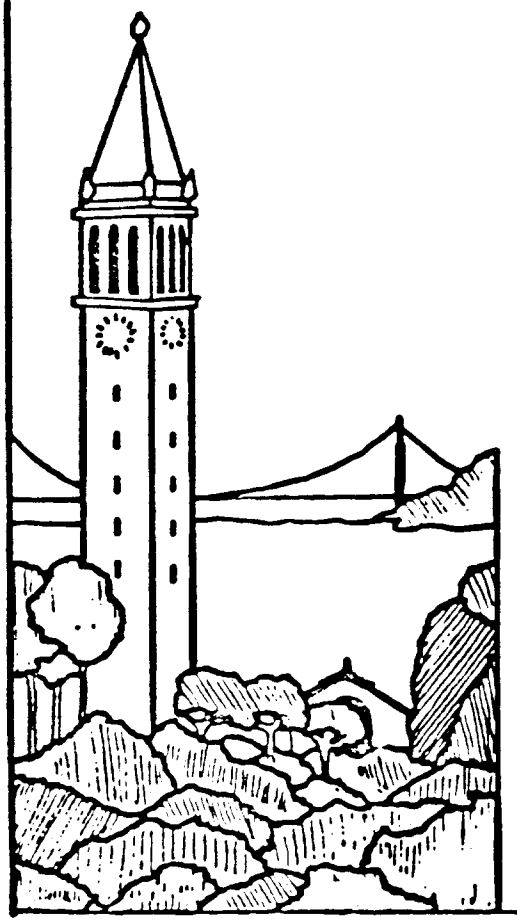
GAFFES:
The Design of a Globally Distributed File System

Shai Gozani, Mary Gray, Srinivasan Keshav,
Vijay Madisetti, Ethan Munson, Mendel Rosenblum,
Steve Schoettler, Mark Sullivan and *Douglas Terry*



Report No. UCB/CSD 87/361

June 1987

Computer Science Division (EECS)
University of California
Berkeley, California 94720

GAFFES: The Design of a Globally Distributed File System*

Shai Gozani
Mary Gray
Srinivasan Keshav
Vijay Madisetti
Ethan Munson
Mendel Rosenblum
Steve Schoettler
Mark Sullivan
Douglas Terry

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720
Telephone: (415)642-5250

ABSTRACT

GAFFES, the Global, Active and Flexible File Environment Study, is a file system designed to facilitate the sharing of information in a global network serving more than a million workstations. We have assumed that this global network has widely varying communication speeds and includes mutually suspicious domains. GAFFES is highly distributed in order to promote high capacity, modular growth, increased availability, and natural autonomy. In spite of the system's distributed nature, users are presented with the same basic service regardless of their location in the environment.

GAFFES provides four services which handle naming, replication and caching, security and authentication, and file access primitives (including the triggers which make files active). These services are to a large extent independent, having little interaction without the direct mediation of the user. This independence requires each service to assure its own reliability, availability and performance.

* This work was partially sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

Preface

Someday, world-wide communication networks will connect millions of computers residing in people's homes, offices, cars, and various public places. People will be able to share information easily like never before. One can envision global "knowledge bases" that permit individuals and corporations ready access to enormous amounts of publicly available data. Students in this past semester's course, "Advanced Topics in Distributed Computing Systems", were given the assignment of exploring techniques for building, updating, accessing, and controlling such large storage systems.

This report presents the outcome of these explorations: an architecture for a globally-distributed active file system. Shai Gozani and Steve Schoettler looked at replication mechanisms; Mary Gray and Ethan Munson worked on how to name and locate files; Srinivasan Keshav and Mark Sullivan developed techniques for security and authentication among mutually suspicious organizations; and Vijay Madisetti and Mendel Rosenblum defined the file system semantics and trigger facilities. The resulting design is but a first step towards tackling the problems of large-scale distributed computing.

Doug Terry
June 1987

1. Introduction

1.1. The Environment of GAFFES

GAFFES is a design for a very large-scale distributed file system intended to serve on the order of several million users who could potentially access any file in the system. Clients could use GAFFES as a tool for the storage and retrieval of information ranging from on-line newspapers to computer programs to electronic mail. GAFFES must guarantee certain levels of performance, availability and security in order to satisfy its users.

The network over which GAFFES is designed to operate could connect computers having a wide range of performance and running many different operating systems. The performance of future networks may continue to vary from that of low-speed telephone lines to high-speed optical fibers. GAFFES must allow the user to make use of knowledge about the speed of the network without requiring such knowledge from all users.

The client workstations connected by this network would be quite heterogeneous, ranging in power from desktop personal computers to large mainframes. GAFFES must provide services that are both sufficiently simple and sufficiently powerful to meet the needs of this complex client mixture.

The heterogeneity seen in the hardware and system software of the network extends into its administration. A global network naturally crosses many national boundaries. Each nation's portion of the network would likely be managed by a different organization (usually the national phone company). For security and economic reasons, some corporate and government entities may manage their own independent subnets. The corporate and government subnets would be particularly concerned with assuring the privacy of their files.

1.2. Assumed Facilities

GAFFES was designed to be built on a base of transactional file servers and RPC communications. The file servers provide means for storing unstructured files (each file is a logical sequence of data bytes), identifying files uniquely, and performing common file operations (read, write, etc.). The file servers also allow operations on multiple files stored on multiple servers to be performed with transactional guarantees (serializability and atomicity). The RPC communication mechanism permits reliable transmission of structured (typed) data. RPC communication can be performed between any two machines in the network, though no performance guarantees are made.

Each client workstation and server machine has a globally-unique identifier. This identifier is not necessarily human-readable. GAFFES also assumes that each workstation and server contains hardware encryption devices that can be used in security procedures. Mechanisms exist for generating the public-key/private-key pairs used by the security system on demand.

1.3. Required Features

From the start, GAFFES was expected to provide clients with many of the operations and assurances that one expects from a single-machine file system. In addition, some of GAFFES' features represent extensions to traditional file systems that we considered important.

Files in GAFFES may contain embedded references to other files. Files can be made 'active' through the use of triggers, which are programs invoked by file operations. Examples of the uses of triggers are found in automatic compilation or remote distribution of files.

GAFFES files have unique names that are independent of a user's location, the file's location, or the number of replicas of the file. This supports sharing and shelters users from many of the complexities of the distributed file system. Moreover, to reduce the complexity of searching for files, clients may identify files with descriptive names that allow files to be accessed based on their content.

The security component of GAFFES is designed so that each organization has autonomous control over the security of its own files. The rights of individual users and groups of users to perform operations on a file may be freely granted or revoked. The security system provides a uniform mechanism for authenticating users and ensures the privacy of both data and operations from unauthorized

users.

In any file system, choices are made about the relative importance of performance and availability and about the consistency of shared files. Although GAFFES is intended to be used more as a general-purpose information distribution and sharing facility than as a high-speed transactional system, the GAFFES community would be far too diverse to be served by a single alternative among the various performance, availability and consistency tradeoffs. Therefore, users of GAFFES achieve desired levels of availability and performance through individual decisions about the replication and caching of their files. The creator of a file specifies its degree of replication, thereby determining its availability. Each client performing a file operation specifies whether caching should be used and what degree of consistency is desired for cached copies.

1.4. Overview of Solution

The GAFFES global file system has five major components:

(1) A set of file server primitives;

(2) A name service;

(3) A security system;

(4) A replication service;

(5) A trigger mechanism through which files are made active.

The five components of GAFFES are largely independent of each other. For example, while the naming service stores information about the location of replication servers and about whether files are replicated, it has no knowledge of where file replicas are stored and no means of determining this information other than the operations available to clients. We believe that this independence is a feature that sets GAFFES apart from other distributed systems. To the best of our knowledge, existing system designs either lack this independence or assume that independent subsystems will be available without actually providing them. We show that designing a system comprised of these independent components is possible, while highlighting the problems that arise and the compromises that are necessary to join these components together.

1.5. What Follows

This report is divided into eight sections, the first being this introduction. The next section describes the semantics of the file system. The naming service is discussed in the third section. The following two sections cover the security and replication systems. The seventh section describes triggers. The final section presents overall conclusions. GAFFES is an experimental design; it was not built, nor is it intended to be built.

1.6. Acknowledgements

We gratefully acknowledge Professor Domenico Ferrari for providing the support that made this project possible, and Dr. Dale Skeen for his contributions. We also thank Stuart Sechrest and P. Venkat Rangan for their suggestions and help with the security section of this paper.

2. File System Semantics

In this section on the file system semantics of GAFFES, we describe the design decisions supporting embedded references, version histories for files, transactions, and transparent distributed file access. These design decisions affect the organization and structure of files, and the client interface to the file system. An embedded reference is the name of a file or part of a file which is contained in another file. Embedded references can be used to express relationships between files such as inclusions. We discuss the design and use of triggers for active files in a later section.

2.1. Design Goals for File System Semantics

In a diverse environment there will be clients, referred to in this paper as **complex clients**, who demand high performance and access to the complex features of the file system. Complex clients may want to interface closely with the replication, name, and authentication services to obtain the highest performance and control of the services provided. There will also be clients, called **simple clients**, who may require only simple reading and writing of files. Simple clients would rather not have to deal with very complex interfaces and file system semantics. The design of the semantics should cater to both simple and complex clients.

Implementing functionality on the servers, which would otherwise be implemented on each client machine, has the potential of massive reduction in implementation cost. It is much more desirable to implement a function once on the servers rather than once for each type of client on the system. On the other hand, if the servers provide functionality, client interfaces to the servers can become more complex and harder to implement. Simple clients who desire only a subset of the functionality, may be forced into choosing between a very expensive full implementation of the interface or a partial implementation that may make some information in the file system inaccessible.

Another design consideration is how best to map the presented file system into the client environment. It should be possible to fit the semantics of the file system into a wide range of operating environments.

The design of the file system semantics caters to both complex and simple clients. To accomplish this goal, the design provides the functionality desired by complex clients allowing high performance access to the file system. On top of this interface, the servers also provide a less complex interface for simple clients. Whenever possible, the interface is structured so complex clients can set up transparent access to advance features for simple clients. For example, complex clients can create embedded references, version control mechanisms, and triggers, which are transparently invoked by simple clients' operations.

2.2. File Organization

Files are defined as a contiguous sequence of client-definable blocks. Division of files into blocks lends itself gracefully to the use of embedded references. Intermixing embedded reference control and data in an otherwise unstructured file leads to both implementation and client interface problems. Having embedded references located in special blocks simplifies the task of keeping track of embedded references for house keeping functions such as garbage collection.

In addition to the role blocks play in resolving embedded references, a file system providing this service on the server relieves the client operating system from the need to provide its own file 'blocking' system. In a large scale distributed environment, providing for the division of files into blocks on the servers is in itself a software advantage. The structure of a file into blocks is convenient for many applications. In the case of a mailing system, mail messages for users can be stored as separate blocks of an *inbox* file.

To support both complex and simple clients, two views of file organization are provided. These are the "**logical**" and the "**virtual**" views.

2.2.1. The Logical View

In the logical file view, the file is organized as a sequence of blocks of two types, data blocks and embedded reference blocks. A data block is a logically contiguous array of zero or more bytes of data. Interpretation of the bytes in a data block is left to the clients; no interpretation is done by the file system servers. Clients may create files with any number of data blocks of arbitrary size. The sizes of data blocks in a file are not fixed, and they can be extended or shortened independently from other blocks in the file.

Embedded reference blocks are blocks containing a name of another file and, optionally, a range of numbers of logical blocks of that file. An embedded reference which does not specify a block range implies a range which includes every logical block of the file. Logical blocks are addressed as integers

numbered from one with each block, embedded reference or data, given a unique consecutive integer.

2.2.2. The Virtual View

The virtual file view presents files as a sequence of data blocks. All embedded references in this view are automatically followed and resolved to data blocks. A client using this view sees a file as an array of data blocks which may come from different logical files.

Virtual file blocks are addressed as a sequence of data blocks from one for more different logical files which are numbered as integers from one. Figure 2.1 shows the differences between logical and virtual files. The block numbers on the right side of files in the figure are logical numbers, while numbers on the left are virtual block numbers. Note that when a file includes no embedded references, as in the case of file "foo" and "foo3" in figure 2.1, the logical block numbers are the same as the virtual. Virtual block numbers come from viewing the file as consecutive data blocks which may include data blocks from other logical files via embedded references. Using virtual block numbers, the same data block is assigned different block numbers in different files. For example, virtual block number 4 of file "foo2" is logical data block 3 of file "foo3". This data block is also virtual block 11 of file "myfile". File "myfile" has 8 logical blocks and 12 virtual blocks.

We decided that the ability to include a file without knowledge of the number of blocks in that file is useful to clients. Including this feature in the design led to unfortunate file semantics. Consider the case of a client adding blocks to a file that is included as a embedded reference in another file. In figure 2.1, this would be the case if a client added blocks to file "foo2". Such addition of blocks may cause some of the virtual blocks of the the file embedding the enlarged file to become renumbered. We

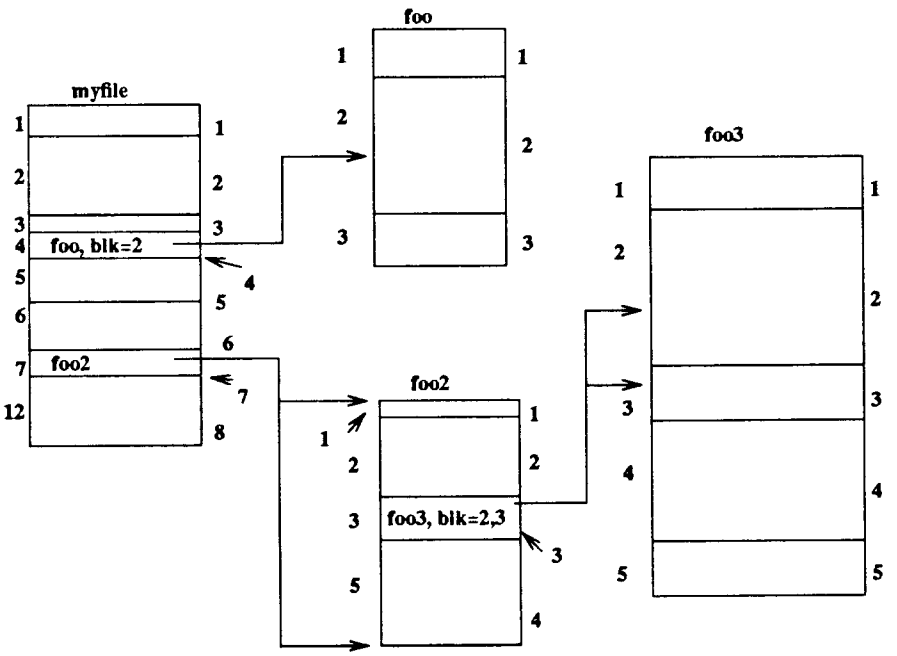


Figure 2.1 — Logical and Virtual Block Numbering

anticipate that correctly operating clients will avoid this situation by including only immutable files, such as files qualified with a version number, in an embedded reference. The usefulness of including files of unknown block counts outweighed the potential hazards introduced.

2.2.3. Reason for Two Views

The design goal of supporting a wide range of clients was our motivation for providing logical and virtual views of the file system. The virtual view allows simple clients to ignore the existence of embedded reference and treat files as containing only data blocks. Such clients still get to use the inclusion facility of embedded reference yet require no extra code above the normal data block read and write primitives. These semantics allow clients transparently to use embedded references created by complex clients. It would be easier to map the semantics of current common file systems to the virtual view rather than the logical view.

The use of the name and block number of a file in an embedded reference provides some interesting options for clients. Using the naming methods for the file system, an embedded reference can point at a block in a specified version of a file or always point at the most recent version of the file. Embedded files are permitted the same benefits of caching and replication that are provided to all clients of the system.

2.2.4. Operations on Blocks

To access files, clients use the normal read and write routines specifying the block numbers and types, virtual or logical. This is done by having different read and write primitives: **read_logical, write_logical, read_virtual, write_virtual.** When writing blocks by logical block numbers, clients must specify whether the block type is an embedded reference or data, for each block. Logical block reads return the block type along with the data or embedded reference block. Since block sizes vary widely, clients can read many blocks in a single request or just a byte range inside a block. Clients using the virtual view routines can ignore the existence of embedded references and treat every block of the file as a data block. The operation returns an error if an embedded reference is unresolvable to a data block. This error can occur if the user does not have the necessary permissions to access an embedded file, the specified block does not exist, or the block is inaccessible. Additionally, a function is provided for returning the size and type of each block in a view of a file. This function is useful for emulating a byte array file structure, such as found in UNIX, on virtual files.

2.3. Client Interface to the File System

Clients access file system services by doing remote procedure calls to a server machine. To support simple clients, the servers hide the distributed nature of the file system and present it as a single file system which may be accessed through any server. Complex clients can communicate directly with the name, replication, storage, and authentication servers, and are aware of the distributed nature of the file system, while simple clients can treat the file system as a black box. It is not unreasonable for a high performance client to wish to communicate directly with a storage server which contains its files. Simple clients would like a trusted server to perform these tasks on their behalf.

2.3.1. File Location Operations

Files are accessed by retrieving a **handle** for the file from the naming and replication services. Among other things, this handle indicates a list of servers that can efficiently process operations on the file. Clients may send a request with such a handle to any server. If a server gets a request with a handle that does not include it, it will try to forward the request to the correct server for processing. A high performance client would bypass the forwarding step by looking at the handle and communicating directly with the correct servers. This automatic forwarding of requests means that clients need not know how to talk with all servers in the system. This is crucial for supporting very large distributed systems. If each client had to communicate directly with the correct server, the requirements placed on the communication subsystem would scale poorly. The interface provided reduces the problem of requiring every client to know how to communicate with every server to the problem of asking every

server to do be able to do so. The number of servers in the system is much smaller than the number of clients. Existing name server and network management techniques, such as used on the Arpanet, can be used amongst the servers.

Additionally, the system allows simple clients to avoid communicating with name and authentication servers by allowing them to pass a name to the open call rather than a handle. The servers do the necessary handle lookup from the name servers and open the file for the client.

2.3.2. Transaction Management

All client requests must be performed as part of a transaction which is run as a set of atomic serializable actions with respect to all other requests. A traditional transactional file server interface is used which supports nested transactions affecting files on any number of servers. Simple clients may have transactions automatically generated for them by the servers.

2.3.2.1. Explicit Transaction Control

The level of client involvement in transaction management is left up to the client. Complex clients may have explicit control by calling the **open_transaction** primitive, which returns a transaction ID under which operations on files are performed. All requests for operations sent to servers must present this ID. Nested transactions may be created by passing this ID to another open_transaction request. When all the operations of this transaction have returned, the transaction may be committed or aborted by calling **commit_transaction** or **abort_transaction.** The semantics of the operation are totally independent of the location of files. No "add server" call is needed. A server automatically joins a transaction when a file located at its site is used. In addition to the user being able to abort a transaction, the file system reserves the right to abort transactions and inform the user upon the next operation which references that transaction. Clients must be prepared to handle such aborted transactions.

2.3.2.2. Implicit Transaction Control

The transaction-oriented interface may be too complex for simple clients. To support these clients, the file system supports a non-transactional interface. The primitive operations sent without a transaction ID are automatically given a valid transaction ID by the server and processed under this transaction. The client does not have to open or commit the transaction. Each open request creates a transaction under which all reads and writes are processed. Closing the file commits the transaction. The close will return an error if the transaction cannot be committed. Clients can be unaware of any transactions or the distribution of the files in the system.

2.4. Support for Versions

The file system semantics provide support for clients that wish to maintain a version history of files. File system support is provided for creating, accessing and deleting versions of files. The support consists of naming conventions, triggers, and file modification semantics which permit clients to implement version files. The file system provides mechanisms but does not make many policy decisions about versions. Policy decisions such as how many versions to allow and how long to keep old versions are left to the clients to decide.

2.4.1. Naming of Versions

The naming system provides a consistent way to address versions of files. File names may be given with or without a version number specified. We refer to Files names with a version number specified as **qualified names** while those without are termed **unqualified names.**

2.4.2. Version Creation Semantics

The file modification operation depends on whether the file name given is qualified or unqualified. When unqualified file names are used, a copy of the current highest version of the logical file is made and all modifications are made on that copy. When the transaction controlling the

modification commits, the modified copy becomes the highest available version number of the file. Qualified references do not perform the initial copy operations. Instead, committed modifications appear in the original file. This implies that older versions need not necessarily be immutable. Clients can make this policy decision for themselves by using a special version trigger, as explained below.

Supporting versions at this level has several benefits. If a file is consistently referenced without an explicit version number, new versions are created with each modification, and the naming support insures that clients always get the highest-numbered version. If the old versions are not deleted, a complete modification history of the file exists. Clients are free to use the naming support for versions and can substitute their own version creation routines by explicitly referencing the new versions by number. For example, clients wishing to save disk space by implementing versions with forwards or backwards deltas are free to do so. Clients could use embedded reference blocks to reduce the number of copies of a block needed to be keep on disk. Support for versions of this form is best left to the client programs.

2.4.3. Version Triggers

In addition to a copy being made at new version creation time, a user-definable trigger, called the version trigger is executed. This trigger is intended to implement the old version cleanup policy of the client.

The version control trigger may be used by clients to implement an arbitrarily complex cleanup policy for old versions. For example, a client's trigger may delete versions older than a certain number of days or delete old versions until the space used by versions is less than a certain threshold. More complex triggers can implement compression routines to make old versions smaller. A client can implement immutable files with a version trigger that removes all write access to old versions upon creation of a new version. Triggers are discussed in a later section.

2.5. Summary

Semantics of the file system for the design of GAFFES include support for features such as embedded references, triggers, transactions, version histories for files, and transparent distributed access to files. Files are divided into user-defined blocks allowing support of embedded references. Two different views of the file system semantics are presented to the clients to support different levels of processing sophistication.

3. Naming and File Location

The GAFFES name service provides the functions that permit users and the applications they use to map string names of files to the objects which manage the files. Precisely what type of system service manages the files depends on whether the file is replicated. Unreplicated files are managed by disk servers. Replicated files are managed by replication servers. Each replicated file is managed by a particular replication server, but since there may be multiple copies of that replication server, the name service must map the file name to a list of replication server instances. Logically, the name service supports the function:

FindFileName(name) -> <disk server handle or list of replication server handles>.

A server handle has two parts, a network address to be used for remote procedure calls and a public key to allow the user to authenticate the server. Those names which can be successfully mapped to server handles form the GAFFES namespace.

The name service supports several other operations besides the name lookup function. Users may add and delete file names. There are also administrative functions for the addition and deletion of name servers. The name service also provides some support for the security system. As is described more fully in the security section of this report, the keys which are used to authenticate roles are stored in files. Thus, a portion of the namespace is taken up by the key files corresponding to each role. The

name servers also support the security system by having public key/private key pairs themselves.

The next five sections describe the GAFFES name service in more detail. The goals and objectives that motivated the design are presented in the second section. The third section describes the namespace, the structure of the name service, and the user level operations. Descriptive names, which supplement the primary namespace, are discussed in the next section. The operations which allow system administrators to adapt the name service to alterations in usage are outlined in the fifth section. The last section presents some conclusions.

3.1. Design Objectives

The design of the GAFFES name service was based on four goals: *performance*, *availability*, *scalability*, and a *consistent, low complexity user interface*. Since the meanings of performance, availability, and scalability are related, it is worthwhile to define these goals more carefully. In this paper, the goals of performance and availability refer to the behavior of a *static* system of millions of workstations. Scalability is the goal of maintaining the same level of performance and availability in the face of substantial changes in the size of the system. Our goal for the user interface was that the user only need to specify a logical name to be able to access a file and that once this name was assigned initially, it need not change despite system changes. These four goals gave rise to the following objectives which, in turn, guided the design of the name service. The first set of objectives is primarily concerned with insuring adequate performance, availability, and scalability.

Decentralized Servers

Decentralization of the name service is critical to insure performance, availability, and scalability. Centralized services are almost certain to become bottlenecks that degrade performance. Even if such bottlenecks can be prevented, new bottlenecks are likely to be created by changes in system scale. Centralized services also threaten availability because loss of the central service cripples the whole system instead of just a subset of the system. In GAFFES, this objective is primarily fulfilled through replication of the name servers.

Decentralized Knowledge

Knowledge of the namespace must also be decentralized. One way to have decentralized servers is to give many hosts in the system full copies of the name server database. While this will improve the performance of lookups, it places considerable overhead on the operations which update the namespace, since these operations must alter all copies of the database. Certain system faults, such as network partitions, could make the update operations unavailable. Scalability would also be poor, since the full name server database for a million workstations would be huge, requiring large storage facilities on each machine running a server instance. Thus, the name service's knowledge should be divided so that each machine supporting an instance of the name service need only keep track of a reasonable subset of the total knowledge.

Adaptability

Since GAFFES is not a static system, its name service must adapt to maintain performance and availability in the face of changes in demand. These changes may be either growth or shrinkage and may occur either globally or locally. The name service must adapt to these changes 'in place', because it would be intolerable to have to 'take the system down' for each reconfiguration.

The following objectives are primarily related to the goal of a consistent, low complexity user interface. Most of these objectives attempt to let the user ignore the details of file storage and name service configuration.

Uniform Naming

A single naming convention should prevail throughout the entire namespace. The structure of

the namespace should appear the same regardless of the user's position in the namespace. This distributed file system should appear to be one unified file system. This is similar to the file system view provided in Locus [Walker 83].

Location Transparency
It is important that services (e.g. disk servers or name servers) be allowed to come and go without requiring objects to be renamed. If object names are tied to the location of services handling them, objects will either be subject to frequent renaming or poor availability.

Replication Transparency
A user should not need to know whether a file is replicated in order to access it.

Scale Change Transparency
The name service should be allowed to grow or shrink without requiring renaming of files. This is similar to location transparency, but its primary goal is increased scalability as opposed to consistency of the user interface.

Easy generation of unambiguous names
It is not possible for a user of a large system to keep track of the names of all files in the system. It must be simple and inexpensive for a user to choose a useful, globally unique name for a new file.

Descriptive names
While unique file names are an essential part of the user interface, it should be possible to use other naming conventions. In particular, the user should be able to access a file by describing its qualities instead of its label (i.e. name). This descriptive naming feature allows the user to access a file on the basis of its content or contextual or user-specific knowledge.

3.2. Structure of Names and the Name Server Hierarchy

3.2.1. Hierarchical Names

The GAFFES namespace is hierarchical. Each successive component of a name further qualifies or "locates" the object in the namespace. For example,

/U.S.A./EDU/University-of-California/Berkeley/EECS/CS-division/Xavier/recipes/flying-pancakes

might be the name of the file holding a recipe of which the student named Xavier in UC Berkeley's CS department is particularly fond. The actual syntax used to separate components in an object's name is unimportant and we chose the syntax in the example simply because we are accustomed to it. What is important is that whatever syntax is chosen be used globally in the system. Users should not need to name objects differently because their physical or logical location in the file system has changed. This would violate our uniform naming objective.

The name of this recipe file is location-independent in that it is unnecessary to know what machine stores the recipe file, or what name servers would be used to resolve the name. While the file name does specify the organizations to which the file is attached, it does not specify a storage device. No component of an object's name needs to describe the object's actual location. Instead, the name is passed to successive name servers based upon substrings of the name until one of the name servers recognizes that it stores the location information for the part of the namespace that includes the named object.

We chose a hierarchical namespace because it satisfies our goal of providing an easy choice of unambiguous (globally unique) names. A hierarchical namespace has a tree structure, and because a

simple path from the root to any node in a tree is unique, it is easy to choose a unique name. Users need only check that the last component of the name they choose is unique at the level of the hierarchy in the tree represented by the prefix of the name. This is inexpensive to do and does not place a severe burden on a user provided he is allowed to add more levels of hierarchy to his branch of the namespace. For this reason, we have also decided to allow arbitrary depth in the namespace. Using hierarchical names with arbitrary depth gives organizations and users the freedom to create sub-namespaces without affecting other organizations and users.

3.2.2. Operations on Names

There are three operations on file names:

```
/*
 * This routine is called iteratively.
 * Given a name and a server, resolve the name
 * to the next list of name server handles, or to a file server
 * handle or list of replication server handles if the given server
 * recognizes the entire name.
 */
FileServer or ListOfReplicationServers or ListOfNameServers
FindFile(server, name)


/*
 * Add a file name to the file namespace.
 * Return success or failure.
 */
Status
AddFileName(server, name)


/*
 * Delete a file name from the file namespace.
 * Return success or failure.
 */
Status
DeleteFileName(server, name)
```

3.2.3. Name Resolution

3.2.3.1. The Logical View

Although names are hierarchical, name resolution need not be hierarchical in all cases. It will be hierarchical only in the most expensive and most general case. Figure 3.1 describes the most general case.

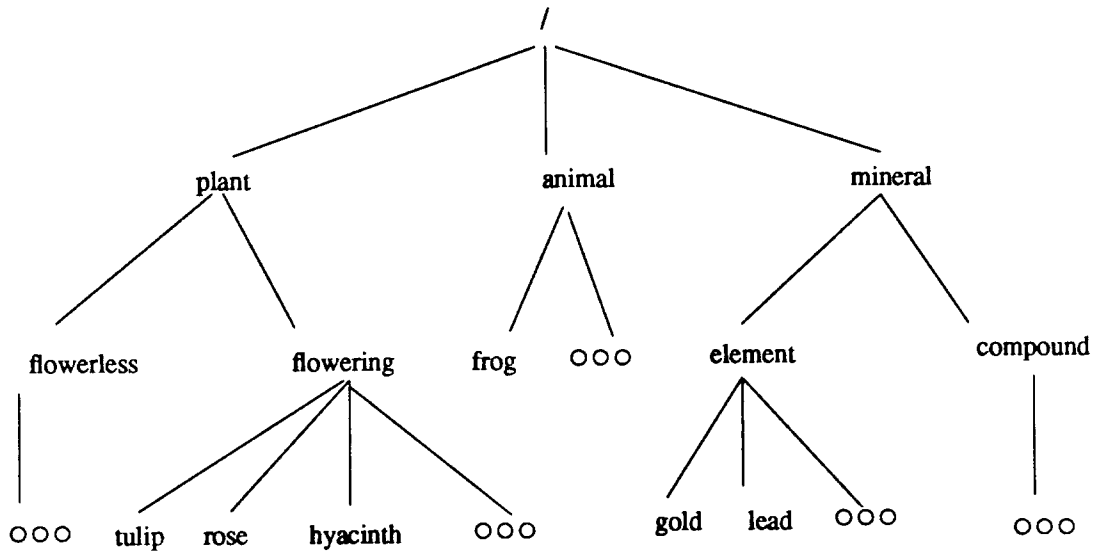


Figure 3.1 — Example of a Simple Name Server Hierarchy

A name server is handed the name */plants/flowering/hyacinth*, and, not recognizing the name, hands the name intact to its parent name server. If the parent name server still does not recognize the name, it, in turn, hands the name to its parent name server. Eventually the name makes its way up to a top-level name server that recognizes the first '/' component of the name. (Actually, it recognizes the empty name before the first '/' delimiter.) This server then knows the server (or how to find the server) that will recognize the */plants* portion of the name and hands the name resolution responsibility to that server. Finally, the name comes to a server that is responsible for the */plants/flowering* section of the namespace and recognizes the full name */plants/flowering/hyacinth*. This name server is able to find the disk server or replication server that stores the file.

An alternative method of name resolution could require that any name be passed to a top-level name server at which point the name resolution process could begin. For reasons of trust and performance, we feel this is too great a restriction. In a system such as Grapevine or Clearinghouse [Schroeder 83] [Oppen 83], where there are only a few levels to the namespace, it is not terribly wasteful to start name resolution from the root of the namespace tree. In a global file system, where there could be an arbitrary number of levels, this requirement could be very wasteful. If names are always passed to top-level servers to begin the chain of resolution, then name resolution will almost always require many steps since in a large system, most objects will be fairly deep in the namespace tree. If names are always passed to parent servers when unrecognized by the initial name server, then in the worst case any name may require a chain of resolution all the way up the tree to a top-level server and then down the tree until a final server claims the name. The resolution chain of the second scheme is potentially longer than that of the first scheme, but because we believe its worst case will not occur often, and because the first scheme will almost always require as many resolution steps are there are components in the name, we chose the second scheme.

We believe that the worst case of the second scheme will not often occur, because users will tend to refer to files localized in some subtree of the namespace. Only servers handling that subtree need participate in name resolution for these files. Often, one server will handle a subtree of a namespace, and if this is the same server with which the user initiates the name resolution process, only that server need participate in resolving the name. Users and organizations will for the most part

refer to names near them in the hierarchy, perhaps only up a level or so and then down another level or so. We have optimized our name resolution process for this case. In the case where a user wishes to refer to an object far away in the namespace tree, our name resolution scheme is more wasteful. Also (as described below), we allow servers to pass names not just to their parent and child servers, but to arbitrary other servers.

Of the designs mentioned in [Terry 85, p. 47], our design appears to be a simplified version falling between the first design requiring a "metacontext" and the second decentralized design. In Terry's metacontext design, the resolution of names not otherwise recognized by a server must eventually involve a a top-level server that stores a metacontext that tells it which of the other servers will be able to begin the name resolution process. In the decentralized design, the information about which server should next participate in the resolution of a name is spread out through the servers so that there is no need to have "top-level" servers where storing a metacontext. Our scheme has top-level servers through which the name resolution process may need to pass, but the information about how to reach those top-level servers is spread out through the servers by using a notion of parent and child servers. Servers pass unrecognized names to their parents and they pass further-resolved names to their children. Servers can also pass names to arbitrary other servers, as explained below, but this is not the default case.

Our design also lacks the full generality of contexts in Terry's designs. Name resolution in our design amounts to passing the full name of an object to each successive server that recognizes some increased portion (usually a prefix) of the object's name. We believe enough load balancing is available through splitting the responsibilities of the servers along these syntactic lines and that more general information (beyond the full name of the object) is not necessary.

Our design also contrasts with a more restricted design where each successive name server strips off the portion of the name that it has recognized and hands the shortened name to the next server. We chose this design for two reasons. In GAFFES, there is not a one-to-one mapping between servers and nodes in the namespace hierarchy. A name server thus needs to inspect some prefix of the full name to determine whether it or another server is responsible for the file. As described below, name servers are able to communicate with other servers far removed from them in the namespace and must be able to hand those other servers a full name. This flexibility allows name servers to make decisions based not just on syntax but also on trust and load considerations.

There are two patterns of server communication that we could follow, an iterative method and a recursive, nested method. In the iterative method, a client passes a name to a nearby name server. The nearby name server determines what other name servers could next be talked to in order to resolve the name and returns this list back to the client. Based upon trust, availability and performance reasons, the client makes a choice as to which server from this list to try next. This server in turn returns a list of the next servers to resolve the name further. After several iterations of this procedure, the client talks to a server that is able to return a disk server handle or a list of replication server handles.

In the recursive method, each successive name server makes the decision as to which other name server to pass the name to, without returning any information to the client. The initial server finally returns the resolution of the name to the client. ([Terry 85] mentions a third, transitive method. We rejected the transitive method because it does not deal as well with server failures and does not adapt as well to the use of remote procedure calls.) The iterative method allows a client to choose whichever name servers he most trusts at each step of the name resolution process. The recursive method leaves trust issues up to the name servers themselves. Depending upon trust considerations, the iterative method of name resolution can be used for all of the steps in the name resolution process until a server is reached that the user trusts. At this point, the resolution process can continue in the recursive fashion.

A variant of the recursive method, tentatively called the don't-care method, permits relatively high performance access to files which the user wishes to access in spite of his lack of trust. In the don't-care method, the name servers may pass the name request to any other server, regardless of whether that server is considered trustworthy or not.

We decided to use full path names rather than relative names because relative names mean different things in different parts of the name space. This would violate our goal of providing a uniform namespace globally unique names. Descriptive and generic names, as described later, give a location-sensitive or context-sensitive way of naming objects.

The name server hierarchy need not match the name space hierarchy exactly. Figure 3.2 shows several ways in which the name server hierarchy can differ from the syntactic hierarchy.

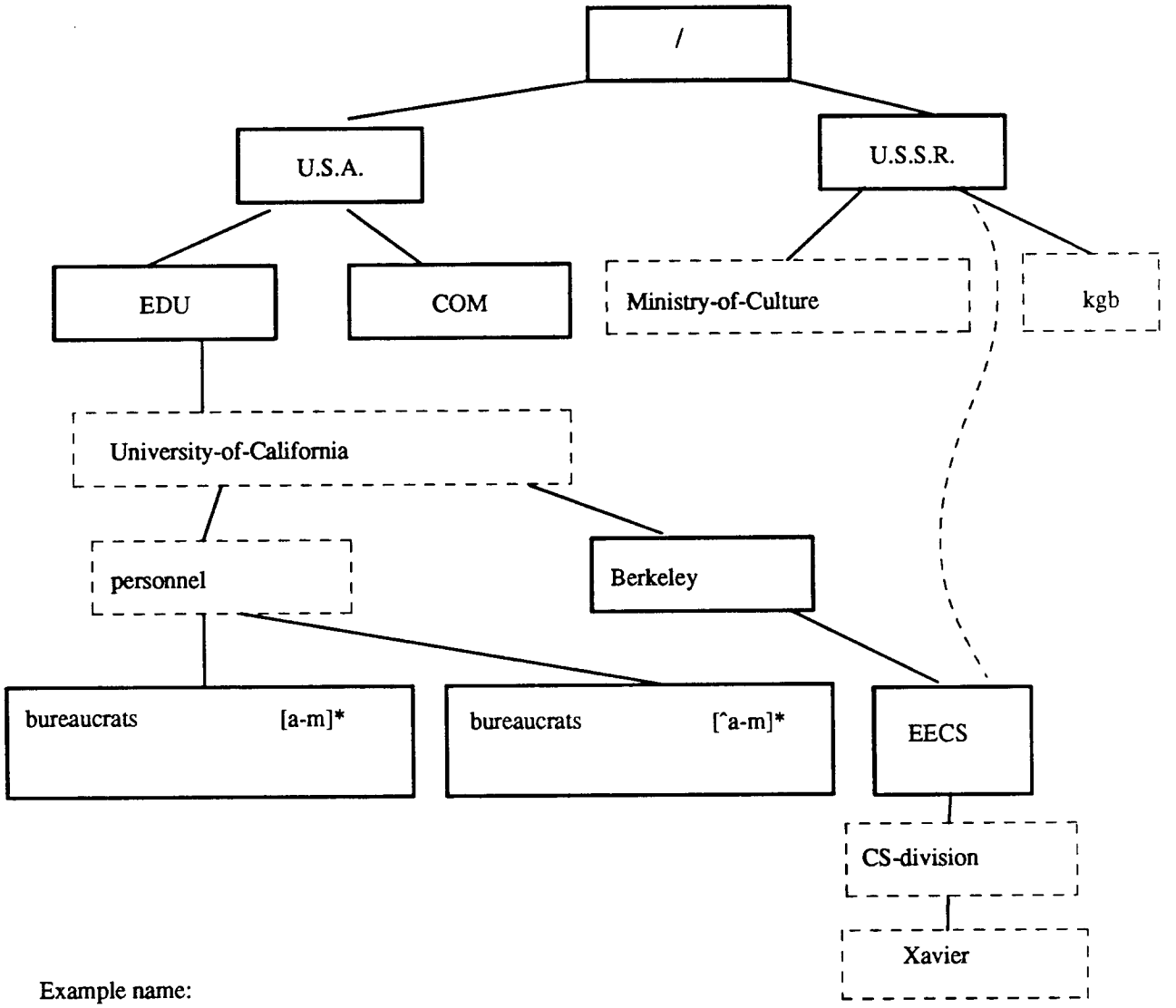


Figure 3.2 — Example Mapping Between Name Server Hierarchy and Syntactic Hierarchy

An individual user may wish to add arbitrary depth to some branch of the hierarchy, without the need to add new name servers, particularly if he does not anticipate much of a performance bottleneck. In this case, the student Xavier in the CS-division can add whatever depth of subtree he wishes for storing his own files and avoid naming conflicts if he gives the files names with the prefix

/U.S.A./EDU/University-of-California/Berkeley/EECS/CS-division/Xavier

For security reasons, an organization may want to have only one name server handling a subtree of the hierarchy. In figure 3.2, if the kgb owns the /U.S.S.R. name server, then it has control over all files with the prefix /U.S.S.R., even if the files are in the Ministry-of-Culture subtree.

If there is a part of the namespace that is heavily used and for which the necessary data to store is very large, several name servers can handle one node in the syntactic hierarchy. In figure 3.2, the amount of information to store about the "bureaucrats" node is too large for a single name server. Two name servers handle this syntactic node by splitting the name resolution tasks in some fashion. In this example, one name server recognizes names with the pattern *.../bureaucrats/[a-m]** (all names starting with an 'a' through 'm'), and the other name server recognizes names with the pattern *.../bureaucrats/[^a-m]** (all names starting with anything except an 'a' through 'm').

This last example is an important feature for systems that wish to grow and shrink gracefully. In Grapevine, when a registry became too large, splitting the registry into pieces forced users to change their names. In this design, name resolution databases can be split (or combined) without the need to make syntactic changes to the namespace [Terry 85].

We also wish to allow name servers to resolve names by passing a name directly to another name server that isn't a parent or child name server in the hierarchy. (A parent name server is the server directly above the particular server in the hierarchy, and a child server is one directly below.) For instance, the EECS department name server could resolve a name */USSR/kgb/recipes* directly by handing the name to the kgb name server without resolving the name all the way up the tree. If two organizations trust each other and wish to communicate frequently, it is a performance advantage for them to set up their name servers to recognize each other directly. This sort of a "hard link" is also to their advantage for security reasons, if they trust each other but do not trust arbitrary higher-level name servers.

3.2.3.2. The Mechanism

Name resolution would be easy if a name server could store the locations of every single object in the global file system. This is not a feasible solution in such a large system. The necessary amount of information is far too large for one server to handle. Even if the server could store all of the information, updates to the global file system would swamp the server. It is therefore necessary to divide up the location information among various servers. As soon as this is done, it is necessary to define how servers cooperate to resolve a name of which the servers that initially see the name have an incomplete understanding.

We have modeled our name resolution mechanism on a subset of the idea presented in [Terry 85]. The patterns of name resolution described above are handled by name servers storing information about which other name servers they talk to. We can think of each name server recognizing or rejecting a name based upon some arbitrary function.

Boolean
NameServerRecognize(name)

In our design, the function will make its decision based upon syntactic information. In the usual case, a name server given a name will recognize or reject a name based on some prefix. If the name server stores no information about its child and parent name servers, then if it accepts a prefix of a name, it

will need to forward the name on to all of its children and wait for a response indicating that a particular child has recognized the name. It is preferable that a name server store information about what its children name servers are likely to recognize. This saves time by avoiding the expense of handing a name off to all the children and waiting for a child to respond in the affirmative. A name server is "bound" to another name server if it knows how to address the other name server (a network address or port number) and also knows something about what kinds of names that other server is likely to recognize. A server is bound to its parent server and its child servers, and it can be bound to arbitrary other servers as in the case of the EECS server in Figure 3.2.

When a local name server is handed a name that resolves to an object far away in the name space, it may not recognize any prefix of the name, and may not know of any server that will. If this is the case, it hands the name to its parent, and its parent repeats the process until a top-level server receives the name. If the local name server is bound to a top-level server, it could pass the name directly to the top-level server, but this is an option and not a requirement in our system.

3.2.4. Naming Versions

Unlike file replicas, the versions of a file represent distinct objects. For the user to be able to distinguish between the versions, they must have different names. We have chosen to name versions by putting a ',#' after the name. For example,

/U.S.A./EDU/University-of-California/Berkeley/EECS/CS-division/Xavier/292j/final_paper,3

might be the the third version of Xavier's Computer Science 292J paper. The user is not required to specify the version number when requesting a file. If a user specifies a name without a version number on the end, and the file has versions, this name will be interpreted to mean the most recent version of the file (the version with the highest number). Also, while problems of file archiving and source code control are important, GAFFES was not intended to solve these problems. Therefore, more powerful version naming is not provided.

This is not the only reasonable convention for the naming of versions. It could be argued that the versions of a file should not have separate names or that versions other than the most recent should be named relative to the current version. We chose to give each version a separate name to maintain the one-to-one relationship between the file and its name. Relative naming of versions was rejected because it would have meant that a file's name would change as new versions were created. Users desiring a more sophisticated version system could use the existing version system as a set of primitives from which to construct more advanced features.

We considered naming versions by using the descriptive name (attributes) mechanism. We rejected this possibility because we decided that attributes should not be used to have special meaning to the system. Our reasons for this decision are described below, in the section on descriptive names.

3.3. Descriptive Names

3.3.1. Design Decisions

It is not always convenient or intuitive for a user to refer to a file by its hierarchical name. Often, a user will not care precisely which file he gets access to as long as it has certain content. For example, he may need to see some sample source code from a certain language to remind himself of a programming convention, but he may not care which particular file he looks at. Another possibility is that the user will want to access files on the basis of his current context. For instance, a programmer maintaining programs for several different machines might have a context corresponding to each machine. He could use descriptive names for the files and let the context 'choose' the files for the machine he was currently working on. This would be particularly useful if the different sets of code were not being revised at the same rate, since the unique, hierarchical name of a file contains version information. By using descriptive names the programmer could ignore the differences in version

numbers for the different machines. Descriptive names also permit a user to underspecify a file, either because the user doesn't know the exact description of the file, or because he wishes to inspect all files that match the description.

In GAFFES, a file is given a descriptive name by attaching "attributes" to the file. An attribute is simply an <name, value> pair, where both the attribute and value are strings. Originally, we thought that certain attributes would have special meaning to the file system. Examples of possible "special-meaning" attributes are file type, owner, and alias. The decision to implement files as untyped blocks made the file type attribute pointless. The ownership attribute also became pointless when ownership was defined in terms of access rights. The final special-meaning attribute we rejected was the alias attribute. We thought that one way to make it easier for a user to find certain files would be to attach alias attributes to those files. When a normal file access attempt failed, the name service would automatically search the user's context for a file whose alias matched the original request. This would have only be done for file descriptions that did not include name hierarchy information. We decided that enforcing this convention at the level of the name service was a bad idea, because it would at least double the time necessary to return from a file access failure and could be implemented at the level of a user's shell if so desired. Thus, the system does not attach special meaning to any attributes.

Descriptive names have several disadvantages. A descriptive name is not likely to be unique. For example, there are many different random number generators in the world. If one were to attach the attribute <function, random number generation> to such a library routine, it would not be certain that that particular file would be the one the name service would find when one later requested a random number generator. This lack of uniqueness is unavoidable unless users are required to assure that every file has a unique set of attributes at the time of creation. Such a restriction would violate the underlying goals of providing descriptive names in the first place, by forcing the file's owner to modify the name in ways that had nothing to do with the file's purpose.

There is also a very large set of possible attribute names that could be used. It is not reasonable to require every user to specify even a moderately large subset of these attributes for each new file, regardless of the expected lifetime of the file or the user's desire to name it descriptively. As a result of these two problems, GAFFES files are not required to have any attributes and no attempt is made to assure that a file's attributes are unique. In fact, this lack of uniqueness is considered a positive <name, value> pair with absolute precision. To balance these two concerns, it was decided to permit the use of UNIX-style regular expressions in the value field of an attribute during lookups. This allows the user some flexibility in attribute specification without imposing a high processing burden on the name service.

3.3.2. Operations on Descriptive Names

There are five operations which manipulate attributes:

```
/*
 * Add attributes to a file.
 * Return success or failure.
 */
Status
AddFileAttribute(filename, attName, attValue)


/*
 * Delete attributes from a file.
 * Return success or failure.
 */
Status
DeleteFileAttribute(filename, attName)
```

```
/*
 * List the attributes of a file.
 */
ListofAttrValuePairs
ListAttributes(filename)


/*
 * Find any file in the given context matching the given attributes.
 * Return empty string if no file matches.
 */
FileName
FindAnyFileByAttr (AttrValuePairs, context)


/*
 * Find all the files in the given context that match the given attributes.
 * Return empty list if no files match.
 */
FileNameList
FindAllFilesByAttr (AttrValuePairs, context)
```

The first three operations are simply the standard add, delete, and list actions that would be expected. However, since descriptive names can map to more than one file, it is necessary to provide two lookup functions. The first, FindAnyFileByAttr, returns a handle for the first file matching a list of <attribute, value> pairs. FindAllFilesByAttr returns a list containing handles for every file which matches its argument. Both lookup functions take as an additional argument, a context. A context is actually a set of prefixes for sections of the name space that are to be searched for the file, similar to the PATH environment variable in UNIX. Thus, a user can make his own decision about how far to search, picking his own point on the performance/completeness continuum.

3.4. Operations on Name Servers

There are several sorts of changes that can occur in the server hierarchy. These changes are generally made in response to performance problems as a result of growth or organizational changes. A server may be added in two ways. A new server may take over a level of the syntactic hierarchy previously handled by a node that will become its parent. We call this "adding a child server." A server may instead be added to split the handling of a syntactic node of the naming hierarchy with a previously existing server. In this case, the new server will have the same parent as the server with which it splits the syntactic node. We call this "adding a sibling server." Both child and sibling servers may be added. Deletions of servers are similar. One complication to this description occurs when a child server is added (or deleted) and it itself has a child.

In general, the addition operations are carried out by copying the relevant portions of the naming database to new 'ghost' servers, informing parents and children of the existence of the new servers, and then activating the new servers. Deletion operations are similar. The database information stored on the servers to be removed is copied to the servers that will handle the information after the deletion. The parents and children are informed of the new servers they must talk to, and the old servers are deleted.

We originally thought that informing parent and child servers of additions and deletions of servers required a transaction. We have decided instead that the performance loss resulting from requiring complete consistency is too severe. Eventual consistency should be adequate. When the server structure changes, available parent and child servers are informed of the new servers with which they must communicate. The operation is not performed unless there is at least one parent and one child server available. Parent and child server connections are treated as hints. When previously

unavailable servers rejoin the system and find that servers they communicated with are non-existent, or that a server no longer recognizes a set of names, they ask their replicas for the new server information. In addition, we propose a method similar to that of Grapevine in which replicas make periodic consistency checks with each other. These consistency checks should not be too expensive as replicas will ordinarily be close to each other. The replications themselves will be supported by the replication facilities described in a separate paper.

3.5. Summary

We have been able to meet a number of important goals for naming and file location in GAFFES. We provide uniform, globally unique names that are independent of file location and services that are tolerant of hardware and software failure. The system adapts to growth by allowing changes to the server structure without requiring name changes. Through the use of context and descriptive names we provide the user with powerful name lookup facilities.

4. Security and Protection

The GAFFES design should support on the order of a million users who could potentially access any file in the system, yet the system would not be administered by a single authority. Instead, many mutually suspicious organizations would use the facilities of GAFFES and cooperate in its maintenance. Each organization participating in GAFFES would support GAFFES system software necessary to identify and store local files. It would also need to authenticate users from its own organization.

Although allowing cooperation between organizations is one of the primary features of GAFFES, users in each organization will need to be assured that involvement in GAFFES does not expose them to the following security threats [Voydock 83]:

- unauthorized release of information,
- unauthorized modification of data,
- denial of service.

In order to protect against the first two threats, GAFFES must provide a means of authenticating users and establishing secure communication channels. We do not consider the issue of denial of service.

We have divided the proposal for security in GAFFES into five sections. The first gives some background. It lists several assumptions, defines some terms used throughout the paper, and discusses previous work on network security. The second section enumerates the set of security guarantees GAFFES provides its users. These guarantees are GAFFES' security *policies*. A short third section lists the difficulties involved in implementing these policies. The fourth section of the paper describes the *mechanisms* in GAFFES designed to enforce its security policies in the face of these difficulties. A final section gives some conclusions.

4.1. Background

4.1.1. Assumptions

- It is possible to generate public/private key pairs on demand.
- Our protection mechanisms are only as secure as public key cryptography.
- DES or other private key encryption hardware is available.
- The network subsystem is susceptible to packet smashing, wire tapping, and replay.
- All objects in the global file system (files, name servers, replication servers, etc) have unique network addresses. The network address can be interpreted by the communication subsystem to allow delivery of messages to the object. Most network addresses will last only for the lifetime of the object manager's process.
- Where necessary, security provisions take precedence over performance concerns.

4.1.2. Definitions

4.1.2.1. Users and Roles

Users are humans who use the file system and the processes that exist on their behalf. A *role* [Birrell 86] is an abstract entity which has permission to do certain operations on a file. Associating rights with roles instead of users allows file access rights to be granted to logical users instead of GAFFES system accounts. For example, one user may use roles /NewsReader, /SystemAdministratorUCBArpa, /Berkeley/user/Terry, and /GAFFES_Programmer. Some of these roles may be shared with other users. A role may be shared with a partially-trusted server (such as a printer) without potentially compromising the other rights possessed by the user. Damage done due to a security breach is limited to the subset of a user's rights owned by the compromised role.

Users acquire (or "take on") roles in order to exercise the rights associated with that role. Just as access to files requires rights, acquiring roles requires rights; users must use one role to take on another. At login, a user is given the permanent *user-role* unique to that user. Since presenting a local kernel with a password is a necessary component of beginning a login session, a password is the ultimate key to a user's rights.

Roles and their implementation are discussed in greater detail in a later section.

4.1.2.2. Owner

An *access right* is the permission to do a particular operation on a particular file. Access rights to an object are granted by the *owner* of the object. By definition, the owner is the role (or roles) given the right to do the *ChangeAccessRights* operation on the object. Similarly, roles have owners who grant the right to acquire the role. The creator of a file or role is its first owner.

4.1.2.3. Trust

In GAFFES, all services have owners. The owner is a role representing a user (or organization) who guarantees the semantics of the operations defined on the service. A *trust* is the assumption that a potential user of the service makes about the validity of the owner's guarantee. Since operations are invoked by roles, each role must have established for it a set of trusts acceptable for the role.

A *belief system* of a role is a set of statements. With respect to this role, every statement in the belief system is a true assertion. GAFFES expects roles to create their belief systems using external mechanisms. Further, a role can place an arbitrary set of statements in the belief system. GAFFES will guarantee only that actions carried out on behalf of a role are consistent with the beliefs expressed in its belief system.

In GAFFES belief systems, we place statements of trust. Individual trusts are represented in a belief system as triples containing:

- The owner of a service,
- An operation on this service,
- A set of constraints on the parameters and result values acceptable for this operation.

An operation is invoked on a server owned by the listed owner only if the parameters satisfy the given constraints. A role believes in the results only if the operation is invoked in a manner consistent with its belief system, and the results also satisfy the stated constraints. GAFFES clients (and services) use belief systems when choosing a service on which to invoke an operation.

For an example of this notation, consider the situation where a role trusts the Berkeley name service to resolve names of objects at Berkeley correctly. This trust is represented in the role's belief system as:

(/EDU/Berkeley/Name_Service/admin, resolve_name, names beginning /EDU/Berkeley)

Note that this formalization of trust allows us to express such subtleties as the fact that a name service may be trusted to resolve names in one part of the name space and no others, or the fact that a name service may not be trusted for some operations. The formulation and use of belief systems is discussed more fully in a later section.

4.1.3. Previous Work

An early paper in this field, [Needham 78], introduced the concept of authentication servers and presented several schemes for authenticating principals using public and private key encryption techniques. The algorithms described, however, assume that authentication servers are trusted globally. Since the GAFFES system cannot include universally-trusted components, the algorithms proposed are unsuitable for our design.

[Birrell 86] and [Lampson 86] describe work being done at DEC SRC for trusted authentication in systems without global trust. Our notion of roles arises from their work. These papers are also our motivation in using a hierarchical role name space, with non-hierarchical authentication paths.

The work at DEC SRC is based on a hierarchy of secure channels between authentication servers. There are pairwise secret keys between principals at either end of a secure channel. Secure communication is established by composing these channels to form a trusted path. Our scheme differs in that it uses public keys for authentication. Further, our formalization of trust is different from the model they describe. We have introduced the concept of a belief system, which the DEC SRC model lacks.

At UC Berkeley, the DASH project [Anderson 87] is currently investigating issues in the design of distributed systems with non-global trusts. A formal statement of trust and theorems using this formalism are described in [Rangan 87b]. However, their notions of trust differ from ours in several respects, particularly from our idea of trusts as linked to guarantees.

Using cached public keys as hints is based on the material in [Terry 87]. Maintaining loose consistency among public key caches uses an idea of a "time to live" as described in [Terry 84].

Traditional network security problems are discussed in [Denning 79]. Some solutions are proposed in [Taylor 85] as well as [Mills 87]. The ITC distributed file system [Satyanarayanan 85] uses access list protection and a form of group access rights. Their *groups* are similar to our roles.

4.2. Policies

GAFFES provides the user a set of security/protection guarantees about the file system. These guarantees will be stated in the form of *policies* regarding security/protection issues. The policies are not meant to imply the use of any particular mechanisms in their implementation. Clearly separating policies from mechanisms gives us a vehicle for discussing the function of GAFFES security separately from the mechanisms available to implement it. This separation also gives us a means of evaluating alternative mechanisms.

- Every file has at least one owning role. An owning role is a role that has access to the operation that changes access rights to the file. Clearly, a role with this right may use it to acquire any other rights to the file as well.
- Roles also have owners. The owner of a role determines the roles that may use that role to do operations on files. GAFFES does *not* guarantee that these roles will not surreptitiously or accidentally distribute access to the role once it is granted them.
- Our central protection guarantee for GAFFES is that a file will not be accessed other than in a manner authorized by the owner of the file.
- The multiple domain structure of GAFFES should not result in the violation of the above policies. In particular, the presence of untrusted domains in the system should not affect a file's security.

4.3. Problems

Given these policies and the environment described above, the implementors of GAFFES must deal with these problems :

(1) How is unauthorized access to a file prevented ?

(2) How is a role implemented ? What is the user interface to a role ?

(3) How should secure communication links be established ?

(4) How can caching of keys improve performance without making security violations harder to prevent/detect?

(5) What rights do the operations invoked by triggers have?

(6) How are roles authenticated between domains?

(7) How can users be sure the names of roles are resolved to the correct public and private keys?

Answers to these and other interesting questions are described in the section that follows.

4.4. Mechanisms

4.4.1. Overview of the Implementation of GAFFES

The security mechanisms described here depend on four components of the global active file system:

The hierarchical name service
The storage service
The key-caching service

Note that GAFFES does not use key distribution servers or authentication servers. The actions performed by these traditional security services have been subsumed in our design by the components listed above.

The security-related functions performed by these services are

1) Role creation
2) Role authentication
3) Public key revocation
4) Use of belief systems for trusted operations
5) Public key caching

4.4.1.1. Name Service

As far as the present discussion is concerned, "the" name service must be thought of as a collection of name services. Each organization supports a name service capable of resolving *some, but not all* of the names of objects in the global file system. From any given location, the resolution of some names will require the cooperation of several name services. The details of the individual name services are described elsewhere. Here we view a name service as a black box that supports two operations: *recursive* name resolution and *iterative* name resolution. The first, when invoked, returns the network address and public key bound to a name. The second, returns the owner, public key, and address of a *name server* that can resolve the name. The reason for this distinction is security-related and explained in a section below.

An important point about name servers is that they communicate with each other using *secure channels*. Thus communications between components of the name service are secure to the extent that all messages are encrypted using a one time conversation key.

4.4.1.2. File Manager

A file manager is the server that actually controls the files; for unreplicated files it is a disk server, and for replicated files it is a replication server. We require the file manager to authenticate roles and check permissions for the files it maintains. Details of this are described in a later section.

File managers have public/private key pairs and network addresses. Public key and network address are assumed to be registered with the name service by human intervention or some other

measure outside of normal GAFFES operation. Changing public key may be accomplished with a name server operation so long as the old key has not been compromised. The private key of a file manager is stored in some secure manner on the disk it is managing.

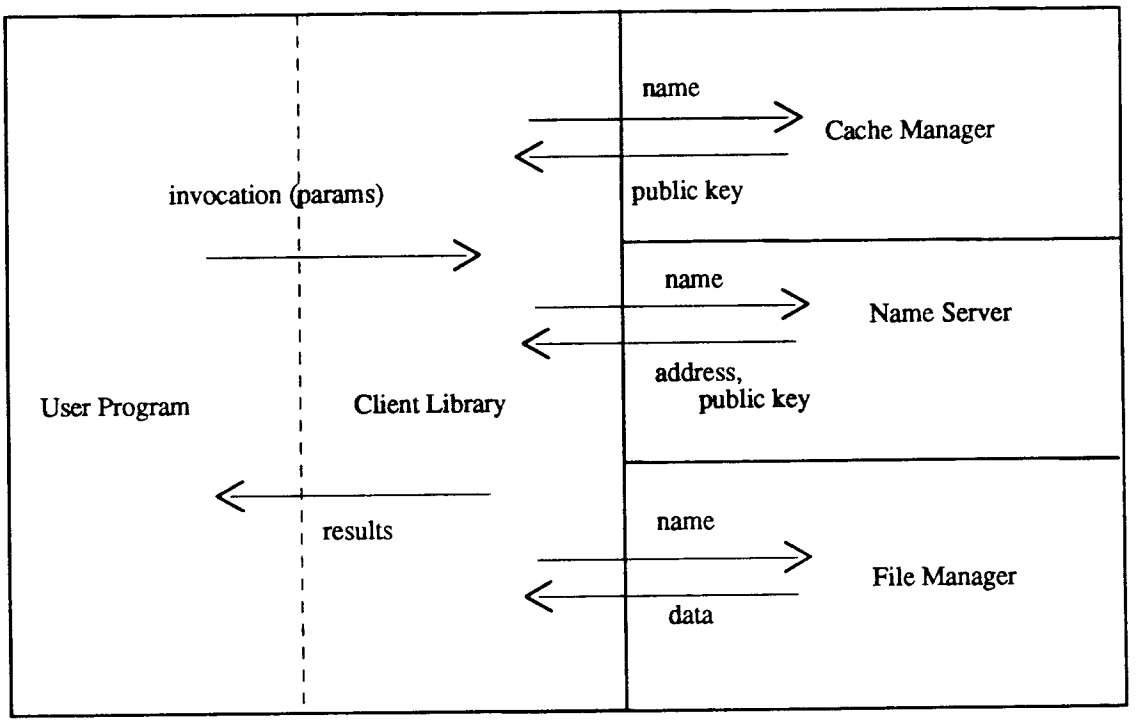
4.4.1.3. Cache Manager

The cache manager maintains a cache of public keys that have been resolved at that site. Resolving public keys is time consuming. Hence, in the interests of performance and availability, it is useful to maintain a cache of public keys on a per site basis. We describe the nature of our cache in a section below.

In addition to the cache of public keys that is maintained per site, we also expect public and private keys to be cached in main memory on a per role basis. Routines in the client library are responsible for maintaining this cache. By maintaining a hierarchy of caches, we expect to improve performance.

4.4.1.4. Client Library

We do not expect a user to directly invoke the above services. Instead, GAFFES will provide a library of routines as an interface to GAFFES (not unlike the UNIX standard I/O library). The library routines can be thought of as existing at a level above the cache manager, the name service, and the file manager. The client library manages interactions between these entities as shown in the following figure.



The library must be able to tell which roles should be used to access which files. To do so, it maintains mappings from file name to name of role used to access file in a definition file (similar to .cshrc in UNIX). The library routines read the definition file and automatically acquire the correct role to access a file. Since groups of files accessed by the same role will probably be near each other in the name space, the definition file can actually contain *prefixes* to file names instead of complete file names. Using prefix mappings will prevent the definition file from growing too large. Most files in the global file system will be accessible only if unprotected so a mapping from "/" to user-role should suffice as a default.

4.4.2. Implementation of Roles

Public key cryptography serves as the basis for security in GAFFES. A public/private key pair is associated with each role. Although users and file managers will cache the private and public keys they use, each role has a file suite associated with it which stores the authoritative key pair for that role.

We prefer public key to secret key cryptography because public keys are more readily cached and replicated. There are no penalties for accidentally divulging a public key, in contrast to the severe penalty on loss of a secret key. In addition, a role's private key authenticates it to anyone who has access to a reliable public key. This property is useful in a global system. Finally, in order to acquire a secret key, everyone along the path transferring the secret key has to be trusted not to misuse it. This is far more trust than in a public key scheme, where a propagating node has to be trusted only to the extent that the public key propagated has not been changed. An intermediate node cannot misuse the public key in itself [Rangan 87a].

The file suite for each role consists of two files-- one containing the role's public key and one containing its private key. The read operation on the public key file is not protected, However, only users (in roles) allowed to take on this role may read the private key file. Only the owner of the role may write the key files.

Roles are created by a user program "CreateRole (owner rolename)" that generates a key pair, establishes both files, and checks that the name given the role is unique. By convention, private key files have the extension ".prv"; public key files are identical to the role name. Private key files also store the belief system of the role. Anyone can create a role. Since new roles presumably haven't been given any rights, there is no reason to protect this operation. Roles may be deleted by their owners.

Changing the keys stored by a role is done by a similar program. Only an owner of the key file may change the keys. This program uses a transaction mechanism to make sure that all versions of the role files are consistent.

Implementing key storage as a file suite allows authentication servers to be replicated through GAFFES replication service, named through the GAFFES name service, and protected by the GAFFES file protection mechanism. An important implication of this design is that role names are part of the same name space as file names and are guaranteed unique by the restrictions of the GAFFES naming hierarchy. This implication is discussed in detail in a later section on trust and trusted authentication.

4.4.3. Standard Network Security Problems

Many network security problems such as wire tapping, packet replay and packet smashing have been extensively studied in [Taylor 85]. Solutions to these problems are well documented. Essentially, they involve encryption of data put on exposed wires and placing sequence numbers or time stamps on messages in order to provide a secure *window* (for example the Sun RPC protocol). We refer the reader to [Taylor 85], [Denning 79], and [Needham 78] for further discussion regarding these problems and their resolution.

The algorithms normally used for security in distributed systems assume that there is a way that two communicating principals can get hold of a shared secret key. We will call this the *conversation key*. If both role and file manager have well-known public keys, a conversation key is easy to arrange. The requester authorizes the original request for file access with a digital signature generated by its private key and seals it in the public key of the file manager. The reply contains a randomly-generated conversation key, encrypted by the public key of the requesting role, and authenticated by the file manager's digital signature. Continued communications between the file manager and the requester can use this conversation key. Therefore, once the two communicating principals know each other's public keys, a conversation key and solutions to the classical problems immediately follow.

Unfortunately, in a distributed system without global trust, obtaining the correct public keys for a role or a file is not necessarily easy. Consider the following example - suppose file manager for file A, (henceforth called A), wants to authenticate request from a user claiming to have taken on role B.

The requester's digital signature should be ((operation) private_b)public_a); A needs to know the public key of B in order to decipher the signature. A must read the public key of B from the authentication server administering B.

While resolving the role name, A could accidentally do an operation on a malicious server (authentication or name). That malicious server could substitute (a path to) the public key of one of its evil henchlings instead of the real key of B. Assuming that B had some access privileges to file A, these can now be usurped by the impostor. This would enable file A to be accessed by an unauthorized role, which voids our guarantee.

The problem here is that public keys are only as secure as the method used to obtain them. In a system like GAFFES, users may only know a few servers that can be relied upon to find authentic public keys. By insuring that the process of acquiring public keys does not involve servers outside of the user's *trust domain*, GAFFES can guarantee those users that keys are authentic as long as the user's trusts are warranted. Informally, a trust domain is a set of name servers whose operations a role trusts. We will discuss trusted authentication more completely below.

GAFFES expects the communication lines to be insecure and susceptible to wire tapping, packet smashing etc. Hence all communications between principals must use a conversation key to encrypt messages. Each principal must arrange a conversation key for each other principal that it wishes to communicate with. This key is easy to establish if the public key of the other

4.4.4. Implementation of The Client Library

Users interact with the file system through a client library similar to the standard I/O library in UNIX. The client library includes RPC stub routines to package the user's request into the messages that the file manager actually receives. Those messages must contain digital signatures encrypted with the private key of a role and the public key of the file manager. Library routines go through the protocols necessary to obtain public keys for file managers and to assume roles for users. Several of the more important library routines are discussed next.

One of the major routines present in the client library is *FindPublicKey*. Given a role name, this routine returns the public key for a role in a manner consistent with the belief system of the invoking role (henceforth called "trusted manner").

The FindPublicKey routine first contacts the local cache in order to check if the public key is available locally. If so, the cache returns the key, along with the time it was acquired and the path(s) used to acquire it. The routine then reads in the belief system of the role on whose behalf it is doing the role name resolution. It uses the belief system to decide if the cached value is acceptable. If it is, the routine is done. Otherwise, the name server is asked to determine (in a trusted manner) the address and public key of the manager of a public key file. Sending a read request to the file manager of the public key file allows the client library routine to obtain the desired public key. The cache is subsequently updated to reflect a new path for the public key, or at least a new time of acquisition.

We next describe the file open request. This is similar in spirit to other file operations (read, write, close, seek etc.) and in the interest of brevity, we will discuss only this operation.

To open a file, the name of the file is first presented to the name service. This returns the location (network address) and public key of the file manager responsible for this file. To open the file the routine must send the file manager a request, authenticating it with the digital signature of a role that is used to access this file. The role to be used is found from the lookup table. The private key of the role is now obtained by doing a read from the private key file of that role. (A read request is sent to the file manager for the private key file, authenticating the request with the private key of the current role). Finally, the open request is encrypted using the private key, and this is sent off to the file manager of the file. The file manager authenticates the request and verifies that the role has correct access rights. If this is satisfied, it opens the file.

4.4.5. Access Lists

The file has a control block area in which to store access lists. As mentioned previously, file owners create and modify these access lists with the operation *ChangeAccessRights(filename, mode (= add, delete), rolename)*. Any access list can contain the role "unprotected" which turns off the access control mechanism for an operation. We chose to implement access control with access lists instead of capabilities for several reasons:

(1) Revocation of widely distributed capabilities would be difficult and extremely expensive.

(2) Access list protection is much more convenient for users than capability-based protection. Users will usually have fewer roles than capabilities so they are easier to keep track of.

(3) One major disadvantage of access list protection schemes in large systems is that the lists themselves may become large. Because of the existence of roles to specify group rights, we think that this can be controlled. Instead of giving access to particular individuals, owners of widely-accessed files will find it convenient specify groups of users in access lists. The group role owner, then, helps manage access to the file. If necessary, additional roles could even be created expressly for the purpose of shortening the access lists of popular files. There is obviously a tradeoff here between the number of roles a user must maintain and the size of access lists.

(4) The CPU overhead in checking an access list is not substantial; the check is only a pattern match. As CPU speeds increase, we feel that CPU overhead will not be a significant factor in using access lists.

(5) Finally, we can also take advantage of *classifying* roles into sets such as group, within_organization, outsider etc. as in UNIX. Thus, checking the access list will just be a matter of finding in which class a role lies (this could be syntactic), and a lookup in a small access list that contained permissions for a class. UNIX gets by with just nine bits. We do not expect to need much more

4.4.6. Caching Public Keys in Cache Managers

Resolving the names of roles to public keys is a complicated operation. We would like to save public keys in a cache once they have been acquired in order to avoid the expense of reacquiring them on next use. Keys are small and most applications won't require secure communication between millions of nodes, so we do not consider the size of the cache to be a limiting factor. Maintaining consistency, however, is still a problem since a remote role may change its public key arbitrarily. In addition, there is the problem that if the cache has an old key, and the key is compromised, during the time that the cache retains the old key, it cannot guarantee authentication.

4.4.6.1. Structure of Cache Manager

Probably, each site (machine) will maintain its own cache manager. Physically, the cache will be a series of entries of the form:

Rolename,
public key,
time of acquisition,
{trusted path used to acquire key},
{roles currently accepting the key},

where the braces denote a set of items. The time of acquisition field is for calculating whether the cache entry is valid according to a particular role's belief system. A trust concerning CacheLookup operations may specify a maximum acceptable age for cached keys in the "parameter limitations" field of its entry in the belief system.

Each public key in the cache is acquired through a sequence of iterative and recursive resolve-name calls to the name server, depending upon the belief system of the role that resolved the name to the public key. In order for some other role to rely on the key, it is necessary that the trusts used to

obtain the public key be consistent with the belief system of the other role. Thus, when the cache returns the public key for a role, it should also return the name servers that have been trusted to obtain the key. We call this the 'trusted path'. If the cache has the public key for a role and the path used to obtain it is not a trusted path according to the role that requires the key, then the name has to resolved once again according to that role's belief system. If the result obtained is the same as the public key already in the cache, the new path, as well as the role name is added to the set maintained in the cache. The idea is that if

a role whose trusted path lies in one of the paths maintained in the cache

or

a role for whom the public key has been obtained already using its trusted path

then the cache entry can be used directly. Since we expect public keys to be long lived, we hope that the overheads that we incur in maintaining all this elaborate context will help in quick resolution of public keys in the average case.

4.4.6.2. Revocation and the Consistency of Caches

Assume for the moment that revocation is not a problem. We solve the consistency problem by treating the cache as a cache of hints. Verifying the hint is easy, because if a file manager is unable to decipher a digital signature, chances are that the hint is wrong (though it may be someone masquerading). To regain the correct public key, it has to repeat the role name resolution process.

Let us now consider revocation. Suppose a role discovers to its horror that its private key has been compromised, and so has to change the public key/private key pair. A server that has cached the old private key might continue to use that key indefinitely. Security might remain compromised at these servers despite the change of key.

To handle this problem, we use the idea of a time to live for a cached public key. A cached key that has lived past this time is automatically invalidated. The idea is that keys used by resources needing high security should be reacquired every time a message is received. For file managers with lesser security requirements the time to live can be longer. In some cases, the invalidation of the cache entry will take place due to the receipt of an undecipherable signature rather than due to the time to live.

Critical applications may need to send out explicit cache invalidations. Fortunately, a normal file operation whose signature is generated by the new key will invalidate a cache's old key. If, in the course of authenticating an operation, a file manager discovers that a cached public key does not match the requester's signature, the file manager will check the role's public key file to make sure the key has not changed. A role administrator (or any user) who wants to invalidate the old key may store a list containing file managers of interest who might have cached the old key. Sending dummy file requests to these file managers will cause cache invalidations; thus, explicit revocation can be handled using exactly the same mechanism as is used for updating a wrong hint.

4.4.7. The Proxy Problem

When a role requests a file operation, the operation may set off a *trigger*. The trigger may need to carry out other file operations and will need access rights for those operations. Since these operations are carried out on behalf of the role who initiated the trigger in the first place, the trigger should able to assume the rights of this role. The *proxy problem* is to design a mechanism that allows this to happen.

Three entities interact in the proxy problem - the *right holder* , the *right wielder* , and the *right verifier*. The right holder has the right to some file operation and is authenticated by the right verifier. In our case, the right verifier is the file manager of the active file. The proxy problem arises because we wish to allow a right wielder, who is different from the right holder, to be authenticated at the right verifier. Having established this terminology, we are now in a position to discuss some solutions.

The trivial way of solving the proxy problem is for the wielder to acquire the private key of the holder role. However, a security conscious invoker could not do file operations on files whose owner

may introduce a malicious trigger. Giving the wielder the holder's private key would allow the wielder to misuse the holder's rights at other verifiers. (This is analogous to giving away one's credit card number to a trusted server). We feel that this solution forces holders to put an unreasonable amount of trust in the wielder. Further, revoking such a trust is potentially troublesome since it involves a key change for the holder.

Another possibility is for the user to supply a *token* when invoking an operation that has triggers. A token is a binding of a particular holder and wielder to a verifier. In order to authenticate a wielder, we will need to associate a role with it. In our case, the trigger of an active file is the wielder, and hence will need to be associated with a role. This recognizes the fact that a trigger can carry out certain operations, and hence has certain rights to these operations. A role encapsulates the rights owned by the trigger. It also allows the trigger to be authenticated.

Tokens can be generated by using the private key of the holder to encrypt the name of the wielder together with the name of the verifier. Since the verifier and wielder are bound to the token, the token cannot neither be transferred to another wielder, nor used at some other verifier. Adding a time to live field to the token, would limit rights given the wielder to a time interval. This time to live field can be replaced by a count of the number of times the wielder may exercise its rights (an invocation count), if the verifier is prepared to maintain additional state.

Given a token, the verifier must decrypt the token with the public key of the holder. Presumably, it will be told who this is by the wielder. The verifier then authenticates the wielder by the digital signature of the wielder in the operation request. Having checked the credentials of the holder and wielder, it verifies that the time to live (or invocation count) is valid. Once these tests are passed, the verifier can correctly carry out the desired operation.

The major problem with this method is that it requires that the holder be aware of every verifier that the wielder will use. This means that the user of a file will need to know what subsequent operations are invoked by every trigger on the file. This prohibits any transparency of triggers and limits modularity. A modification to a trigger (by adding calls to other verifiers, say) may involve changes throughout the system. This is unacceptable.

One can regain transparency by not binding the verifier in the token. This means that the holder can use the token for any verifier where the holder has access. This may be tolerable if the time to live can be fine tuned to permit a limited operational capability.

If the token lacks even the wielder binding, then we need not associate triggers with roles. However, this means that the token can be passed around to other wielders. Again, the time to live may provide an optimistic access control.

Another solution is for holders to create a very restricted role whose private key is passed to a wielder as a parameter, along with a time to live. (This can be done by encrypting the current time plus the time to live with the private key of the restricted role). The wielder may misuse the private key, but since the rights associated with the role are so limited, the extent of possible harm is contained.

Finally, the solution we prefer is to embed a role with a trigger. All files to which the trigger should be granted access are asked to add the trigger's role to their access list. For example, let us assume that a print spooler is activated if the spool file is written on. That is, the write operation of the spool file triggers off the spooler. If the spooler need access to some files in the writer's directory, then the files that will need to be accessed add the trigger's role to their access list. This solution is clean in that trigger rights are well defined, and can be easily revoked. The private key associated with the trigger has to be protected. This may be stored in an area accessible only to the file manager and supplied to the trigger on demand. The invoker of the trigger has to trust that the rights granted to the trigger are not misused.

Access to embedded references causes a similar problem. In most cases, the right holder must generate the request to open, read, or write the embedded reference file with his or her own permissions -- not those of the file manager (who is the wielder in this case). Fortunately, these additional operations may be handled in the client library. If a read or write operation comes upon an embedded

reference in a file, the file manager returns to the client library. The return parameters of the operation direct the library routine to continue the operation invocation on the embedded reference instead. This process can continue recursively through levels of embedded references. Embedded references will cause some performance degradation since the extra file operations will cause extra RPCs. These RPCs, however, will be hidden in the library, so simple clients will not notice the presence of embedded references.

Note that the methods used to resolve embedded references will not help simplify security in trigger-initiated operations. The file containing the embedded reference is not a right wielder. The file manager does not need to invoke an operation on the referenced file or see the results of the operation. On the other hand, if a trigger initiated a read operation, the data would be delivered to the trigger directly. Similarly, on a write, it would have to be trusted to supply the data.

4.4.8. Trust and Trusted Authentication

4.4.8.1. Implications of a Multiple Organization Distributed System

In a system administered by a single organization, a service can be considered to provide identical semantics everywhere the service is offered. Since GAFFES is administered cooperatively by organizations with different needs and different standards, the "same" service offered by two different organizations may have slightly different characteristics. The general interfaces will be the same throughout the system, e.g. name services everywhere will support the same set of operations; the operations will do roughly the same things. However, the quality of the instances of a service supported by different organizations may vary.

For example, services may vary in:

(1) speed. The hardware supporting one instance of a service may be faster than another. The speed of a communication link from a particular client to some service instances will vary.

(2) compatibility. Different organizations may have different versions of the service software.

(3) reliability. The service might have different implementations at different sites. This implies some sites could be untrustworthy because of bugs in their software. Others might simply cut corners for the sake of speed or expense.

(4) security. On-site security mechanisms might be less strict at some organizations than at others. Some systems might be easier to compromise than others.

(5) trusts. The machines supporting a service may be physically secure, but the service still depend on data from the file manager of a neighbor some clients consider insecure.

(6) costs. It is conceivable that services charge their clients. Some (faster or more secure) would be more expensive than others.

In short, a uniform service interface allows access to a spectrum of service instances that differ somewhat in the reliability and expense of the service provided. We view this as desirable. Different applications will require (and be willing to pay for) different degrees of reliability from the service. Since, throughout GAFFES, the service interface is the same, client organizations can shop around for instances of services to suite their current needs.

The definition of trust described earlier lends itself to this kind of environment. Each service has an owner who asserts a place in the spectrum of service quality through a guarantee. Potential clients of services assess the guarantees and the integrity of owners before determining which services are reliable. Clients (aided by software) use these assessments to generate coherent belief systems. The intangible issues in trust, describing and assessing guarantees, are outside of the scope of GAFFES and handled by human beings. Belief systems, listing what operations are acceptable under what circumstances, are simple enough to be used by software.

Clearly, roles could require arbitrarily large and complex belief systems, but, fortunately, most useful trusts can be expressed compactly. Related beliefs could be grouped into a single statement by wildcards. Also, we think most trusts will be determined at an organizational level and shared by all

roles administered by the organization. Individual roles may have several more trusts than their organizational belief system, or fewer, but most of the role's belief system would be represented by implicit reference to the shared organizational belief system. For example, if a a user delivers all mail to the departmental mail server, he or she implicitly uses all of its trusts. The mail server may have to use a complex system of trusts to decide how to route a message, but the user's belief statement has only one statement concerning the mail delivery operation.

Formally, a belief system is a series of triples-- {owner, operation, acceptable parameters}. (Belief systems could actually be implemented as relational data bases.) Before invoking a remote operation on behalf of a role, the client library or a service will use the data base to map an (operation, set of parameter values) pair to a list of (guarantor, condition) pairs. A guarantor is a reference to a trusted service; conditions are limitations on valid invocation results.

Guarantors fall into several classes: "meta-guarantors," owners, any-service-at-all, or no-service-at-all. A meta-guarantor is a (network address, public key) pair that is assumed *a priori* to lead to an acceptable server. Meta-guarantors are established by means outside of the GAFFES system. They are the necessary "hard" links to gateways, local name servers, and file servers that bootstrap the security mechanism. The addresses and public keys of services other than meta-guarantors will be determined indirectly (through a name service or a reference by a server related to the service requested). These indirect references will include the owner of the referenced service so the invoker may check the service against its belief system. A service address received through an indirect reference is valid if the owner, or any-service-at-all is listed as a guarantor in the referees belief system.

Most of the widely-used services supported by GAFFES will have to be partitioned or otherwise distributed. Invoking an operation on a distributed service might involve remote operations on servers in several different organizations. The choice of which, if any, of these servers to invoke must be made with respect to some belief system. If these indirect invocations were made by the service that the client invoked originally, that service would either need to have access to its invoker's belief system or use some other belief system (presumably that of the owner of the service). It would be impractical for an invoker to ship its belief system in the invocation, but forcing clients to accept the service's beliefs would severely limit the trusts available to some applications.

Our solution to this problem is to have services support a "iterative" and "recursive" operations. The invoked object completes an iterative operation only if it can do so using information obtained locally. If not, the operation returns enough information about other remote services (owner, address, and public key) that the invoker may continue the operation using the service of its choice.

4.4.8.2. Combining Iterative and Recursive Operations

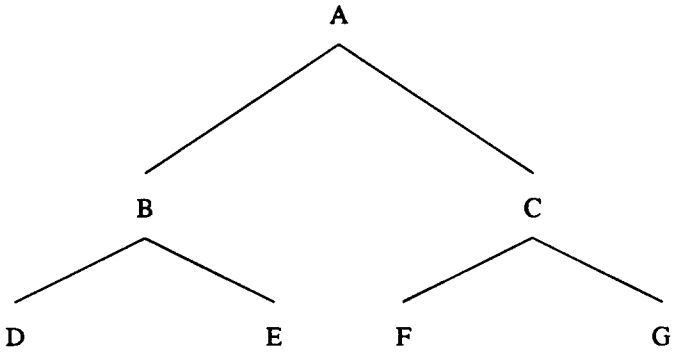
The GAFFES collective name service must support both iterative and recursive operations. When a high level service (/EDU) cannot resolve a name, it usually knows of several lower level name services (/EDU/California/Berkeley, /EDU/California/Stanford) that can complete the resolution. Choosing one of these services to complete the resolution must be done on the basis of some role's belief system. If the original name service (/EDU) were invoked through the recursive resolve_name operation, it would choose a lower level name service based on the belief system of it's *owner*. If the higher level name service were invoked iteratively, the resolve_name operation would return an indirect reference (address, public key, and owner) to the remote name services. The client, in this case, would use its belief system to choose one of these name services for a second invocation.

Iterative operations allow clients of cooperatively-administered distributed services to use their own trust relationships to generate *trusted paths* through the services. At each point in the path, the client decides which instance of the required service is acceptable to it and invokes another iterative operation at that service. At some point in the trusted path, the client will come to a service that it trusts enough to call using a recursive operation. Since the process of iterating through partially trusted services is expensive, clients will probably cache trusted paths to end services. Using timeouts as conditions in the belief systems will help the client's services make decisions about how to tradeoff performance versus security in the cache.

4.4.8.3. Trusted Authentication and Iterative Name Resolution

Access lists must refer to roles indirectly, through names, instead of directly by public key. It is convenient to have role names reside in the hierarchical file name space so that the GAFFES need not implement a parallel name service for the sake of roles. However, this convenience should not compromise security by forcing a particular service to trust its parent in the name space. Fortunately, while name assignment must follow the hierarchical organization, resolution of a name (key or address) need not. By using hard links (in the form of meta-guarantors) to connect disjoint subtrees of the name space, a role can build up a set of relative name resolution paths which skip untrusted parts of the name space.

Let us consider an example. The tree below represents a section of the naming hierarchy. Each of the nodes is an authentication or naming server.



If authentication were hierarchical, then D would have to trust B, A and C in order to get the public key of F. However, in our scheme, B may maintain a soft link to F of the form (A/C/F , public key for F). If D trusts B to always return a correct public key, but not A or C , then a call to B to determine the public key of F would return a resolved value without going through an untrusted server.

Relative name resolution could introduce hidden security violations in some cases. The presumption that the hierarchical name space insures unique names isn't strictly true. This presumption implies that a node in the hierarchy must be trusted to insure that all the nodes under it are named in a unique fashion. A malicious node may indeed create a spurious node under it that has the same name as some other node. As long as name resolution is relative, this will not have any affect on users that do not trust the malicious parent node. If the parent node is not trusted, it is not in the set of servers used to resolve role names during the authentication process. However, passing a name between users creates a potential security violation since, although the name of an object looks the same to both users, they may use different relative paths to resolve the name.

4.4.9. Summary

To recapitulate, the GAFFES authentication and security scheme associates file access rights with roles; users in roles invoke operations on files. The GAFFES security mechanism must prevent users without authorized roles from carrying out operations on file objects. To enforce this, GAFFES must make sure that (1) the role has the correct access rights and (2) every invoking role is authenticated.

Validating access rights is trivial, it means merely checking a list of authorized roles before allowing the operation. GAFFES authenticates roles with a system of public keys. After the request is authenticated, the requester and a file object establish a conversation key. With it, they may continue file operations using standard network security techniques-- encryption of text, windowing to prevent replay etc.

The difficulty of public-key-based authentication lies in obtaining the correct public key for a role. To map the name of a role to a public key in a trusted manner, a GAFFES file server must establish a path to the key holder that does not depend on untrusted servers. By using iterative operations

and choosing the path based on trusts specified in a belief system, the file server can generate such a path-- even if it requires the participation of partially trusted servers. Once in possession of a public key, it is a straightforward task to authenticate the role by its digital signature.

4.5. Concluding Remarks on Security

Our formalization of trust as statements in a belief system is a new idea. We think that it provides a useful framework in which to discuss issues regarding systems without global trust. Our use of this formalization in our design shows its utility in the design process.

We think that associating access rights with roles is a good idea. Partitioning a user's rights into logical units contains the damage due to a security leak. Because roles provide an extra level of indirection between the user and the system, we are able to gain the standard benefits of indirection such as sharing, easy revocation and added flexibility.

Basing security on public keys seems to be a good decision. Public keys allow a clean authentication interface, and delimit trust in the system. Systems that use secret keys need more extensive trust relationships.

Triggers are a potential security problem. Since they act on behalf of a role in hidden ways, it is much harder to keep them in check. We feel that in a system without global trust, users should avoid operating on a file outside of the organization they belong to if the file uses triggers.

In conclusion, the design of security in GAFFES shows it is possible to build a distributed system in which all system components are not assumed reliable or trustworthy.

5. Replication and Caching

Replication of files in a distributed system has the primary goals of supporting high file availability and improved performance of heavily-used files. The design of GAFFES requires both of these features to support its use as a general purpose information exchange system. The replication service proposed in this section is designed to give file owners control over the availability of their files and file users control over the performance of the files they use. The replication and caching section is divided into several subsections. The first defines the replication service goals. The second states several assumptions. The third defines the replication service model. The fourth provides specific details of the replication service. The fifth describes some alternate mechanisms. The sixth suggests some unanswered questions and sources of future work, and the last summarizes this section.

5.1. Replication Service Goals

The replication service model we propose for GAFFES is intended to support client-selectable levels of file availability and file access performance. Naming transparency will be supported to hide the issues of replication naming. The first two goals are related and conflicting. Availability is supported through *consistent* replication of a particular file in multiple locations. No performance assumptions are made. High performance is attained by caching files on machines local to the source of the file request. Consistency guarantees are necessarily relaxed. Transparency is a obtained by hiding from the client the extent of replication and the locations of replicated files. The following paragraphs enumerate the specific properties which the authors' replication model will support.

Availability Property

Subject to the degree of replication, a client can access the desired file if at all possible. Performance degradation is a reasonable cost to pay for high availability.

Performance Property

In general, performance is obtained at the expense of availability and visa-versa. However, high file read performance can be obtained if the client can be satisfied with a loose consistency guarantee.

Transparency Property

As mentioned earlier, the basic transparency property is uniform naming in the face of replication.

5.2. Assumptions

Several assumptions directed the design of our replication model. These assumptions fall into two categories: those related to the expected use of GAFFES and those related to the other components of GAFFES.

Assumptions about expected use of GAFFES:

We assume that GAFFES will be used mostly as a general purpose information distribution and sharing facility where performance and consistency requirements are flexible. We do not intend that GAFFES be used as a high-speed transactional system (e.g. banking). File reading is assumed to be more common than writing (this is a condition which seems to be true in many circumstances [Ousterhout 85]). In addition, there will probably be few writers per file (usually just the creator). This makes write conflicts rare. These assumptions are essential since the optimal design of a system, primarily one on the scale of GAFFES, is based on the the typical or average case [Lampson 83].

Assumptions about other components of GAFFES:

The **naming service** is abstractly viewed as a database of information on a per file basis. The naming service is expected to return handles to files. Each client has a basic set of file operations (e.g. open, read, write), which contain the appropriate replication service calls. An example of how this is done will be given. There is an underlying **security** mechanism which is used to authenticate machines participating in GAFFES and to validate file access.

5.3. Replication Service Model

The replication model can be viewed as set of interactions among five components.

requester	client machine and role[1] of user requesting file operation
name service	distributed naming service
replication server	machine providing replication services
replication sites	file server storing file replica
cache machine	machine on which a file is cached (if different from client machine)

Two of the above components require elaboration (complete details will be provided later in paper)

replication server

Replication servers provide replication functionality and store replication information. Each replication server is responsible for some subset of the globally active files. Thus each file has some set of servers which handle its replication. A replication server provides the following support for each file it is responsible for.

[1]A user or group of users having access to a file. When a file is created the creator specifies a list of roles having read access and a list of roles having write access.

functions	return handle to a replica
	mediate write locks
	notify cached sites of updates
storage	maintain list of replication sites
	maintain list of cache sites (for automatic update notification)

replication site

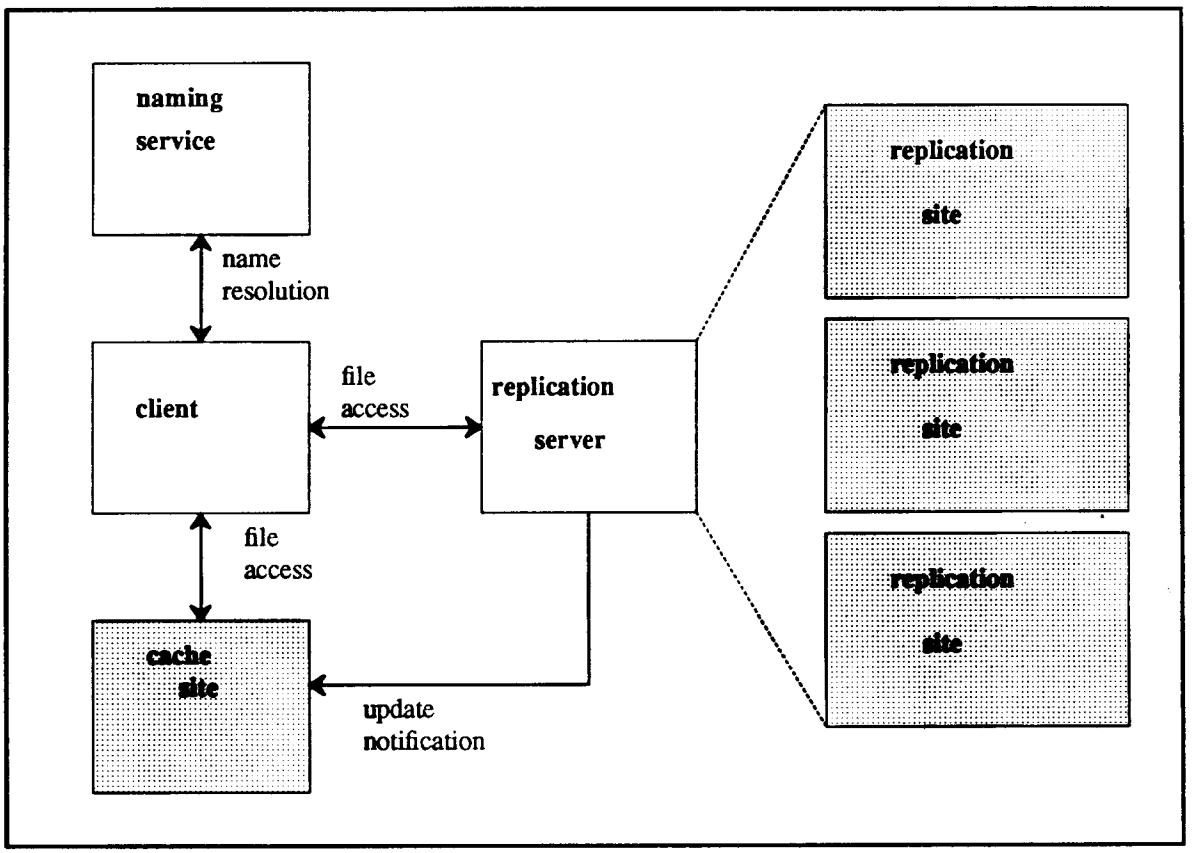
A replication site is a file server, within the *trusted domain*[2], which stores a replica of the file.

There is an important distinction between replication servers and replication sites. Replication servers are machines dedicated to providing the replication functionality (e.g. much like the name servers provide naming functionality). Replication sites are simple file servers which store copies of a file. There is no inherent reason why replication servers and replication sites can or can not physically exist on the same machine.

There are five interactions which take place among the aforementioned components.

- requester and name service
- requester and replication server
- replication servers among themselves
- replication server and replication sites
- replication server and cache site

[2]Collection of machines trusted by file creator (see section on security).



The requester of a file operation makes a call to the name service (e.g. a name server). The request returns a *handle* to the file, which is used to access the file. The handle can contain one of two kinds of information. If the file is not a replicated file then the naming service returns the machine identifier (MID) of the file server storing the file. In this case, the replication service does not get involved. If the file is replicated then the naming service returns a list of replication server MIDs which have responsibility for the file.

One of the replicated servers is chosen from the list returned by the naming service. The choice is made by the client machine because it is the only place capable of making an *intelligent* decision about which replication server is closest (assuming global topology and network status information is not kept). Clients can keep long term replication server availability information to facilitate this decision. All interaction with a replication server takes place in the form of remote procedure calls. There are six replication service functions.

rep_open_read	open replicated file for read
rep_open_write	open replicated file for write
rep_write_commit	commit update made to a file
rep_write_abort	abort an update to a file
rep_register_cache	register file cacher
rep_add_site	add replication site for file
rep_delete_site	delete replication site for file

These six functions provide the functionality necessary to support replication.

Replication servers communicate among themselves to insure that a majority have the same replication state for those files for which they are responsible. Replication state includes information

such as whether the file is in the process of being updated (e.g. after rep_open_write but before rep_write_commit).

Replication servers communicate with the replication sites to add new versions and to verify the storage of files.

Replication servers communicate with cache sites to send file update notification.

5.3.1. Availability Issues

Availability is a function of the number of replication sites storing a particular file and the number of replication servers responsible for the file. The replication servers are responsible for finding an available replica of a file when requested by a **rep_open_read** or a **rep_open_write** call. A client machine opening a file with either of these two procedures is assured of contact with a replica if at all possible.

The client can specify a consistency requirement which guides the replication server in finding the file. Consistency can be specified in two ways: need the the latest version or willing to accept a possibly out-dated version. Because of the expected uses of GAFFES, no read locks are provided. Consistency requirements are only valid for read operations since writes must be performed on the latest version after a write lock has been obtained.

5.3.2. Performance Issues

Performance of file access is improved by caching files at *nearby* machines (file server or client machine). Caching improves performance by reducing the access cost to files since the files are located in a more optimal fashion (e.g. closer). The authors believe that, in GAFFES, only read caching is necessary. Thus a mechanism to flush back updated cache copies is not provided. The reason is that if a cache copy can be globally updated (the cache copies can be locally manipulated) a lock on the file will be required to avoid conflicts. This is the same situation as the normal file updating (without caching) when a replication site is near the writer. This is generally the case because the file creator defines the write access list and the replication site list. Thus there is very little if any added benefit in supporting cache updating. The three main issues related to caching are when to cache, where to cache, and how to maintain cache validity and consistency.

5.3.2.1. Cache Decision

The decision of whether to cache and where to cache a file is made by the user requesting access to a file. The user determines the location of the cache copy. There are two ways of making cache decision other than explicit user control.

(1) File access operations can automatically cause files caching. This is the model used in the ITC system at CMU [Satyanarayanan 85].

(2) An automatic caching mechanism which dynamically responds to file usage patterns could be used. Such a facility would be extremely complex. Deciding if and when to cache would require a great deal of global system knowledge, such as storage capacities and communication bandwidths of nearby machines. Such knowledge would be particularly difficult to accumulate in a distributed system such as GAFFES due to the size of the network and the issues of trust (i.e. can you trust the information others give you). It may be feasible for a local cluster (e.g. LAN) to maintain such knowledge and share a cache.

5.3.2.2. Cache Consistency

File caching improves read performance at the cost of consistency. To alleviate this problem, an automatic notification mechanism is provided. In this scheme, cache sites register their location with a replication server responsible for the file. The replication server will then send notification to these sites when the file is updated. There are several disadvantages to such a mechanism.

- The replication servers incur an extra storage cost associated with maintaining cache notification lists.
- The replication servers have an extra processing and communication burden as a result of notifications.

However, notification updates have several advantages.

- Subject to propagation delays and communication failures, the cache site will eventually contain the latest version, as in the Grapevine system [Birrell 82].
- The decision of what to do with the notification is left up to the caching site. The cache site could choose to ignore the notification, or it could request the latest version if it was needed. Thus notification can be less expensive than sending the entire file.
- Notifications allow replication servers to handle the update problem all at once, rather than each time a cache site validates its cache on open. This situation occurred in ITC, and the designers of that system suggested that a notification scheme is preferable and in fact scales better [Satyanarayanan 83].

5.3.3. Example Client File Operation

In order to clarify the operation of the replication service, an example of a file access operation on a client is depicted in algorithmic form. The following example, shows how a client would implement an *open for read* file operation.

```
/*
 * Function takes a global file name as parameter, and returns the
 * replication site MID where the file is stored
 */
client_open_read(file_name)



begin
        /* call name service to obtain list of MID or single MID */
        name_information = name_service(file_name)
        /* if not replicated file then simply use the MID returned */
        if name_information.type = NOT_REPLICATED
                return(name_information.mid)
        else /* name_information.type = REPLICATED */
                /*
                 * choose closest replication server from among those returned
                 * from name service.
                 */
                rep_server = closest(name_information.mid_list)
                /*
                 * call replication server to find out machine id of file
                 * server which is storing a replica of the file
                 */
                mid = rep_open_file(rep_server, file_name, READ)
                return(mid)
        endif
end
```

5.4. Replication Service Specifics

The replication service provides four distinct services: replicated file reading, replicated file updating, caching mechanisms, and replication database control.

5.4.1. Replicated File Reading

Reading a replicated file is the simplest of the replication services. The basic operation is to map a logical file name, which represents the collection of replicas, into one of the physically replicated copies. The operation is provided by the **rep_read_open** remote procedure call to a replication server. The procedure is straightforward if the requester is willing the accept a possibly out-dated version. The replication server obtains the replication site list for the file (it has this because the called replication server has responsibility for the file), and then chooses one. The chosen replication site is contacted to verify that it is available and stores the desired file (note that the file is a file name plus a file version). If the answer to this question is negative, then the replication server will try another replication site from the list. It is possible that no replication site stores the requested version of the file, in which case the open operation fails.

Some complexity is introduced when the requester must have the latest version. If an update is in progress then the replication server will refuse the read request. This means that the replication server will have to communicate with a majority of the other replication servers to determine if a write lock has been granted (unless the contacted replication server has a write lock set). If the requester is willing to accept a possibly out-dated version, then whether or not an update is in progress, the replication server can proceed.

summary of open for read operation

1. obtain replication site list for file
2. choose optimal replication site
3. verify availability of file on replication site
4. return machine id of replication site

5.4.2. Replicated File Updating

Updating a replicated file is an involved process. The complexity results from the necessity of avoiding inconsistency that would result from the update of two (or more) replicas of a file concurrently. Replicated file updating is a two step process: update and commit. Note that file updating can only be done to the latest version, so if some other client has a write lock the write request fails. In order to update a file, the file is opened for writing with the remote procedure call **rep_open_write** to a replication server with responsibility for the file. The replication server attempts to establish a write lock on the file, on behalf of the requester. The write lock is necessary to prevent concurrent updates. Obtaining a file write lock is an atomic operation performed by the replication server, in which a majority of replication servers responsible for the file grant a lock to the requester. The atomic nature of the operation prevents deadlock. To avoid starvation, requesters wait a random time interval prior to retrying. If the write lock is granted, a copy of the latest version of the file is created on a replication site and the write can proceed. The file writer determines which replication sites will store replicas of the new file. The file writer can also choose which one of these replication sites to use as the update site, assuming it is available. This copy serves a similar purpose as the shadow page in Locus [Walker 83]. The copy does not become a version of the file until the update is successfully committed. The commit operation is accomplished by the remote procedure call **rep_write_commit** to the same replication server that granted the write lock. As part of the commit process, the update server sends to each other replication server the list of replication sites which contain the new version. Also during this operation the new version is copied to the replication sites determined by the file creator, and the naming service is notified of the updated version. A two-phased commit protocol is not necessary since replication site lists are interpreted as hints to replica storage rather than fact (i.e. if a

replication site does not contain the desired file, another replication site is contacted).

summary of file update operations

update
1. replication server contacts other replication servers
2. replication server establishes lock with majority of servers
3. replication site chosen to serve as update site

commit
1. notify other replication server that update is committed and send list of replication sites
2a. propagate update to the replication sites
2b. notify naming service of new version

5.4.3. Caching

There are two important aspects to caching: cache machine perspective and replication server perspective.

5.4.3.1. Cache Machine Perspective

When a cache copy of a file is created, it receives a name local to the caching machine. The caching machine maintains a mapping of global names to locally cached file names. On file operations, this list is searched before invoking the global naming service. This accomplishes two goals of the replication service by reducing overhead when accessing cached files and maintaining naming transparency. Access performance to cached files is optimal because the replication and naming overhead is bypassed and the file is located at a *nearby* machine (of course the actual choice of where to cache is left up to the user or the user's operating system).

The operation of caching a file can be summarized as follows.

1. copy file to cache site (from replication site, by means of a replication server)
2. save global to local name mapping
3. register with a replication server (if update notification is desired)

5.4.3.2. Replication Server Perspective

Replication servers only need to know about cached files if automatic update notification is desired. The remote procedure call **rep_register_cache**, is made to any replication server responsible for the file. If a cache file is registered, then whenever the file is updated, a notification of the fact will be sent to the cache machine. This notification can be used in a multitude of ways by the caching machine. For example, an operation could be initiated which would copy the file (with **rep_open_file** and then read the entire file over), thus updating the cache. A trigger can be explicitly activated (e.g. by an open operation on a cached file) which will cause the latest version of the file to be copied. The cache validation and update policy is client definable.

5.4.4. Modifying the Replication Database

There are two types of replication database modifications. Changing the replication site list for a file version and changing the replication server list for a file.

Replication *site* list changes can be made to reflect the changing usage patterns of a file. Consider the following:

(1) Old versions may be rarely accessed and probably do not need to be replicated in large number of places. In addition, replication sites may delete old versions of files.

(2) It may be difficult to predict the way a particular file will be used, and the creator may have made a poor choice of replication sites and wish to modify the list to better reflect actual usage patterns.

The addition or deletion of replication sites for a particular version of a file does not pose a consistency problem because the list is interpreted as hints to replica storage. Thus the list is modified as soon as the add/delete request is received by a replication server. Subsequently, the same request is forwarded to the other responsible replication servers (for the file in question). The replication sites are updated as required. If a replication site is added then the appropriate file is copied, and if a replication site is deleted then the file is deleted.

1. modify list to reflect add/delete of entry in replication site list
2. send add/delete to other responsible replication servers
3. add or delete file from the appropriate replication site

Modifying the replication *server* list for a particular file would allow one to dynamically control the file access availability to replicated files (since every file access has to go through a replication server). This poses a significant consistency problem. Replication servers participate in majority algorithms and so dynamic changes to the set of replication servers for a particular file compromises the integrity of the algorithms. The marginal advantage of adding this capability in comparison to the added complexity that is introduced, does not justify this feature. In any case, the file could be copied to a new file with the desired replication server list.

5.5. Alternative Mechanisms

There are many ways of designing a replication service. The authors note some alternatives and reasons for not using them.

combining name service and replication service
: Both the name service and the replication service contain per-file information, so combining the two might conserve storage space. However, replication information changes more often than naming information and only requires majority agreement rather than absolute agreement as in the case of the naming service. Forcing atomic replication information update would unnecessarily degrade performance. Nevertheless, it may be possible to integrate the two services if naming consistency is relaxed.

replication service functionality
: The authors' proposed model has most of the replication service functionality contained within the replication servers. The responsibility could be placed upon the clients (system might scale better), however it would require that each client implement the same replication service. In addition, issues of trust would make the the design very difficult.

5.6. Future Work and Remaining Questions

The authors have defined the basic operation of a replication model for GAFFES. Without performance simulations, benefits of some features are unclear. Several issues remain partially or entirely unanswered.

? Can the assignment of replication server responsibility to files be automated? Currently, it has to be defined by the file creator.

? Likewise, can the assignment of replication sites be automated? Could the file creator (or file updator) simply define the degree of availability and performance desired (how would these be defined?) and then have the replication sites automatically chosen? Issues of trust would have to considered.

? Can the naming service and replication service be physically and/or logically combined?

? If replication servers do not impose too much overhead, it might be advantageous to have access to cached files directed through them, thus freeing client machines from keeping mappings of the files they have cached.

? How much overhead do replication servers impose?

? Would read locks be useful? How can they be designed to avoid significant performance degradation?

5.7. Summary

The proposed replication service model is designed to provide three essential features for GAFFES:

- availability control
- performance control
- transparency

Availability is controlled in two ways: by the number of replication servers and by the number of replication sites. Increasing the number of replication servers increases the chance of being able to perform a file operation. Increasing the number of replication sites increases the availability of the file.

Performance is controlled by carefully selecting the location and number of replication sites and replication servers and caching files. Replication sites should be chosen "close to" where the file will be most heavily accessed. There should be few replication servers if high write performance is required, but a larger number if high availability is important. If there is only one replication server for a file, a write lock can be quickly obtained. If replication servers are well connected, this may be a satisfactory configuration for most files. Caching is used when read performance is important (to avoid global communication) and absolute consistency is not essential.

Transparency of the replicated nature of files is maintained in two ways. A global name of a file refers to the collection of replicas of the file (a mapping controlled by the replication servers). References to cached files are mapped from the global file name to the local file name by the client machine.

We believe that the replication model proposed in this section is appropriate for the read-oriented, few-writers type use that GAFFES would encounter.

6. Triggers

The design of GAFFES allows extensive use of triggers. Triggers have not received much attention in current literature; this section includes a detailed discussion of triggers and their operation in a distributed file system.

6.1. Definition of Triggers

A trigger is defined as a program initiated by and associated with a standard system operation *(its parent)* and executed in addition to it.

A **trigger** is represented by a triple consisting of a file name, the operation to trigger on, and a file containing an executable program. There may be multiple triggers associated with an operation. A trigger can execute simple commands itself, but cannot influence its parent (see below). Since a trigger executes a program, it can return results for future use. The trigger routine has access to all the data arguments available to the parent primitive operation.

6.2. Objectives of Triggers

Triggers have many uses in a globally distributed file system.

In Active Files

Files can be treated as active objects which may respond to a request. Numerous possibilities for the use of triggers exist. Examples include text files which reformat themselves when they are modified and files which decompress themselves upon use.

For Consistency

Relationships between files may be explicitly expressed in program semantics. Consistency relationships can be enforced. For example, deleting a record from a file updates the B-tree index of that file.

For Extensibility

Triggers provide a limited form of system primitive extensibility. The semantics of system operations may be extended by additional functionality of trigger programs.

6.3. Characteristics of Triggers

Triggers satisfy the following conditions.

Causality

A trigger cannot influence its parent operation, leading to the concept of causality of operation. This ensures consistency. Atomicity of trigger operations is not guaranteed if triggers are permitted to abort or delay indefinitely the parent operation.

Given a primitive operation S and its associated triggers $T_1, T_2, T_3, \ldots\ldots$, infinite loops cannot exist.

Justification: The *causality* property of the triggered operations ensures that recursion is precluded. Let $S_1, T_{1,1}, T_{1,2}, S_2, T_{2,1}, T_{2,2}, \ldots$ be a sequence of operations allowed by a system, where S_i for all i denotes a primitive operation and $T_{i,j}$ refer to triggers associated with primitive S_i. Causality implies that the precedence order $S_i, \ldots\ldots, T_{i,j}, \ldots\ldots, S_i$ is invalid for all j and any i. This eliminates any recursion.

Independence

If triggers T_1 and T_2 are two triggers initiated simultaneously by the parent operation S, then they run independently of each other, and of S. This condition allows the triggers for an operation to be executed in arbitrary order or in parallel.

Flexibility of functionality

Triggers are flexible, their functionality being determined either by the system or the user as will be discussed below. Since triggers can be arbitrary client programs, the functionality of triggers is potentially unlimited.

6.4. Control of Triggers

Triggers may be client programs; their access rights are strictly controlled. Triggers run under the following modes of control:

User:

Triggers may run with the access rights of the user of the trigger. The advantage of providing for user control is the facility for utilizing the user's *a priori* knowledge. However, if the triggers run under the user's authority they must have access to the user's private key. The triggers may misuse this information. This creates a problem with security and authentication, which must be solved.

Role:

A trigger will require access rights in order to carry out file operations. Since access rights in GAFFES can only be assigned to roles, the creator of a trigger must create a role to serve as an

identity for the trigger. The *proxy problem* discussion in the security section of this paper gives more detail.

System:

The system administrator of the controlling domain may add triggers. This system control is intended to provide for routine housekeeping triggers, using **invisible triggers** as described in the following paragraph.

6.5. Classification of Triggers

On the basis of their domain of control, triggers can be classified into the following two types:

Invisible Triggers

These are provided by the system to the primitive operation. The user or role under normal conditions would not be aware that they exist. These triggers are programs which are routinely run, e.g. appending a compiled file with a ".o" suffix. These triggers could also be used for accounting operations and other system housekeeping jobs associated with the primitive. In addition, invisible triggers are used to enforce atomicity of trigger operations.

Primitive operations invoked may also be aborted. In a large scale distributed system, the atomicity of the primitive operation is important. It is necessary to ensure that triggers running at the behest of a primitive operation on a remote site are aborted too, in case the parent operation is aborted. Therefore triggers and their parents need to come under one atomic operation.

Optional Triggers

These triggers can be selected by the user/role. The creator of a trigger specifies whether this is allowed. One of the advantages of an active file system is the utilization of semantic information to speed up operations. Optional triggers offer the system a method of making the system user-interactive. Since the role best predicts its actions, the role can choose the triggers it may wish to run. The triggers are available as options displayed to the role as a trigger library. For example, a trigger need not compile source code each time the file is edited, it may do so if the user/role opts for it. This would avoid loading down the CPU with avoidable jobs. It may be argued that the user need not know about the existence of triggers, but performance improvements encourage such interaction. For example, many users are familiar with the operations they intend to execute, and their accompanying effects, and it would be advantageous to use this information [Garcia-Molina 84].

6.6. Execution of Triggers

There remains the question as to whether triggers should run on the client machine or the server. If the triggers run on the server machine they could degrade performance. Running on the client machine would imply that the client machine is available during the operation. This situation is further complicated by the heterogeneous nature of clients.

With the above considerations in mind, the following guidelines are suggested for triggers:

A. The action of firing a trigger invokes the program on the server containing the file. Triggers must specify files containing code executable on the server storing the file. To support heterogeneous server types, the program could be a command script which selects the correct executable image for the server.

B. In order to avoid overloading of servers, triggers may be subject to resource limitations specified by the server's administrators (for example, CPU, file access and real-time limitations). Any trigger exceeding these limits can be terminated. The primitive operation has to be rerun without the trigger, to maintain atomicity.

C. Triggers are permitted to use remote procedure calls to procedures registered on the client's machine. Using the RPC mechanism, resource-intensive triggers can be run without overloading the servers.

The above design has several important features. Simple triggers are executed quickly on the server machines while resource-intensive triggers are forced to be offloaded. Heterogeneous clients are supported trivially because of the use of remote procedure calls.

6.7. Primitive Operations Using Triggers

Triggers are associated with standard system operations. Trigger primitives are provided to the user only in the context (system state) of the standard primitive already used. That is, the system executes these commands in the context established by the standard operation.

The following are the primitives for the active file system:

No.	Primitive	Description
1.	display_trigger_options	lists optional triggers.
2.	set_trigger_mask	selects triggers to be executed.
3.	add_trigger	adds a program as a trigger.
4.	delete_trigger	deletes a program as a trigger.

The invisible triggers are executed by the system and the user does not control them. The invisible triggers also execute *anti-triggers* for maintaining atomicity of operations, as explained later.

6.8. Atomicity at the Trigger Layer

The file system assumes a transactional support layer which guarantees transactional recovery and atomicity of operations. The choice presented to the system designers was either to use this layer to support atomicity of triggers or add a new layer for guaranteeing consistency in a globally distributed environment. We chose the latter solution, and a new layer of consistency at the trigger level was constructed, which we believe has a number of advantages in a distributed environment.

The most significant advantage in building a new layer for guaranteeing atomicity, rather than assigning it to the underlying operating system transaction support system is simplicity [Stonebraker 83]. An operating system log manager is unable to use the semantic content of the transactions; hence the operations have to be logged in detail. A logical logging system we propose simplifies recovery at the cost of computation only at recovery time. This speeds up the normal operation of an active distributed file system.

Another advantage is the locking overhead saved by using logical locking in which only the triggers and primitives are stored. An operating system transaction manager locks relations for the entire duration of the transaction, reducing transaction throughput and concurrency.

Anti-triggers introduced below simplify the rollback and atomicity constraints in an elegant manner. The transaction support mechanism has a coarser granularity of locks. We now describe in detail the atomicity of trigger operations.

6.8.1. Anti-triggers

Given a trigger T_i there exists an anti-trigger A_i which rolls back the effect of the operation T_i on the state of the system. As introduced before, each trigger associated with a primitive operation is independent of the other triggers, and consequently the anti-triggers are also independent of each other. In addition, in any sequence of operations, the anti-trigger T_i can only be invoked by the system if the trigger T_i has been invoked previously. Therefore, if a trigger has been executed, it is assumed that there exists an anti-trigger which undoes the effect of that operation.

This suggests a guideline for programs which qualify as candidates for triggers: triggers should always have anti-triggers and vice versa. This may limit the flexibility of triggers, but it has the advantage of ensuring atomicity by allowing for the rollback of operations. Any program which does not have an anti-trigger may, however, be run separately.

6.8.2. Atomicity of Triggered Operations

In this section we make precise the notion of atomicity of transactions using triggers and anti-triggers. Let S be the standard system operation being invoked, and let T_i be the triggers set, and A_i be their corresponding anti-triggers. Then either of the following sequence of events is executed;

$$[\, S\ , T_1\ , T_2\ , \ldots\ldots , T_N\ ]$$

or

$$[\, T_1\ , T_2\ , \ldots\ .\ , T_J\ , A_J\ , A_{J-1}\ , \ldots\ldots\ , A_1\ ]\ , J < N\ .$$

is executed.

The above definition for atomicity ensures that if the standard system primitive S has been aborted, then all the triggers are undone. The entire operation is governed by a three-phase protocol, which we discuss next.

6.9. Three-phase Trigger Protocol

The three-phase protocol, guaranteeing atomicity at the trigger layer, consists of the following phases.

Phase 1

The system primitive is invoked, but the system does not permit the execution of the primitive until it validates the task. This could mean that the system authenticates the request, searches for versions which the primitive wishes to access and related tasks. Until the primitive receives the acknowledgement from the system, the triggers cannot be activated.

Phase 2

When the acknowledgment from the system is received, the selected triggers run. The parent primitive could be aborted at any time.

Phase 3

If the primitive has been executed then all the triggers are in turn executed. The transaction log simply stores the primitive and the triggers invoked. In the case the parent primitive aborts, the anti-triggers are invoked and the operation rolls back. In this way atomicity at the trigger level is guaranteed.

Triggers can operate at either the second phase or before the termination of the atomic operation, and the triggers are designated as **pre-triggers** and **post-triggers** respectively.

The first two phases of the protocol are standard, the third phase is introduced as an atomicity and integrity constraint verifier.

7. Overall Conclusions

The design of GAFFES brings forth two important issues: the problems that arise in scaling services to a very large-scale distributed system, and the difficulties of combining global authentication, naming and replication services. The replication service in GAFFES allows users to choose the level of availability, performance, and consistency that they desire for their files. The naming service provides unique, location-independent names. Descriptive names are also provided, making it easier for users to find files based on their content, purpose, or other attributes. The security system in GAFFES authenticates users and allows them to control the protection of their data and the operations on their files. The use of roles to partition a user's rights into separate units helps to restrict the damage due to a security leak. The formalization of trust as statements in a belief system is a new and useful idea for discussing security in systems without global trust. The triggers that GAFFES provides on files make it possible for file operations to cause further programs to execute. The design of the file system supports embedded references in files and versions of files. It provides a simple interface to the file system for clients that do not have the power and resources to communicate directly with the naming,

authentication and replication services, but it also allows clients with the required power and resources to interface closely with such services.

In designing each of the components of GAFFES, we focussed on the needs of a globally distributed system: availability, reliability, performance, security, and the heterogeneity of the network and clients. The authentication service must protect users from unauthorized release of information and modification of data in an environment that lacks global trust and has no single administrative authority. The naming service must make it easy for users to avoid naming conflicts and must help locate files without fully qualified names. It must be possible to adjust the naming service as the system grows and shrinks without the need to rename any files. In a system as large as GAFFES, it is not possible to pick a single balance of performance and availability that meets the needs of all clients, and as a result, the replication service in GAFFES provides flexible degrees of performance and availability. It is not practical for a global system to demand the same degree of computing power and sophistication from all its clients. We therefore provide both simple and complex interfaces to the file system, making it possible for both simple and powerful clients to use GAFFES.

We have described several examples of friction between the various components of GAFFES. Triggers present a security problem, since they act on behalf of a role in hidden ways. The cleanest solution to this problem requires giving a role to a trigger. The naming and authentication services are particularly tricky to combine. Users who trust only certain name servers must be able to make direct links to those servers and circumvent the usual name resolution process. Security provisions also complicate the naming service by requiring both recursive and iterative methods of name resolution in order that users be able to direct name resolution in accordance with their trust requirements. In addition, there are conflicts between other features of GAFFES; embedded references are most easily handled by defining a file as a sequence of client-definable data blocks and embedded reference control blocks, but the ability to embed references to files without knowing the number or structure of the embedded file's blocks complicates matters if the embedded file is not immutable.

A globally-distributed file system is only one example of the services that will be built as huge world-wide communication networks become publicly available. Most of these systems will need to incorporate security, and object naming, location, and replication mechanisms. Our design for GAFFES is a start towards combining these features in a large-scale globally-distributed system.

References

[Anderson 87]
Anderson D.P., et al.: "The DASH Project : Issues in the Design of Very Large Distributed Systems", UCB CSD Tech Report 87/338, January 1987

[Birrell 80]
Birrell, A. D., et al.: "A universal file server," *IEEE Transactions on Software Engineering*, SE-6, September 1980, pp. 450-453.

[Birrell 82]
Birrell A. D., et al.: "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM*, Vol. 25, No. 4, April 1982.

[Birrell 84]
Birrell A. D., et al.: "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984, pp. 39-59.

[Birrell 86]
Birrell A. D., et al.: "A Global Authentication Service Without Global Trust", *Proc. IEEE Symposium on Security and Privacy*, April 1986, pp. 223-230.

[Denning 79]
Denning D. E., et al.: "Data Security", *Computing Surveys*, September 1979.

[Garcia-Molina 84]
Garcia-Molina H.: *Transaction Processing in a Distributed Database*, *ACM Transactions on Database Systems*, Vol.8, No.2, June 1983.

[Lampson 83]
Lampson B. W.: "Hints for Computer System Design," Proceedings of the 9th SOSP, *Operating Systems Review*, Vol. 17, No. 5, pp. 33-48.

[Lampson 86]
Lampson B. W.: "Designing a Global Name Service", ACM Reprint No. 0-89791-198-9/86/0800-0001, September 1986.

[Mills 87]
Mills D. L.: "A Distributed-Protocol Authentication Scheme", RFC1004, Network Working Group, April 1987.

[Needham 78]
Needham R. M., et al.: "Using Encryption for Authentication in Large Networks of Computers", *Communications of the ACM*, December 1978.

[Oppen 83]
Oppen D. C., et al.: "The Clearinghouse: A decentralized agent for locating named objects in a distributed environment", *ACM Transactions on Office Information Systems*, Vol.1, No. 3, July 1983, pp. 230-253.

[Ousterhout 85]
Ousterhout J. K., et al.: "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," Proceedings of the 10th SOSP, *Operating Systems Review*, Vol. 19, No. 5, December 1985, pp. 15-24.

[Rangan 87a]
Rangan P. V.: private communication to the authors.

[Rangan 87b]
Rangan P. V.: "Security and Trust Relationships in Large Distributed Systems", Submitted for Publication, March 1987.

[Satyanarayanan 85]
Satyanarayanan M., et al., "The ITC Distributed File System: Principles and Design", Tenth Symposium on Operating Systems Principles, December 1985, pp. 35-50.

[Schroeder 83]
Schroeder M. D.: "Experience with Grapevine: the Growth of a Distributed System", *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984, pp. 3-23.

[Stonebraker 83]
M. Stonebraker et al. "Problems in Supporting Database Transactions in an Operating System Transaction Manager," *preprint. 1983* .

[Sturgis 80]
Sturgis H., et al.: "Issues in the Design and Use of a Distributed File System," *Operating Systems Review*, Vol. 14, No. 3, July 1980, pp. 55-69.

[Svobodova 84]
Svobodova L.: "File Servers for Network-Based Distributed Systems", *ACM Computing Surveys*. Vol. 16, No. 4, December 1984, pp. 353-398.

[Swinehart 79]
Swinehart D., et al.: "WFS: A simple shared file system for distributed environment," *Proceedings of Seventh Symposium on Operating System Principles*, December 1979, pp. 9-17.

[Taylor 85]
Taylor B., et al.: "Secure Networking in the Sun Environment," USENIX Summer Conference Proceedings, June 1985, pp 28-37.

[Terry 84]
Terry D. B., et al.: "The Berkeley Internet Domain Server", USENIX Summer Conference Proceedings, June 84, pp 23-31.

[Terry 85]
Terry D. B.: "Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments," UCB/C85/228, Computer Science Division, EECS, UCB, Berkeley, CA, March 1985.

[Terry 87]
Terry D. B.: "Caching Hints in Distributed Systems," *IEEE Transactions on Software*

Engineering, SE-13(1), January 1987, pp. 48-54.

[Voydock 83]
Voydock V.L., et al.: "Security Mechanisms in High-Level Network Protocols," Computing Surveys, Vol. 15, No.2, June 1983, pp. 135-171.

[Walker 83]
Walker B., et al.: "The LOCUS Distributed Operating System," Proceedings of the 9th SOSP, *Operating Systems Review*, Vol. 17, No. 5, pp. 49-70.