CARNEGIE-MELLON UNIVERSITY

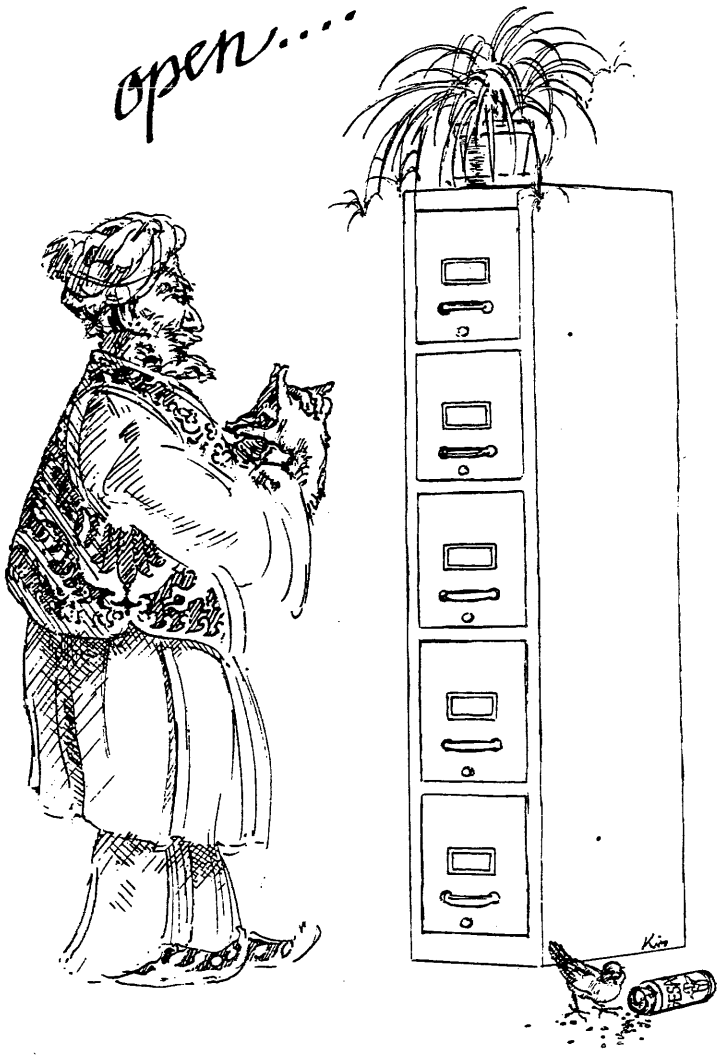
DEPARTMENT OF COMPUTER SCIENCE

SPICE PROJECT

Sesame: The Spice File System

Draft

Michael B. Jones
Mary R. Thompson
Richard F. Rashid



24 August 1984

Spice Document S170

Location of machine-readable file: [cfs]/usr/spicedoc/aug84/program/sesame

Abstract

Sesame provides several distinct but interlated services needed to allow protected sharing of data and services in an environment of personal and central computers connected by a network. It provides a smooth memory hierarchy between the local secondary storage and central file system. It provides a global name space and a global user authentication protocol.

This design evolved from the earlier Network-Based Central File System design document in Technical Report CMU-CS-80-134 written by M. Accetta, G. Robertson, M.Satyanarayanan, and M. Thompson. Gene Ball, Greg Harris, Peter Hibbard and others at CMU have also contributed to this document.

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Projects Agency or the U.S. Government.

Table of Contents

1. Overview 1
1.1 Status 1
1.2 Environment 1
1.3 Goals 1
2. General Design 3
2.1 Authentication System 3
2.1.1 Overview 3
2.1.2 Establishing a connection 3
2.1.3 Proving identity 4
2.1.4 Passwords and Secure Connections 4
2.2 Authorization System 5
2.2.1 Overview 5
2.2.2 Access Groups 5
2.3 File System 7
2.3.1 Overview 7
2.3.2 Implementation and use of files 7
2.3.3 File System functions 8
2.4 Name System 8
2.4.1 Overview 8
2.4.2 Structure of a directory 10
2.4.3 Versions 11
2.4.4 Name syntax 12
2.4.5 Canonical form of names 13
2.4.6 Symbolic Links 13
2.4.7 Links and Copies 14
2.4.8 Access control 15
2.4.9 Directory access rights 16
2.4.10 Deleting Names 17
2.4.11 Recovering expunged names 17
2.5 Migration System 17
2.5.1 Overview 17
2.5.2 Migration policy 18
2.5.3 Functions of the Migration Server 18
3. Primitive Operations 19
3.1 Common Characteristics 19
3.2 Authentication Primitives 20
3.2.1 Primitives for establishing and breaking a connection 20
3.3 Authorization Primitives 23
3.4 Data transmision primitives 28
3.4.1 File I/O primitives 28
3.4.2 File header manipulation primitives 32
3.5 Name Manipulation Primitives 34
3.6 Name Policy Primitives 44
3.7 Access Control primitives 46
Acknowledgements 51
A. File header fields 52

B. Directory fields 53
C. Format of the Group Attribute List 54
D. Using Sesamoid, the first version of Sesame 55
E. Summary of Primitives 56
References 59

1. Overview

1.1 Status

The Sesame system (Name, File and Authorization Servers) will become part of the Spice environment by the end of 1984. Until then, the Sesamoid File Server provides a subset of the Sesame Name and File server calls and an interim Authorization Server provides a list of authorized system users.

This document describes the Sesame system. The only calls that are currently provided by Sesamoid are the ones not marked with an asterisk (*) in the Primitives summary at the end of the document. Read this document if you wish to understand the direction in which the Spice file facilities are evolving. If you are simply looking for the calls that you can currently use, read the section on Sesamoid and the Authorization Server in *Servers* document.

1.2 Environment

- User community of 200 to 300 faculty, staff and graduate students.
- Many members of the community have personal Spice machines with at least the power of a Perq and a reasonable amount of secondary storage, e.g. 24 megabytes.
- Some Spice machines may be shared amoung a small number of users.
- Some Spice machines may be available for public use.
- All the Spice machines in the environment are connected by one or more local networks.
- Also on the local network are two or more secure central machines with larger amounts of secondary and archival storage. E.g. 300 megabytes secondary storage, and as many dismountable 300 megabyte disk packs as necessay for archival storage.

1.3 Goals

- Secure and reliable storage and migration of users' files on local secondary storage, central secondary storage and archival storage.
- Protected sharing of files and services amoung users.
- Uniform name space for data and services.
- Uniform access to data regardless of what level of storage it is on.
- Very high availability of basic system (no system-wide downtime).

- Graceful expansion to a potential user community of over 10,000.

2. General Design

This section describes the functions of the various subsystems which belong to Sesame. These subsystems represent functions that are grouped together mainly because of the commonality of the data they use. The user interface documented in chapter 3 is grouped according to more logical areas of functionality, and so is not exactly parallel to this section.

2.1 Authentication System

2.1.1 Overview

The Authentication System has the responsibility of authenticating a user's identity by checking a login name and password against the list of recognized users and their valid passwords. Since only software running on the central service machines is secure, only Authentication Servers running on the the central machines can be trusted to provide authentication between users on different machines. Local Authentication Servers can be used to provide authentication between processes on the local machine. *Login* and *registration* requests made to the Local server are normally also reflected to the Central Authentication Server. The local Authentication Server may maintain a cache of users and their group access lists, to be used in case the local machine is detached from the central system.

Since the traffic to the Authentication System is not particularly heavy, especially the requests that modify the data base, and since it is essential that an Authentication Server is always available, there will be Authentication Servers running on at least two of the central machines. The data base that is used by the Authentication Server will be duplicated on each machine that is running a server. Thus a *verification* request can be handled by either server, but a *login* or *registration* request must be reflected to all the servers.

2.1.2 Establishing a connection

In order to use the services provided by Sesame, a user must first establish a secure connection with the Authentication Server. This connection is established by sending a login message, consisting of a Sesame user name and an encrypted login code (see 2.1.4), to a public port of the Authentication Server. Depending on the local authentication policy, this login connection may be established explicitly at the request of the user or implicitly when a user logs in to the local machine. As a result of a successful login, a user service port to the Authentication Server is established and returned, and the message reply port registered so that it can be used as subsequent proof by the user of his identity. The Authentication Server defines the identity associates with each service port to be a list of the access groups (see section 2.2.2) of which the user is a member.

The connection to the Authentication Server is terminated by sending a logout message to the user service

port. This deallocates the service port, deallocates any sublogin service ports, and deregisters any other ports that these service ports may have registered.

The Authentication connection may also be broken implicitly by events such as a time-out due to local machine crash. In these cases too, the connection is terminated appropriately.

2.1.3 Proving identity

Many other servers need to have a way to identify a processes that is sending a request for service and to associate its user with class of users that has specific rights to objects or services. The Authentication Server provides a uniform mechanism for this purpose. Once a user has a service port connection to the Authentication Server, he may create and *register* ports that belong to him. He can then send a *registered port* to another server as a token of his identity. The server then *verifies* the *registered port* via its own connection to the Authentication Server and receives back the name and Access Group list of the user that registered the port.

If a user wishes to operate with a restricted set of rights, he can perform a restricted login through his original Authentication Server connection. This *restriction* request returns a new Authetication service port associated with an identity that is a subset of the user's complete set of access groups.

2.1.4 Passwords and Secure Connections

Because the network is considered a public medium, some care is taken not to transmit secrets such as passwords in clear text. The general mechanism is to use a secure connection between network servers. The sender turns on the "secure" bit in the message type word, which tells the network server to send the message encrypted. Each network server must know the correct encryption key for talking to every other network server.

Shortly after the Spice machine comes up, the owner of the machine logs in to the central system. This login message (*SendLogin*) is sent before a secure channel is established, since it is part of the handshake between the new host and the central,secure hosts. The local system chooses a random encryption key. The login message consists of the user's name in plain text and the name and encryption key encrypted with the password that the user has typed in. Passwords are stored by the Authentication Server on a secure machine. *SendLogin* uses the password to decrypt the login code and find out the encryption code. The presence of name in the encrypted message provides enough redundancy for the Authentication Server to know that the message was actually encrypted with the correct password. Thus passwords act as the initial encryption key for communication between the user and the Authentication Server. Now both parties to the central login can inform their local network server of the new network server's encryption key without ever sending that key or

the password in the clear. Subsequent logins from the same host will only establish authentication service ports; the network servers will ignore the second generation of encryption keys.

2.2 Authorization System

2.2.1 Overview

A user gains rights to objects and services of Sesame by being a member of one or more *Access Groups* as defined in section 2.2.2. It is the responsibility of the Authorization System to maintain all the access group lists. As in the case of the Authorization System, the software implementing this system must be secure and always available, but is not too heavily used. Thus the Authorization System is implemented by servers running on at least two of the central machines, all using the same replicated data base. There is no local Authorization Server, so a user on a detached Spice is unable to call any of the authorization primitives.

2.2.2 Access Groups

An access group defines a subset of users to whom privileges may be collectively awarded or from whom these privileges may be collectively revoked. Access groups are partitioned into two classes: *primary access groups* and *secondary access groups.* Each user of Sesame is the sole member of exactly one primary access group. Since there is a one-to-one relationship between primary access groups and users, the terms "user" and "user's primary access group" will be used interchangeably in this document. Secondary access groups are created by users and differ from primary access groups in two ways:

1. Secondary access groups may not login, and
2. A secondary access group may contain more than one user.

A user may belong to any number of secondary access groups, and at any instant of time, the protection environment of a user is defined by the union of the rights of all the access groups of which he is an *authenticated* member at that instant. The user may establish a restricted authentication to deliberately deprive himself of certain rights; for example, while debugging a file manipulation program, a user may not wish to have *Delete* privileges on a sensitive file.

Associated with every access group is a data structure called the *group attribute list* or GAL (see appendix C). The GAL contains all relevant information about that access group, and is uniquely identified by an integer called the *access group ID.*

Every access group also has a name that identifies it uniquely. In the case of a primary access group, this is the name by which the corresponding user logs in to the Authentication Server, and in the case of a secondary access group, this is the name by which that access group is referenced. Access groups may be renamed;

however, their IDs cannot be altered. Every user is automatically a member of the group called ALL, which consists of all Sesame users. When a user logs in to the Authentication Server, the authentication service port returned to him has associated with it a list containing all the access groups of which the user is a member.

While primary access groups are restricted to having exactly one member, there is no restriction on either the number or type of members of secondary access groups. One implication of this generality is that it is possible to construct a hierarchy of secondary access groups. This is done by defining a secondary access group to consist of a set of other secondary access groups (or a mixture of primary and secondary access groups). Thus, if A and B are two access groups, one can define C to consist of these two access groups. Every member of either A or B is automatically a member of C. Adding or deleting a member from A or B thus implicitly adds him to C or deletes him from C. The relationship *"is a member of"* is thus a transitive one. Consequently, the reflexive transitive closure[1] of this relationship applied to any access group yields a list containing every member, direct or indirect, of that group. Thus, the set of rights that a user has on any entity is the union of the rights that his access groups have on that entity.

This construction of the transitive closure (referred to as "flattening") of a user's access list is done at login time. The flattened access list is associated with the port to the Authentication Server and is returned to any server who asks for verification of a user's *registered port.* The Name Server gets the access group list for a user by a *Verify* request when a connection is first established.

Primary access groups can only be created and deleted by the System Administrator whereas secondary access groups can be created and deleted by users. An access group's attributes are protected against unauthorized modification by an access control list mechanism. Each access group has an access control list associated with it. This list consists of a set of access group IDs and the rights that each of them has on that access group. The set of access rights are:

AddMembers members may be added to the group

RemoveMembers members may be removed from the group

DeleteGroup the group may be deleted

ChangePassword the password of the primary group may be changed (?)

ChangeFullName the (name and) full name of the group may be changed

GetAuxAccess the access control list for the group may be read

---

[1]For brevity, the phrase "transitive closure" will mean "reflexive transitive closure" in the rest of this document.

SetAuxAccess the access control list for the group may be modified

2.3 File System

2.3.1 Overview

The basic function of the File System is to allow the storing and retrieving of files on one's local disk and/or the central file systems disks. A file is simply a collection of constant data that can be named by a single unique identifier. Thus a unique file identifier, (*FID*), refers to a unique collection of data. Possession of a *FID* logically implies the possession of the data. Writing a file causes a new *FID* to be generated. If necessary, pages of data may be shared between files in order to implement minor modifications of large files efficiently. The maximum size of a file is 4 gigabytes.

The constraints on the File System services are the opposite of those on the Authentication and Authorization Servers. It is expected that the service load on the File System will be quite heavy and the security requirements not so severe. In fact, File System security need only be insured to a level equivalent to the reliability the storage media on which the data is kept. In order to store a very large number of files and to be able to retrieve them quickly, the global file space is spread among all the machines on the network. In fact, copies may be kept on many different machines for files that are used frequently.

A File Server is running on each machine that has secondary storage and that server has primary responsibility for all the files on its storage medium. The servers running on local Spice machines are referred to as Spice File Servers, those on the central machines controlling the main large capacity secondary storage media are called the Central File Servers, and those on the central machines controlling the tertiary storage are called Archival File Servers. All of these servers are running the same basic software.

2.3.2 Implementation and use of files

A File Server keeps a table of the FIDs of all the files on its disk, and can return data for only those files. The FID consists of a random, unique part and a hint part that specifies on what machine/server the file might be found. If a File Server fails to find the file locally it forwards the request to the Migration Server who uses the hint portion of the FID and perhaps a local database to begin searching for the file on other machines. The Migration Server maintains a policy on how far to search for a file. For example, whether or not to go to other insecure Spice machines, whether or not to go to the Central File Servers and whether or not to go to the Archival File Servers.

In order to efficiently implement large, sparse files, the File Server can recognize areas of a file or virtual memory that have never been referenced or have been explicitly cleared. These areas of a file take no space on disk or in physical memory, but logically contain all zeroes.

The File System interface assumes that both it and its users have available a large amount of virtual memory and demand paging such as the Accent Kernel [2] provides. Unlike a more conventional file system, the only way to modify file data is:

- To read the **entire** file into virtual memory, (this is a mapping operation and no data transfer takes place).
- Make the desired data modifications in virtual memory, (this creates new pages containing the modified data).
- Write out the entire piece of virtual memory (this writes out only the modified pages and creates a new file consisting of those pages plus the non-modified pages of the original file).

2.3.3 File System functions

The basic function of the File System is to take a *FID* and return the data that it references, and to put new data from virtual memory into a file and to return the new *FID*. The file system must be able to retrieve and store data both on its local disk, and/or forward the data to another file server which controls the storage on another disk. Files on the local disk are stored as permanent Spice segments. Thus all direct disk I/O is done by the Accent Kernel and not by the File System.

The File Server functions are:

- To locate and return files stored on the local disk.
- To request the Migration Server to find files that are not currently on the local disk.
- To keep track of the total disk space that is used and to request the migration server to remove files from the local disk when it is getting full.
- To maintain some additional information about files on the local disk, such as file size, creation, access dates, author ID, local reference count, etc.

2.4 Name System

2.4.1 Overview

The Name System provides a global name space for all objects or services intended to be available to users of the Spice environment. As in the case of the File System, the Name System must maintain a large data base and respond quickly to requests for service. As a result of these requirements there is a Name Server running on each machine, and the local Name Server should be able to satisfy the majority of requests that come from local processes. The global name space is partitioned into disjoint sections, with only one Name Server at a time having responsibility for each partition.

The Name System provides four primary functions:

1. It maps names chosen by users to unique identifiers.[2]
2. It provides a directory structure that aids users in organizing their files in a logical manner.
3. It aids sharing of files by supporting shared directories and provides control over the extent of this sharing.
4. It maintains type information on all named objects.
5. It provides access control on all named objects.

These services are provided independently of the services provided by the File System. Consequently, it is possible to use the name space as a repository for names that have nothing to do with files. One such use, described in section 2.4.2, is to provide a network-wide naming scheme for IPC ports. Other uses of the name space are left to the imagination of the reader and the ingenuity of users.

The global name space provided by Sesame appears to the user as a hierarchy of directories, similar to that in the UNIX file system [4]. This directory hierarchy can be viewed as a rooted graph with non-terminal nodes corresponding to directories and terminal nodes corresponding to names of entries in directories. An arbitrary path from the root to a leaf specifies the entry name at the leaf unambiguously (but not necessarily uniquely). Similarly, an arbitrary path from the root to an internal node unambiguously specifies a directory. Such a specification of a file or directory name is called an *absolute pathname.*

The global name space is maintained by several secure Central Name Servers on central machines, and by the less secure Spice Name Servers on the Spice machines. In order to share the work among many server processes the name space is partitioned along subtree boundaries. This partitioning is done dynamically. That is, a Name Server may check out a subtree from the current maintainer if allowed, and can return the subtree when it can no longer maintain it. This grants to that Name Server the exclusive right to make modifications in that portion of the name space. There is an index of Name Servers and the (absolute) pathnames of the directory subtrees that they are currently maintaining. When a given Name Server is unable to find a pathname in its domain, it can check the index and forward the request to the Name Server for the subtree, if any, on which the pathname should be found. Names in those domains that are being maintained by Central Name Servers are findable by all users of the system. Finding a name in a domain that is checked out to Spice Name Server is more probabilistic, since software running on personal machines can never be fully trusted. Thus the objects that are intended to be shared between users should be kept in directories that

---

[2] In most cases this can be interpreted as "translates the name of a file to its FID."

are always maintained by one of the Central Name Servers. Files that are primarily used by one user may in kept in directories that can be checked out to the Spice Name Server on that user's Spice machine.

2.4.2 Structure of a directory

Conceptually, a directory is a list of entries, each entry being a mapping from a *entry name* to a typed *entry value*. In the great majority of cases the *entry value* will be of type *File*. This represents the simplest situation, where a name in a directory is directly mapped to the FID of a file. Since entries are merely mappings, it is perfectly possible and meaningful for a given file to have many names associated with it, in the same or different directories.

Subdirectories have entries of type *Directory* in their parent directories. When parsing a pathname which contains a subdirectory name, the Name Server must first check to see that the pathname resides in its domain. If the name is not in the local domain the Name Server index is checked and the request is forwarded to the appropriate Name Server. If the name is in the local domain the Name Server recursively searches down the directory hierarchy until an entry which is not of type *Directory* is encountered. At this point, with the one exception described below, the corresponding entry value is returned as the result of the name lookup.

A second kind of entry is of type *File*. Here, the entry in the directory is simply a File ID, thus the name is simply a mapping to that particular File ID. Normally file accesses are done by name rather than by ID, since access control and migration policy is done on the basis of names. If the user has the right to do a *LookupName* and obtain the File ID from the name however, he can then read the file by ID.

A third kind of entry is of type *Symbolic Link*. A symbolic link is the absolute pathname of an entry in some directory and is used like a macro in a programming language. When the Name Server encounters a symbolic link while trying to resolve a name reference, it continues the search using the pathname corresponding to the symbolic link, thereby effectively performing a macro expansion. This may be applied recursively. See section 2.4.6 for more details.

Another type of entry value is *IPC port*. This feature is used to provide a simple name, IPC port mapping service so that users who wish to share IPC ports can remember names rather than IPC ports. Looking up an agreed-upon name in the global name space will yield an IPC port which can then be used for whatever purposes are necessary.

To assist users who wish to use the Name Server for non-standard purposes, a special range of user-defined entry types is recognized by the Name Server. Entries of these types are 256 byte records whose values are

neither checked nor interpreted by the Name Server. Entries of this type may be entered or looked up in directories just as entries of type *FID*. It is expected that higher-level mechanisms will check and interpret these entries suitably.

2.4.3 Versions

The naming mechanism supported by the Name Server reflects the expectation that most users will view individual files and possibly other objects as being members of families rather than as isolated entities. For instance, modifying existing files will typically lead to the creation of a sequence of files, closely related yet distinct from each other. These files are certainly distinct, yet they only represent snapshots in the life history of what would be called a file in conventional file systems.

The Name Server captures this view by composing a full entry name from two parts: a *family name* and a *version number*. The family name is a text string that identifies a family of entry names. Version numbers are integers used to distinguish between members of a family. The version numbers of the members of a family reflect the chronological order in which the members were entered into the directory.

It is often the case that a user wishes to refer to a particular distinguished version of a name without bothering to remember or find out what its version number is. Moreover, the particular version number desired is influenced by the nature of the call. The Name Server therefore adopts the convention that a name without a version number refers to a distinguished version appropriate to the specific call. The set of distinguished versions recognized are as follows:

Low	the least non-deleted version number of a name currently in the directory
High	the greatest non-deleted version number of a name currently in the directory
New	a new version one greater than the greatest (deleted or non-deleted) version in the directory is to be created
All	for calls that allow wildcarding, this distinguished set of versions is meaningful

For most calls a versionless name refers to the highest version. For those calls which enter a new name such as *EnterName* and *WriteFile*, an unspecified version results in a new version number one greater than the highest. Finally, in the *DeleteName* call, the default version number is the lowest. Bear in mind that the user may always override the default version number by either providing a specific positive integral version number, or by explicitly requesting one of the above distinguished "versions".

Directories and symbolic links are both exceptions to the normal naming scheme in that they may have only family names, and no version numbers. Keeping multiple versions of directories would vastly complicate the

global name space and the name syntax without adding much reliability. A name archive is kept instead for each directory. In the case of links, the complexity of the name syntax required to determine if a version number should be applied to the link name or to the expanded object name was too complicated to be worth the small increase in recoverability of keeping old versions of links around. (See sections 2.4.4 and 2.4.6 for more details.)

A final caveat on versions: the proper use of version numbers is up to the user. A malicious (or merely foolish) user may enter totally unrelated files as versions of a family. Version numbers will then be of dubious value.

2.4.4 Name syntax

An *absolute pathname* is written as a "/" followed by the entry names, in order of traversal, of nodes encountered while traversing that path, with the "/" character serving as a name separator. Entry names accepted by the Name Server may be of arbitrary length and may contain any of the ASCII characters in the set { letters, digits, "_", "-", "$", "." }. The version number (if specified) follows the terminal entry name, and is delimited by a "#" character. The version number may be either a positive decimal integer, or one of the keywords: `Low`, `High`, `New`, or `All`. The character "*" is an abbreviation for the keyword `All`.

Name matching *not* case distinctive. The original case is however, preserved. Successive enters of names which are equivalent (other than case) cause additional versions to be created, with the case being taken from the original version.

Any reserved 7-bit ASCII character (including letters, for matching purposes) may be included in a name by quoting it with a single quotation mark (" ' "). For instance, I may enter a name with a question mark in it by preceding the question mark with a quote character. Likewise, if I want to enter a name which differs from an existing name only by the case of letters, I can do this by quoting the differing letters. Once two names differing only by case have been entered, they can only be matched by quoting all of the ambiguous letters, or by entering the case of each letter exactly.

Two primitives, *ScanNames* and *ScanArchivedNames* permit wildcarding in the terminal component of the absolute pathnames that they accept as input. Two forms of wildcarding are recognized within entry names. A "*" character in a name will match zero or more arbitrary ASCII characters. A "%" character will match exactly one arbitrary ASCII character. Both forms of wildcarding may be used within one name. Additionally, the `#All` option performs version number wildcarding.

Two special names are recognized in every directory. They are "." and "..". Rather than being entered into

the name space, these names are actually macros which expand in pathnames. The name "." merely refers to the directory itself. The name ".." refers to the directory's parent directory. Usage of ".." in the root directory is illegal.

Some examples of legal pathnames are:

```
/Root/Spice/Source/Oregano/Oregano.Pas
/User/Jones/XYZ/abc...xyz#8930
/User/Jones/Finger.Plan#Low
/Spice_System/Mail_System/EtherNet/Mailer.Run
/Root/Spice/Source/Nutmeg/%%%Nut*.Pas#All
/'Unix/'Names/'A'R'E/case/'Di'St'In'Ct'Iv'E.
/User/Jones/Source/../Binary/
```

Note: While the Name Server itself requires full paths from the root directory to the terminal nodes to be specified for all operations, the user will in most cases not be burdened with this chore. User-interface code will provide such abstractions as a *current directory* which specifies initial path elements of a name, as well as *search lists*, *logical names*, and other name-related functions. Thus, the user will usually be able to specify abbreviated pathnames relative to the current directory or an explicit logical name. Certain programs, it is presumed, will also choose to use an implicit search list when looking up names.

2.4.5 Canonical form of names

When absolute pathnames are returned from the name server to the user, they are always returned in a canonical form. This form is as follows:

- Symbolic links are expanded (unless being returned as the terminal component of a pathname).
- Version numbers are returned as decimal integers with no leading zeros following the terminal pathname component and version delimiter. Directories have a trailing directory delimiter. Symbolic links have no punctuation following following their terminal component.
- Any extraneous quote characters are removed. Letters are never returned quoted, although quoting them may be necessary to achieve an unambiguous match.
- All characters are returned as 7-bit codes (no "meta-bits" set).

2.4.6 Symbolic Links

As already outlined in section 2.4.2, symbolic links may be entered into directories, and serve as macros which expand into other names when they are used. The value of a symbolic link must be a versionless absolute pathname. The actual semantics of symbolic links differ slightly depending upon their use. The usual rule is that whenever a link is encountered in a pathname, either as an imbedded or terminal component, that the link is expanded. This means, for instance, that ordinarily an *EnterName* (and

consequently *WriteFile*) which references a symbolic link's name will enter a new version of the name which the link points to, and not make an new entry under the link's name. Similarly, a *DeleteName* will ordinarily delete the name pointed to by the link, and not the link itself. This behavior is consistent with both Multics [1] and UNIX [5] symbolic links.

There are however, cases where one needs to talk about a link itself, and not the object pointed to by a link. Forms of *TestName*, *LookupName*, *EnterName*, the friends of *EnterName* such as *DeleteName*, *Rename* and *CopyName* are provided to override the default behavior, and actually reference links themselves rather than referencing through links. Finally, since *ScanNames* is an operation on an entire directory and not on individual names, any symbolic links in a directory are returned as such, and are not expanded.

As already stated in section 2.4.3, symbolic links may not be entered with version numbers. Thus, a version number following a name which is a symbolic link is applied to the macro-expanded name, and not to the link's name. Whenever a link itself is being referenced without being expanded, a version number should not be specified.

2.4.7 Links and Copies

Although an argument can be made that one might want to have links where the default behavior was to "write on" rather than "write through" links, Sesame chooses the "write through" behavior for several reasons. The most effective use for "write on" links is to "shadow" a subsystem in another directory, allowing a user to keep copies only of the components which he has changed. While this is a valid use, Sesame provides equivalent semantics to "write on" links through the use of the *CopyName* primitive.

CopyName makes a copy of the entry for an existing name under a new name. In the case of files, for instance, the file id would be copied and entered under the new name as well as the old. Note that this does not make a copy of the file data, it only makes two names for the same file. Then any subsequent *WriteFile* requests would enter a new version of the written name, but leaving the copy under the other name untouched. Compare this to the behavior of making a link to the file instead of using *CopyName* to copy the name's file id; when a *WriteFile* is performed to the link's name, the link is expanded to be the original family name, and a new version of the original name is entered. Thus unlike the *CopyName* case, links cause both names to refer to the changed version. Thus Sesame effectively provides the semantics of both "write through" and "write on" links.

2.4.8 Access control

Corresponding to each name is an associated *access control list*, which is a mapping from access groups to access rights on the object. When a user requests an operation on an object, the Name Server examines the entries corresponding to each of the user's access groups in the access control list on that name. If any of these access groups possesses sufficient rights to perform the requested operation on the object, the access check succeeds and further processing of the request occurs. Otherwise access is denied.

The access rights for a particular access group are defined by a collection of bits describing the operations which may legally be performed on the object by members of that access group. The set of access bits for a file is divided into *system access bits* and *auxiliary access bits*. Logically, one can view a file as having two access control lists: system and auxiliary. System access bits are interpreted by the Name Server. Auxiliary access bits are stored with the name but normally have no meaning to the Name Server; they are interpreted only by the programs or subsystems using them. The Name Server and the Authentication Server use auxiliary access bits on directories and GALs to implement their own access checks.

The system access bits are:

Supersede a new version of this name may be added

Delete the entry name and object may be physically deallocated. This right is also implied by *DeleteNames* permission in the name's directory.

Visible the name will match a wildcarded pathname. If this bit is not set the name may still be used, but will not be returned by the *ScanNames* call unless the user has *ReadNames* permission on the name's directory.

Lookup the value, e.g. FID or port, of the name may be returned.

Copy the value, e.g. FID or port, of the name may be copied to another name. (In most cases, the *Lookup* right provides at least the equivalent capabilities.)

Read the object entered under the name may be "read". For a file this right allows the user to read a file by name, even though he may not have the *Lookup* right necessary to retrieve the file's File ID from the name. For a link, this right allows the user to use the link's pathname value to continue the name resolution.

GetSysAccess the system access control list may be examined

GetAuxAccess the auxiliary access control list may be examined

SetSysAccess the system access control list may be modified

SetAuxAccess the auxiliary access control list may be modified

2.4.9 Directory access rights

The current implementation of the Name System controls access to directories as well as to the individually named objects. The Name Server uses the auxiliary access bits of a directory to define a set of directory access rights. These directory access rights are:

CreateNames new names may be added to the directory

DeleteNames names may deleted from the directory (see section 2.4.10)

ReadNames the directory names and name types may be examined

GetDefAccess the default directory access control list may be examined

SetDefAccess the default directory access control list may be modified.

CheckoutDir the directory may be placed in the exclusive domain of this user, so that all update and search requests must be routed through his Name Server.

Access checks on directories are performed by the Name Server before carrying out a user's request.

Unlike conventional file systems, there is no notion of "ownership" of a directory. All access control is done via the access control list of the directory, and this list may be used to give a set of users equal rights on the directory. In some sense one can say that the "owners" of a directory are the access groups which have all rights on the directory. Thus, a set of users may collectively "own" a directory and share its use.

To assist in the sharing of files, every directory contains system and auxiliary *default access control lists.* The purpose of the default access control lists is to automatically provide an access control lists for names entered in that directory. When a name is entered in a directory the access control list is set in one of the following ways.

- If no previous version of the name exists, the default access control list is used.
- If a previous version exists, the access control list from the highest existing (non-deleted) version is used.
- The user may subsequently call *SetAccess* to change the access control list.

A set of users having access to a common directory may set up the default access lists so that the mere entering of a name in that directory causes the corresponding file to be accessible to everyone in that set.

2.4.10 Deleting Names

In order to allow easy recovery from the not uncommon user mistake of deleting the wrong names, a delete name request is implemented as a delayed action. When the delete request is made, the name is flagged in the directory as deleted and a request is sent to the Migration Server to queue the name and object for future removal. If the name remains deleted until the removal request comes due, then the name is removed from the directory and the object is removed from the local disk if there is no other name referring to it.

If a subsequent undelete of the name occurs before the removal time arrives, the delete flag is reset and it appears as if the name was never deleted at all. To force immediate removal of deleted names an *ExpungeName* operation on the directory is provided. Once removed, the corresponding entry name and entry value associations can only be recovered from the Name Archive (see section 2.4.11). Users will, in general, never directly use the *ExpungeName* operation except for in unusual circumstances.

2.4.11 Recovering expunged names

Corresponding to every directory is a Name Archive. This is a data base to which only the Name System has direct access. It is partitioned in a manner parallel to the subtrees of the global name space so that each Name Server maintains matching domains of the name space and name archive space. Each time a name is entered in a directory, it is also entered in the corresponding Name Archive, along with its entry value and the date and time of entry. The Name Archive is never deleted or truncated, even if the corresponding directory is. The Name Server supports a name retrieval service which can be used to recover the entry value that corresponded to a given pathname on a certain date and time. Since directories themselves have entries in their parent directories, this recovery mechanism will work even if some of the directories present in a pathname no longer exist. To use this retrieval service, a user must possess *ReadNames* rights on the directory from which the pathname is to be retrieved.

2.5 Migration System

2.5.1 Overview

The primary resposibility of the Migration System is to smoothly move files between the various File Servers on the Spice, Central and Archival machines. This should be accomplished with a minimum of explicit user control, but must be controllable by general policies set by the administrators of the specific machines. A Migration Server will run on every machine connected to the Sesame network. These servers must have knowledge of all the File Servers and their related media on the network.

A Migration Server's policy data base is local to it and its machine. The local Migration and File Servers must be able to share the information about all the files on the local disk, how recently they have been accessed, and how many local names reference each file.

2.5.2 Migration policy

The migration policy on a machine should be setable by the administrator of that machine. In the case of a personal Spice machine that is the owner. In the case of public Spice machines or the central service machines it is the system administrator. The parameters of the policy include the following:

- The minimum amount of storage to be kept free.
- The suggested storage quotas for subtrees of the name space.
- Directories or subtrees in which no files are to be migrated to other machines. This creates a class of files that are kept only locally.
- Directories or subtrees in which no files are to be automatically removed from the local machine. This creates a class of files that are always available from the local disk.
- Directories or subtrees into which no files are to be automatically migrated. This creates a class of files that are never kept on the local disk, but are paged in across the network whenever they are referenced.
- Archival time delays for directories or subtrees. (i.e. the delay between when a file is created locally and when it is copied to the next more secure File Server.)
- Removal time delays for directories or subtrees. (i.e. the delay between a delete name request and the removal of the name and possibly the file)
- Default values for all the parameters for files that are not named or are in directories that do not have policy parameters set.

2.5.3 Functions of the Migration Server

The Migration Server must provide a mechanism for the administrator to set and display the migration policy. It provides primitives to *retrieve* a file from another File Server to the local one, to *store* a copy of a local file to another File Server, and to *move* the local copy of a file to another File Server. It must also keep track of all files that are queued for automatic archiving or removal, and to perform the necessary action when the appropriate time comes. The Migration Server also has the responsibility of keeping track of total free disk space and selecting less used files for automatic removal when the disk is getting full.

3. Primitive Operations

The text which follows is an outline of the primitive operations provided by Sesame. Migration primitives and primitives to be used for communication between the various Sesame servers are not speicifed. Call which have not been implemented in the first release are marked with an asterisk (*).

3.1 Common Characteristics

All requests of Sesame are made through IPC messages sent to ports of the File, Name, Migration, Authorization or Authentication Servers. See [3] for a description of IPC message formats. All request messages have a standard format. Each message consists of a message ID, a reply port, and one or more request parameters. The message ID is some value agreed upon between a server process and its clients. The Sesame servers have defined these IDs to identify the request when responding to messages. The reply port specifies the destination for responses from the server. When a server responds to a request, the response message is sent to the specified reply port.

All requests have a *request code* which specifies the type of request being made. This is followed by a sequence of typed input parameters, which are interpreted according to the request code.

Response messages from the Sesame Servers also have a common form. Each contains the source port from which the message was sent, the message ID of the request invoking the response, the completion code of the request, and one or more typed result values. The completion code specifies either that the request was successfully completed or that the request could not be completed successfully due to some error. When a completion code of *Success* is returned, the result values (if any) are the results of the completed operation. If an error occured, the completion code indicates the reason for the failure.

This request/reply message protocol is entirely asynchronous. Client processes need not wait for a completion response from server before proceeding. However, it is the responsibility of these processes to insure that no dependent actions are attempted before a completion response is received from the server.

In addition to the asynchronous message interface documented here, synchronous remote procedure calls are provided by Pascal modules (and eventually Lisp modules) generated by the Matchmaker program [6]. These calls correspond to the documentation here with the result being returned as the function value, and the result parameters being returns as var parameters following the in parameters.

Several of the calls are provided with two or more forms, with one being a subset of the other. Where this is the case, the subset form will have a name prefix of *Sub*, whereas the full form of the call will have a name prefix of *Ses*.

3.2 Authentication Primitives

The Authentication Server primitives provide the verification mechanisms used in establishing an authenticated connection with Sesame.

3.2.1 Primitives for establishing and breaking a connection

Logging in to the Authentication Server

RequestCode: *SendLogin* *

Port: *public authentication port*

Parameters: *user name* - a name which uniquely identifies each user. This is the same as the name of the primary access group to which the user belongs.

login code - the password of the primary access group corresponding to the user, applied to the tuple <0, *Rkey*, *user name*>, where *Rkey* is going to be sent to the network server, associated with the message's reply port.

Result: *ANPort* - an IPC port for all further communication with the Authentication Server.

Completion Code: *success* - Login successful

failure - User name not recognized or password incorrect

The *SendLogin* request is used to establish a connection with Authentication Server. It verifies the authenticity of the user and creates and returns the user's authenticated service port for further communication to the Authentication Server. It associates the access group information from the GAL for the user with this port and registers the reply port so that it is available for verification when necessary.

User calls packages have a *LoginOwner(user name, password) => (ANPort)* call to generate the *SendLogin* primitive. *LoginOwner* chooses the *Rkey* and sends it to the network server.

Logging off

RequestCode: *Logout* *

Port: *ANPort*

Completion Code: *success* - Logout successful

The *Logout* request is used to break the connection previously established with the *Login* request. It is sent

to the Authentication Server port created at login time and causes that connection and all subordinate connections created with *Restrict* to be closed down.

Registering a port

RequestCode: *Register* *

Port: *ANPort*

Parameters: *user port* - a port to which the user typically has *ownership* and *receiverights* which is to be registered with the Authentication Server as having the user's identity.

Completion Code: *success* - the port is now registered

Ports that are registered with the Authentication Server may be passed to other servers as a token of a user's authenticated identity. (See *Verify*.)

Deregistering a port

RequestCode: *Deregister* *

Port: *ANPort*

Parameters: *user port* - a registered port.

Completion Code: *success* - the port is no longer registered

Deregister is used to break the association of *user port* with the *ANPort* user identity. It only breaks off the user's own registration of *user port*.

Create a new restricted authentication port

RequestCode: *Restrict* *

Port: *ANPort*

Parameters: *can restrict* - a boolean that determines if the new connection can execute this *Restrict* call again.

knows name - a boolean that determines if the new connection has a user name associated with it

remove list - a boolean indicating whether the list that is supplied is a list of the only ones in the restricted identity or the only ones *not* in the restricted identity.

list option flag - when set to *ID*, only access group IDs will be supplied; when set to *names*, only access group names will be supplied; when set to *both*, a list of name, ID pairs will be supplied

count - number of group ids and names that follow

IDS - a pointer to the array of group IDs that the new connection is to have

Result: *Authentication Port* - an authenticated port to send messages to the Authentication Server

Completion Code: *success* - A new connection to the Authentication Server has been established

This allows a user to create a new authenticated identity with a subset of his maximum rights. Now ports can be registered through calls to this new Authentication port and they become tokens of the diminished identity.

Verify a user's identity

RequestCode: *Verify* *

Port: *ANPort*

Parameters: *user port* - a registered port that has been sent as a token of a user's identity

list option flag - when set to *ID*, only access group IDs will be returned; when set to *names*, only access group names will be returned; when set to *both*, a list of name, ID pairs will be returned

Result: *user name* - the login name of the user whose port it is

count - the number of access groups that are being returned

IDs - a pointer to an array of the access group IDs that the user has

Completion Code: *success* - the port is registerd to this user

invalid port - the port is not a registered port

The *Verify* primitive returns the identity that is registered with *user port.* The *user name* will be null if the *knows name* flag in *Restrict* was false.

Verify is checked against the local Authentication server database first, and if that does not have *user port* then the request is passed through to the central Authentication server.

3.3 Authorization Primitives

Connecting to the authorization server

RequestCode: *AuthorConnect* *

Port: *public authorization port*

Parameters: *valid port* - a registered port from the authentication server.

Result: *ARPort* - the registered user's own port for authorization service.

The *AuthorConnect* primitive is used for establishing contact with the authorization server.

(The authentication server also uses this to get its connection to the authorization database. In that case, *valid port* signifies the system in some way.)

Implementation of Login-style password checking

RequestCode: *UserValidation* *

Port: *ARPort*

Parameters: *user name* - the name of the user attempting to log in.

Result: *antipassword* - the stored encrypted form of the password, for recognition purposes.

group ID - the user's primary group ID

The authentication server uses this call to look up the user's stored encrypted password. This information is public since anyone may check a password by encrypting it and comparing it with the stored *antipassword.*

Creating a new access group

RequestCode: *CreateGroup* *

Port: *ARPort*

Parameters: *access group name* - the name to be given to this group for unique identification.

access group type - *primary* or *secondary*

full name (optional) - the full, descriptive name of the access group, used only for informational purposes

Result: *access group ID* - the group ID of the GAL for the newly created access group.

Completion Code: *success* - new access group created

duplicate name - an access group whose name is *access group name* already exists.

access violation - only the System Administrator may create primary access groups

The *CreateGroup* request is used to create a new access group. The same primitive is used to create both primary and secondary access groups. However, only the System Administrator may create primary access groups. The access control list of the newly-created access group gives the creator all Authentication Server rights and gives no rights to anyone else. If the access group created is a primary access group, its password is set to null.

Changing a primary access group password

RequestCode: *ChangePassword* *

Port: *ARPort*

Parameters: *user name* - the name of the primary access group whose password is to be altered.

old password - the old password for the access group.

new password - the new password for the access group.

Completion Code: *success* - password changed

invalid password - old password incorrectly specified

invalid access group - specified access group does not exist

The *ChangePassword* request is used to to change the password for a primary access group. The old password must be supplied as a parameter before the new password may be changed. The System Administrator, however, need not supply the old password.

Changing an access group full name

RequestCode: *ChangeFullName* *

Port: *ARPort*

Parameters: *user name* - the name of the primary access group whose password is to be altered.

full name - the new full name of the access group.

Completion Code: *success* - full name changed

invalid access group - specified access group does not exist

The *ChangeFullName* request is used to to change the full name for an access group.

Deleting an access group

RequestCode: *DeleteGroup* *

Port: *ARPort*

Parameters: *access group name* - the name of the access group to be deleted

Completion Code: *success* - access group deleted

access violation - the user does not have *DeleteGroup* rights on this access group

nonexistent group - the specified access group does not exist.

The *DeleteGroup* request is used to delete an access group. Anyone possessing *DeleteGroup* rights on this access group is allowed to delete the group. Only the System Administrator may delete primary access groups.

Renaming an access group

RequestCode: *RenameGroup* *

Port: *ARPort*

Parameters: *old name* - the current name of the access group

new name - the new name of the access group

Completion Code: *success* - access group renamed

access violation - the user does not have *DeleteGroup* rights on this access group

duplicate name - the new name specified already exists

nonexistent group - the specified access group does not exist.

The user must possess *DeleteGroup* rights on an access group in order to rename it. The System Administrator may rename any access group.

Adding members to an access group

RequestCode: *AddToGroup* *

Port: *ARPort*

Parameters: *Mod group* - the name of the secondary access group to be modified.

Add group - the name of the access group to be added.

Completion Code: *success* - *add group* made a member of *mod group*

access violation - the user does not possess *AddMember* rights on *mod group*

redundant addition - *add group* is already a member of *mod group*

invalid access group - one of the specified access groups does not exist

The *AddToGroup* request is used to add a member to a secondary access group. A user must possess *AddMember* rights on an access group in order to add members to it.

Removing members from an access group

RequestCode: *RemoveFromGroup* *

Port: *ARPort*

Parameters: *Mod group* - the name of the secondary access group to be modified.

Rem group - the name of the access group to be removed

Completion Code: *success* - removal successful

access violation - the user does not possess *RemoveMember* rights on *Rem group*

not a member - *Rem group* is not currently a member of *Mod group*

invalid access group - one of the specified access groups does not exist

The *RemoveFromGroup* request is used to remove members from a secondary access group. Removal of currently logged in members takes effect only when they log out or when they perform an action that causes their flattened access list to be recomputed.

Mapping Access Group Names to IDs

RequestCode: *TrGroupName* *

Port: *ARPort*

Parameters: *access group name* - the name of the access group whose ID is desired

Result: *access group ID* - the ID of *access group name*

full name - the full name of the access group

Completion Code: *success* - the given name was successfully translated

invalid name - the given name does not belong to any access group

Mapping group IDs to names

RequestCode: *TrGroupID* *

Port: *ARPort*

Parameters: *access group ID* - the ID of the access group whose name is desired

Result: *access group name* - the name of the access group whose ID was supplied

full name - the full name of the access group

Completion Code: *success* - the translation was successful

invalid ID - the given ID does not refer to an existing access group

Listing the members of a secondary access group

RequestCode: *ListMembers* *

Port: *ARPort*

Parameters: *access group name* - name of the secondary access group whose members are to be listed

transitive closure flag - when this flag is set, both direct and indirect members of the access group are listed. Otherwise only direct members are listed.

list option flag - when set to *ID*, only access group IDs will be returned in *member list*; when set to *names*, only access group names will be returned; when set to *both*, a list of name, ID pairs will be returned

Result: *member list* - a list of of access groups which are members of this secondary access group. This may contain both primary and secondary access groups. The format of this result depends on *list option flag*

Completion Code: *success* - the members have been listed

nonexistent group - the secondary access group specified does not exist.

Listing the membership of an access group

RequestCode: *ListMembership* *

Port: *ARPort*

Parameters: *access group name (optional)* - name of the primary or secondary access group whose membership in secondary access groups is to be listed. If none is specified, this parameter defaults to the current primary access group.

transitive closure flag - when this flag is set, both direct and direct membership of the access group is returned. Otherwise only direct membership is returned.

list option flag - when set to *ID*, only access group IDs will be returned in *membership list*; when set to *names*, only access group names will be returned; when set to *both*, a list of name, ID pairs will be returned

Result: *membership list* - the list of access groups of which *access group name* is a member. This list will contain only secondary access groups. The format of this result is determined by *list option flag*.

Completion Code: *success* - the membership list has been generated.

nonexistent group - the access group specified does not exist.

3.4 Data transmision primitives

In the usual case all data transmision primitives refer to files by name and go through the Name Server. However, there are limited facilities for a client to refer to files by their File ID's and thus to deal with unamed files.

3.4.1 File I/O primitives

Reading data from a file to memory

RequestCode: *SubReadFile*

Port: *NSPort*

Parameters: *apathname* - the absolute pathname to read the data from

Result: *data pointer* - a pointer to the data read in memory

file size - the total number of bytes read

Completion Code: *success* - the data was successfully mapped into memory

no such name - no entry was found under apathname

access violation - the requesting client tried to read a file for which he did not possess rights

not a file - the entry found under apathname was not a file

no such file ID - the file associated with the entered ID could not be found

The *SubReadFile* request is used to map the data associated with the given filename into memory. Note that we map the entire file in, thus there are no arguments specifying the size or position of the data to be read. We can get away with mapping the whole file since we actually will only bring in the pages which are touched, the rest being backed to secondary storage. Holes in the file will be mapped in as valid zero pages which will actually be created if written to.

Reading data from a file to memory

RequestCode: *SesReadFile*

Port: *NSPort*

Parameters: *apathname* - the absolute pathname to read the data from

Result: *data pointer* - a pointer to the data read in memory

file size - the total number of bytes read

data format - one of {unspecified, text, bit8, bit16, bit32, bit36, press, ...}

creation date - the date and time the file was written

actual name - the absolute pathname actually read from

name status flags - low version, high version

Completion Code: *success* - the data was successfully mapped into memory

no such name - no entry was found under apathname

access violation - the requesting client tried to read a file for which he did not possess rights

not a file - the entry found under apathname was not a file

no such file ID - the file associated with the entered ID could not be found

The *SesReadFile* request is a long form of the *SubReadFile* call. It returns several parameter values which the short form does not.

Reading data from a file to memory

RequestCode: *SesIDReadFile* *

Port: *FSPort*

Parameters: *FID* - the File ID of the file to be read

file search flags (optional) - subsets of {inhibit/allow local file system access, inhibit/allow CFS access, inhibit/allow archive access, inhibit/allow foriegn Spice machine access}. Defaults will be applied for each flag if unspecified.

Result: *data pointer* - a pointer to the data read in memory

file size - the total number of bytes read

file status flags - flags indicating properties (stable/normal, location (local, CFS, archive, foriegn)) of the file

Completion Code: *success* - the data was successfully mapped into memory

no such file ID - the file associated with the given ID could not be found

The *SesIDReadFile* request is used to map the data associated with the given file ID into memory. Note that we map the entire file in, thus there are no arguments specifying the size or position of the data to be read. We can get away with mapping the whole file since we actually will only bring in the pages which are touched, the rest being backed to secondary storage. Holes in the file will be mapped in as valid zero pages which will actually be created if written to.

Writing data from memory to a file

RequestCode: *SubWriteFile*

Port: *NSPort*

Parameters: *apathname* - the absolute pathname to write the data to

data pointer - a pointer to the data in memory

file size - the number of bytes to be written

data format (optional) - one of {unspecified, text, bit8, bit16, bit32, bit36, press, ...}

Result: *actual name* - the absolute pathname actually written to

creation date - the date and time the file was written

Completion Code: *success* - the data was written under the name returned

access violation - the requesting client tried to write a file for which he did not possess rights

conflicting version - the explicit version number was less than or equal to that of an existing version

allignment error - the data was not on a page boundry

The *SubWriteFile* request is used to enter a new name into the directory structure and write a file under that name. This short form of the call picks up defaults for unspecified parameters from previous versions or directory defaults. The user must have *CreateNames* rights in the directory or *Supercede* rights on the previous version. Empty pages need no disk pages assigned to them.

Writing a file allowing control over defaults

RequestCode: *SesWriteFile* *

Port: *NSPort*

Parameters: *apathname* - the absolute pathname to write the data to

data pointer - a pointer to the data in memory

file size - the number of bytes to be written

file properties (optional) - flags indicating desired properties (stable/normal) of file. Defaults will be applied if unspecified.

data format (optional) - one of {unspecified, text, bit8, bit16, bit32, bit36, press, ...}

sys/aux flag (optional) - if set to *sys* the system access control for the entry is modified; if set to *aux* the auxiliary access control list is modified; *both* causes both lists to be modified

access control list (optional) - pointer to a structure of access control lists values for each type of access

Result: *actual name* - the absolute pathname actually written to

creation date - the date and time the file was written

Completion Code: *success* - the data was written under the name returned

access violation - the requesting client tried to write a file for which he did not possess rights

conflicting version - the explicit version number was less than or equal to that of an existing version

allignment error - the data was not on a page boundry

The *SesWriteFile* request is a long form of the *SubWriteFile* call. It may be used to specify parameter values which simply default in the *SubWriteFile* form when required. Note that defaults may still be taken for any of the optional parameters. The user must have *CreateNames* rights in the directory or *Supercede* rights on the previous version.

3.4.2 File header manipulation primitives

Returning file header information

RequestCode: *SesGetFileHeader*

Port: *NSPort*

Parameters: *apathname* - the absolute pathname with which to find the file

Result: *file header block* - a data structure containing the fields of the file header. See appendix A for details on file header fields.

Completion Code: *success* - the header data was successfully returned

no such name - no entry was found under apathname

access violation - the requesting client tried to read a file header for which he did not possess rights

not a file - the entry found under apathname was not a file

no such file ID - the file associated with the entered ID could not be found

The *SesGetFileHeader* request is used to return fields from the file header.

Returning file header information

RequestCode: *SesIDGetFileHeader**

Port: *FSPort*

Parameters: *FID* - the File ID of the file whose header is to be read

file search flags (optional) - subsets of {inhibit/allow local file system access, inhibit/allow CFS access, inhibit/allow archive access, inhibit/allow foriegn Spice machine access}. Defaults will be applied for each flag if unspecified.

Result: *file header block* - a data structure containing the fields of the file header. See appendix A for details on file header fields.

Completion Code: *success* - the header data was successfully returned

no such file ID - the file associated with the entered ID could not be found

The *SesIDGetFileHeader* request is used to return fields from the file header.

Returning file data and header information

RequestCode: *SesReadBoth*

Port: *NSPort*

Parameters: *apathname* - the absolute pathname with which to find the file

Result: *data pointer* - a pointer to the data read in memory

file size - the total number of bytes read

file header block - a data structure containing the fields of the file header. See appendix A for details on file header fields.

actual name - the absolute pathname actually read from

name status flags - deleted/undeleted, low version, high version

Completion Code: *success* - the header data was successfully returned

no such name - no entry was found under apathname

access violation - the requesting client tried to read a file header for which he did not possess rights

not a file - the entry found under apathname was not a file

no such file ID - the file associated with the entered ID could not be found

The *SesReadBoth* request is used to return both the data from a file and fields from its header.

3.5 Name Manipulation Primitives

The Name Server provides primitives to map names to their entry values, to create and delete directories, and to control access to directories.

Looking up names

RequestCode: *SubLookupName*

Port: *NSPort*

Parameters: *apathname* - the name to be looked up (may not contain wildcard characters)

Result: *actual name* - the absolute pathname actually found

entry type - the type value of the entry found. Some example values are *File*, *Directory*, and *Port*. See appendix B for a more complete description.

entry data - a variant field dependent upon the *entry type*. A File ID will be returned here for a *File* entry, whereas there will be no data returned for a *Directory* -- all that this call will tell you about a directory is that it exists. Again, see appendix B for a more complete description.

name status flags - deleted/undeleted, low version, high version

Completion Code: *success* - the name was successfully looked up.

name not found - the specified name was not found

access violation - the client does not have sufficient rights to look up the given name

This function provides a simple way to look up a name. If any entry encountered during the parsing of

apathname is of type *Symbolic Link*, a macro expansion is performed using the value of that entry in place of the corresponding name in *apathname*. If the final result is of type *FID* or *IPC Port*, the corresponding FID or IPC port is returned. If it is of type *Directory* no *entry data* is returned, but *entry type* specifies the fact that a lookup on a directory name was done. If *apathname* does not contain a version number, the most recent version is assumed.

Looking up names allowing special name treatment

RequestCode: *SesLookUpName* *

Port: *NSPort*

Parameters: *apathname* - the name to be looked up (may not contain wildcard characters)

name flags - subsets of {expand/return symbolic links, inhibit/allow deleted/undeleted names}

Result: *actual name* - the absolute pathname actually found

entry type - the type value of the entry found. Some example values are *File*, *Directory*, and *Port*. See appendix B for a more complete description.

entry data - a variant field dependent upon the *entry type*. A File ID will be returned here for a *File* entry, whereas there will be no data returned for a *Directory* -- all that this call will tell you about a directory is that it exists. Again, see appendix B for a more complete description.

name status flags - deleted/undeleted, low version, high version

Completion Code: *success* - the name was successfully looked up.

name not found - the specified name was not found

access violation - the client does not have sufficient rights to look up the given name

The *SesLookupName* call is a superset of *SubLookupName*, allowing the additional *name flags* parameter in addition to those provided by *SubLookupName*.

Determining the type of a name

RequestCode: *SubTestName*

Port: *NSPort*

Parameters: *apathname* - the name to be looked up (may not contain wildcard characters)

Result: *actual name* - the absolute pathname actually found

entry type - the type value of the entry found. Some example values are *File*, *Directory*, and *Port*. See appendix B for a more complete description.

name status flags - deleted/undeleted, low version, high version

Completion Code: *success* - the name was successfully found

name not found - the specified name was not found

access violation - the client does not have *UseNames* rights on the name's directory

SubTestName is like *SubLookupName* except that it never returns the *entry data* associated with a name, but only the *entry type*. Note that this only requires *UseNames* rights on the parent directory, whereas *SubLookupName* requires *Lookup* rights on the name itself -- a much stronger requirement. It is anticipated that this call will be used when all that is desired is to test for the existance of a name, and to determine its type.

Determining the type of a name allowing special name treatment

RequestCode: *SesTestName* *

Port: *NSPort*

Parameters: *apathname* - the name to be looked up (may not contain wildcard characters)

name flags - subsets of {expand/return symbolic links, inhibit/allow deleted/undeleted names}

Result: *actual name* - the absolute pathname actually found

entry type - the type value of the entry found. Some example values are *File*, *Directory*, and *Port*. See appendix B for a more complete description.

name status flags - deleted/undeleted, low version, high version

Completion Code: *success* - the name was successfully found

name not found - the specified name was not found

access violation - the client does not have *UseNames* rights on the name's directory

The *SesTestName* call is a superset of *SubTestName*, allowing the additional *name flags* parameter in addition to those provided by *SubTestName*.

Entering an object into a directory

RequestCode: *SubEnterName*

Port: *NSPort*

Parameters: *apathname* - the name to be entered into the directory structure

entry type - the type value of the object to be entered. Some example values are *File*, *Directory*, and *Port*. See appendix B for a more complete description.

entry data - a variant field dependent upon the *entry type*. For instance, this field must contain a File ID for type *File*, and an IPC port for type *Port*. For type *Directory*, this field is left empty, seeing as how the user can't write any directory data directly. Use of this call with *entry type Directory* enters a new directory, thus no special call is needed for that purpose. See appendix B for a more complete description.

Result: *actual name* - the absolute pathname, including version number, actually entered

Completion Code: *success* - *apathname* was successfully entered

conflicting version - the version number specified in *apathname* was less than or equal to that of an already existing version

access violation - the client does not possess *CreateNames* rights in the specified directory

invalid directory version - only one version number of a directory is allowed

The *SubEnterName* request is used to place a name and entry value pair in the directory structure. It requires *CreateNames* permission in the directory within which the name is being entered if no version of the name already exists else if a version of the name already exists, then *Supersede* privileges on the name are needed. If a previous version of the name already exists in the directory and no version number is specified in *apathname*, the name is entered with the next higher version number. If a previous version of the name already exists and a version number is specified in the name, then it must be greater than the highest version number so far. If no previous version of the name has ever existed, then version one is assigned if none is specified in *apathname*, otherwise the specified version is used.

Entering an object allowing special name treatment

RequestCode: *SesEnterName* *

Port: *NSPort*

Parameters: *apathname* - the name to be entered into the directory structure

name flags (optional) - expand/supersede symbolic links

entry type - the type value of the object to be entered. Some example values are *File*, *Directory*, and *Port*. See appendix B for a more complete description.

entry data - a variant field dependent upon the *entry type*. For instance, this field must contain a File ID for type *File*, and an IPC port for type *Port*. For type *Directory*, this field is left empty, seeing as how the user can't write any directory data directly. Use of this call with *entry type Directory* enters a new directory, thus no special call is needed for that purpose. See appendix B for a more complete description.

sys/aux flag (optional) - if set to *neither* then the default access control list is used; if set to *sys* the system access control for the entry is specified; if set to *aux* the auxiliary access control list is specified; *both* causes both lists to be taken from *access control list*

access control list (optional) - pointer to a structure of access control lists values for each type of access

Result: *actual name* - the absolute pathname, including version number, actually entered

Completion Code: *success* - *apathname* was successfully entered

conflicting version - the version number specified in *apathname* was less than or equal to that of an already existing version

access violation - the client does not possess *CreateNames* rights in the specified directory

invalid directory version - only one version number of a directory is allowed

The *SesEnterName* request is a superset of *SubEnterName*, allowing both the additional *name flags* parameter and the specification of a non-default access control list for the name. An explicit access control list is necessary if the default access control list would deny the user rights to the entry which he may want, and would not have the right to grant himself otherwise. Note that defaults may still be taken for any of the optional parameters.

Deleting a name in a directory

RequestCode: *SubDeleteName*

Port: *NSPort*

Parameters: *apathname* - the name to be deleted

Completion Code: *success* - the name was successfully deleted

nonexistent name - the name specified does not exist

access violation - the user does not possess *DeleteNames* rights on the name specified

name already deleted - the given name was already deleted

directory not empty - illegal to delete a non-empty directory

The *SubDeleteName* request is used to delete a name from a directory. It requires *DeleteNames* permission in the directory or *Delete* permission on the name. The name is marked as deleted, but remains in the directory and is queued for expunging by the Migration Server. If no version number is specified then the lowest version in the directory will be deleted.

Deleting a name in a directory allowing special name treatment

RequestCode: *SesDeleteName* *

Port: *NSPort*

Parameters: *apathname* - the name to be deleted

name flags (optional) - expand/delete symbolic links

expunge flag (optional) - set to true if an expunge is also desired. If false or ommitted, no expunge will be done.

Completion Code: *success* - the name was successfully deleted

nonexistent name - the name specified does not exist

access violation - the user does not possess *DeleteNames* rights on the name specified

name already deleted - the given name was already deleted

directory not empty - illegal to delete a non-empty directory

The *SesDeleteName* call is a superset of *SubDeleteName*, allowing the additional *name flags* parameter in addition to those provided by *SubDeleteName*. Unless the expunge option is specified, the name is marked as deleted, but remains in the directory and is queued for expunging by the Migration Server. However, if an expunge is also requested, then it is removed from the directory.

Restoring a deleted name

RequestCode: *SesUndeleteName* *

Port: *NSPort*

Parameters: *apathname* - the name to be undeleted

name flags (optional) - expand/undelete symbolic links

Completion Code: *success* - the name was successfully undeleted

nonexistent name - the name specified does not exist

access violation - the client does not possess the proper rights to undelete the name

name not deleted - the given name was not deleted

The *SesUndeleteName* request is used to restore a deleted name. It requires either *CreateNames* permission on the directory in which the entry resides or permission to have deleted the entry. The Migration Server is informed that the entry is no longer to be expunged.

Expunging deleted names from a directory

RequestCode: *SesExpungeDirectory* *

Port: *NSPort*

Parameters: *apathname* - name of the directory from which files are to be expunged.

Completion Code: *success* - the expunge was successful

access violation - the requesting client tried to expunge a directory for which he did not have the necessary rights

All deleted names in the directory are expunged. The client must possess *DeleteNames* permission in the directory for this operation to succeed. Note that with the *SesExpungeName* call a client may be able to expunge individual names in directories for which he doesn't possess *DeleteNames* access.

Changing the name of an entry

RequestCode: *SubRename*

Port: *NSPort*

Parameters: *old apathname* - absolute pathname of the object to be renamed

new apathname - new name to enter the object under

Result: *actual name* - the new absolute pathname the object was entered under

Completion Code: *success* - the name was successfully changed

access violation - the client either does not have the proper rights to delete *old apathname* or to enter *new apathname*

SubRename enters the object specified by *old apathname* into into the global name space as *new apathname* and then removes the *old apathname.* The access control list of the object is moved with it.

Changing the name of an entry allowing special name treatment

RequestCode: *SesRename* *

Port: *NSPort*

Parameters: *old apathname* - absolute pathname of the object to be renamed

name flags - expand/rename symbolic links

new apathname - new name to enter the object under

Result: *actual name* - the new absolute pathname the object was entered under

Completion Code: *success* - the name was successfully changed

access violation - the client either does not have the proper rights to delete *old apathname* or to enter *new apathname*

The *SesRename* request is a superset of *SubRename*, allowing the additional *name flags* parameter.

Copying an entry to a new name

RequestCode: *SubCopyName* *

Port: *NSPort*

Parameters: *old apathname* - absolute pathname of the entry to be copied

new apathname - name to make a copied entry under

Result: *actual name* - the new absolute pathname the object was entered under

Completion Code: *success* - a new entry was successfully made

access violation - the client either does not possess *Copy* rights on *old pathname* or the proper rights to enter *new pathname*

SubCopyName enters the object specified by *old apathname* into into the global name space as *new apathname* and without removing the *old apathname.* This is equivalent to performing a lookup and then an enter on an entry except that in certain cases an access control violation would occur if the actual lookup were attempted where a copyname is allowed. One example of this is a file entry for which the user does not possess lookup rights, but for which he does possess the *Copy* right. The access control list of the old object is copied to the new name.

Copying an entry to a new name allowing special name treatment

RequestCode: *SesCopyName* *

Port: *NSPort*

Parameters: *old apathname* - absolute pathname of the entry to be copied

name flags - expand/copy symbolic links

new apathname - name to make a copied entry under

Result: *actual name* - the new absolute pathname the object was entered under

Completion Code: *success* - a new entry was successfully made

access violation - the client either does not possess *Copy* rights on *old pathname* or the proper rights to enter *new pathname*

The *SesCopyName* request is a superset of *SubCopyName*, allowing the additional *name flags* parameter.

Scanning directories for names

RequestCode: *SesScanNames*

Port: *NSPort*

Parameters: *wild apathname* - the absolute pathname to be scanned (may contain wildcard characters in the terminal component)

name flags (optional) - inhibit/allow deleted/undeleted names

entry type - the entry type being scanned for. The special type designator *All* may be given, in which case names of all entry types are returned.

Result: *directory nameentry list* - the absolute pathname of the directory in which matches occured

actual name count - the actual number of list elements returned. Will be zero if no match occured.

Completion Code: *success* - the given wild absolute pathname was successfully scanned

access violation - the client does not possess *ReadNames* rights on the directory to be scanned

illegal pathname - wildcards were found in *wild apathname* at other than the terminal component

The *SesScanNames* call is used to search for a given pattern in a specified directory. It will sort and return all the matches to the given pattern in the directory. Optionally, names only of a specific entry type can be scanned for. Symbolic links are not expanded, and are returned by this call. Depending on the value of *name flags*, only active names, deleted names or both are considered. If no version number is specified, then the highest existing version of the name is returned. If the version number is wildcarded, then all existing versions of the name are returned. The version field for directory and symbolic link entries will be returned as zero. Note that wildcarding is permitted **only** in the terminal component of *wild apathname*. This call requires either *ReadNames* access on the directory being scanned, in which case all matches will be returned, or else *Visible* access on each match which is to be returned.

Retrieving archived names

RequestCode: *SesScanArchivedNames**

Port: *NSPort*

Parameters: *wild apathname* - the name to be scanned for; may be wildcarded

entry type - the entry type being scanned for. The special type designator *All* may be given, in which case names of all entry types are returned.

start time - a date and time which determines the maximum age of the names scanned; here "age" is determined by the date of entry in the directory whose name archive is being scanned

end time - a date and time which determines the minimum age of the names scanned; here

"age" is determined by the date of entry in the directory whose name archive is being scanned

Result: *entry list* - a list of the names, entry types and creation dates corresponding to *wild apathname* between *start time* and *end time*

actual name count - the actual number of list elements returned

Completion Code: *success* - the entries were successfully retrieved

access violation - the client does not possess *ReadNames* rights the directory whose name archive is being scanned

Retrieves the entry types corresponding to the given name(s) between the specified dates and times. The client must possess *ReadNames* rights on the directory whose name archive is begin scanned.

3.6 Name Policy Primitives

Returning the retention count of a name

RequestCode: *SesGetRetentionCount* *

Port: *NSPort*

Parameters: *apathname* - the absolute pathname whose retention count is to be returned

Result: *retention count* - The number of versions of name being retained in the directory before the oldest version is automatically deleted. A zero result specifies an infinite retention count.

Completion Code: *success* - the retention count for the name was returned

name not found - the specified name was not found

access violation - client did not have the proper rights

version number specified - no pathname version number is allowed in this call

name is a directory - this call is not allowed for directories

The retention count on a name specifies the number of versions of a name which are to be retained when a new version is created. When a new version is entered enough versions will be deleted to bring the number of undeleted versions down to *retention count.* Deletion is done in ascending version number order.

Setting the retention count for a name

RequestCode: *SesSetRetentionCount* *

Port: *NSPort*

Parameters: *apathname* - the absolute pathname whose retention count is to be set

retention count - The number of versions of name to retain in the directory before the oldest version is automatically deleted. A zero argument specifies an infinite retention count.

Completion Code: *success* - the retention count for the name is now set

name not found - the specified name was not found

access violation - client did not have rights to delete the name

version number specified - no pathname version number is allowed in this call

illegal retention count - a negative retention count was specified

name is a directory - this call is not allowed for directories

The retention count on a name specifies the number of versions of a name which are to be retained when a new version is created. When a new version is entered enough versions will be deleted to bring the number of undeleted versions down to *retention count.* Deletion is done in ascending version number order.

Returning the default retention count in a directory

RequestCode: *SesGetDefRetentionCount* *

Port: *NSPort*

Parameters: *apathname* - the directory absolute pathname whose default is to be returned

Result: *retention count* - The number of versions of a name to retain in the directory before the oldest version is automatically deleted. A zero argument specifies an infinite retention count.

Completion Code: *success* - the default retention count for the directory was returned

name not found - the specified name was not found

access violation - client did not have the proper rights

version number specified - no pathname version number is allowed in this call

not a directory - *apathname* did not specify a directory

The retention count on a name specifies the number of versions of a name which are to be retained when a new version is created. When a new version is entered enough versions will be deleted to bring the number of undeleted versions down to *retention count.* Deletion is done in ascending version number order.

Setting the default retention count in a directory

RequestCode: *SesSetDefRetentionCount* *

Port: *NSPort*

Parameters: *apathname* - the directory absolute pathname whose default is to be set

retention count - The number of versions of a name to retain in the directory before the oldest version is automatically deleted. A zero argument specifies an infinite retention count.

Completion Code: *success* - the default retention count for the directory was set

name not found - the specified name was not found

access violation - client did not have rights to delete the name

version number specified - no pathname version number is allowed in this call

illegal retention count - a negative retention count was specified

not a directory - *apathname* did not specify a directory

The retention count on a name specifies the number of versions of a name which are to be retained when a new version is created. When a new version is entered enough versions will be deleted to bring the number of undeleted versions down to *retention count.* Deletion is done in ascending version number order.

3.7 Access Control primitives

Reading the access control list

RequestCode: *SesGetAccess* *

Port: *NSPort*

Parameters: *apathname* - the name of the entry to return the access control list for

name flag (optional) - expand link/set link access.

sys/aux flag - if set to *sys* the system access control for the entry is returned; if set to *aux* the auxiliary access control list is retrieved; *both* causes both lists to be retrieved

Result: *access control list* - the access control list of the entry, If *sys/aux flag* is set to *sys* or *aux* a list of two-tuples consisting of access group ID, access rights pairs is returned. If the *sys/aux flag* specified both lists, a list of three-tuples consisting of access group ID, system rights and auxiliary rights is returned.

access control list count - the number of elements in the acess control list.

Completion Code: *success* - the access control list was successfully read

access violation - The client does not have the rights to read the specified access control list. *GetSysAcess* rights are needed to read the system access control list and *GetAuxAccess* rights are needed to read the auxiliary access list.

The access control list of the named entry is returned

Writing the access control list

RequestCode: *SesSetAccess* *

Port: *NSPort*

Parameters: *apathname* - the name of the entry to set the access control list for

name flag (optional) - expand link/set link access.

sys/aux flag - if set to *sys* the system access control for the entry is modified; if set to *aux* the auxiliary access control list is modified; *both* causes both lists to be modified

access control list - the access control list of the entry. If *sys/aux flag* is set to *sys* or *aux* this consists of a list of two-tuples corresponding to access group ID, access rights pairs. If the *sys/aux flag* specified both lists, this list consists of three-tuples corresponding to access group ID, system rights and auxiliary rights.

access control list count - the number of elements in the access control list

Completion Code: *success* - the access control list was successfully modified

access violation - The client does not have the rights to modify the specified access control list. *SetSysAcess* rights are needed to modify the system access control list and *SetAuxAccess* rights are needed to modify the auxiliary access list.

The *SesSetAccess* request is used to update the access rights for one or more access groups in the access control list of an entry. If the access rights are null, the access group is completely removed from the access control list. If the access rights are non-null and the access group is already on the access control list, the new rights are substituted for the old rights. If the access rights are non-null and the access group is not currently on the access control list, the access group is added to the access control list with the specified access rights.

Checking access on a file

RequestCode: *SesCheckAccess* *

Port: *NSPort*

Parameters: *apathname* - the name of the entry whose access is to be checked.

name flag (optional) - expand link/check link access

access group list (optional) - the list of IDs of the access groups whose rights are to merged for the purposes of this check. If absent, the set of currently enabled access groups is used.

access group list count - the number of elements in the access group list

Result: *rights mask* - a bit mask corresponding to system and auxiliary rights.

Completion Code: *success* - the rights list for the specified access groups on the file have been returned

name not found - the specified apathname cannot be found.

access violation - the client does not have *GetAccess* privileges on this entry

The *SesCheckAccess* primitive is used to check the access that any arbitrary collection of access groups have on a file. The *GetAccess* privilege is needed to perform this check unless the access groups being checked are the ones belonging to the client.

Reading the directory default access control list

RequestCode: *SesGetDefAccess* *

Port: *NSPort*

Parameters: *directory name* - the name of the directory whose default access control list is to be read.

sys/aux flag - if set to *sys* the default system access control list is retrieved; if set to *aux* the default auxiliary access control list is retrieved; *both* causes both lists to be retrieved.

access control list count - the number of elements in the access control list

Result: *access control list* - the default access control list of the directory. If the *sys/aux flag* is set to *sys* or *aux*, this consists of a list of access group, access rights pairs. If the flag is set to *both*, this list consists of a set of three-tuples corresponding to access group IDs, system rights and auxiliary rights.

Completion Code: *success* - the access control list was successfully returned

invalid name - *directory name* either does not exist or does not refer to a directory

access violation - the enabled access groups do not possess *Lookup* on the directory

The *SesGetDefAccess* request is used to retrieve the default access control list associated with a directory. It requires *GetDefAccess* permission in the directory whose default access control list is being read.

Changing the directory default access control list

RequestCode: *SesSetDefAccess* *

Port: *NSPort*

Parameters: *directory name* - the name of the directory whose default access control list is to be updated.

sys/aux flag - if set to *sys* the default system access control list is modified; if set to *aux* the default auxiliary access control list is modified; *both* causes both lists to be modified.

access control list - the new access control list for the corresponding access group(s). If the *sys/aux flag* is set to *sys* or *aux*, this consists of a list of access group, access rights pairs. If the flag is set to *both*, this list consists of a set of three-tuples corresponding to access group IDs, system rights and auxiliary rights.

access control list count - the number of elements in the access control list

Completion Code: *success* - the access control list was successfully modified

invalid name - *directory name* either does not exist or does not refer to a directory

access violation - the enabled access groups do not possess *SetDefAccess* rights on the directory

The *SesSetDefAccess* request is used to update the access rights for one or more access groups in the default access control list of a directory. It requires *SetDefAccess* permission in the directory whose default access control list is being changed. If the access rights are zero, the access group is completely removed from the

default access control list of the directory. If the access rights are non-zero and the access group is already on the access control list, the new rights are substituted for the old rights. If the access rights are non-zero and the access group is not currently on the access control list, the access group is added to the access control list with the specified access rights.

Checking default access on a directory

RequestCode: *SesCheckDefAccess* *

Port: *NSPort*

Parameters: *directory name* - the name of the directory whose default access control list is to be updated.

sys/aux flag - if set to *sys* the default system access control list is modified; if set to *aux* the default auxiliary access control list is modified; *both* causes both lists to be modified.

access group list (optional) - the list of IDs of the access groups whose rights are to merged for the purposes of this check. If absent, the set of currently enabled access groups is used.

access group list count - the number of elements in the access group list

Result: *rights mask* - a bit mask corresponding to system and auxiliary rights.

Completion Code: *success* - the access control list was successfully modified

invalid name - *directory name* either does not exist or does not refer to a directory

access violation - the enabled access groups do not possess *SetDefAccess* rights on the directory

The *SesCheckDefAccess* primitive is used to check the default access that any arbitrary collection of access groups has on a directory. The *GetDefAccess* privilege is needed to perform this check.

Acknowledgements

The authors wish to acknowledge the contributions of George Robertson, M. Satyanarayanan and Mike Accetta who are co-authors of the Central File System design from which Sesame has evolved, and to Gene Ball and Peter Hibbard who took part in a number of the design meetings and initial critiques. Credit is also due to members of the Three Rivers Computer Corporation Accent group, who have been involved in the review process and initial implementation effort. There special credit is due to David Golub, who has actively worked with us on file system issues. The authors also wish to express their appreciation to the members of the CMU Department of Computer Science, in general, and the members of the Spice group, in particular, for their comments on the design and on this document.

A. File header fields

The following table summarizes the information that the File Server keeps about each of the files on its local disk.

- **Global information**
  - File ID
  - File Size
  - Advisory Data Format
  - Author ID
  - Creation Date
  - File Print Name
- **Local information**
  - Last Access Date
  - Has Been Archived
  - Do Not Remove
  - Reference Count
  - Segment ID
  - Storage Map

The next table specifies the fields that are returned by the *SesGetFileHeader* primitive.

- File size
- Data format
- Print name
- Author ID
- Creation Date
- Access Date

B. Directory fields

The following table sumarizes the information in the name data base (directory structure).

- **Directory information**
  - Default access control list
  - Number of versions of names to retain
  - Directory status (primary or advisory)
  - File ID of corresponding name archive

Entry Information

- Name of Entry -- just this component of the pathname for the entry
- Status of Entry -- undeleted/deleted and maybe other flags
- Retention Count -- number of versions to retain
- Access Control List -- list of access control groups and rights
- Entry Type -- one of {File, Directory, Symbolic Link, IPC Port, User Defined, Empty}
- Entry Data -- variant dependent upon entry type. Contents are listed for each entry type:

File	the File ID
Directory	no contents. A directory entry has no user readable or writable components in the normal sense. He can only manipulate them through name name server calls. Thus the user's calls will view the data for a directory as an empty variant field. The hidden representation is what is actually being described in this appendix.
Symbolic Link	the substitute pathname
IPC Port	the actual IPC port
User Defined	A block of storage at least as big as a 255 byte string.

C. Format of the Group Attribute List

For every access group in the system there is one GAL containing the following information:

Access Group Name	a unique name that identifies this access group. For primary access groups this is the login name of the corresponding user.
Full Name	the full descriptive name of the access group. For primary access groups this will be the full name of the corresponding user; consequently it may not be unique.
Access Group ID	a unique integer that identifies this access group. Used instead of *Access Group Name* wherever a fixed length identification is needed for the access group, such as in the access control list of files.
Group Type	(Primary/Secondary)
Password	For primary access groups this is the login password, transformed by an encryption function. Not present for secondary access groups.
Direct Members List	the list of direct (i.e., without applying transitive closure) members of this access group
Direct Membership List	the list of access groups of which this access group is a direct member.

D. Using Sesamoid, the first version of Sesame

Sesamoid is the current Spice filesystem. It consists of a subset of the data transmission and name manipulation primitives that are described in this document, plus some temporary primitives needed to retain disk structure compatibility with the POS filesystem. No authorization, authentication or migration primitives have been implemented. No automatic archiving, delayed deletion, or access control is done. There are no links, or other types of name entries except for files, directories and ports. There is an existing message interface and a matchmaker call interface to the primitives that are not flagged by an "*".

The Sesamoid file server provides a naming convention that allows a local filesystem to reference files on a remote Spice machine.

Each machine should have a file by the name <Boot>SysName (usually this is Sys:Spice>SysName) which contains the name of that machine. The first line of this file is an ascii name by which his filesystem will be known. To reference a file on a remote machine a user must know the name of that machine (as defined in the remote SysName file) and then reference the remote file by its absolute pathname, starting with the remote machine name. For expample, if a machine has the name mrt, remote users can refer to a file on that machine with the name >sys>User>Guest>foo.pas by the name >mrt>User>Guest>foo.pas from their machines.

It is also possible to control remote access to your filesystem. There are three possible levels of access: no access, read access, and full access, on a per-machine basis. When a system is booted read access is automatically granted. The shell command *Access* returns the current access. *Access* takes the arguments *None, Read,* or *Full* and sets the access accordingly.

The exported Pascal procedures for Sesamoid are found in SesameUser.pas, the interface routines are in PathName.pas, with the Evnironment Manager procedures (to set search lists) being in EnvMgrUser.pas. The type definitions files are SesameDefs.pas and EnvMgrDefs.pas. All of these files are retrieved from the update set /usr/spice/libpascal/src/ on CMU-CS-CFS.

E. Summary of Primitives

The following is a summary of the primitives provided by Sesame. The page on which the primitive is fully described appears within square brackets. Calls which will not be implemented with the first release are marked with an asterisk (*).

Authentication Primitives

[20] * ***SendLogin*** *(public authentication port, user name, login code) ⇒ (ANPort)*

[20] * ***Logout*** *(ANPort) ⇒ ( )*

[21] * ***Register*** *(ANPort, user port) ⇒ ( )*

[21] * ***Deregister*** *(ANPort, user port) ⇒ ( )*

[21] * ***Restrict*** *(ANPort, can restrict, knows name, remove list, list option flag, count, IDS) ⇒ (Authentication Port)*

[22] * ***Verify*** *(ANPort, user port, list option flag) ⇒ (user name, count, IDs)*

Authorization Primitives

[23] * ***AuthorConnect*** *(public authorization port, valid port) ⇒ (ARPort)*

[23] * ***UserValidation*** *(ARPort, user name) ⇒ (antipassword, group ID)*

[23] * ***CreateGroup*** *(ARPort, access group name, access group type, full name) ⇒ (access group ID)*

[24] * ***ChangePassword*** *(ARPort, user name, old password, new password) ⇒ ( )*

[24] * ***ChangeFullName*** *(ARPort, user name, full name) ⇒ ( )*

[25] * ***DeleteGroup*** *(ARPort, access group name) ⇒ ( )*

[25] * ***RenameGroup*** *(ARPort, old name, new name) ⇒ ( )*

[26] * ***AddToGroup*** *(ARPort, Mod group, Add group) ⇒ ( )*

[26] * ***RemoveFromGroup*** *(ARPort, Mod group, Rem group) ⇒ ( )*

[27] * ***TrGroupName*** *(ARPort, access group name) ⇒ (access group ID, full name)*

[27] * ***TrGroupID*** *(ARPort, access group ID) ⇒ (access group name, full name)*

[27] * ***ListMembers*** *(ARPort, access group name, transitive closure flag, list option flag) ⇒ (member list)*

[28] * ***ListMembership*** *(ARPort, access group name, transitive closure flag, list option flag) ⇒ (membership list)*

Data Transmission Primitives

[28] ***SubReadFile*** *(NSPort, apathname) ⇒ (data pointer, file size)*

[29] ***SesReadFile*** *(NSPort, apathname) ⇒ (data pointer, file size, data format, creation date, actual name, name status flags)*

[30] * ***SesIDReadFile*** *(FSPort, FID, file search flags) ⇒ (data pointer, file size, file status flags)*

[30] ***SubWriteFile*** *(NSPort, apathname, data pointer, file size, data format) ⇒ (actual name, creation date)*

[31] * ***SesWriteFile*** *(NSPort, apathname, data pointer, file size, file properties, data format, sys/aux flag, access control list) ⇒ (actual name, creation date)*

[32] ***SesGetFileHeader*** *(NSPort, apathname) ⇒ (file header block)*

[33] * ***SesIDGetFileHeader*** *(FSPort, FID, file search flags) ⇒ (file header block)*

[33] ***SesReadBoth*** *(NSPort, apathname) ⇒ (data pointer, file size, file header block, actual name, name status flags)*

Name Manipulation Primitives

[34] ***SubLookupName*** *(NSPort, apathname) ⇒ (actual name, entry type, entry data, name status flags)*

[35] * ***SesLookUpName*** *(NSPort, apathname, name flags) ⇒ (actual name, entry type, entry data, name status flags)*

[35] ***SubTestName*** *(NSPort, apathname) ⇒ (actual name, entry type, name status flags)*

[36] * ***SesTestName*** *(NSPort, apathname, name flags) ⇒ (actual name, entry type, name status flags)*

[37] ***SubEnterName*** *(NSPort, apathname, entry type, entry data) ⇒ (actual name)*

[37] * ***SesEnterName*** *(NSPort, apathname, name flags, entry type, entry data, sys/aux flag, access control list) ⇒ (actual name)*

[38] ***SubDeleteName*** *(NSPort, apathname) ⇒ ( )*

[39] * ***SesDeleteName*** *(NSPort, apathname, name flags, expunge flag) ⇒ ( )*

[40] * ***SesUndeleteName*** *(NSPort, apathname, name flags) ⇒ ( )*

[40] * ***SesExpungeDirectory*** *(NSPort, apathname) ⇒ ( )*

[40] ***SubRename*** *(NSPort, old apathname, new apathname) ⇒ (actual name)*

[41] * ***SesRename*** *(NSPort, old apathname, name flags, new apathname) ⇒ (actual name)*

[41] * ***SubCopyName*** *(NSPort, old apathname, new apathname) ⇒ (actual name)*

[42] * ***SesCopyName*** *(NSPort, old apathname, name flags, new apathname) ⇒ (actual name)*

[42] ***SesScanNames*** *(NSPort, wild apathname, name flags, entry type) ⇒ (directory name, entry list, actual name count)*

[43] * ***SesScanArchivedNames*** *(NSPort, wild apathname, entry type, start time, end time) ⇒ (entry list, actual name count)*

Name Policy Primitives

[44] * ***SesGetRetentionCount*** *(NSPort, apathname) ⇒ (retention count)*

[45] * ***SesSetRetentionCount*** *(NSPort, apathname, retention count) ⇒ ( )*

[45] * ***SesGetDefRetentionCount*** *(NSPort, apathname) ⇒ (retention count)*

[46] * ***SesSetDefRetentionCount*** *(NSPort, apathname, retention count) ⇒ ( )*

Access Control primitives

[46] * ***SesGetAccess*** *(NSPort, apathname, name flag, sys/aux flag) ⇒ (access control list, access control list count)*

[47] * ***SesSetAccess*** *(NSPort, apathname, name flag, sys/aux flag, access control list, access control list count) ⇒ ( )*

[48] * ***SesCheckAccess*** *(NSPort, apathname, name flag, access group list, access group list count) ⇒ (rights mask)*

[48] * ***SesGetDefAccess*** *(NSPort, directory name, sys/aux flag, access control list count) ⇒ (access control list)*

[49] * ***SesSetDefAccess*** *(NSPort, directory name, sys/aux flag, access control list, access control list count) ⇒ ( )*

[50] * ***SesCheckDefAccess*** *(NSPort, directory name, sys/aux flag, access group list, access group list count) ⇒ (rights mask)*

References

[1] Organick, E.I.
The Multics System: an Examination of its Structure.
MIT Press, Cambridge, Mass., 1972.

[2] Rashid, R. F. & G. G. Robertson.
Accent: A comunication oriented network operating system kernel.
Technical Report CMU-CS-81-123, Department of Computer Science, Carnegie-Mellon University, April, 1981.

[3] Rashid, R. F.
Accent Kernel Reference Manual.
Technical Report , Department of Computer Science, Carnegie-Mellon University, July, 1982.

[4] Ritchie, D. M. and Thompson, K.
The UNIX Time-Sharing System.
Bell System Technical Journal, July-August, 1978.

[5] McKusick, Joy, Leffler, Fabry.
A Fast File System for UNIX.
Technical Report Draft of September 6, 1982, Computer Systems Research Group, University of California, Berkeley, 1982.

[6] Wright,Keith.
Matchmaker: A Remote Procedure Call Generator.
Technical Report , Department of Computer Science, Carnegie-Mellon University, April, 1982.