir	Directory operation attempted on a non-directory.
nefs_access	Permission denied. The caller does not have the correct permission to perform the operation.
nefs_exist	File already exists.
nefs_badoffset	Illegal offset. The offset given for a file or directory operation does not make sense.
nefs_notdir	Not a directory. Directory operation attempted on a non-directory.
nefs_fbig	File too large. The operation caused the file to grow beyond the server's limit.
nefs_nospace	No space left on device. The operation caused the server's filesystem to reach its limit.
nefs_rofs	Read-only filesystem. Modification attempted to a read-only filesystem object.
nefs_nametoolong	The filename in an operation is too long.
nefs_notempty	Attempt to remove a directory that was not empty.
nefs_stale	Invalid file handle. The file handle cannot be mapped to a filesystem object.
nefs_badname	The filename is illegal on the server.
nefs_xfsop	Operation between filesystems is not supported
nefs_busy	Object is temporarily unavailable on the server.
nefs_warning	A warning.

Overview Diagram



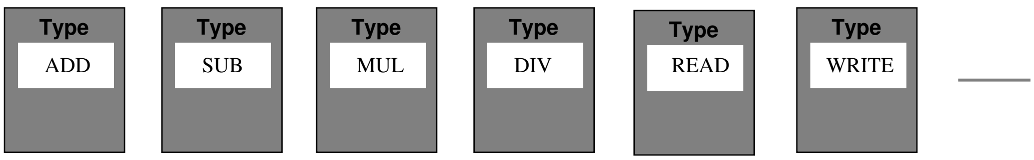
Object Diagram

A request or response comprises a finite sequence of objects.

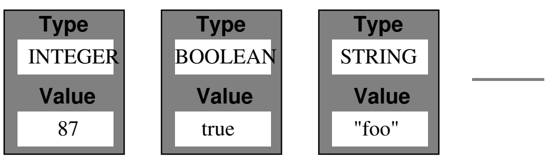


Objects may be classified as operators or data. An operator object represents an operation to be performed by the interpreter. An operator object has no value. A data object represents an item of data - an integer, a string of characters, or a boolean value.

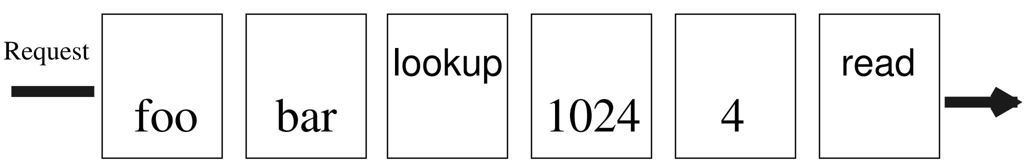
Operators



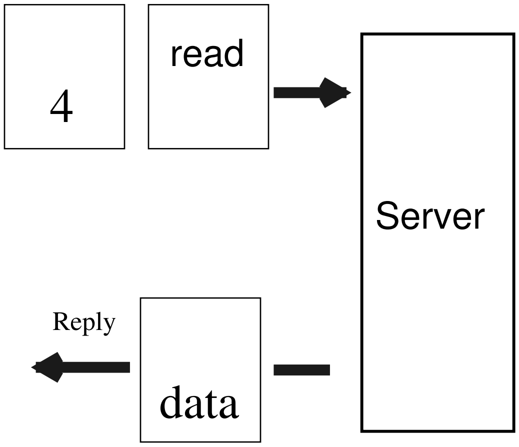
Values



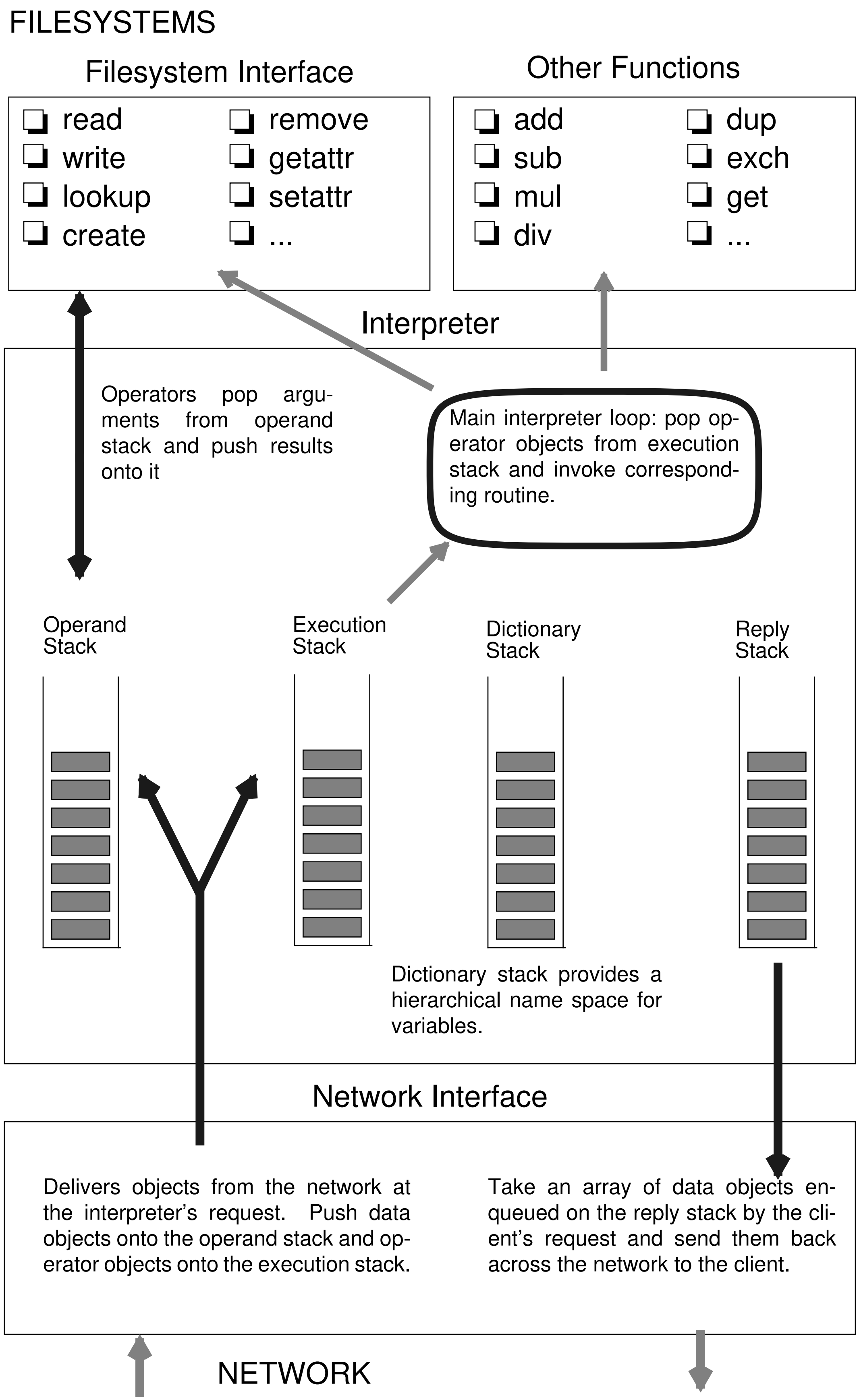
An example: lookup a file “bar” in directory “foo” and read 4 bytes from offset 1024.



Since the interpreter is stack-based the notation used here is *reverse polish form*. Reverse means that the arguments precede the operator that uses them. Value objects are pushed onto the interpreter’s operand stack. Operator objects are executed immediately and fetch their arguments from the operand stack.



Interpreter Diagram



Example: Determine disk usage in bytes

The following request enumerates all the files in filesystem hierarchy on the server and accumulates the total number of bytes that they occupy. The request begins by initializing the **total** to 0. Then a recursive procedure “descend” is defined. It takes a directory filehandle as an argument and enumerates all the entries in it with the readdir operator. The **readdir** operator takes three arguments from the operand stack: the filehandle of the directory, an offset into the directory (initially 0) and a procedure. For each entry in the directory **readdir** pushes the name of the entry and its offset in the directory and calls the procedure.

The procedure in this example first discards “.” and “..” entries since they represent the directory itself and its parent. The **lookup** operator converts the name into a filehandle and the **getattr** operator obtains its attributes as a dictionary object. If the type attribute indicates that the entry is a directory (type = 2) then **descend** is invoked recursively to accumulate the sizes of all the files in the subtree of which this directory is its root. Otherwise the entry is assumed to be a regular file and the **size** attribute is added to the running **total**.

The request as shown below is an ASCII representation of the request. A client would need to parse this request and convert it to a sequence of objects before transmitting it to the server. Text following a % is commentary and would be discarded. Tokens like **def**, **dup**, **exch**, **pop** would be converted to operator objects whereas tokens like **/total**, **0**, **(..)** would become value objects. String objects are delimited by parentheses. Curly braces delimit an executable array or procedure.

```
% du.ps
% NeFS request to recursively descend a directory hierarchy
% and return the total size of all the files contained therein.
%
/total 0 def                            % Initialize the total size

/descend {                              % Define a procedure called "descend"
        dup 0
        {
                exch pop                % Throw away the offset
                dup dup (.) ne exch (..) ne and {
                        1 index exch lookup dup getattr
                        dup /ftype get 2 eq
                                % It's a directory - descend into it
                                { pop descend }
                                % It's a file - add in its size
                                { /fbytes get /total exch total add def }
                        ifelse
                } if
                pop                     % Throw away the name
        }
        readdir
} def

@ descend                               % The "@" is the filehandle for the root directory

total 1 encodereply sendreply           % Send the final total back to the client
```

Example: Read a directory

Given the filehandle of a directory, use the **readdir** operator to enumerate all the entries in the directory. For each entry encode its name and its **fileno** attribute into the reply. When the end of the directory is reached send the reply back to the client.

```
% For each entry in a directory return its
% name and fileno attribute to the client
%
/dfh @ def

dfh 0                                   % filehandle and initial offset for readdir
{
        exch pop                        % discard offset
        dup dfh exch lookup getattr /fileno get
        2 encodereply                   % encode name & fileno for reply
}
readdir

sendreply       % send the reply
```

Example: Copy a File

Make a copy of file (foo) called (bar). Both files exist in the same directory dfh. The request starts by looking up the filehandle for the file to be copied and creates a filehandle for the copy. The **loop** operator executes a procedure that copies the file using 1K reads and writes. It maintains a running count of the number of bytes yet to be copied.

```
% Copy a file
%
dfh (foo) lookup /foofh exch def        % get filehandle for (foo)
dfh (bar) create /barfh exch def        % create filehandle for (bar)

/bytes foofh getattr /fsize get def     % get size of (foo) so we know how much to copy
/offset 0 def                           % initialize offset for (bar)
{
        /data foofh offset 1024 read def        % read up to 1K from (foo)
        barfh offset data write                 % write up to 1K to (bar)
        /bytes bytes 1024 sub def               % decrement byte count by 1024
        bytes 0 le { exit } if                  % if it's < 0 then we're done
        /offset offset 1024 add def             % increment offset by 1024
}
loop

barfh getattr 1 encodereply sendreply   % return the attributes of the new file to client.
```