Sesame: The Spice File System

Mary R. Thompson
Robert D. Sansom
Michael B. Jones
Richard F. Rashid

11 December 1985

Abstract

Sesame provides several distinct but interelated 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 a central file system. It provides a global name space and a global user authentication protocol.

Technical Report CMU-CS-85-172

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 Research Projects Agency or the U.S. Government.

Table of Contents

1 Introduction

Sesame is a distributed file system designed and implemented as part of the Spice Project at the Computer Science Department of Carnegie-Mellon University. It is not just a file storage service but provides as well most of the interlated services needed to allow protected sharing of data and services in a network of personal and central computers. It deals with issues of user verification both locally and between machines, name lookup services for a variety of typed objects, archiving of files to more stable media as well as the fundamental functions of reading and writing files. Sesame is currently running as an alternative file system in the Spice environment on three central machines and a dozen or so user machines. Archival and retrieval to tertiary storage is still under development, but the rest of the system as described in this document is complete.

The Spice environment has the following characteristics:

- It is a user community of 200 to 300 faculty, staff and graduate students.
- Many members of the community have personal Spice machines with the speed of at least 1MIP and a reasonable amount of secondary storage, e.g. 24 megabytes. There may be several different kinds of machines.
- Some Spice machines are shared among a small number of users.
- Some Spice machines are 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.

The goals of the Sesame File System are:

- 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 among users.
- Uniform naming 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).

Sesame is implemented as a set of co-operating processes running on local workstations and secure central machines. Establishing a user's identity and giving him access to the services of Spice is conceptually divided

into two parts: Authorization and Verification. Authorization services are those functions that manage the data base of authorized Spice users. Verification services are those functions that allow dynamic verification of a user's identity to Sesame and other Spice servers. Authorization services are provided by Central Authorization Servers running on secure machines. Verification services are provided by local and central verification servers. Only a Central Verification Server running on a secure machine can be trusted to provide a user's identity to a remote machine. Thus most of the service is provided by the Central Verification Server. The Local Verification Server can only provide verification between processes on the local machine and limited verification service when no Central Verification Server is available.

Name lookup and file storage and retrieval services are provided by local and central Name/File Servers. The code of the local and central servers is identical. Differences in various policy tables and access rights on certain directories can provide any necessary differences in behavior between a central server and a local server.

This document primarily explains the services that are provided by Sesame. It also presents a general description of the information that is kept in order to give the reader a general idea of how Sesame is implemented.

2 Name Service

2.1 Overview

The Name Service provides a global name space for all objects or services intended to be available to users of the Spice environment. The Name Service 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 with the local Name Server 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 the rights to modify each partition.

The Name Service provides six primary functions:

1. It maps names chosen by users to unique identifiers.[1]
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.

---

[1] In most cases this can be interpreted as "typically translates a file name to an internal file identifier."

4. It maintains type information on all named objects.

5. It provides access control on all named objects.

6. It maintains information on caching policy for names and files on a per directory/subtree basis.

These services are provided independently of the services provided by the File Service. 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 2.2, is to provide a network-wide naming scheme for communication ports. Directory entries and symbolic links are treated as typed objects by the Name Server and so have the standard access control and caching mechanisms applied to them. The Name Server also allows a user to define his own types of objects which may be entered into the global name space and managed with the standard Name Server primitives.

The global name space provided by Sesame appears to the user as a hierarchy of directories, similar to that in the Unix[2] file system [5]. 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 directory boundaries. This partitioning is done dynamically. That is, a Name Server may check out a directory from the current maintainer and may return the directory when it no longer wishes to maintain it. A directory that has been checked out is marked as *primary directory* and that server is granted the exclusive right to make modififcations in that directory. A Name Server may also have *cached directories* which are partial or complete copies of directories and may be used only for looking up information.

When a given Name Server does not have the primary directory for a name lookup in its domain it can either use its cached directory if one is present or it can search for the primary copy of the directory. A cached directory may be used if the request is to get information and the caching policy for that directory allows its use. If the primary directory is required there are several ways to search for it. The cached directory may contain a hint as to what machine has the primary directory. The local Name Server may ask all the central

---

[2]Unix is a trademark of Bell Laboratories

Servers that it is connected to if they have the primary copy. A central Server that has a cache of the directory will return its hint for where the primary directory is. If the primary directory exists on a private workstation that is turned off the search will fail. In that case the best complete cache will be used for reading information, and all modify requests will fail.

In general users are advised to keep objects that are intended to be shared in directories that are always maintained by one of the Central Name Servers. These servers are always available and are searched first for unknown directories. A user may keep his unshared files in directories that are checked out to the Spice Name Server on that his machine in order to get more efficient access to them. It is possible for one user to access files on another user's machine, but the search for the file may take longer and will fail if the remote machine is turned off.

2.2 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 internal file identifier (*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. Normally file accesses are done by name rather than by FID, since access control and caching policy is done on the basis of names. If the user has the right to do a *LookupName* and obtain the FID from the name however, he can then read the file by FID.

Subdirectories have entries of type *Directory* in their parent directories. There is no user available data associated with directories. A successful name lookup on a directory entry merely returns the fact that a directory of that name exists.

A third kind of entry is of type *Symbolic Link*. The value of 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 for more details.

Another type of entry value is a *Spice Interprocess Communication Port (IPC port)*. This feature is used to provide a simple name to 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.

2.3 Names

2.3.1 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 any printable ASCII character. 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 is not normally 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.

If it necessary to enter names that are to be case distinctive (e.g. to match names on a foreign file system that are case distictive such as Unix), then each component of the name that has significant cases may be surrounded by single quotation marks. Once two names differing only by case have been entered, they can only be matched properly by quoting each case significant component. If unquoted components are entered they will match the "first" name found.

Two primitives, *ScanNames* and *ScanCacheNames* 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:

```
/Spice/Source/Oregano/Oregano.Pas
/User/Jones/XYZ/abc...xyz#8930
/User/Jones/Finger.Plan#Low
/Spice_System/Mail_System/EtherNet/Mailer.Run
/Spice/Source/Nutmeg/???Nut*.Pas#All
/'Unix'/'Names'/'ARE'/case/'DiStInCtIvE'
/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 in most cases is not burdened with this chore. User interface code provides 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. [2] Thus, the user usually is 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.

2.3.2 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 returns all versions

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 integer 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 checkpoint file 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 2.3.1 and 2.4 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.

2.3.3 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.

2.4 Symbolic Links

As already outlined in section 2.2, 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 [3] and Unix [6] 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*, *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 2.3.2, 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.

Although an argument can be made that one might want the default behavior to "write on" rather than "write through" a link, Sesame provides that capability with the *CopyName* primitive. *CopyName(OldName,NewName)* makes a copy of the entry for *OldName* with the name *NewName*. In the case of files, for instance, the FID 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(name)* requests would enter a new version for the name specified, but leave the copy under the other name untouched. Compare this to the behavior of making a link to the file by *Link(LinkName, OldName)*. Now when a *WriteFile(LinkName)* is performed *LinkName* is expanded to be *OldName*, and a new version of *OldName* 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.

2.5 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 combination 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 uses auxiliary access bits on directories to implement its 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.

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 FID 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

The Name Server controls access to directories as well as to the individually named objects. The Name Server uses the auxiliary access bits on the access control list of a directory entry to define a set of access rights for that directory.

The directory auxilary access rights are:

CreateNames new names may be added to the directory

DeleteNames names may deleted from the directory (see section 2.6)

ReadNames the directory names and name types may be examined

UseNames objects in the directory may be searched for and referenced. UseNames in the directory is required along with the appropriate rights on the object for any action on an object.

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.

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.

- The user may provide an access control list in the WriteFile or EnterName call.
- If no access control list is provided then:
    - If a previous version exists, the access control list from the highest existing (non-deleted) version is used.
    - If no previous version of the name exists, the default access control list 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.

2.6 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 queued 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. A directory must be empty before it can be deleted.

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 entry name and FID can only be recovered via a retrieval request from the Archive Server (see section 2.7). Non-file type entries cannot be recovered. Users will, in general, never directly use the *ExpungeName* operation except in unusual circumstances.

2.7 Recovering expunged names

Corresponding to every directory is a Archived Name Index. This is a data base to which only the Archive Server has direct access. Each time a file is sent to the Archive Server its name (only one name per file) is entered in a Name Index. The Name Indices are never deleted or truncated, but may be migrated to off-line storage. Sesame supports a file retrieval service which can be used to recover the file that corresponded to a given pathname on a certain date and time.

2.8 Migration policy

A migration policy data base is kept local to each machine. Information is kept there about the way to manage objects on the local disk. There is one piece of global information in the policy table. This is the Default Partition list. This is a list of disk partitions on which to store files if no other partition is specified. All other policy information is kept on a per directory or subtree basis. Thus when descions are to be made about a file, the policy that applies to its most immediate superior directory that can be found in the policy table is used. The first time Sesame is started on a machine, it creates a policy entry for the root. That policy applies to all files until entries for more directories are created. The migration policy on a machine is setable by the administrator of that machine. In the case of a personal Spice machine this is the owner. In the case of public Spice machines or the central service machines this is the system administrator.

The parameters kept for each subtree are:

- Whether or not to archive files. If archiving is not allowed the files are kept only locally.
- Whether or not files are to be automatically removed from the local machine. If migration is not allowed the files will always be available on the local disk.
- The caching policy for names and files which may be one of the following:

UseCache — When a remote name is referenced use cached information about that name if any exists otherwise get the information from the remote directory and enter it in the locally cached directory. This is the recommended setting for directories that are large and only some of the entries are of interest.

CacheEntire — When the first name in this directory is referenced, cache all the names in the directory. Then subsesquent references to any name in the directory can be handled locally. Recommended for heavily used directories and directories that are placed on search lists.

UsePrimary — Keep a cache of referenced names, but do not use it unless the primary directory is not available. Recommended for directories where the most current version of rapidly changing files is needed, but where old versions are better than none.

NoCache — Always go to primary directory, keep no names or files cached on the local machine. Recommended for directories containing files too large to be cached on the local disk or where older versions are not better than none.

- Archival time delay (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 delay (i.e. the delay between a delete name request and the removal of the name and possibly the object).

- Cache refresh time delay (i.e. how long the cached names may be used before a request triggers a call to the primary directory to update all the cached values).

- The name of the partition on which to place files with this pathname. This partition is used when a file is created on or migrated to this machine. The name may be a single disk partition or may be specified as **Default_Partition** in which case the member of the default partition list which has the most space on it at the time at the file creation is used.

3 File Service

3.1 Overview

The basic function of the File Service is storing files on the local disk and retrieving files from the local disk or from any other machine in the Spice environment. 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.

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. Since a FID refers to a unique set of data any copy of a file that matches the FID being sought can be used. There is no need to check last modified dates or versions at this level. That type of checking is done at the name lookup stage.

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.

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 disk pack the file might be found. If a local File Server fails to find the file locally it forwards the request to its preferred Central File Server. The Central File Servers keep a table of packids and servers who have the packs mounted. A Central

File Server will first look for a copy of the file on its local disk, and then see if it has a port to server that corresponds to the packid hint in the FID. If so the request is forwarded to that server. If not the request is forwarded to a Central Archive machine.

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 Service interface assumes that both it and its users have available a large amount of virtual memory and demand paging such as the Accent[3] Kernel [4] 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).

3.3 File Service functions

The basic function of the File Service 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 server 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 server.

The File Service functions are:

- to locate and return files stored on the local disk.
- to forward requests for files that are not found on the local disk.
- to keep track of the total disk space that is used and 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, local access date, author ID, local reference count, etc.

---

[3] Accent is a trademark of Carnegie Mellon University

4 Authorization Service

4.1 Overview

A user gains access to the resources and services of Sesame and other Spice services by being a member of one or more *Access Groups* as defined in section 4.2. It is the responsibility of the Authorization Service to maintain the access group lists. The Authorization Servers implementing this service must be secure and available. Thus the Authorization Service is provided by servers running on at least two of the trusted central machines, all maintaining the same replicated data base. Maintaining a consistent replicated authorization database is practical because it is infrequently updated. There is no local Authorization Server, so a user on a detached node is not able to call any of the authorization primitives.

4.2 Access Groups

An access group defines a subset of users to whom privileges may be awarded or from whom privileges may be revoked. There are two types of access groups: *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" are used interchangeably in this document. Primary access groups may only be created by the *system administrator* - a highly privileged user. Secondary access groups may be created by any user and can consist of a list of primary and secondary access groups. Only primary access groups may login.

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 a *verified* member (see Section 5.1) . The user may establish a restricted sublogin 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.

An access group is uniquely identified by a 32 bit integer called the *access group ID*. Each access group also has a name and this name must also be unique. For primary access groups, this is the name under which a user logs in to the Verification Server. Access groups may be renamed, but their IDs cannot be altered. In addition each access group may be given a *fullname* which is typically used to provide a more descriptive name of the group. The other important information associated with each access group is a list of which other groups this group is a member, and, for secondary access groups only, a list of the members of this group. (For more details on information associated with access groups see appendix C.) Every user is automatically a member of the group called *ALL*: the default group consisting of all Sesame users.

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. Thus it is possible to construct a hierarchy of secondary access group by allowing a secondary access group to have members which are other secondary access groups. The membership relationship is transitive and a user effectively belongs to more access groups than just the ones of which he is a direct member. Consequently to find what rights a user has, the authorization server must compute the *reflexive transitive closure* of the user's memberships. This computation is normally done at login time and yields a list of all the groups of which that user is a member. Thus the user's access rights are the union of the access rights of each group in the access group list. When a user logs in to the Verification Server (see the next section), the private connection returned to him has associated with it this membership list.

4.3 Rights on Access Groups

An access group is protected against unauthorized modification by an access control list mechanism. Each access group has an access control list associated with it and this list consists of a set of access group IDs and the rights that each of them has on that access group. The access rights are:

- ***AddMembers*** : members may be added to a secondary group.
- ***DeleteMembers*** : members may be removed from a secondary group.
- ***DeleteGroup*** : a group may be deleted (or renamed).
- ***ChangePassword*** : the password of a primary access group may be changed.
- ***ChangeFullName*** : the full name of an access group may be changed.
- ***GetAuxAccess*** : the access control list for a group may be read.
- ***SetAuxAccess*** : the access control list for a group may be modified.

5 Verification Service

5.1 Overview

The Verification Service provided by Sesame permits the verification of a user's identity (in cooperation with the authorization service) and the registration of tokens with which users can prove their identity to other Spice services. The initial verification of a user is done by presenting a login name and password to a verification server. The user sends this information to a Local Verification Server, which is assumed to be running on the user's machine. The local server then encrypts this information and forwards it to a Central Verification Server which performs the actual authentication. The software on the Central Verification Server is trusted to check the authenticity of the users and can also be trusted to provide verification between users on different machines. The Local Verification Servers can be used to provide verification between processes on the local machine and also to provide a limited verification service when the Central Verification Servers are unavailable.

Since it is important that there should always be a verification service available, there will be Central Verification Servers running on at least two of the trusted machines. All of the central servers retain the same dynamic information and communicate with each other to maintain consistency. In particular all login and registration request must be reflected to all of the Central Verification Servers. This reflection is the responsibility of the Central Verification Server to which the request is initially addressed. Since all the central servers have the same information, verification requests can be directed to any central server.

5.2 Establishing a connection

To establish a secure connection with a Central Verification Server, a login message, consisting of a Sesame user name and an encrypted login code (see Section 5.4), is sent to the server's public port. Normally this connection is established when a user logs into a local machines, but it may occur explicitly at the request of the user. The login request made by the user, whether explicit or implicit, is sent to the Local Verification Server, which then constructs the login code and forwards the request to a Central Verification Server. A successful login results in a private port being established for the user at the Verification Server. The user can use the private port to make further authenticated requests to the server. The server associates this private port with the user's identity and with the list of access groups (see Section 4.2 above) of which the user is a member.

The connection to the Verification Server is terminated by sending a logout message to the user's private port. This deallocates the private port, deallocates any private ports created by sublogin requests, and removes any state associated with the user. The verification connection may also be broken implicitly by events such as a time-out caused by the user's machine crashing.

5.3 Proving identity

Other servers in Spice need to be able to identify a client process that is requesting service or access to a resource, so that the server can identify what access rights the client has to the service or the resource. The Verification Server provides a registration service for this purpose. Once a user has a private port connection to a Verification Server, the user can *register* ports that he has created. The user can then present a registered port to a server as a token of his identity. To determine the user's identity, the server *verifies* the registered port via its own connection to the Verification Server and receives back the user's name and a list of access groups to which the user belongs.

Sometimes a server is required to do work on behalf of a user and this work requires access to a user's resource at a second server. To allow the first server to access the user's resource, the user must pass it a registered port. But doing this produces the undesirable result of the first server obtaining access to all of the user's resources at all servers. To avoid this problem the user must give the first server a registered port which

only allows access to the resources that the first server needs to access. To obtain such a registered port the user must first perform a *restrict* operation which results in a private port associated with a subset of the user's access groups. Registering a port via this new private port results in a port which can be used as a token representing the restricted rights and can be safely passed to the first server.

5.4 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 with which it is in communication.

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 Central Verification Server on a secure machine. The Central Verification Server uses the stored password to decrypt the login code and find out the encryption key. The presence of name in the encrypted message provides enough redundancy for the Central Verification 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 Central Verification 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 cleartext. Subsequent logins to the central machine can be distinguished from the initial login because they will be sent encrypted whereas the initial login code was sent in an unencrypted message.

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 [1] 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. During the implementation stage Jeff Eppinger made significant suggestions for improving the robustness of the distributed servers. Credit is also due to members of the Perq Systems 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. Directory fields

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

- **Directory information**
  - Default access control list
  - Default number of versions of names to retain
  - Directory status (primary or advisory)
  - Synchronization time (if primary, time directory was last modified; if cached, time directory was last updated from primary)
  - Network name of server with primary directory (hint)
  - Port to server with primary directory (may be null)

Entry Information

- Name of Entry -- just this component of the pathname for the entry
- Version Number of Entry
- Status of Entry -- undeleted/deleted
- 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	index of the IPC port and date that port was entered
User Defined	A block of storage 255 bytes long

B. 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

C. Access Group Information

For every access group in the system the Authorization Servers maintain the following information:

Access Group ID	a 32 bit integer that uniquely 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 for files.
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. The full name need not be unique.
Group Type	Primary or Secondary.
Password	For primary access groups this is the login password. It is not present for secondary access groups.
Members List	the list of direct (i.e., without applying transitive closure) members of this access group. It is not present for primary access groups.
Membership List	the list of access groups of which this access group is a direct member.

D. Summary of Sesame calls

Name Service Functions

```
Function SubReadFile(ServPort: Port;var APathName: APath_Name;var Data: File_Data;
        var Data_Cnt: long): GeneralReturn;

Function SesReadFile(ServPort: Port;var APathName: APath_Name;var Data: File_Data;
        var Data_Cnt: long;var DataFormat: Data_Format;
        var CreationDate: Internal_Time;var NameStatus: Name_Status
            ): GeneralReturn;

Function SesReadBoth(ServPort: Port;var APathName: APath_Name;var Data: File_Data;
        var Data_Cnt: long;var FileHeader: File_Header;
        var NameStatus: Name_Status): GeneralReturn;

Function SubWriteFile(ServPort: Port;var APathName: APath_Name; Data: File_Data;
        Data_Cnt: long; DataFormat: Data_Format;
        var CreationDate: Internal_Time ): GeneralReturn;

Function SesWriteFile(ServPort: Port;var APathName: APath_Name; Data: File_Data;
        Data_Cnt: long; DataFormat: Data_Format;
        PrintName: Print_Name; AccessFlags: Access_Flags;
        ACLp: pAccControlList; ACLp_Cnt: long;
        var CreationDate: Internal_Time): GeneralReturn;


Function SesSetCreationDate(ServPort: Port;var APathName: APath_Name;
        CreateDate: Internal_Time ): GeneralReturn;

Function SesGetFileHeader(ServPort: Port;var APathName: APath_Name;
        var FileHeader: File_Header): GeneralReturn;

Function SubLookUpName(ServPort: Port;var APathName: APath_Name;
        var EntryType: Entry_Type;var EntryData: Entry_Data;
        var NameStatus: Name_Status): GeneralReturn;

Function SesLookUpName(ServPort: Port;var APathName: APath_Name;
        NameFlags: Name_Flags;var EntryType: Entry_Type;
        var EntryData: Entry_Data;var NameStatus: Name_Status
            ): GeneralReturn;

Function SubTestName(ServPort: Port;var APathName: APath_Name;
        var EntryType: Entry_Type;var NameStatus: Name_Status
                ): GeneralReturn;

Function SesTestName(ServPort: Port;var APathName: APath_Name;
        NameFlags: Name_Flags;var EntryType: Entry_Type;
```

```
        var NameStatus: Name_Status): GeneralReturn;

Function SubEnterName(ServPort: Port;var APathName: APath_Name;
        EntryType: Entry_Type; EntryData: Entry_Data
            ): GeneralReturn;

Function SesEnterName(ServPort: Port;var APathName: APath_Name;
        NameFlags: Name_Flags; EntryType: Entry_Type;
        EntryData: Entry_Data; AccessFlags: Access_Flags;
        ACLp: pAccControlList; ACLp_Cnt: long): GeneralReturn;

Function SubDeleteName(ServPort: Port;var APathName: APath_Name ): GeneralReturn;

Function SesDeleteName(ServPort: Port;var APathName: APath_Name;
        NameFlags: Name_Flags; ExpungeFlag: boolean
            ): GeneralReturn;

Function SesUndeleteName(ServPort: Port;var APathName: APath_Name;
        NameFlags: Name_Flags ): GeneralReturn;

Function SesExpungeDirectory(ServPort: Port;var APathName: APath_Name
            ): GeneralReturn;

Function SubReName(ServPort: Port;var OldAPathName: APath_Name;
        var NewAPathName: APath_Name ): GeneralReturn;

Function SesReName(ServPort: Port;var OldAPathName: APath_Name;
        NameFlags: Name_Flags;var NewAPathName: APath_Name
            ): GeneralReturn;

Function SubCopyName(ServPort: Port;var OldAPathName: APath_Name;
        var NewAPathName: APath_Name ): GeneralReturn;

Function SesCopyName(ServPort: Port;var OldAPathName: APath_Name;
        NameFlags: Name_Flags;var NewAPathName: APath_Name
            ): GeneralReturn;

Function SesScanNames(ServPort: Port;var WildAPathName: Wild_APath_Name;
        NameFlags: Name_Flags; EntryType: Entry_Type;
        var DirectoryName: APath_Name;var EntryList: Entry_List;
        var EntryList_Cnt: long ): GeneralReturn;

Function SesGetRetentionCount(ServPort: Port;var APathName: APath_Name;
        var RetCount: integer ): GeneralReturn;

Function SesSetRetentionCount(ServPort: Port;var APathName: APath_Name;
        RetCount: integer ): GeneralReturn;

Function SesGetDefRetentionCount(ServPort: Port;var APathName: APath_Name;
        var RetCount: integer ): GeneralReturn;
```

```
Function SesSetDefRetentionCount(ServPort: Port;var APathName: APath_Name;
            RetCount: integer ): GeneralReturn;

Function SesConnect(ServPort: Port; RegPort: Port;var NSPort: Port
            ): GeneralReturn;

Function SesDisconnect(ServPort: Port ): GeneralReturn;

Function SesSynchronize(ServPort: Port ): GeneralReturn;

Function SesReconnectCFS(ServPort: Port ): GeneralReturn;
```

Caching Policy Functions

```
Function SesSetDirPolicy(ServPort: Port;var APathName: APath_Name;
            PolicyRecord: MP_Record ): GeneralReturn;

Function SesGetDirPolicy(ServPort: Port;var APathName: APath_Name;
            var PolicyRecord: MP_Record;var inherited: boolean
                ): GeneralReturn;
Function SesDelDirPolicy(ServPort: Port;var APathName: APath_Name): GeneralReturn;

Function SesGetDefaultPartList(ServPort: Port;var DefPartList: APath_Name
                ): GeneralReturn;

Function SesSetDefaultPartList(ServPort: Port; DefPartList: APath_Name
                ): GeneralReturn;
```

Cache Control Functions

```
Function SesScanCacheNames(ServPort: Port;var WildAPathName: Wild_APath_Name;
            NameFlags: Name_Flags; EntryType: Entry_Type;
            var DirectoryName: APath_Name;var EntryList: Entry_List;
            var EntryList_Cnt: long ): GeneralReturn;

Function SesPurgeCache(ServPort: Port;var APathName: APath_Name): GeneralReturn;

Function SesUpdateCache(ServPort: Port;var APathName: APath_Name): GeneralReturn;

Function SesCheckoutDir(ServPort: Port;var APathName: APath_Name): GeneralReturn;

Function SesCheckinDir(ServPort: Port;var APathName: APath_Name): GeneralReturn;
```

Access Control Functions

```
Function SesGetAccess(ServPort : Port; var APathName: APath_Name;
            NameFlags : Name_Flags; AccessFlags: Access_Flags;
            var AccList: pAccControlList; var AccListCnt: long
                ) : GeneralReturn;

Function SesSetAccess(ServPort: Port;var APathName: APath_Name;
            NameFlags: Name_Flags;AccessFlags: Access_Flags;
            AccList: pAccControlList;AccListCnt: long
                ) : GeneralReturn;

Function SesCheckAccess(ServPort: Port;var APathName: APath_Name;
            NameFlags: Name_Flags;AccGrpList: Access_ID_List;
            AccGrpListCnt: long;var Rights: RightsMask
                ) : GeneralReturn;

Function SesGetDefAccess(ServPort: Port;var APathName: APath_Name;
            AccessFlags: Access_Flags;var AccList: pAccControlList;
            var AccListCnt: long) : GeneralReturn;

Function SesSetDefAccess(ServPort: Port;var APathName: APath_Name;
            AccessFlags: Access_Flags;AccList: pAccControlList;
            AccListCnt: long) : GeneralReturn;

Function SesCheckDefAccess(ServPort: Port;var APathName: APath_Name;
            AccGrpList: Access_ID_List;AccGrpListCnt: long;
            var Rights: RightsMask) : GeneralReturn;
```

File Service Functions

```
Function FS_SubReadFile(ServPort: Port;DelayedReplyPort: Port;var FID: File_ID;
            var Data: File_Data;var Data_Cnt: long ): GeneralReturn;

Function FS_ReadBoth(ServPort: Port; DelayedReplyPort: Port;
            var FID: File_ID; var FSHeader: File_Header;
            var Data: File_Data; var Data_Cnt: long)
                : GeneralReturn;

Function FS_WriteFile(ServPort: Port;Data: File_Data;Data_Cnt: long;
            DataFormat: long;PrintName: Print_Name;AuthorID: Group_ID;
            NonMigrate: boolean;PartName: DevPartString;
            var FID: File_ID;var CreationDate: Internal_Time
                ): GeneralReturn;

Function FS_SetCreationDateServPort : Port; FID: File_ID;
            CreateDate: Internal_Time): GeneralReturn;

Function FS_GetFSHeader(ServPort: Port; DelayedReplyPort: Port;
```

```
        var FID: File_ID;
        var FSHeader: File_Header)
            : GeneralReturn;

Function FS_DeleteFile(ServPort: Port; FID: File_ID; force: boolean)
            : GeneralReturn;

Function FS_IncRefCnt(ServPort: Port; FID: File_ID): GeneralReturn;

Function FS_DecRefCnt(ServPort: Port; FID: File_ID): GeneralReturn;

Function FS_MountPartition(ServPort : Port; PartName : DevPartString;
        PartPort: Port; PTSegNum: SegID; PartS: long;
        PartE: long; DefPartList: APath_Name): GeneralReturn;

Function FS_DismountPartition(ServPort: Port; PartName: DevPartString)
            : GeneralReturn;
```

Authorization Service Functions

```
function AuthorConnect (PublicPort : port; ValidPort : port;
        var ARPort : port) : GeneralReturn;

function CreateGroup (ARPort : port; AccGpName : UserNameString;
        AccGpType : AGType; FullName : FullNameString;
        var AccGpID : Group_ID) : GeneralReturn;

function ChangePassword (ARPort : port; UserName : UserNameString;
        OldPwd, NewPwd : PassWordString) : GeneralReturn;

function ChangeFullName (ARPort : port; UserName : UserNameString;
        FullName : FullNameString) : GeneralReturn;

function DeleteGroup (ARPort : port; AccGpName : UserNameString)
            : GeneralReturn;

function RenameGroup (ARPort : port; OldName, NewName : UserNameString)
            : GeneralReturn;

function AddToGroup (ARPort : port; ModGroup, AddGroup : UserNameString)
            : GeneralReturn;

function RemoveFromGroup (ARPort : port; ModGroup : UserNameString;
        RemGroup : UserNameString) : GeneralReturn;

function TrGroupName (ARPort : port; AccGpName : UserNameString;
        var AccessGpType : AGType; var AccGpID : Group_ID;
```

```
        var Fullname : FullNameString) : GeneralReturn;

function TrGroupID (ARPort : port; AccGpID : Group_ID;
        var AccessGpType : AGType; var AccGpName : UserNameString;
        var Fullname : FullNameString) : GeneralReturn;

function ListMembers (ARPort : port; AccGpName : UserNameString;
        TransClos : boolean; ListOpt : ListOptions;
        var NameList : Access_Name_List; var NameList_Cnt : long;
        var IDList : Access_ID_List; var IDList_Cnt : long)
            : GeneralReturn;

function ListMemberships (ARPort : port; AccGpName : UserNameString;
        TransClos : boolean; ListOpt : ListOptions;
        var NameList : Access_Name_List; var NameList_Cnt : long;
        var IDList : Access_ID_List; var IDList_Cnt : long)
            : GeneralReturn;

function ListGroups (ARPort : port; ListOpt : ListOptions;
        var NameList : Access_Name_List; var NameList_Cnt : long;
        var IDList : Access_ID_List; var IDList_Cnt : long)
            : GeneralReturn;

function GetAuxAccess (ARPort : port; OnGroupName : UserNameString;
        OfGroupName : UserNameString; var right : ASRight)
            : GeneralReturn;

function SetAuxAccess (ARPort : port; OnGroupName : UserNameString;
        right : ASRight; OfGroupName : UserNameString;
        add : boolean) : GeneralReturn;

function GetPassword (ARPort : port; UserName : UserNameString;
        var Password : PassWordString; var User_ID : Group_ID)
            : GeneralReturn;

function ListRights (ARPort : port; AccGpName : UserNameString;
        var IDList : Access_ID_List; var IDList_Cnt : long;
        var RightsList : Access_ID_List; var RightsList_Cnt : long)
            : GeneralReturn;
```

Verification Service Functions

```
function Sendlogin (VSPort : port; Username : UserNameString; Code :LoginCode;
        var PrivatePort : port; var User_ID : Group_ID;
        var GroupIDs : Access_ID_List; var GroupIDS_Cnt : long)
            : GeneralReturn;

function Locallogin (VSPort : port; UserName : UserNameString;
```

```
                Password : PassWordString; var PrivatePort : port)
                    : GeneralReturn;

function Logout (VSPort : port) : GeneralReturn;

function Register (VSPort, UserPort : port) : GeneralReturn;

function DeRegister (VSPort, UserPort : port) : GeneralReturn;

function Restrict (VSPort : port; CanRestr, KnowsUserName, Remove : boolean;
                IDs : Access_ID_List; IDs_Cnt : long;
                var NewPort : port; var Rest_IDs : Access_ID_List;
                var Rest_IDs_Cnt : long) : GeneralReturn;

function Verify (VSPort, UserPort : port;
                var UserName : UserNameString; var UserID : Group_ID;
                IDoption : ListOptions; var IDs : Access_ID_List;
                var IDs_Cnt : long) : GeneralReturn;

function VS_CVSReConnect (VSPort : port; CVSPort : port) : GeneralReturn;
```

References

[1] Accetta, M., Robertson, G., Satyanarayanan, M. and Thompson, M.
The Design of a Central File System for a Local Network.
Technical Report CMU-CS-80-134, Department of Computer Science, Carnegie-Mellon University, 1980.

[2] Spice Documentation Group.
The Spice Programmers's Manual: The Server Manual.
Technical Report, Department of Computer Science, Carnegie-Mellon University, 1984.

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

[4] 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.

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

[6] 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.