NAME

intro – introduction to the Plan 9 File Protocol, 9P

SYNOPSIS

```
#include <fcall.h>
```

DESCRIPTION

A Plan 9 *server* is an agent that provides one or more hierarchical file systems — file trees — that may be accessed by Plan 9 processes. A server responds to requests by *clients* to navigate the hierarchy, and to create, remove, read, and write files. The prototypical server is a separate machine that stores large numbers of user files on permanent media; such a machine is called, somewhat confusingly, a *file server*. Another possibility for a server is to synthesize files on demand, perhaps based on information on data structures maintained in memory; the *plumber*(4) server is an example of such a server.

A *connection* to a server is a bidirectional communication path from the client to the server. There may be a single client or multiple clients sharing the same connection.

The *Plan 9 File Protocol*, 9P, is used for messages between *clients* and *servers*. A client transmits *requests* (*T-messages*) to a server, which subsequently returns *replies* (*R-messages*) to the client. The combined acts of transmitting (receiving) a request of a particular type, and receiving (transmitting) its reply is called a *transaction* of that type.

Each message consists of a sequence of bytes. Two-, four-, and eight-byte fields hold unsigned integers represented in little-endian order (least significant byte first). Data items of larger or variable lengths are represented by a two-byte field specifying a count, *n*, followed by *n* bytes of data. Text strings are represented this way, with the text itself stored as a UTF-8 encoded sequence of Unicode characters (see *utf*(7)). Text strings in 9P messages are not NUL-terminated: *n* counts the bytes of UTF-8 data, which include no final zero byte. The NUL character is illegal in all text strings in 9P, and is therefore excluded from file names, user names, and so on.

Each 9P message begins with a four-byte size field specifying the length in bytes of the complete message including the four bytes of the size field itself. The next byte is the message type, one of the constants in the enumeration in the include file `<fcall.h>`. The next two bytes are an identifying *tag*, described below. The remaining bytes are parameters of different sizes. In the message descriptions, the number of bytes in a field is given in brackets after the field name. The notation *parameter*[*n*] where *n* is not a constant represents a variable-length parameter: *n*[2] followed by *n* bytes of data forming the *parameter*. The notation *string*[*s*] (using a literal *s* character) is shorthand for *s*[2] followed by *s* bytes of UTF-8 text. (Systems may choose to reduce the set of legal characters to reduce syntactic problems, for example to remove slashes from name components, but the protocol has no such restriction. Plan 9 names may contain any printable character (that is, any character outside hexadecimal 00-1F and 80-9F) except slash.) Messages are transported in byte form to allow for machine independence; *fcall*(3) describes routines that convert to and from this form into a machine-dependent C structure.

MESSAGES

size[4] `Tversion` *tag*[2] *msize*[4] *version*[*s*]
size[4] `Rversion` *tag*[2] *msize*[4] *version*[*s*]

size[4] `Tauth` *tag*[2] *afid*[4] *uname*[*s*] *aname*[*s*]
size[4] `Rauth` *tag*[2] *aqid*[13]

size[4] `Rerror` *tag*[2] *ename*[*s*]

size[4] `Tflush` *tag*[2] *oldtag*[2]
size[4] `Rflush` *tag*[2]

size[4] `Tattach` *tag*[2] *fid*[4] *afid*[4] *uname*[*s*] *aname*[*s*]
size[4] `Rattach` *tag*[2] *qid*[13]

size[4] `Twalk` *tag*[2] *fid*[4] *newfid*[4] *nwname*[2] *nwname**(*wname*[*s*])
size[4] `Rwalk` *tag*[2] *nwqid*[2] *nwqid**(*wqid*[13])

size[4] `Topen` *tag*[2] *fid*[4] *mode*[1]
size[4] `Ropen` *tag*[2] *qid*[13] *iounit*[4]

size[4] `Topenfd` *tag*[2] *fid*[4] *mode*[1]
size[4] `Ropenfd` *tag*[2] *qid*[13] *iounit*[4] *unixfd*[4]

size[4] `Tcreate` *tag*[2] *fid*[4] *name*[*s*] *perm*[4] *mode*[1]
size[4] `Rcreate` *tag*[2] *qid*[13] *iounit*[4]

size[4] `Tread` *tag*[2] *fid*[4] *offset*[8] *count*[4]
size[4] `Rread` *tag*[2] *count*[4] *data*[*count*]

size[4] `Twrite` *tag*[2] *fid*[4] *offset*[8] *count*[4] *data*[*count*]
size[4] `Rwrite` *tag*[2] *count*[4]

size[4] `Tclunk` *tag*[2] *fid*[4]
size[4] `Rclunk` *tag*[2]

size[4] `Tremove` *tag*[2] *fid*[4]
size[4] `Rremove` *tag*[2]

size[4] `Tstat` *tag*[2] *fid*[4]
size[4] `Rstat` *tag*[2] *stat*[*n*]

size[4] `Twstat` *tag*[2] *fid*[4] *stat*[*n*]
size[4] `Rwstat` *tag*[2]

Each T-message has a *tag* field, chosen and used by the client to identify the message. The reply to the message will have the same tag. Clients must arrange that no two outstanding messages on the same connection have the same tag. An exception is the tag `NOTAG`, defined as `(ushort)~0` in `<fcall.h>`: the client can use it, when establishing a connection, to override tag matching in `version` messages.

The type of an R-message will either be one greater than the type of the corresponding T-message or `Rerror`, indicating that the request failed. In the latter case, the *ename* field contains a string describing the reason for failure.

The `version` message identifies the version of the protocol and indicates the maximum message size the system is prepared to handle. It also initializes the connection and aborts all outstanding I/O on the connection. The set of messages between `version` requests is called a *session*.

Most T-messages contain a *fid*, a 32-bit unsigned integer that the client uses to identify a "current file" on the server. Fids are somewhat like file descriptors in a user process, but they are not restricted to files open for I/O: directories being examined, files being accessed by *stat*(3) calls, and so on — all files being manipulated by the operating system — are identified by fids. Fids are chosen by the client. All requests on a connection share the same fid space; when several clients share a connection, the agent managing the sharing must arrange that no two clients choose the same fid.

The fid supplied in an `attach` message will be taken by the server to refer to the root of the served file tree. The `attach` identifies the user to the server and may specify a particular file tree served by the server (for those that supply more than one).

Permission to attach to the service is proven by providing a special fid, called `afid`, in the `attach` message. This `afid` is established by exchanging `auth` messages and subsequently manipulated using `read` and `write` messages to exchange authentication information not defined explicitly by 9P. Once the authentication protocol is complete, the `afid` is presented in the `attach` to permit the user to access the service.

A `walk` message causes the server to change the current file associated with a fid to be a file in the directory that is the old current file, or one of its subdirectories. `Walk` returns a new fid that refers to the resulting file. Usually, a client maintains a fid for the root, and navigates by `walk`s from the root fid.

A client can send multiple T-messages without waiting for the corresponding R-messages, but all outstanding T-messages must specify different tags. The server may delay the response to a request and respond to later ones; this is sometimes necessary, for example when the client reads from a file that the server synthesizes from external events such as keyboard characters.

Replies (R-messages) to `auth`, `attach`, `walk`, `open`, and `create` requests convey a *qid* field back to the client. The qid represents the server's unique identification for the file being accessed: two files on the same server hierarchy are the same if and only if their qids are the same. (The client may have multiple fids pointing to a single file on a server and hence having a single qid.) The thirteen-byte qid fields hold a one-byte type, specifying whether the file is a directory, append-only file, etc., and two unsigned integers: first the four-byte qid *version*, then the eight-byte qid *path*. The path is an integer unique among all files in the hierarchy. If a file is deleted and recreated with the same name in the same directory, the old and new path components of the qids should be different. The version is a version number for a file; typically, it is incremented every time the file is modified.

An existing file can be opened, or a new file may be created in the current (directory) file. I/O of a given number of bytes at a given offset on an open file is done by read and write.

A client should clunk any fid that is no longer needed. The remove transaction deletes files.

Openfd is an extension used by Unix utilities to allow traditional Unix programs to have their input or output attached to fids on 9P servers. See *openfd*(9p) and *9pclient*(3) for details.

The stat transaction retrieves information about the file. The *stat* field in the reply includes the file's name, access permissions (read, write and execute for owner, group and public), access and modification times, and owner and group identifications (see *stat*(3)). The owner and group identifications are textual names. The wstat transaction allows some of a file's properties to be changed.

A request can be aborted with a flush request. When a server receives a Tflush, it should not reply to the message with tag *oldtag* (unless it has already replied), and it should immediately send an Rflush. The client must wait until it gets the Rflush (even if the reply to the original message arrives in the interim), at which point *oldtag* may be reused.

Because the message size is negotiable and some elements of the protocol are variable length, it is possible (although unlikely) to have a situation where a valid message is too large to fit within the negotiated size. For example, a very long file name may cause a Rstat of the file or Rread of its directory entry to be too large to send. In most such cases, the server should generate an error rather than modify the data to fit, such as by truncating the file name. The exception is that a long error string in an Rerror message should be truncated if necessary, since the string is only advisory and in some sense arbitrary.

Most programs do not see the 9P protocol directly; on Plan 9, calls to library routines that access files are translated by the kernel's mount driver into 9P messages.

Unix

On Unix, 9P services are posted as Unix domain sockets in a well-known directory (see *getns*(3) and *9pserve*(4)). Clients connect to these servers using a 9P client library (see *9pclient*(3)).

DIRECTORIES

Directories are created by create with DMDIR set in the permissions argument (see *stat*(9P)). The members of a directory can be found with *read*(9P). All directories must support walks to the directory .. (dot-dot) meaning parent directory, although by convention directories contain no explicit entry for .. or . (dot). The parent of the root directory of a server's tree is itself.

ACCESS PERMISSIONS

This section describes the access permission conventions implemented by most Plan 9 file servers. These conventions are not enforced by the protocol and may differ between servers, especially servers built on top of foreign operating systems.

Each file server maintains a set of user and group names. Each user can be a member of any number of groups. Each group has a *group leader* who has special privileges (see *stat*(9P) and Plan 9's *users*(6)). Every file request has an implicit user id (copied from the original attach) and an implicit set of groups (every group of which the user is a member).

Each file has an associated *owner* and *group* id and three sets of permissions: those of the owner, those of the group, and those of "other" users. When the owner attempts to do something to a file, the owner, group, and other permissions are consulted, and if any of them grant the requested permission, the operation is allowed. For someone who is not the owner, but is a member of the file's group, the group and other permissions are consulted. For everyone else, the other permissions are used. Each set of permissions says whether reading is allowed, whether writing is allowed, and whether executing is allowed. A walk in a directory is regarded as executing the directory, not reading it. Permissions are kept in the low-order bits of the file *mode*: owner read/write/execute permission represented as 1 in bits 8, 7, and 6 respectively (using 0 to number the low order). The group permissions are in bits 5, 4, and 3, and the other permissions are in bits 2, 1, and 0.

The file *mode* contains some additional attributes besides the permissions. If bit 31 (DMDIR) is set, the file is a directory; if bit 30 (DMAPPEND) is set, the file is append-only (offset is ignored in writes); if bit 29 (DMEXCL) is set, the file is exclusive-use (only one client may have it open at a time); if bit 27 (DMAUTH) is set, the file is an authentication file established by auth messages; if bit 26 (DMTMP) is set, the contents of the file (or directory) are not included in nightly archives. (Bit 28 is skipped for historical reasons.) These bits are reproduced, from the top bit down, in the type byte of the Qid: QTDIR, QTAPPEND, QTEXCL, (skipping one bit) QTAUTH, and QTTMP. The name QTFILE, defined to be zero, identifies the value of the type for a plain file.

NAME

attach, auth – messages to establish a connection

SYNOPSIS

size[4] `Tauth` *tag*[2] *afid*[4] *uname*[*s*] *aname*[*s*]
size[4] `Rauth` *tag*[2] *aqid*[13]

size[4] `Tattach` *tag*[2] *fid*[4] *afid*[4] *uname*[*s*] *aname*[*s*]
size[4] `Rattach` *tag*[2] *qid*[13]

DESCRIPTION

The `attach` message serves as a fresh introduction from a user on the client machine to the server. The message identifies the user (*uname*) and may select the file tree to access (*aname*). The *afid* argument specifies a fid previously established by an `auth` message, as described below.

As a result of the `attach` transaction, the client will have a connection to the root directory of the desired file tree, represented by *fid*. An error is returned if *fid* is already in use. The server's idea of the root of the file tree is represented by the returned *qid*.

If the client does not wish to authenticate the connection, or knows that authentication is not required, the *afid* field in the `attach` message should be set to `NOFID`, defined as `(u32int)~0` in `<fcall.h>`. If the client does wish to authenticate, it must acquire and validate an *afid* using an `auth` message before doing the `attach`.

The `auth` message contains *afid*, a new fid to be established for authentication, and the *uname* and *aname* that will be those of the following `attach` message. If the server does not require authentication, it returns `Rerror` to the `Tauth` message.

If the server does require authentication, it returns *aqid* defining a file of type `QTAUTH` (see *intro*(9P)) that may be read and written (using `read` and `write` messages in the usual way) to execute an authentication protocol. That protocol's definition is not part of 9P itself.

Once the protocol is complete, the same *afid* is presented in the `attach` message for the user, granting entry. The same validated *afid* may be used for multiple `attach` messages with the same *uname* and *aname*.

ENTRY POINTS

Fsmount and *fsauth* (see *9pclient*(3)) generate `attach` and `auth` transactions.

SEE ALSO

9pclient(3), *version*(9P), Plan 9's *authsrv*(6)

NAME

clunk – forget about a fid

SYNOPSIS

size[4] `Tclunk` *tag*[2] *fid*[4]
size[4] `Rclunk` *tag*[2]

DESCRIPTION

The `clunk` request informs the file server that the current file represented by *fid* is no longer needed by the client. The actual file is not removed on the server unless the fid had been opened with `ORCLOSE`.

Once a fid has been clunked, the same fid can be reused in a new `walk` or `attach` request.

Even if the `clunk` returns an error, the *fid* is no longer valid.

ENTRY POINTS

`Clunk` transactions are generated by *fsclose* and *fsunmount* (see *9pclient*(3)) and indirectly by other actions such as failed *fsopen* calls.

NAME

error – return an error

SYNOPSIS

size[4] `Rerror` *tag*[2] *ename*[*s*]

DESCRIPTION

The `Rerror` message (there is no `Terror`) is used to return an error string describing the failure of a transaction. It replaces the corresponding reply message that would accompany a successful call; its tag is that of the failing request.

By convention, clients may truncate error messages after `ERRMAX`−1 bytes; `ERRMAX` is defined in <`libc.h`>.

NAME

flush – abort a message

SYNOPSIS

size[4] `Tflush` *tag*[2] *oldtag*[2]
size[4] `Rflush` *tag*[2]

DESCRIPTION

When the response to a request is no longer needed, such as when a user interrupts a process doing a *read*(9p), a `Tflush` request is sent to the server to purge the pending response. The message being flushed is identified by *oldtag*. The semantics of `flush` depends on messages arriving in order.

The server should answer the `flush` message immediately. If it recognizes *oldtag* as the tag of a pending transaction, it should abort any pending response and discard that tag. In either case, it should respond with an `Rflush` echoing the *tag* (not *oldtag*) of the `Tflush` message. A `Tflush` can never be responded to by an `Rerror` message.

The server may respond to the pending request before responding to the `Tflush`. It is possible for a client to send multiple `Tflush` messages for a particular pending request. Each subsequent `Tflush` must contain as *oldtag* the tag of the pending request (not a previous `Tflush`). Should multiple `Tflush`es be received for a pending request, they must be answered in order. A `Rflush` for any of the multiple `Tflush`es implies an answer for all previous ones. Therefore, should a server receive a request and then multiple flushes for that request, it need respond only to the last flush.

When the client sends a `Tflush`, it must wait to receive the corresponding `Rflush` before reusing *oldtag* for subsequent messages. If a response to the flushed request is received before the `Rflush`, the client must honor the response as if it had not been flushed, since the completed request may signify a state change in the server. For instance, `Tcreate` may have created a file and `Twalk` may have allocated a fid. If no response is received before the `Rflush`, the flushed transaction is considered to have been canceled, and should be treated as though it had never been sent.

Several exceptional conditions are handled correctly by the above specification: sending multiple flushes for a single tag, flushing after a transaction is completed, flushing a `Tflush`, and flushing an invalid tag.

ENTRY POINTS

The *9pclient*(3) library does not generate `flush` transactions.. *9pserve*(4) generates `flush` transactions to cancel transactions pending when a client hangs up.

NAME

open, create – prepare a fid for I/O on an existing or new file

SYNOPSIS

size[4] Topen *tag*[2] *fid*[4] *mode*[1]
size[4] Ropen *tag*[2] *qid*[13] *iounit*[4]

size[4] Tcreate *tag*[2] *fid*[4] *name*[*s*] *perm*[4] *mode*[1]
size[4] Rcreate *tag*[2] *qid*[13] *iounit*[4]

DESCRIPTION

The open request asks the file server to check permissions and prepare a fid for I/O with subsequent read and write messages. The *mode* field determines the type of I/O: 0 (called OREAD in <libc.h>), 1 (OWRITE), 2 (ORDWR), and 3 (OEXEC) mean *read access, write access, read and write access,* and *execute access,* to be checked against the permissions for the file. In addition, if *mode* has the OTRUNC (0x10) bit set, the file is to be truncated, which requires write permission (if the file is append-only, and permission is granted, the open succeeds but the file will not be truncated); if the *mode* has the ORCLOSE (0x40) bit set, the file is to be removed when the fid is clunked, which requires permission to remove the file from its directory. All other bits in *mode* should be zero. It is illegal to write a directory, truncate it, or attempt to remove it on close. If the file is marked for exclusive use (see *stat*(9P)), only one client can have the file open at any time. That is, after such a file has been opened, further opens will fail until *fid* has been clunked. All these permissions are checked at the time of the open request; subsequent changes to the permissions of files do not affect the ability to read, write, or remove an open file.

The create request asks the file server to create a new file with the *name* supplied, in the directory (*dir*) represented by *fid*, and requires write permission in the directory. The owner of the file is the implied user id of the request, the group of the file is the same as *dir*, and the permissions are the value of

```
perm & (~0666 | (dir.perm & 0666))
```

if a regular file is being created and

```
perm & (~0777 | (dir.perm & 0777))
```

if a directory is being created. This means, for example, that if the create allows read permission to others, but the containing directory does not, then the created file will not allow others to read the file.

Finally, the newly created file is opened according to *mode*, and *fid* will represent the newly opened file. *Mode* is not checked against the permissions in *perm*. The *qid* for the new file is returned with the create reply message.

Directories are created by setting the DMDIR bit (0x80000000) in the *perm*.

The names . and .. are special; it is illegal to create files with these names.

It is an error for either of these messages if the fid is already the product of a successful open or create message.

An attempt to create a file in a directory where the given *name* already exists will be rejected; in this case, the *fscreate* call (see *9pclient*(3)) uses open with truncation. The algorithm used by the *create* system call is: first walk to the directory to contain the file. If that fails, return an error. Next walk to the specified file. If the walk succeeds, send a request to open and truncate the file and return the result, successful or not. If the walk fails, send a create message. If that fails, it may be because the file was created by another process after the previous walk failed, so (once) try the walk and open again.

ENTRY POINTS

Fsopen and *fscreate* (see *9pclient*(3)) both generate open messages; only *fscreate* generates a create message. The iounit associated with an open file may be discovered by calling *fsiounit*.

For programs that need atomic file creation, without the race that exists in the open–create sequence described above, *fscreate* does the following. If the OEXCL (0x1000) bit is set in the *mode* for a *fscreate* call, the open message is not sent; the kernel issues only the create. Thus, if the file exists, *fscreate* will draw an error, but if it doesn't and the *fscreate* call succeeds, the process issuing the *fscreate* is guaranteed to be the one that created the file.

NAME

openfd – prepare a fid for I/O using a file descriptor

SYNOPSIS

size[4] `Topenfd` *tag*[2] *fid*[4] *mode*[1]
size[4] `Ropenfd` *tag*[2] *qid*[13] *iounit*[4] *unixfd*[4]

DESCRIPTION

The *openfd* request behaves like open, except that it prepares and returns a Unix file descriptor corresponding to the opened fid.

After a successful *open* transaction, *fid* is considered by the client to have been clunked and can be reused.

The returned Unix file descriptor is one end of a Unix pipe. A proxy process at the other end transfers data between the pipe and the 9P server. Because it is a pipe, errors on reads and writes are discarded and *mode* must be `OREAD` or `OWRITE`; it cannot be `ORDWR`.

Openfd is implemented by *9pserve*(4). 9P servers that post their services using *9pserve*(4) (or indirectly via *post9pservice*(3)) will never see a `Topenfd` message.

ENTRY POINTS

Fsopenfd (see *9pclient*(3)) generates an `openfd` message.

NAME

read, write – transfer data from and to a file

SYNOPSIS

size[4] `Tread` *tag*[2] *fid*[4] *offset*[8] *count*[4]
size[4] `Rread` *tag*[2] *count*[4] *data*[*count*]

size[4] `Twrite` *tag*[2] *fid*[4] *offset*[8] *count*[4] *data*[*count*]
size[4] `Rwrite` *tag*[2] *count*[4]

DESCRIPTION

The `read` request asks for *count* bytes of data from the file identified by *fid*, which must be opened for reading, starting *offset* bytes after the beginning of the file. The bytes are returned with the `read` reply message.

The *count* field in the reply indicates the number of bytes returned. This may be less than the requested amount. If the *offset* field is greater than or equal to the number of bytes in the file, a count of zero will be returned.

For directories, `read` returns an integral number of directory entries exactly as in `stat` (see *stat*(9P)), one for each member of the directory. The `read` request message must have `offset` equal to zero or the value of `offset` in the previous `read` on the directory, plus the number of bytes returned in the previous `read`. In other words, seeking other than to the beginning is illegal in a directory.

The `write` request asks that *count* bytes of data be recorded in the file identified by *fid*, which must be opened for writing, starting *offset* bytes after the beginning of the file. If the file is append-only, the data will be placed at the end of the file regardless of *offset*. Directories may not be written.

The `write` reply records the number of bytes actually written. It is usually an error if this is not the same as requested.

Because 9P implementations may limit the size of individual messages, more than one message may be produced by a single *read* or *write* call. The *iounit* field returned by *open*(9P), if non-zero, reports the maximum size that is guaranteed to be transferred atomically.

ENTRY POINTS

Fsread and *fswrite* (see *9pclient*(3)) generate the corresponding messages. Because they take an offset parameter, the *fspread* and *fspwrite* calls correspond more directly to the 9P messages. Although *fsseek* affects the offset, it does not generate a message.

NAME

remove – remove a file from a server

SYNOPSIS

size[4] `Tremove` *tag*[2] *fid*[4]
size[4] `Rremove` *tag*[2]

DESCRIPTION

The `remove` request asks the file server both to remove the file represented by *fid* and to `clunk` the *fid*, even if the remove fails. This request will fail if the client does not have write permission in the parent directory.

It is correct to consider `remove` to be a `clunk` with the side effect of removing the file if permissions allow.

If a file has been opened as multiple fids, possibly on different connections, and one fid is used to remove the file, whether the other fids continue to provide access to the file is implementation-defined. The Plan 9 file servers remove the file immediately: attempts to use the other fids will yield a "phase error." *U9fs* follows the semantics of the underlying Unix file system, so other fids typically remain usable.

ENTRY POINTS

Fsremove (see *9pclient*(3)) generates `remove` messages.

NAME

stat, wstat – inquire or change file attributes

SYNOPSIS

size[4] `Tstat` *tag*[2] *fid*[4]
size[4] `Rstat` *tag*[2] *stat*[*n*]

size[4] `Twstat` *tag*[2] *fid*[4] *stat*[*n*]
size[4] `Rwstat` *tag*[2]

DESCRIPTION

The `stat` transaction inquires about the file identified by *fid*. The reply will contain a machine-independent *directory entry*, *stat*, laid out as follows:

size[2] total byte count of the following data

type[2] for kernel use

dev[4] for kernel use

qid.type[1]
the type of the file (directory, etc.), represented as a bit vector corresponding to the high 8 bits of the file's mode word.

qid.vers[4]
version number for given path

qid.path[8]
the file server's unique identification for the file

mode[4]
permissions and flags

atime[4]
last access time

mtime[4]
last modification time

length[8]
length of file in bytes

name[s]
file name; must be / if the file is the root directory of the server

uid[s] owner name

gid[s] group name

muid[s]
name of the user who last modified the file

Integers in this encoding are in little-endian order (least significant byte first). The *convM2D* and *convD2M* routines (see *fcall*(3)) convert between directory entries and a C structure called a `Dir`.

The *mode* contains permission bits as described in *intro*(9P) and the following: 0x80000000 (`DMDIR`, this file is a directory), 0x40000000 (`DMAPPEND`, append only), 0x20000000 (`DMEXCL`, exclusive use), 0x04000000 (`DMTMP`, temporary); these are echoed in `Qid.type`. Writes to append-only files always place their data at the end of the file; the *offset* in the `write` message is ignored, as is the `OTRUNC` bit in an open. Exclusive use files may be open for I/O by only one fid at a time across all clients of the server. If a second open is attempted, it draws an error. Servers may implement a timeout on the lock on an exclusive use file: if the fid holding the file open has been unused for an extended period (of order at least minutes), it is reasonable to break the lock and deny the initial fid further I/O. Temporary files are not included in nightly archives (see Plan 9's *fossil*(4)).

The two time fields are measured in seconds since the epoch (Jan 1 00:00 1970 GMT). The *mtime* field reflects the time of the last change of content (except when later changed by `wstat`). For a plain file, *mtime* is the time of the most recent `create`, open with truncation, or `write`; for a directory it is the time of the most recent `remove`, `create`, or `wstat` of a file in the directory. Similarly, the *atime* field records the last `read` of the contents; also it is set whenever *mtime* is set. In addition, for a directory, it is set by an `attach`,

`walk`, or `create`, all whether successful or not.

The *muid* field names the user whose actions most recently changed the *mtime* of the file.

The *length* records the number of bytes in the file. Directories and most files representing devices have a conventional length of 0.

The `stat` request requires no special permissions.

The `wstat` request can change some of the file status information. The *name* can be changed by anyone with write permission in the parent directory; it is an error to change the name to that of an existing file. The *length* can be changed (affecting the actual length of the file) by anyone with write permission on the file. It is an error to attempt to set the length of a directory to a non-zero value, and servers may decide to reject length changes for other reasons. The *mode* and *mtime* can be changed by the owner of the file or the group leader of the file's current group. The directory bit cannot be changed by a `wstat`; the other defined permission and mode bits can. The *gid* can be changed: by the owner if also a member of the new group; or by the group leader of the file's current group if also leader of the new group (see *intro*(9P) for more information about permissions, users, and groups). None of the other data can be altered by a `wstat` and attempts to change them will trigger an error. In particular, it is illegal to attempt to change the owner of a file. (These conditions may be relaxed when establishing the initial state of a file server; see Plan 9's *fsconfig*(8).)

Either all the changes in `wstat` request happen, or none of them does: if the request succeeds, all changes were made; if it fails, none were.

A `wstat` request can avoid modifying some properties of the file by providing explicit "don't touch" values in the `stat` data that is sent: zero-length strings for text values and the maximum unsigned value of appropriate size for integral values. As a special case, if *all* the elements of the directory entry in a `Twstat` message are "don't touch" values, the server may interpret it as a request to guarantee that the contents of the associated file are committed to stable storage before the `Rwstat` message is returned. (Consider the message to mean, "make the state of the file exactly what it claims to be.")

A *read* of a directory yields an integral number of directory entries in the machine independent encoding given above (see *read*(9P)).

Note that since the `stat` information is sent as a 9P variable-length datum, it is limited to a maximum of 65535 bytes.

ENTRY POINTS

`Stat` messages are generated by *fsdirfstat* and *fsdirstat* (see *9pclient*(3)).

`Wstat` messages are generated by *fsdirfwstat* and *fsdirwstat*.

BUGS

To make the contents of a directory, such as returned by *read*(9P), easy to parse, each directory entry begins with a size field. For consistency, the entries in `Twstat` and `Rstat` messages also contain their size, which means the size appears twice. For example, the `Rstat` message is formatted as "(4+1+2+2+*n*)[4] `Rstat` *tag*[2] *n*[2] (*n*-2)[2] *type*[2] *dev*[4]...," where *n* is the value returned by `convD2M`.

NAME

version – negotiate protocol version

SYNOPSIS

size[4] `Tversion` *tag*[2] *msize*[4] *version*[*s*]
size[4] `Rversion` *tag*[2] *msize*[4] *version*[*s*]

DESCRIPTION

The `version` request negotiates the protocol version and message size to be used on the connection and initializes the connection for I/O. `Tversion` must be the first message sent on the 9P connection, and the client cannot issue any further requests until it has received the `Rversion` reply. The *tag* should be `NOTAG` (value `(ushort)~0`) for a `version` message.

The client suggests a maximum message size, `msize`, that is the maximum length, in bytes, it will ever generate or expect to receive in a single 9P message. This count includes all 9P protocol data, starting from the `size` field and extending through the message, but excludes enveloping transport protocols. The server responds with its own maximum, `msize`, which must be less than or equal to the client's value. Thenceforth, both sides of the connection must honor this limit.

The `version` string identifies the level of the protocol. The string must always begin with the two characters "`9P`". If the server does not understand the client's version string, it should respond with an `Rversion` message (not `Rerror`) with the `version` string the 7 characters "`unknown`".

The server may respond with the client's version string, or a version string identifying an earlier defined protocol version. Currently, the only defined version is the 6 characters "`9P2000`". Version strings are defined such that, if the client string contains one or more period characters, the initial substring up to but not including any single period in the version string defines a version of the protocol. After stripping any such period-separated suffix, the server is allowed to respond with a string of the form `9P`*nnnn*, where *nnnn* is less than or equal to the digits sent by the client.

The client and server will use the protocol version defined by the server's response for all subsequent communication on the connection.

A successful `version` request initializes the connection. All outstanding I/O on the connection is aborted; all active fids are freed ('clunked') automatically. The set of messages between `version` requests is called a *session*.

ENTRY POINTS

Fsversion (see *9pclient*(3)) generates `version` messages; it is called automatically by *fsmount*.

NAME

walk – descend a directory hierarchy

SYNOPSIS

size[4] `Twalk` *tag*[2] *fid*[4] *newfid*[4] *nwname*[2] *nwname**(*wname*[*s*])
size[4] `Rwalk` *tag*[2] *nwqid*[2] *nwqid**(*qid*[13])

DESCRIPTION

The `walk` request carries as arguments an existing *fid* and a proposed *newfid* (which must not be in use unless it is the same as *fid*) that the client wishes to associate with the result of traversing the directory hierarchy by 'walking' the hierarchy using the successive path name elements `wname`. The *fid* must represent a directory unless zero path name elements are specified.

The *fid* must be valid in the current session and must not have been opened for I/O by an `open` or `create` message. If the full sequence of `nwname` elements is walked successfully, *newfid* will represent the file that results. If not, *newfid* (and `fid`) will be unaffected. However, if *newfid* is in use or otherwise illegal, an `Rerror` is returned.

The name "`..`" (dot-dot) represents the parent directory. The name "`.`" (dot), meaning the current directory, is not used in the protocol.

It is legal for `nwname` to be zero, in which case *newfid* will represent the same file as *fid* and the `walk` will usually succeed; this is equivalent to walking to dot. The rest of this discussion assumes `nwname` is greater than zero.

The `nwname` path name elements `wname` are walked in order, "elementwise". For the first elementwise walk to succeed, the file identified by *fid* must be a directory, and the implied user of the request must have permission to search the directory (see *intro*(9P)). Subsequent elementwise walks have equivalent restrictions applied to the implicit fid that results from the preceding elementwise walk.

If the first element cannot be walked for any reason, `Rerror` is returned. Otherwise, the walk will return an `Rwalk` message containing *nwqid* qids corresponding, in order, to the files that are visited by the *nwqid* successful elementwise walks; *nwqid* is therefore either `nwname` or the index of the first elementwise walk that failed. The value of *nwqid* cannot be zero unless `nwname` is zero. Also, *nwqid* will always be less than or equal to `nwname`. Only if it is equal, however, will *newfid* be affected, in which case *newfid* will represent the file reached by the final elementwise walk requested in the message.

A `walk` of the name "`..`" in the root directory of a server is equivalent to a walk with no name elements.

If *newfid* is the same as *fid*, the above discussion applies, with the obvious difference that if the walk changes the state of *newfid*, it also changes the state of *fid*; and if *newfid* is unaffected, then *fid* is also unaffected.

To simplify the implementation of the servers, a maximum of sixteen name elements or qids may be packed in a single message. This constant is called `MAXWELEM` in *fcall*(3). Despite this restriction, the system imposes no limit on the number of elements in a file name, only the number that may be transmitted in a single message.

ENTRY POINTS

Fswalk (see *9pclient*(3)) generates walk messages. One or more walk messages may be generated by any call that evaluates file names: *fsopen*, *fsopenfd*, *fsdirstat*, *fsdirwstat*.