# The Sprite RPC Interface

## 1. Introduction

This is a brief summary of the network interface to Sprite. This is done via a description of each RPC used in the implementation. Most of the calls have to do with the file system, although there are a few related to process migration, signals, and remote waiting. The parameters of each RPC are specified, and a few words about the use of the RPC are given. This is only a terse reference guide; there is not necessarily justification or explanation included in the descriptions.

It is important to understand that this RPC interface is between two operating system kernels. The terms ‘‘client’’ and ‘‘server’’ refer to instances of the kernel acting in these roles. The term ‘‘application process’’ will be used when necessary to refer to the process triggering the use of an RPC.

### 1.1. The RPC Protocol

It is useful to review the RPC protocol itself. The main thing to understand is that the information included in each RPC request and reply is broken into three parts, a standard RPC header, a parameter area, and a data area. The actual size of the parameter and data areas are extracted from the packet header and reported to the stub procedures. The ID of the client is also reported to the server-side stub procedures. The stubs have to deal with the distinct parameter and data areas. The parameter area is restricted to contain only integers, while the data area can contain arbitrary data. The low-levels of the RPC protocol handle communication between hosts of different byte order by automatically byteswapping the RPC header and the parameter area when a packet is received from a host with a different byte order. This means that both the client and server stubs can treat the parameter area as a C structure (of integers), and not worry about alignment problems. If the data area is used for other than file data or strings, then the stub, or even a higher-level procedure, has to do its own byteswapping.

Lastly, there is a return code from each RPC, with SUCCESS (zero) meaning successful completion. Sprite return codes are defined in the file ‘‘/sprite/lib/include/status.h’’.

The complete set of RPCs used in the Sprite implementation is given in Table 1. The number associated with the RPC is the procedure number used in the RPC packet header. The RPCs are described in more detail below.

### 1.2. Source Code References

The RPC parameters described below are given as C typedefs. These have been extracted from the C source code of the Sprite implementation. It may help to consult the code in order to fully understand the use of a particular RPC. Most of the RPC stubs are organized so the client and server stub for a given RPC are together along with the

| Sprite Remote Procedure Calls | | |
|---|---|---|
| Procedure | # | Description |
| ECHO_1 | 1 | Echo. Performed by server's interrupt handler (unused). |
| ECHO_2 | 2 | Echo. Performed by Rpc_Server process. |
| SEND | 3 | Send. Like Echo, but data only transferred to server. |
| RECEIVE | 4 | Receive. Data only transferred back to client. |
| GETTIME | 5 | Broadcast RPC to get the current time. |
| FS_PREFIX | 6 | Broadcast RPC to find prefix server. |
| FS_OPEN | 7 | Open a file system object by name. |
| FS_READ | 8 | Read data from a file system object. |
| FS_WRITE | 9 | Write data to a file system object. |
| FS_CLOSE | 10 | Close an I/O stream to a file system object. |
| FS_UNLINK | 11 | Remove the name of an object. |
| FS_RENAME | 12 | Change the name of an object. |
| FS_MKDIR | 13 | Create a directory. |
| FS_RMDIR | 14 | Remove a directory. |
| FS_MKDEV | 15 | Make a special device file. |
| FS_LINK | 16 | Make a directory reference to an existing object. |
| FS_SYM_LINK | 17 | Make a symbolic link to an existing object. |
| FS_GET_ATTR | 18 | Get the attributes of the object behind an I/O stream. |
| FS_SET_ATTR | 19 | Set the attributes of the object behind an I/O stream. |
| FS_GET_ATTR_PATH | 20 | Get the attributes of a named object. |
| FS_SET_ATTR_PATH | 21 | Set the attributes of a named object. |
| FS_GET_IO_ATTR | 22 | Get the attributes kept by the I/O server. |
| FS_SET_IO_ATTR | 23 | Set the attributes kept by the I/O server. |
| FS_DEV_OPEN | 24 | Complete the open of a remote device or pseudo-device. |
| FS_SELECT | 25 | Query the status of a device or pseudo-device. |
| FS_IO_CONTROL | 26 | Perform an object-specific operation. |
| FS_CONSIST | 27 | Request that cache consistency action be performed. |
| FS_CONSIST_REPLY | 28 | Acknowledgement that consistency action completed. |
| FS_COPY_BLOCK | 29 | Copy a block of a swap file. |
| FS_MIGRATE | 30 | Tell I/O server that an I/O stream has migrated. |
| FS_RELEASE | 31 | Obsolete. |
| FS_REOPEN | 32 | Recover the state about an I/O stream. |
| FS_RECOVERY | 33 | Signal that recovery actions have completed. |
| FS_DOMAIN_INFO | 34 | Return information about a file system domain. |
| PROC_MIG_COMMAND | 35 | Used to transfer process state during migration. |
| PROC_REMOTE_CALL | 36 | Used to forward system call to the home node. |
| PROC_REMOTE_WAIT | 37 | Used to synchronize exit of migrated process. |
| PROC_GETPCB | 38 | Return process table entry for migrated process. |
| REMOTE_WAKEUP | 39 | Wakeup a remote process. |
| SIG_SEND | 40 | Issue a signal to a remote process. |
| FS_RELEASE_NEW | 41 | Tell source of migration to release I/O stream. |

Table 1. The Remote Procedure Calls that are used in the Sprite implementation.

| Important Source Files | |
|---|---|
| fs/fs.h | Basic FS definitions. |
| fs/fsNameOps.h | Definitions for the naming interface. |
| fsrmt/fsNameOpsInt.h | More definitions for the naming interface. |
| fsrmt/fsrmtInt.h | Definitions for the I/O interface. |
| fsrmt/fsSpriteDomain.c | RPC stubs for the naming interface. |
| fsrmt/fsSpriteIO.c | RPC stubs for the I/O interface |
| fsrmt/fsRmtAttributes.c | RPC stubs for attribute handling. |
| fsrmt/fsRmtDevice. | RPC stubs for remote devices. |
| fsrmt/fsRmtMigrate.c | RPC stubs for migration. |
| fsio/fsStream.c | RPC stub for migration callback. |
| fsconsist/fsCacheConsist.c | RPC stubs for cache consistency. |
| fsutil/fsRecovery.c | RPC stubs for state recovery. |

Table 2. Important source files concerning the RPC interface to Sprite. The file names are relative to the ‘‘/sprite/src/kernel’’ directory. This list isn’t guaranteed to be complete. There are likely to be other definition files needed to fully resolve the parameter definitions of an RPC. (All required definitions are given in this appendix.)

definition of their parameters. However, these definitions also include structures that are common to the rest of the implementation and may be defined elsewhere. Table 2 has a list of the ‘‘.c’’ files that contain most of the RPC stubs and the ‘‘.h’’ files that contain most of the relevant typedefs. Finally, note that these structures are passed around within the kernel as well as between kernels using RPC, and there are some structure fields that are not valid during an RPC.

**2. ECHO_2** This echoes data off another host. This operation is handled by an Rpc_Server process, so it exercises the full execution path involved in a regular RPC. (The unsupported ECHO_1 was used to echo off the interrupt handler.) This is used for benchmarking the RPC system, and it is also used by the recovery module to verify that another host is up. Equal amounts of data are transferred in both directions. The data is uninterpreted, and it is put in the data area of the RPC packet.

**3. SEND** This transmits data to another host. This is just used for benchmarking the RPC system. Data is only transferred one way, from the client to the server.

**4. RECEIVE** This receives data from another host. This is only used for benchmarking the RPC system.

**5. GETTIME** This is a broadcast RPC used to get the time-of-day. This is used by hosts during boot strap in order to set their clocks. The clock value is also used to set their rpcBootID, which is included in the header of all subsequent RPC packets. This is the first RPC done by a Sprite host, and the bootID in its RPC header is zero. The parameters and data areas of the RPC request are empty. The parameter area of the reply contains the following structure:

```
typedef struct RpcTimeReturn {
        Time    time;
        int     timeZoneMinutes;
        int     timeZoneDST;
} RpcTimeReturn;
```

The Time data type is defined as:

```
typedef struct Time {
        int     seconds;
        int     microseconds;
} Time;
```

The time is the number of seconds since Jan 1, 1970 in universal (GMT) time. The timeZoneMinutes is the offset of the local timezone from GMT. The timeZoneDST is a flag indicating if daylight savings time is allowed in the timezone (not if it is in effect at the current date).

**6. FS_PREFIX** This is a broadcast RPC used to locate the server for a prefix. The request parameter area is empty. The request data area contains the null-terminated prefix, whose length can be determined from the RPC packet header's dataLength field. The reply data area is empty. The reply parameter area contains the following structure:

```
typedef struct FsPrefixReplyParam {
        FsrmtUnionData      openData;
        Fs_FileID       fileID;
} FsPrefixReplyParam;
```

The fileID identifies the root directory of the *domain* identified by the prefix. The client should cache the mapping from the prefix to the fileID. It specifies this fileID as the *prefixID* in lookup RPCs (FS_OPEN, etc.). The openData is used by Sprite clients to set up internal data structures associated with an open I/O stream. The FsrmtUnionData typedef is described under FS_OPEN. The Fs_FileID is defined as follows:

```
typedef struct Fs_FileID {
        int     type;
        int     serverID;
        int     major;
        int     minor;
} Fs_FileID;
```

The serverID is the Sprite hostID of the I/O server for the object. The major and minor fields identify the object to the I/O server. The values for the type field are defined in Table 3. With the FS_PREFIX RPC the returned type is either FSIO_RMT_FILE_STREAM for regular Sprite file systems, or FSIO_PFS_NAMING_STREAM for pseudo-file-systems.

| Sprite Internal Stream Types | | |
|---|---|---|
| # | Name | Description |
| 0 | FSIO_STREAM | For top-level stream descriptor. |
| 1 | FSIO_LCL_FILE_STREAM | Local file. |
| 2 | FSIO_RMT_FILE_STREAM | Remote file. |
| 3 | FSIO_LCL_DEVICE_STREAM | Local device. |
| 4 | FSIO_RMT_DEVICE_STREAM | Remote device. |
| 5 | FSIO_LCL_PIPE_STREAM | Anonymous pipe. |
| 6 | FSIO_RMT_PIPE_STREAM | Remote anonymous pipe. |
| 7 | FSIO_CONTROL_STREAM | Pseudo-device control stream. |
| 8 | FSIO_SERVER_STREAM | Pseudo-device server stream. |
| 9 | FSIO_LCL_PSEUDO_STREAM | Attached to a server stream. |
| 10 | FSIO_RMT_PSEUDO_STREAM | Indirectly attached to a server stream. |
| 11 | FSIO_PFS_CONTROL_STREAM | Records pseudo-file-system server. |
| 12 | FSIO_PFS_NAMING_STREAM | Pseudo-file-system naming stream. |
| 13 | FSIO_LCL_PFS_STREAM | Pseudo-file-system I/O stream. |
| 14 | FSIO_RMT_PFS_STREAM | Remote Pseudo-file-system I/O stream. |
| 15 | FSIO_RMT_CONTROL_STREAM | Fake type for GET_ATTR of a pseudo-device. |
| 16 | FSIO_PASSING_STREAM | Fake type when passing open streams. |

Table 3. The internal stream types used in the Sprite implementation. Most types have corresponding local (LCL) and remote (RMT) types. The CONTROL and SERVER streams, however, are always local because they are between the kernel and the pseudo-device or pseudo-file-system server process. The PFS_NAMING stream is always remote. It is returned in response to prefix broadcasts in the case of a pseudo-file-system. Some additional types are defined for use with the kernel's internal I/O interface, and these are included here for completeness.

**7. FS_OPEN** This is used to open a file system object in preparation for further I/O operations. The parameters to an open RPC include a pathname, which may have several components, and a file ID that indicates where the pathname starts. It also includes information about how the object will be used, and identification of the user and the client host that is doing the open. The reply to an open is one of two things. Ordinarily data is returned that is used to set up data structures for an I/O stream to the object. Depending on the type of object opened, which is specified in the reply, the reply contains different data used for this purpose. However, it is also possible that the pathname leaves the server's domain, in which case a new pathname is returned to the client, and perhaps also a new prefix. These details are described in more detail below.

**7.1. FS_OPEN Request Format** The request data area contains a null terminated path name. The request parameter area contains the following structure:

```
typedef struct Fs_OpenArgs {
        Fs_FileID       prefixID;
    Fs_FileID   rootID;
        int     useFlags;
        int     permissions;
        int     type;
        int     clientID;
        int     migClientID;
        Fs_UserIDs    id;
} Fs_OpenArgs;
```

The prefixID specifies where the pathname begins. There are two cases for this. If the client initially has an absolute pathname, then it can match this against its prefix cache and use the fileID associated with the longest matching prefix. The prefix should be stripped off before sending the pathname to the server. If the client initially has a relative pathname, then the prefixID is the fileID associated with the current working directory. This is obtained by a previous FS_OPEN RPC on the current directory. The rootID is a prefix ID, and it is used to trap out pathnames that ascend out the root directory of a domain. It is either the same as the prefixID, or it is the ID of the prefix that identifies the domain of the current directory. Note that this supports implementation of chroot(), and it also allows servers to export prefixes that don't correspond to the ''natural'' root directory of a domain.

The permissions field contains the permission bits to set on newly created files. These are defined in Table 4.

The type field constrains the type of object that can be opened. If the type is FS_FILE, then any type can be opened. This is the way the FS_OPEN RPC is used by the open() system call. However, FS_OPEN is also used in the implementation of readlink(),

| Permission Bits | |
|---|---|
| FS_OWNER_READ | 00400 |
| FS_OWNER_WRITE | 00200 |
| FS_OWNER_EXEC | 00100 |
| FS_GROUP_READ | 00040 |
| FS_GROUP_WRITE | 00020 |
| FS_GROUP_EXEC | 00010 |
| FS_WORLD_READ | 00004 |
| FS_WORLD_WRITE | 00002 |
| FS_WORLD_EXEC | 00001 |
| FS_SET_UID | 04000 |
| FS_SET_GID | 02000 |

Table 4. Permission bits for the Fs_OpenArgs structure. These are octal values

symlink(), and mknod(). For these calls specific types are indicated so that the type of the named file must match. (Warning, note the bug report concerning FS_REMOTE_LINK and FS_SYMBOLIC_LINK in the last section of this appendix.) The types are defined in Table 5, and they correspond to types in the file descriptors kept on disk.

The clientID is the hostID of the process doing the open. The migClientID is the home node of a migrated process, which may be different than the clientID if a process has migrated. This is used when opening devices so that a migrated process can open devices on its home node. This only applies to device files with the FS_LOCALHOST_ID serverID attribute. Device files with a specific host ID (>0) for their serverID attribute always specify the device on that host. (See FS_MAKE_DEV below.)

The id field contains the user and group IDs of the process doing the open. The Fs_UserIDs typedef is defined as follows:

```
#define FS_NUM_GROUPS       8

typedef struct Fs_UserIDs {
        int user;
        int numGroupIDs;
        int group[FS_NUM_GROUPS];
} Fs_UserIDs;
```

The useFlags indicate how the object is going to be used. Valid useFlag bits are defined in Table 6. They are divided into two sets, those passed into the Fs_Open system call from user programs, and those set by the Sprite kernel for its own use. (For

| File Descriptor Types | |
|---|---|
| FS_FILE | 0 |
| FS_DIRECTORY | 1 |
| FS_SYMBOLIC_LINK | 2 |
| FS_REMOTE_LINK | 3 |
| FS_DEVICE | 4 |
| (not used) | 5 |
| FS_LOCAL_PIPE | 6 |
| FS_NAMED_PIPE | 7 |
| FS_PSEUDO_DEV | 8 |
| (not used) | 9 |
| (reserved for testing) | 10 |

Table 5. Types for disk-resident file descriptors. Also used for the type field in the Fs_OpenArgs structure.

| Usage Flags | | |
|---|---|---|
| Value | Name | Description |
| 0xfff | FS_USER_FLAGS | A mask used to prevent user programs from setting kernel bits. |
| 0x001 | FS_READ | Open the object for reading. |
| 0x002 | FS_WRITE | Open the object for writing. |
| 0x004 | FS_EXECUTE | Open the object for execution, if it is a regular file, or for changing the current directory if it is a directory. |
| 0x008 | FS_APPEND | Open the object for append mode writing. |
| 0x020 | FS_PDEV_MASTER | Open a pseudo-device as the server process. |
| 0x080 | FS_PFS_MASTER | Open a remote link as the server for a pseudo-file-system. |
| 0x200 | FS_CREATE | Create a directory entry for the object if it isn't there already. |
| 0x400 | FS_TRUNC | Truncate the object after opening. |
| 0x800 | FS_EXCLUSIVE | If specified with FS_CREATE, then the open will fail if the object already exists. |
| 0xfffff000 | FS_KERNEL_FLAGS | Bits are set in this field by the client kernel. |
| 0x00001000 | FS_FOLLOW | Follow symbolic links when traversing the pathname. |
| 0x00004000 | FS_SWAP | The file is being used as a VM backing file. |
| 0x00010000 | FS_OWNERSHIP | Set when changing ownership or permission bits. The server should verify that the opening process owns the file. |
| 0x00020000 | FS_DELETE | Set when deleting a file. (FS_UNLINK RPC, not FS_OPEN.) |
| 0x00040000 | FS_LINK | Set when creating a hard link. (FS_LINK RPC, not FS_OPEN.) |
| 0x00080000 | FS_RENAME | Set during rename. (FS_RENAME RPC, not FS_OPEN.) |

Table 6. Values for the useflags field of Fs_OpenArgs and Fs_LookupArgs.

historical reasons the UNIX open() call is mapped to Fs_Open, and some of the flag bits differ from the UNIX O_* flags, mainly the READ and WRITE bits.)

**7.2. FS_OPEN Reply Format** The data area of an FS_OPEN reply may contain a returned pathname, in the case where the input pathname leaves the server's domain. This is indicated by the FS_LOOKUP_REDIRECT return code. If the returned name begins with a '/' character then it has been expanded by the server and can be matched against the client's prefix table. If the prefixLength parameter (defined below) is non-zero then the initial part of the expanded pathname is a domain prefix that should be added to the client's prefix table. The client should use the FS_PREFIX RPC to locate the domain's server. If the returned pathname begins with ''../'', then the client has to combine the returned pathname with the prefix it used for the server's domain as follows. (This means a client implementation has to remember the domain prefix associated with a current working directory.) If the prefix is ''/a/b'' and the returned pathname is ''../x/y'', then these are combined into ''/a/b/../x/y'', which further reduces to ''/a/x/y''. Note this no longer matches on the ''/a/b'' prefix.

The parameter area of the FS_OPEN reply contains data used to initialize the I/O stream data structures. The format of the rely parameters is defined as follows:

```
typedef struct FsrmtOpenResultsParam {
        int     prefixLength;
        Fs_OpenResults       openResults;
        FsrmtUnionData       openData;
} FsrmtOpenResultsParam;

typedef struct Fs_OpenResults {
        Fs_FileID    ioFileID;
        Fs_FileID    streamID;
        Fs_FileID    nameID;
        int     dataSize;
        ClientData   streamData;
} Fs_OpenResults;

typedef union FsrmtUnionData {
        Fsio_FileState fileState;
        Fsio_DeviceState     devState;
        Fspdev_State pdevState;
} FsrmtUnionData;
```

The Fs_OpenResults contain three object identifiers, one for the object that was opened, one for the I/O stream to that object (the streamID is type FSIO_STREAM), and one for the file that names the object (the nameID is type FSIO_RMT_FILE_STREAM). For devices and pseudo-devices the file that represents the name is different than the object itself, and the nameID is used when getting the attributes of an object. The I/O stream also has an identifier because I/O streams are passed between machines during process migration. The rest of the data is type-specific and is described below.

```
typedef struct Fsio_FileState {
        int     cacheable;
        int     version;
```

```
            int    openTimeStamp;
                   Fscache_Attributes attr;
                   int       newUseFlags;
        } Fsio_FileState;

        typedef struct Fscache_Attributes {
                   int       firstByte;
                   int       lastByte;
                   int       accessTime
                   int       modifyTime;
                   int       createTime;
                   int       userType;
                   int       permissions;
                   int       uid;
                   int       gid;
        } Fscache_Attributes;
```

The Fsio_FileState is returned when a regular file or directory is opened. In this case the type in the ioFileID is FSIO_RMT_FILE_STREAM. The cacheable flag indicates if the client can cache the file. The version number is used to detect stale data in the client's cache. The openTimeStamp should be kept and used later when handling FS_CONSIST RPCs. (The openTimeStamp is redundant with respect to the version number, and it may be eliminated in the future.) The Fscache_Attributes are a sub-set of the file attributes stored at the file server. The lastByte, modifyTime and accessTime are updated by the client if it caches the file. These attributes are pushed back to the server at close time. (firstByte is unused. It is a vestige of a named pipe implementation.) The permission bits and ownership IDs are used with setuid and setgid programs. The newUseFlags are a modified version of the useFlags passed in the FS_OPEN request. The client stores these in its I/O stream descriptor and does consistency checking on subsequent I/O operations by the user-level application.

```
        typedef struct Fsio_DeviceState {
                   int       accessTime;
                   int       modifyTime;
                   Fs_FileID       streamID;
        } Fsio_DeviceState;
```

The Fsio_DeviceState is returned from the file server when a device is opened. The type in the ioFileID is either FSIO_LCL_DEVICE_STREAM or FSIO_RMT_DEVICE_STREAM. In the latter case, the client will pass the Fsio_DeviceState to the I/O server in a FS_DEV_OPEN RPC. The accessTime and modifyTime are maintained at the I/O server. (Currently they are never pushed back to the file server. This is bug.) The streamID is the same as in the Fs_OpenResults, but it is included in Fsio_DeviceState so it can be passed to the I/O server.

```
        typedef struct Fspdev_State {
                   Fs_FileID       ctrlFileID;
```

```
    Proc_PID    procID;
        int     uid;
        Fs_FileID       streamID;
} Fspdev_State;
```

The Fspdev_State is returned when a pseudo-device is opened. The type in the ioFileID is FSIO_CONTROL_STREAM when the server process opens the pseudo-device. In this case the Fspdev_State structure is not returned.

When any other process opens the pseudo-device then the type in the ioFileID is either FSIO_LCL_PSEUDO_STREAM or FSIO_RMT_PSEUDO_STREAM. In the latter case the client passes the Fspdev_State to the I/O server with a FS_DEV_OPEN RPC. The ctrlFileID identifies the control stream of the server process. The procID and uid should be filled in by the client before the FS_DEV_OPEN RPC. The streamID is the same as that in the Fs_OpenResults, but it is included in Fspdev_State so it can be passed to the I/O server. The I/O server sets up a shadow stream descriptor that matches the client's.

When a process opens a file in a pseudo-file-system then the type in the ioFileID is FSIO_RMT_PFS_STREAM. (If the pseudo-file-system server is on the same host as the process doing the open, the the FS_OPEN RPC is not used, so the FSIO_LCL_PFS_STREAM is not seen here.)

**8. FS_READ** This call is used to read data from a file system object. The data area of the request message is empty. The parameter area of the request contains the following structure.

```
typedef struct FsrmtIOParam {
        Fs_FileID       fileID;
        Fs_FileID       streamID;
        Sync_RemoteWaiter waiter;
        Fs_IOParam      io;
} FsrmtIOParam;

typedef struct {
        List_Links      links;
        int     hostID;
        Proc_PID        pid;
        int     waitToken;
} Sync_RemoteWaiter;

typedef struct Fs_IOParam {
        Address         buffer;
        int     length;
        int     offset;
        int     flags;
        Proc_PID        procID;
        Proc_PID        familyID;
        int   uid;
              int       reserved;
    } Fs_IOParam;
```

The fileID and the streamID together specify the I/O stream and the object it references. These have been returned from a previous FS_OPEN RPC. The Sync_RemoteWaiter structure contains information about the process in case the read operation would block, in which case it is used in a subsequent REMOTE_WAKEUP RPC. (The links field of this is not valid during the RPC.) The length in the Fs_IOParam indicates how much data is to be read, and the offset indicates the byte offset at which to start the read. (The buffer field is not valid during the RPC.) The flags are the flags from the I/O stream descriptor, which are derived from the useFlags in FS_OPEN. The procID, familyID, and uid are used for ownership checking by certain devices. The reserved field is currently unused, but may eventually be used for a file's version number.

The reply message for a read contains the data in the data area, and a Fs_IOReply in the parameter area.

```
    typedef struct Fs_IOReply {
              int       length;
              int       flags;
              int       signal;
              int       code;
    } Fs_IOReply;
```

The length indicates the number of bytes returned. The flags indicate the select state of the object. They are an or'd combination of the FS_READ, FS_WRITE, and FS_EXECUTE bits defined above. The signal and code are used to return a signal from certain kinds of devices. If non-zero, the signal field will result in that signal being sent to the application process, along with the code that modifies the signal.

If the return code of the FS_READ RPC is FS_WOULD_BLOCK, then length may be greater than or equal to zero. This indicates that less than the requested amount of data was returned, and that the I/O server has saved the Sync_RemoteWaiter information. A REMOTE_WAKEUP RPC will be generated by the I/O server when the object becomes readable.

**9. FS_WRITE** This is similar to the FS_READ RPC, except that the request data area contains the data to be transfered. The request parameters are the same as for FS_READ. The reply parameters are also the same. If the I/O server doesn't accept all the data transferred to it then the client will block the application process if FS_WOULD_BLOCK is returned. Otherwise a short write and a SUCCESS error code will prompt an immediate retry of the rest of the data.

There are some additional flags in the Fs_IOParam structure that pertain to writes. These are described below.

0x00100000 FS_CLIENT_CACHE_WRITE

When a client writes back data from its cache this flag is set. In this case the file server does not reset the modify time because the client maintains the modify time while caching the file.

0x02000000 FS_LAST_DIRTY_BLOCK
: This is set when a client is writing back its last block for a particular file. At the receipt of the last block a server in write-back-ASAP mode will schedule the file to be written to disk, although the RPC will return before the file actually gets to disk.

0x10000000 FS_WB_ON_LDB
: This tells the server to write through the file if it is the last dirty block. This will appear with FS_LAST_DIRTY_BLOCK. At the receipt of a block marked with this flag the server will block until the file is written through, and the RPC will return after the file is on disk.

**10. FS_CLOSE** When the last process using an I/O stream closes its reference, then an FS_CLOSE RPC is made to the I/O server. The request data area is empty. The request parameter area is described below. The reply message has no data or parameters, only a return code.

```
typedef struct FsRemoteCloseParams {
        Fs_FileID       fileID;
        Fs_FileID       streamID;
        Proc_PID        procID;
        int     flags;
        FsCloseData     closeData;
        int     closeDataSize;
} FsRemoteCloseParams;

typedef union FsCloseData {
        Fscache_Attributes      attrs;
} FsCloseData;
```

The fileID is the ioFileID from the FS_OPEN RPC. The streamID is also from the FS_OPEN RPC. The procID identifies the process doing the close. The flags are the flags from the stream descriptor, and they may also include the FS_LAST_DIRTY_BLOCK and FS_WB_ON_LDB flags if a file is in write-back-on-close mode. The closeData is a type-specific union used to propagate attributes back to the file server at close time. Currently this is only implemented for regular files, in which case the closeData the Fscache_Attributes already described. (It should also be implemented for devices, but the file server is not contacted at close time for devices.) The closeDataSize is the number of valid bytes in the closeData union.

**11. FS_UNLINK** This RPC is used to remove a directory entry for an object. When the last directory entry that references an object is removed, the underlying object is deleted. However, if the object is still referenced by an open I/O stream then the deletion is postponed until the I/O stream is closed. The request data area contains a null terminated pathname. The request parameter area contains an Fs_LookupArgs record, which is a sub-set of the Fs_OpenArgs record previously defined.

```
typedef struct Fs_LookupArgs {
        Fs_FileID       prefixID;
        Fs_FileID       rootID;
        int     useFlags;
        Fs_UserIDs      id;
        int     clientID;
        int     migClientID;
} Fs_LookupArgs;
```

The reply to an RPC_UNLINK is ordinarily only a return code. However, as with all operations on pathnames, the server may return a new pathname if the input pathname leaves its domain. In this case the return code is FS_LOOKUP_REDIRECT, and the return data area contains the new pathname (see NB below), and the return parameter area contains an integer indicating the length of the prefix embedded in the returned pathname, or zero.
NB: For implementation reasons, the return data area also contains room for the prefix length (4 bytes) in front of the returned pathname. See FS_OPEN for a fuller explanation of FS_LOOKUP_REDIRECT.

**12. FS_RENAME** This is an operation on two pathnames, and it is used to change the name of an object in the file system. The request parameters include the Fs_LookupArgs previously defined, plus another fileID for the prefix of the second pathname. The data area contains two pathnames, and currently this is defined as two maximum length character arrays.

```
typedef struct Fs_2PathParams {
        Fs_LookupArgs       lookup;
        Fs_FileID       prefixID2;
} Fs_2PathParams;

#define FS_MAX_PATH_NAME_LENGTH       1024

typedef struct Fs_2PathData {
        char    path1[FS_MAX_PATH_NAME_LENGTH];
        char    path2[FS_MAX_PATH_NAME_LENGTH];
} Fs_2PathData;
```

There can also be a pathname redirection during a FS_RENAME, in which case the return code is FS_LOOKUP_REDIRECT, and the return data area has the returned pathname. Again, for implementation reasons two fields in the parameter area (Fs_2PathReply) are repeated in the data area before the returned pathname (Fs_2PathRedirectInfo). The name1ErrorP flag that is returned indicates whether the error code applies to the first name (if name1ErrorP is non-zero), or to the second pathname. (Please excuse the fact that prefixLength and name1ErrorP occur in opposite orders! ugh. The Fs_2PathRedirectInfo is passed around the kernel internally, and these two fields are copied into the parameter area so they get byteswapped correctly.)

```
typedef struct Fs_2PathReply {
    int    prefixLength;
           Boolean          name1ErrorP;
} Fs_2PathReply;

typedef struct Fs_2PathRedirectInfo {
           int name1ErrorP;
           int       prefixLength;
           char fileName[FS_MAX_PATH_NAME_LENGTH];
} Fs_2PathRedirectInfo;
```

Note that redirection makes lookups involving two pathnames slightly more complicated than an operation on a single pathname. A Sprite client will use its prefix cache to get the prefixFileID for both pathnames, and they may specify different servers. The client always directs the FS_RENAME (or FS_LINK) to the server for the first pathname. If the second pathname is also in the same domain then the FS_RENAME (or FS_LINK) can complete with a single RPC. If the first pathname leaves the server's domain, the server returns FS_WOULD_BLOCK and sets name1ErrorP to a non-zero value. If the second pathname begins in the server's domain but subsequently leaves it, the server returns FS_WOULD_BLOCK and sets name1ErrorP to zero. If the second pathname doesn't begin in the server's domain, it returns FS_CROSS_DOMAIN_OPERATION. At this point the client does a FS_GET_ATTR on the *parent* of the second pathname to make sure that further redirections do not lead the second pathname back to the server's domain. The name of the parent is easily computed by trimming off the last component of the second pathname. If the FS_GET_ATTR gets a redirect the client reiterates, otherwise the FS_RENAME (or FS_LINK) fails.

**13. FS_MKDIR** This is used to create a directory. The request data is a null terminated pathname. The request parameters are the Fs_OpenArgs described above for FS_OPEN, with the type equal to FS_DIRECTORY. If the return code is FS_LOOKUP_REDIRECT, then the reply data is a new pathname (again preceded by 4 bytes of junk), and the reply parameters is the length of the embedded prefix.

**14. FS_RMDIR** This is used to remove a directory. The request data is a null terminated pathname. The request parameters are the Fs_LookupArgs described above for FS_UNLINK. If the return code is FS_LOOKUP_REDIRECT, then the reply data is a new pathname (again preceded by 4 bytes of junk), and the reply parameters is the length of the embedded prefix.

**15. FS_MKDEV** This is used to create a device file. The request data is a null terminated pathname. The request parameters, Fs_MakeDeviceArgs, are defined below. If the return code is FS_LOOKUP_REDIRECT, then the reply data is a new pathname (again preceded by 4 bytes of junk), and the reply parameters is the length of the embedded prefix.

```
typedef struct Fs_MakeDeviceArgs {
        Fs_OpenArgs open;
        Fs_Device device;
} Fs_MakeDeviceArgs;

typedef struct Fs_Device {
        int     serverID;
        int     type;
        int     unit;
        ClientData      data;
} Fs_Device;
```

The Fs_MakeDeviceArgs are a slight super-set of the Fs_OpenArgs. They additionally contain a Fs_Device structure that defines the server, type, and unit of the peripheral device. (The data field is not used in the RPC.) The serverID is a Sprite Host ID. If the special value FS_LOCALHOST_ID is used, then this device file always specifies the device attached to the local host. Otherwise, it specifies a devices at a particular host.

```
#define FS_LOCALHOST_ID    -1
```

Some of the device types are defined in Table 7, although this list is not guaranteed to be complete. These types are defined in <kernel/dev/devTypes.h>. The unit number is device specific, and no attempt is made to specify them here. (For example, the ethernet device encodes the protocol number in the unit. The SCSI devices encode the controller number, target ID, and LUN.)

| Device Types | |
|---|---|
| DEV_TERM | 0 |
| DEV_SYSLOG | 1 |
| DEV_SCSI_WORM | 2 |
| DEV_PLACEHOLDER_2 | 3 |
| DEV_SCSI_DISK | 4 |
| DEV_SCSI_TAPE | 5 |
| DEV_MEMORY | 6 |
| DEV_XYLOGICS | 7 |
| DEV_NET | 8 |
| DEV_SCSI_HBA | 9 |
| DEV_RAID | 10 |
| DEV_DEBUG | 11 |
| DEV_MOUSE | 12 |

Table 7. Definitions for device types.

**16. FS_LINK** This creates another directory entry to an existing object. The two objects are restricted to be within the same file system domain. The request and reply messages have the same format as FS_RENAME. This is an operation on two pathnames, and the comments regarding pathname redirection from FS_RENAME apply.

**17. FS_SYM_LINK** This is defined but not yet supported. Instead, symbolic links and remote links are created by creating a file using FS_OPEN and type FS_SYMBOLIC_LINK or FS_REMOTE_LINK, and then writing the value of the link with FS_WRITE. This should change because it interacts poorly with systems that have a different format for their remote links. (For example, for no good reason Sprite includes a null character in its implementation of symbolic links, while UNIX does not. NFS access to Sprite are confused by this, and it is possible to create bad symbolic links on an NFS server via the Sprite NFS pseudo-file-system.)

**18. FS_GET_ATTR** This is used to get the attributes of the object behind an open I/O stream. The parameter area of the request contain a Fs_FileID, which has been defined above. This is the same as the ioFileID returned from an FS_OPEN. The request and reply data areas are empty. The reply parameter area contains a Fs_Attributes structure defined below.

```
typedef struct Fs_Attributes {
        int             serverID;
        int             domain;
        int             fileNumber;
        int             type;
        int             size;
        int             numLinks;
        unsigned int    permissions;
        int             uid;
        int             gid;
        int             devServerID;
        int             devType;
        int             devUnit;
        Time            createTime;
        Time            accessTime;
        Time            descModifyTime;
        Time            dataModifyTime;
        int             blocks;
        int             blockSize;
        int             version;
        int             userType;
        int             pad[4];
} Fs_Attributes;
```

**19. FS_SET_ATTR** This is used to set the attributes of an object behind an I/O stream. The request parameters contain the FsRemoteSetAttrParams structure, which is defined below. The request data area, reply data area, and reply parameter area are all empty.

```
typedef struct FsRemoteSetAttrParams {
        Fs_FileID       fileID;
        Fs_UserIDs      ids;
        Fs_Attributes   attrs;
        int             flags;
} FsRemoteSetAttrParams;
```

The fileID identifies the object, and was returned as the ioFileID from the FS_OPEN RPC. The Fs_UserIDs structure is needed to verify that the attributes can be changed, and this has been defined above with FS_OPEN. The flags field indicates which attributes are to be set. These flags are described in Table 8.

**20. FS_GET_ATTR_PATH** This is used to get the attributes of a file system object from the file server. This is an operation on a pathname, so the request data area contains a null terminated pathname. The request parameters are the Fs_OpenArgs previously

| Set Attribute Flags | | |
|---|---|---|
| Value | Name | Description |
| 0x1F | FS_SET_ALL_ATTRS | Set all the attributes possible. See the following definitions. |
| 0x01 | FS_SET_TIMES | Set the access and modify times. Used to implement UNIX utimes(). |
| 0x02 | FS_SET_MODE | Set the permissions. Used to implement UNIX chmod(). |
| 0x04 | FS_SET_OWNER | Set the owner and group. If either of the gid and uid fields in the Fs_Attributes structure are -1, then they are not changed. |
| 0x08 | FS_SET_FILE_TYPE | Set the user-defined file type. The user-defined types currently in use are defined in the next table. |
| 0x10 | FS_SET_DEVICE | Set the device attributes. Used to implement Fs_MakeDevice (UNIX mknod()). |

Table 8. Values for the flags field in FsRemoteSetAttrParams.

| User-defined File Types | | |
|---|---|---|
| FS_USER_TYPE_UNDEFINED | 0 | Not used. |
| FS_USER_TYPE_TMP | 1 | Temporary file. |
| FS_USER_TYPE_SWAP | 2 | VM swap file. |
| FS_USER_TYPE_OBJECT | 3 | Program object file. |
| FS_USER_TYPE_BINARY | 4 | Complete program image. |
| FS_USER_TYPE_OTHER | 5 | Everything else. |

Table 9. Values for the user-defined file type attribute.

described. The reply parameters are a Fs_GetAttrResultsParam structure, which is defined below. If the return code is FS_LOOKUP_REDIRECT, then the reply data area contains a returned pathname. NB: in this case there is *no* 4 bytes of padding in the data area! (Sorry for this ugly inconsistency.)

```
typedef union Fs_GetAttrResultsParam {
        int        prefixLength;
        struct AttrResults {
        Fs_FileID       fileID;
        Fs_Attributes  attrs;
        } attrResults;
} Fs_GetAttrResultsParam;
```

The prefixLength is returned with the FS_LOOKUP_REDIRECT status. Otherwise, the fileID is the same as the ioFileID returned from an FS_OPEN, and this is used to do a FS_GET_IO_ATTR RPC with the I/O server. The Fs_Attributes structure has been defined above.

**21. FS_SET_ATTR_PATH** This is used to set the attributes of a named file system object. The request data area contains a null terminated pathname. The request parameters are defined below. The reply data area contains a new pathname if the return code is FS_LOOKUP_REDIRECT. (No padding bytes.) The reply data area contains a Fs_GetAttrResultsParam structure which has been defined above. If FS_LOOKUP_REDIRECT is returned then only the prefixLength is defined in the Fs_GetAttrResultsParam.

```
typedef struct Fs_SetAttrArgs {
        Fs_OpenArgs  openArgs;
        Fs_Attributes  attr;
        int        flags;
} Fs_SetAttrArgs;
```

Each of these fields have been described above. The flags define which attributes to set. See FS_SET_ATTR for a description.

**22. FS_GET_IO_ATTR** This is used to get the attributes that are maintained by the I/O server, typically the access and modify times. The general approach to getting attributes is to first contact the file server (with either FS_GET_ATTR or FS_GET_ATTR_PATH) to get the initial version of the attributes, and then use this RPC to get the most up-to-date access and modify times. This is only applicable to remote devices and remote pseudo-devices.

The request parameter area contains a Fs_GetAttrResultsParam that has been initialized by a FS_GET_ATTR or FS_GET_ATTR_PATH RPC. The reply parameter area contains a Fs_Attributes structure. The request and reply data areas are empty.

**23. FS_SET_IO_ATTR** This is used to update the attributes that are maintained by the I/O server, typically the access and modify times. Setting attributes is structured the same as with getting them, so the file server is contacted first (with FS_SET_ATTR or FS_SET_ATTR_PATH). The complete attributes are returned from these calls, and then used as the request parameters in this RPC.

The request message contains a FsRemoteSetAttrParams structure, which is defined below. The request data area, the reply parameter area, and the reply data area are empty.

```
typedef struct FsRemoteSetAttrParams {
        Fs_FileID       fileID;
        Fs_UserIDs      ids;
        Fs_Attributes   attrs;
        int     flags;
} FsRemoteSetAttrParams;
```

The fileID identifies the object, and is the same as the ioFileID returned from an FS_OPEN RPC. The Fs_UserIDs and Fs_Attributes have been described above. The flags indicate what attributes to set, and these are indicated above as well.

**24. FS_DEV_OPEN** This is used to complete an Fs_Open (UNIX open()) of a remote device or remote pseudo-device. This is done after an FS_OPEN RPC has returned an ioFileID with a type of FSIO_RMT_DEVICE_STREAM or FSIO_RMT_PDEV_STREAM. The request parameter area contains a FsDeviceRemoteOpenParam structure, which is defined below. The reply parameter area contains a new ioFileID so the I/O server can modify this if it wants to. The request and reply data areas are empty.

```
typedef struct FsDeviceRemoteOpenParam {
        Fs_FileID       fileID;
        int     useFlags;
        int     dataSize;
        FsrmtUnionData      openData;
} FsDeviceRemoteOpenParam;
```

The fileID is the ioFileID returned from FS_OPEN. The useFlags are the newUseFlags returned from FS_OPEN. The dataSize indicates the number of valid bytes in openData.

The FsrmtUnionData has been described previously.

**25. FS_SELECT** This RPC is used to poll a remote I/O server to determine if an object is ready for I/O. The input parameters include a fileID that identifies the object, three fields that correspond to the read, write, and execute state of the object, and process information used for remote waiting. If a field is non-zero it means an application is querying that state. The return parameters contains copies of these fields, and the I/O server should clear a field to zero if that state is not applicable. In this case, the I/O server should save the Sync_RemoteWaiter information so it can notify the waiting process when the objects state changes.

```
typedef struct FsRemoteSelectParams {
        Fs_FileID       fileID;
        int     read;
        int     write;
        int     except;
        Sync_RemoteWaiter waiter;
} FsRemoteSelectParams;

typedef struct FsRemoteSelectResults {
        int     read;
        int     write;
        int     except;
} FsRemoteSelectResults;
```

NB: The read, write, and except fields are defined to be zero or non-zero, and the fields in the result should be exact copies of the request fields, or they should be reset to zero. (If non-zero they happen to be the bit that corresponds to the bit in the select mask. It is only appropriate to copy the bit or clear it. Don't blindly set the reply fields to 1.)

**26. FS_IO_CONTROL** This is used to do an object-specific operation. A number of generic I/O controls are defined below, and the implementation of different objects are free to define more I/O controls. The request parameter area contains a FsrmtIOCParam structure, which is defined below. The reply parameter area contains a Fs_IOReply structure that defines the amount of data returned, and a signal to generate, if any. The request and reply data areas contain data blocks that are uninterpreted by generic kernel code. In particular, they cannot be byteswapped except by the implementation of the I/O control handler. The parameters include a byteOrder field so the handler can detect a mismatch. In this case it should byteswap in the request data block so it can properly interpret it, and also byteswap the reply datablock so the application process on the remote client can properly interpret it.

```
typedef struct FsrmtIOCParam {
        Fs_FileID       fileID;
        Fs_FileID       streamID;
        Proc_PID        procID;
```

```
    Proc_PID    familyID;
        int         command;
        int         inBufSize;
        int         outBufSize;
        Fmt_Format    format;
        int         uid;
} FsrmtIOCParam;
```

The fileID and streamID have been returned by FS_OPEN. The procID and familyID identify the process and its process group. The sizes indicate how much data is being sent to the I/O server and the maximum amount that can be accepted in return. The format defines a byte ordering, and it is used in conjunction with the Format library package to do byteswapping. The uid specifies the user ID of the application process. The command is the I/O control operation, and the generic ones are defined below.

1 IOC_REPOSITION

Reposition the current access position of the I/O stream. The input data block contains the following structure to define the new access position.

```
#define IOC_BASE_ZERO        0
#define IOC_BASE_CURRENT  1
#define IOC_BASE_EOF         2

typedef struct Ioc_RepositionArgs {
        int base;
        int offset;
} Ioc_RepositionArgs;
```

2 IOC_GET_FLAGS

Return the flag bits associated with an I/O stream. The reply data block contains an integer with the flag bits. The following bits are defined for all objects, although other objects may define more flags.

```
#define IOC_GENERIC_FLAGS 0xFF
#defineIOC_APPEND    0x01
#define IOC_NON_BLOCKING  0x02
#define IOC_ASYNCHRONOUS0x04
#define IOC_CLOSE_ON_EXEC0x08
```

3 IOC_SET_FLAGS

Set the flags on an I/O stream. The request data block contains an integer which is to be new version of the flag word. It completely replaces the old flag word.

4 IOC_SET_BITS

Set individual bits in the flags word of an I/O stream. The request data block contains an integer with the desired bits set.

5 IOC_CLEAR_BITS

Clear individual bits in the flags word of an I/O stream. The request data block contains an integer with the desired bits set.

6 IOC_TRUNCATE
: Truncate an object to a specified length. The input data block contains an integer which is the desired length.

7 IOC_LOCK
: Place an advisory lock on an object. Used to implement UNIX flock(). The request data block contains the following structure. If the lock cannot be obtained then FS_WOULD_BLOCK should be returned and the process information should be saved at the I/O server for a subsequent REMOTE_WAKEUP RPC back to the client.

```
typedef struct Ioc_LockArgs {
        int         flags;
        int         hostID;
        Proc_PID        pid;
        int         token;
} Ioc_LockArgs;

#define IOC_LOCK_SHARED     0x1
#define IOC_LOCK_EXCLUSIVE          0x2
#define IOC_LOCK_NO_BLOCK           0x8
```

The flag bits are defined above, and specify if the lock should be exclusive, in which case it is blocked by either an existing exclusive lock or by a shared lock, or whether it is a shared lock, in which case it can co-exist with other shared locks but not an exclusive lock. If IOC_LOCK_NO_BLOCK is set then the application process will not be blocked by the client in the case of FS_WOULD_BLOCK, so the I/O server doesn't have to remember the process information.

8 IOC_UNLOCK
: Remove an advisory lock on an object. The complement of IOC_LOCK.

9 IOC_NUM_READABLE
: Returns the number of bytes available on an I/O stream. The input data block contains the current offset of the stream. The return data block contains the number of bytes available on the stream.

10 IOC_GET_OWNER
: Returns the owner of an I/O stream, which is either a process or a process group. The return data block contains the following structure, along with definitions for the procOrFamily field.

```
#define IOC_OWNER_FAMILY 0x1
#define IOC_OWNER_PROC      0x2

typedef struct Ioc_Owner {
        Proc_PID        id;
        int         procOrFamily;
} Ioc_Owner;
```

11 IOC_SET_OWNER
: This defines the owner of an I/O stream. The request data block contains the

Ioc_Owner structure.

12 IOC_MAP
: Obsolete, superseded by IOC_MMAP_INFO.

13 IOC_PREFIX
: This returns the prefix under which a stream was opened. This is used to implement the getwd() library call. The reply data block contains the prefix, which is null terminated.

14 IOC_WRITE_BACK
: This is used to write-back a range of bytes of a file to the server's disk. The cache will align the write-back on block boundaries that include the specified range of bytes. The request data area contains the following structure.

```
typedef struct Ioc_WriteBackArgs {
        int       firstByte;
        int       lastByte;
        Boolean        shouldBlock;
} Ioc_WriteBackArgs;
```

15 IOC_MMAP_INFO
: Tell the I/O server that a client is mapping a stream into memory. The request data area contains the following structure. The isMapped field is 0 if the client is unmapping the stream, and 1 if it is mapping the stream.

```
typedef struct Ioc_MmapInfoArgs {
        int       isMapped;
        int       clientID;
} Ioc_MmapInfoArgs;
```

((1<<16)-1) IOC_GENERIC_LIMIT
: The Sprite kernel reserves the numbers below this for generic I/O control commands. Other device drivers and pseudo-device servers define their own I/O controls. Look at the README file ‘‘/sprite/src/include/dev/README’’ for details.

**27. FS_CONSIST** This is issued by a file server as a side effect of an FS_OPEN RPC. It is a command to a client (not the one doing the FS_OPEN) to control its cache so that future accesses see consistent data. The request parameter area contains the following structure, and the request data area, reply data area, and reply parameter area are empty. The client should respond immediately to this RPC and perform the cache consistency actions in the background. The client issues a FS_CONSIST_REPLY RPC to the server when it has completed the requested actions. This is a crude way of doing parallel RPCs to many clients. The server sets up a short timeout (about 1 minute) for the client to complete its actions, and it will let an open complete anyway if this timeout expires and a rogue client has not responded to a consistency request.

```
typedef struct ConsistMsg {
        Fs_FileID       fileID;
        int       flags;
```

```
    int    openTimeStamp;
           int       version;
} ConsistMsg;
```

The fileID identifies the file, and it is the same as the ioFileID returned from the FS_OPEN RPC. The flags are explained below, and they indicate what action the client should take. The openTimeStamp is the time stamp that the server thinks corresponds to the last open-TimeStamp it returned to the client. The version should match with the last version returned to the client in the Fsio_FileState. (It will eventually replace the openTimeStamp altogether.) The point of the openTimeStamp is that if two clients open the same file at the same time, then the reply to one client's FS_OPEN may loose a race with a FS_CONSIST RPC generated by the second client's open. If a client receives a FS_CONSIST RPC with an openTimeStamp ‘‘in the future’’ it drops the consistency request and returns FAILURE (1). This forces the file server to retry the FS_CONSIST call, giving the reply to the FS_OPEN a chance to arrive at the client. The consistency actions are defined below.

0x01 FSCONSIST_WRITE_BACK_BLOCKS
: The client should write back any dirty blocks that are lingering in its cache.

0x02 FSCONSIST_INVALIDATE_BLOCKS
: The client should stop caching the file because it is now concurrently write shared by different hosts. All future I/O operations on this file should bypass the client cache and go through to the file server.

0x04 FSCONSIST_DELETE_FILE
: This is issued as a side effect of a FS_REMOVE RPC if the client has dirty blocks for the file. This is done even if it is the same client as the one currently making the FS_REMOVE RPC.

0x08 FSCONSIST_CANT_CACHE_NAMED_PIPE
: This is reserved for if we ever re-implement named pipes.

0x10 FSCONSIST_WRITE_BACK_ATTRS
: The client should write-back its notion of the access and modify times of the file that it is caching. This is generated as a side effect of FS_GET_ATTR and FS_GET_ATTR_PATH RPCs by other clients. This is only done if the client is actively using the file, and it is suppressed if the client only has the file open for execution. The attributes are returned with the FS_CONSIST_REPLY RPC.

**28. FS_CONSIST_REPLY** This is issued by the client when it has completed the consistency actions requested by the server. The request parameter area contains the following structure.

```
typedef struct ConsistReply {
        Fs_FileID       fileID;
        Fscache_Attributes    cachedAttr;
        ReturnStatus   status;
} ConsistReply;
```

The fileID indicates the file that was acted on. The client always returns its notion of the attributes of the file because it updates these while caching the file. The status indicates whether it could comply with the request. If the server's disk is so full that a write-back could not be made then this status is FS_DISK_FULL. The data is not lost, but it lingers in the client's cache until the write-back can succeed. However, this means that an open() can fail with a disk full error!

**29. FS_COPY_BLOCK** This is used during fork() to copy a swap file on the file server. This prevents the client from reading a swap file over the network just to copy it and write it back. The request parameter area contains the following structure, and the request data area, reply parameter area, and reply data area are empty.

```
typedef struct FsrmtBlockCopyParam {
        Fs_FileID      srcFileID;
        Fs_FileID      destFileID;
        int      blockNum;
} FsrmtBlockCopyParam;
```

The srcFileID and destFileID have been returned from FS_OPEN when the swap files were opened. The blockNum specifies the FS_BLOCK_SIZE (4096 bytes) block to copy. This is a logical block number because the client has no notion of where the swap files live on disk.

**30. FS_MIGRATE** This is used during process migration to inform the I/O server that an I/O stream has migrated to a new client. This is invoked from the destination client as part of creating the process. The request parameter area contains the following structure.

```
typedef struct Fsio_MigInfo {
        Fs_FileID      streamID;
        Fs_FileID   ioFileID;
        Fs_FileID      nameID;
        Fs_FileID      rootID;
        int      srcClientID;
        int       offset;
        int       flags;
} Fsio_MigInfo;
```

This information is packaged up on the source client when the process migrates away. The streamID, ioFileID, and nameID are those that have been returned from a previous FS_OPEN. The rootID was specified in the FS_OPEN request that created the stream. The srcClientID is the client were the process left. The offset field was made obsolete by the FS_RELEASE_NEW and should be removed. The flags are the flags from the stream.

The reply parameter area of the FS_MIGRATE RPC contains the following structure. The request and reply data areas are empty.

```
typedef struct  FsrmtMigParam {
        int      dataSize;
        FsrmtUnionData        data;
```

```
	FsrmtMigrateReply	migReply;
} FsrmtMigParam;

typedef struct FsrmtMigrateReply {
	int flags;
	int offset;
} FsrmtMigrateReply;

#define FS_RMT_SHARED	0x04000000
```

The I/O server returns the same FsrmtUnionData as it does in an FS_OPEN RPC, and the client uses this to set up its I/O stream data structures. The I/O server also tells the client the new stream offset to use, and it gives the client a new version of the stream flags. If the flags in include the FS_RMT_SHARED bit then processes on different hosts are sharing the stream. In this case the offset in the clients' stream descriptors are not valid, and I/O operations on the object have to go through to the I/O server. The I/O server keeps a shadow stream descriptor that contains the valid stream offset in this case.

**31. FS_RELEASE** This procedure is obsolete.

**32. FS_REOPEN** This is used during the state recovery protocol to inform the I/O server about I/O streams in use by a client. The request and reply parameter areas vary depending on the object being reopened. The following structures are possible, although note that every request structure contains a Fs_FileID as its first element. Also, the client should map its FSIO_RMT stream types to the corresponding FSIO_LCL stream types before making the RPC.

```
typedef struct FsRmtDeviceReopenParams {
	Fs_FileID	fileID;
	Fsutil_UseCounts use;
} FsRmtDeviceReopenParams;

typedef struct Fsutil_UseCounts {
	int	ref;
	int	write;
	int	exec;
} Fsutil_UseCounts;
```

The reopen parameters identify the object and specify how many streams the client has to it. Note that the ref field in Fsutil_UseCounts is not the same as the number of reading streams, but it is the total number of streams. This is a mistake and will be fixed eventually; it makes it impossible for a reader to reopen ''/dev/syslog'', which is a single reader/multiple writer device. The request data area, the reply parameter area, and the reply data area are empty in the case of device reopening.

```
typedef struct Fsio_PipeReopenParams {
    Fs_FileID    fileID;
        Fsutil_UseCounts use;
} Fsio_PipeReopenParams;
```

A pipe may become remote due to process migration, therefore it may have to be reopened if the client looses touch with the server. If the server can crashed then the reopen will fail, but if there has only been a network partition the reopen may succeed. The request data area, the reply parameter area, and the reply data area are empty in the case of pipe reopening.

```
typedef struct Fsio_FileReopenParams {
        Fs_FileID       fileID;
        Fs_FileID       prefixFileID;
        Fsutil_UseCounts        use;
        Boolean         flags;
        int     version;
} Fsio_FileReopenParams;

#define FSIO_HAVE_BLOCKS 0x1
#define FS_SWAP         0x4000
```

The reopen parameters for a file specify the file and the prefix of the domain of the file. This is needed to validate that the server still has the disk mounted. The Fsutil_UseCounts are as described above. The flags include FSIO_HAVE_BLOCKS, FS_SWAP, and FS_MAP to indicate if the client has dirty data blocks, is using the file for VM backing store, or is mapping the file into its memory. The version number is the version that the client has cached. The reply parameters when reopening a file are the Fsio_FileState described above. The client should verify that the version number it has is correct, just as it does during an FS_OPEN.

```
typedef struct FspdevControlReopenParams {
        Fs_FileID       fileID;
        int     serverID;
        int     seed;
} FspdevControlReopenParams;
```

The file server that stores a pseudo-device file also keeps some state as to whether there is currently a server process for the pseudo-device so it can prevent conflicts. If the file server crashes this information has to be restored, and it is done by reopening the pseudo-device control handle. The fileID in the reopen parameters is that returned from FS_OPEN when the server process opened the pseudo-device with the FS_PDEV_MASTER flag. The serverID is -1 if the server process has gone away since the file server crashed. The seed is used by the file server to generate unique fileIDs for the connections to the pseudo-device, and this needs to be restored, too. Under normal operation the file server increments its seed every time a new open is done on the pseudo-device, and it puts the seed into the low-order 12 bits of the minor field in the ioFileID returned from FS_OPEN. The I/O server knows about this, and it extracts the seed from the ioFileID so it can restore it during recovery.

```
typedef struct StreamReopenParams {
        Fs_FileID    streamID;
        Fs_FileID    ioFileID;
        int      useFlags;
        int      offset;
} StreamReopenParams;
```

After all the other kinds of I/O handles have been reopened at a server, the client reopens its stream descriptors that reference the I/O handles. The reopen specifies the streamID and the ioFileID, so the I/O server can verify that its shadow stream descriptor connects to the same I/O handle that the client thinks it should. The offset is used to recover the offset in the server's shadow stream descriptor. (This isn't implemented. If the file server crashes while a stream is shared by processes on different hosts, then the shared offset is lost. This needs to be fixed, perhaps by adding an offset to the Fs_IOReply structure so the client can cache the offset.)

**33. FS_RECOVERY** This is used after a client has completed its state recovery. This is needed because the server drops regular FS_OPEN RPCs while a client is doing FS_REOPEN RPCs. Specifically, after a server gets a FS_REOPEN from a client it drops an FS_OPEN (from that client) until it receives an FS_RECOVERY RPC. The FS_RECOVERY RPC can also be used by a client to signal that it is begining the recovery protocol, but this is not necessary. The parameter area of the request message contains a single integer that contains a flag CLT_RECOV_IN_PROGRESS if the client is initiating recovery, and that has no flag (zero value) when the client completes recovery.

```
#define CLT_RECOV_IN_PROGRESS    0x1
#define CLT_RECOV_COMPLETE       0x0
```

**34. FS_DOMAIN_INFO** This is used to get information about a file system domain, including the amount of disk space available. The request parameter area contains the fileID associated with the prefix of the domain. This is returned from the FS_PREFIX RPC. The reply parameter area contains a FsDomainInfoResults structure. The request and reply data areas are empty.

```
typedef struct FsDomainInfoResults {
        Fs_DomainInfo       domain;
        Fs_FileID    fileID;
} FsDomainInfoResults;

typedef struct {
        int      maxKbytes;
        int      freeKbytes;
        int      maxFileDesc;
        int      freeFileDesc;
        int      blockSize;
        int    optSize;
    } Fs_DomainInfo;
```

The fileID that is returned is the user-visible fileID that an application program would see if it did a GET_ATTR_PATH on the prefix. With a pseudo-file-system this is different than the internal fileID associated with the prefix, which identifies a request-response connection between the kernel and the server process. The Fs_DomainInfo indicates the maximum size of the file system, the number of free kilobytes, the maximum number of file descriptors, the number of free descriptors, the native blocksize of the file system, and the optimal transfer size of the file system.

**35. PROC_MIG_COMMAND** This is used to transfer process state between Sprite hosts. The request parameter area contains a process ID and a command identifier. The request data area contains command specific data. The reply parameter area contains a return status, and optionally some command specific data. The migration commands are described below, along with their command specific data.

```
    typedef struct {
            Proc_PID        remotePid;
            int             command;
    } ProcMigCmd;
```

0 PROC_MIGRATE_CMD_INIT

This is used to request permission to migrate to another host. The remotePid field of the ProcMigCmd is NIL if the process is leaving its home node. During eviction, when a process is migrating back home, the remotePid field is the home node process ID. The request data area contains a ProcMigInitiateCmd structure, and the reply parameter area contains the processID for the process on the remote host.

```
    typedef struct {
            int     version;
            Proc_PID        processID;
            int     userID;
            int     clientID;
    } ProcMigInitiateCmd;
```

The version is a process migration implementation version number to ensure that the two hosts are compatible. The processID is the ID of the process that wishes to migrate. The userID is that of the owner of the process. The clientID is the Sprite hostID of the host issuing the request.

1 PROC_MIGRATE_CMD_ENTIRE

This transfers the process control block. The request data area contains an encapsulated control block. The exact format of the encapsulated control block is machine specific and will not be described here. The reply data area is empty.

2 PROC_MIGRATE_CMD_UPDATE

This is used to update the state of a migrated process. The request data area contains an UpdateEncapState structure, which contains the few fields of a Sprite process control block that a process can modify.

```
typedef struct {
        int     familyID;
        int     userID;
        int     effectiveUserID;
        int     billingRate;
} UpdateEncapState;
```

3 PROC_MIGRATE_CMD_CALLBACK
Not used.

4 PROC_MIGRATE_CMD_DESTROY
This is called to kill a migrated process. The request and reply data areas are empty.

5 PROC_MIGRATE_CMD_RESUME
This is called to continue execution of a suspended migrated process. The request and reply data areas are empty.

6 PROC_MIGRATE_CMD_SUSPEND
This is called to suspend execution of a migrated process. The request and reply data areas are empty.

**36. PROC_REMOTE_CALL** This is used to forward a system call from a migrated process back to its home node. Most system calls are not forwarded, only a few that depend on state maintained at the home node. The format of the request and reply are implementation and system call specific, and are not described here.

**37. PROC_REMOTE_WAIT** This is used when a migrated process waits for child processes. Communication with the home node is required because synchronization with process creation, process exit, and waiting is done there. The parameter area contains a ProcRemoteWaitCmd structure, and the request data area contains an array of processIDs on which to wait. The reply parameter area is empty and the reply data area contains a ProcChildInfo structure. (Byte ordering isn't an issue in the data area because process migration only works between hosts of the same machine architecture.)

```
typedef struct {
        Proc_PID        pid;
        int     numPids;
        Boolean         flags;
        int     token;
} ProcRemoteWaitCmd;

typedef struct {
        Proc_PID        processID;
        int     termReason;
```

```
    int   termStatus;
          int       termCode;
          int       numQuantumEnds;
          int       numWaitEvents;
          Timer_Ticks   kernelCpuUsage;
          Timer_Ticks   userCpuUsage;
          Timer_Ticks   childKernelCpuUsage;
          Timer_Ticks   childUserCpuUsage;
} ProcChildInfo;
```

**38. PROC_GETPCB** This is used to return the process control block of a migrated process for implementation of the **ps** (process status) application program. The request parameter area contains an integer with value GET_PCB (0x1) or GET_SEG_INFO (0x2). With GET_PCB the request data area contains the processID (also an integer). The reply parameter area contains a Proc_PCBInfo structure, and the reply data area contains the argument string of the process. The Proc_PCBInfo is described in ‘‘/sprite/lib/include/proc.h’’. With GET_SEG_INFO the request data area contains a virtual memory segment number, and the reply parameter area contains a Vm_SegmentInfo structure, which is described in ‘‘/sprite/lib/include/vm.h’’.

**39. REMOTE_WAKEUP** This is used to notify a remote process that some event has occurred. The process has presumably registered itself via some blocking call such as FS_READ or FS_WRITE, whose parameters include a Sync_RemoteWaiter structure. The request parameter area of REMOTE_WAKEUP contains a Sync_RemoteWaiter structure, and the request data area, reply parameter area, and reply data area are empty. Note that this wakeup message can race with the process’s decision to wait at the other host. To foil the race condition a process must be marked as in the process of deciding to wait. In the Sprite implementation, this is done by clearing a *notify* bit kept in the process’s control block. When a REMOTE_WAKEUP RPC is received by a Sprite host, the notify bit in the process control block is set. Before actually blocking a process (in response to a FS_WOULD_BLOCK return code) the Sprite kernel checks that the notify bit has not been set asynchronously via this RPC. If the bit has been set, then the process is not blocked and it retries its operation immediately. If this technique were not used then notifications might get lost and hang the process.

**40. SIG_SEND** This is used to issue a signal to a remote process. The request parameter are contains a SigParams structure. The request data area, reply parameter area, and reply data area are empty.

```
typedef struct {
        int       sigNum;
        int       code;
        Proc_PID      id;
        Boolean       familyID;
        int    effUid;
    } SigParms;
```

The sigNum is a Sprite signal, and the code field is used to modify this. The id is a process identifier if the familyID field is zero, otherwise id is a process group identifier. The effUid is the effective user ID of the signaling process, and this is used to verify permissions.

**41. FS_RELEASE_NEW** This is used by the I/O server during process migration to tell the source of a migrated process that it can release an I/O stream that had been associated with the process. Recall that fork() and dup() create extra references to a stream descriptor, so this call is used to release that reference. This cannot be done safely at the time the process leaves, so it is done as a side effect of the FS_MIGRATE RPC issued from the destination client. At this time the current offset in the source client's stream descriptor is also returned to the I/O server, in case it needs to be cached there while the stream is shared by processes on different hosts. The request parameter area contains the ID of the stream that migrated, and the reply contains an inUse flag and the current offset. The inUse flag should be set if their are still processes on the source client that reference the stream descriptor.

```
    typedef struct {
            Fs_FileID streamID;
    } FsStreamReleaseParam;

    typedef struct {
            Boolean     inUse;
            int     offset;
    } FsStreamReleaseReplyNew;
```

## 42. Bugs and Omissions

This specification is based on the Sprite implementation as of Fall 1989. There are a couple of known bugs in it, and it is a bit crufty. However, there is a lot of inertia behind the network interface because changing it requires coordinated changes on all Sprite hosts. Future changes to the interface will ideally be backward compatible with this interface by introducing new RPCs that fix certain bugs, while retaining the original for compatibility with hosts running older versions of Sprite. The known bugs in the interface are summarized below.

**42.1. FS_REMOTE_LINK vs. FS_SYMBOLIC_LINK** The Fs_ReadLink (or UNIX readlink) system call is implemented as an FS_OPEN followed by an FS_READ. The type field in the Fs_OpenArgs is specified as FS_REMOTE_LINK in the current implementation, but the server should also allow regular symbolic links to be opened.

**42.2. Device Attributes** Currently, while the I/O server maintains the access and modify times for a device while it is opened, this information is not pushed back to the file server when the device is closed.

**42.3. Pathname Redirection** There is some cruft in the way pathnames are returned from the server. In some RPCs there is an extra 4 bytes in the data area that precedes the pathname, but in the Attributes RPCs the padding is gone. This is a hold-over from pre-byteswapping days when the 4 bytes in the data area contained the prefix length. Similarly, with the FS_RENAME and FS_LINK, there are 8 bytes of junk before the returned pathname.

**42.4. FS_SERVER_WRITE_THRU** This flag is currently private to the client side of the implementation. It could be passed through to the server to force a write-through to disk. Currently, however, the client and server writing policies are completely independent. Ordinarily clients uses a 30 second delay, and servers use write-back-ASAP. This means that a file ages in the client's cache for 30 seconds, and then gets scheduled for a disk write-back after the last block arrives from the client. Note that a client can use fsync(), in which case the blocks are forced through to the servers disk, and fsync() doesn't return until after that has happened.

# Index

# List of Tables