INTEGRATING CONTENT-BASED ACCESS MECHANISMS WITH HIERARCHICAL FILE SYSTEMS

by

Burra Gopal

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

1 9 9 7

INTEGRATING CONTENT-BASED ACCESS
MECHANISMS WITH HIERARCHICAL FILE
SYSTEMS

Burra Gopal, Ph.D.
The University of Arizona, 1997

Director: Udi Manber

We describe a new file system that provides, at the same time, both name and content based access to files. To make this possible, we introduce the concept of a *semantic directory*. Every semantic directory has a query associated with it. When a user creates a semantic directory, the file system automatically creates a set of *pointers* to the files in the file system that satisfy the query associated with the directory. This set of pointers is called the *query-result* of the directory. To access the files that satisfy the query, users just need to de-reference the appropriate pointers. Users can also create files and sub-directories within semantic directories in the usual way. Hence, users can organize files in a hierarchy and access them by specifying path names, and at the same time, retrieve files by asking queries that describe their content.

Our file system also provides facilities for query-refinement and customization. When a user creates a new semantic sub-directory within a semantic directory, the file system ensures that the query-result of the sub-directory is a *subset* of the query-result of its parent. Hence, users can create a hierarchy of semantic directories to refine their queries. Users can also edit the set of pointers in a semantic directory, and thereby modify its query-result *without* modifying its query or the files in the file system. In this way, users can customize the results of queries according to their personal tastes, and use customized results to refine queries in the future.

Our file system has many other features, including *semantic mount-points* that allow users to access information in other file systems by content. The file system does not depend on the query language used for content-based access. Hence, it is possible to integrate any content-based access mechanism into our file system.

STATEMENT BY AUTHOR

ACKNOWLEDGMENTS

I could not have completed this work without the help of many people. I wish to thank:

Udi Manber, my advisor, for showing me the right way to approach and deal with challenging situations,

Larry Peterson and John Hartman, my committee, for taking time to evaluate and discuss my work with me,

The lab-staff of the department for helping me with every problem I had,

And most of all, my mother, my wife Renu, and my brother Raghu, for their love and support.

TABLE OF CONTENTS

TABLE OF CONTENTS — *Continued*

TABLE OF CONTENTS — *Continued*

LIST OF FIGURES

ABSTRACT

We describe a new file system that provides, at the same time, both name and content based access to files. To make this possible, we introduce the concept of a *semantic directory.* Every semantic directory has a query associated with it. When a user creates a semantic directory, the file system automatically creates a set of *pointers* to the files in the file system that satisfy the query associated with the directory. This set of pointers is called the *query-result* of the directory. To access the files that satisfy the query, users just need to de-reference the appropriate pointers. Users can also create files and sub-directories within semantic directories in the usual way. Hence, users can organize files in a hierarchy and access them by specifying path names, and at the same time, retrieve files by asking queries that describe their content.

Our file system also provides facilities for query-refinement and customization. When a user creates a new semantic sub-directory within a semantic directory, the file system ensures that the query-result of the sub-directory is a *subset* of the query-result of its parent. Hence, users can create a hierarchy of semantic directories to refine their queries. Users can also edit the set of pointers in a semantic directory, and thereby modify its query-result *without* modifying its query or the files in the file system. In this way, users can customize the results of queries according to their personal tastes, and use customized results to refine queries in the future.

Our file system has many other features, including *semantic mount-points* that allow users to access information in other file systems by content. The file system does not depend on the query language used for content-based access. Hence, it is possible to integrate any content-based access mechanism into our file system.

CHAPTER 1
INTRODUCTION

1.1 Name and Content Based Access

There are two well known ways of accessing information: (i) by specifying names that uniquely identify objects that contain the information, or (ii) by describing some characteristics of the information (other than unique identifiers) in the form of a *query* that retrieves objects by content. Name-based access is available in hierarchical file systems like UNIX [39], graph-structured information repositories like the World Wide Web (WWW, [2]), Distributed Information Systems like Gopher [29], and Object Oriented Database Systems [17]. In file systems, to access objects by name, we specify *path names* to files. Each path name points to file or group of files (directory) [1]. Path names allow us to group related files together and organize them in a directory hierarchy. This makes it easier to *browse* through the files and access relevant information. However, it may be difficult to classify all the files in a single directory hierarchy, since some files can belong to several categories. Another drawback of using path names is that it is often difficult to *find* relevant information in a repository, since we have to explicitly traverse path names to discover objects that contain this information. To avoid traversing path names all the time, we must remember the names of the objects we are interested in, but this may be difficult if the total number of objects in the repository is very large.

To access objects by content, we specify queries in a known format called the *query-language*. Queries describe some properties of the information present in these objects. Queries can extract the relevant information from a file system, or return sets of files that contain the information that matches these queries, or both. Content-based access (CBA) can be very useful in a large and diverse information repository where it may be easier to retrieve objects by asking queries rather than remembering their names or discovering

[1]Though more than one path name can point to the same file, every file has at least one path name that uniquely identifies it.

them by browsing the repository. CBA is available in:

- *Search engines* that provide a "quick and dirty" context for viewing results of queries [28].
- *Information discovery tools* that allow us to do *focused* search on specific targets, e.g., host machines or file systems [21, 3].
- *Relational Database Systems* [17] that allow us to create, access, and manipulate information by specifying complex queries in languages such as SQL [17]. (We do not consider Database Systems any further since our focus is on file systems where files are treated as uninterpreted sequences of bytes, and there is no knowledge of any logical relationships between the data stored in them.)

We shall use the term "CBA mechanism" to refer to any system that provides CBA to information. The main drawbacks of CBA are: (i) we must be aware of the query-language used for CBA, (ii) we must have some idea about the kind of information that is stored in the repository in order to formulate queries that return more or less what we are looking for, and (iii) it is difficult to group-together and organize a set of objects based on the queries they satisfy, since different queries can return different subsets or supersets of a given set of objects. Though we may find it relatively easy to learn a new query language, we may have to browse through the repository first before we know how to formulate meaningful queries. However, browsing may be difficult if the objects in the repository are not organized in a meaningful way.

1.2 Combining Name and Content Based Access

The main goal of our research is to combine name and content-based access together and explore the benefits of doing so. In this dissertation, we describe a new approach for combining path name based access in hierarchical file systems with query based access in CBA mechanisms. The main motivation behind our work was to examine whether it is possible to organize both files and results of queries using a single hierarchical file system. In our approach, we define a uniform interface that allows us to integrate any

CBA mechanism into a hierarchical file system. We rejected the opposite approach of extending a given CBA mechanism to be able to access files [13, 22], because hierarchical file systems have proven to be simple and effective ways to organize information retrieved by name, and we believe that they can also be used to organize information retrieved by content. We focus on *hierarchical file systems* because:

- They are simple and easy to use.
- They usually have well defined user-command and application-programmer interfaces. Such interfaces give us very good starting points and can serve as models when we are trying to design a new interface for accessing CBA mechanisms.
- They usually have mechanisms to go beyond the strict hierarchy of objects in the file system (e.g., symbolic links and mount points in the UNIX file system [39]). Hence it is possible to extend hierarchical file systems to support a variety of ways of organizing information, e.g., arbitrary graph structures like the WWW. (We shall return to graph-structured information repositories in Chapter 6.)

Our work was also motivated in part by some existing systems that provide limited ways to access information by name and content. For example:

- WWW browsers (e.g., Mosaic [1], Netscape [20]) allow us to access hypertext-based information [15] and search facilities (e.g., InfoSeek [44], Yahoo [19]) using the same naming mechanism, i.e., using Uniform Resource Locators (URLs, [15]). The result of a search is also a hypertext document that we can browse in the usual way. Hence, WWW browsers allow us to access information by name and by content using a uniform interface.
- Tools like GlimpseHTTP allow us to browse-through a hierarchical file system as usual, but have search facilities that restrict the files that are searched to those present within the current directory we are browsing and all its sub-directories recursively [27]. That is, GlimpseHTTP allows us to use the underlying organization of information to control the "scope" of our search.

- Semantic file systems allow us to access files by asking queries. They allow us to create *virtual* directories where each virtual directory can have *pointers* to the actual files in the file system that satisfy a query [32, 23, 6, 8]. Semantic file systems also allow us to derive the set of pointers in a virtual sub-directory from the set of pointers in its parent, and thereby build a "personal" virtual directory hierarchy that classifies the results of our queries.

1.3 Goals of our File System

The main goals of our file system are:

1. **Integration of CBA mechanisms:** Our file system should have a well-defined interface to CBA mechanisms. That is, it should map (associate) queries and query-results onto (with) some hierarchical file system abstractions, so that queries their results can be accessed and manipulated using the usual file system operations. Moreover, if a query or a query-result is mapped onto an existing file system abstraction, there should be no restriction on the usual file system operations that can be applied to that abstraction. If any special operations on queries or query-results are required, they should be intuitive extensions of existing file system operations.

2. **Full power of hierarchical naming:** All the file naming facilities that are present in hierarchical file systems should also be present in our file system. For example, users should be allowed to give mnemonic aliases to queries and their results, and access them by specifying path names. Users should also be able to use the same file system to organize files (the information retrieved by name) *and* results of queries (the information retrieved by content). Moreover, different users should be able to organize a given set of files in different ways and share their organization of files with others.

3. **Customization of query results:** Users should be allowed to modify results of queries using their judgement and discretion, so that they can *customize* these results according to their tastes. They should also be allowed to use possibly customized query results in future queries, so that they can *refine* queries according to their

tastes. Users should not depend solely on the query language to achieve these objectives.

4. **Definition of consistency model:** Our file system must have a consistency model that defines how query results are kept consistent with the corresponding queries when users modify (i) the files in the file system, and (ii) the query-results themselves (*without* changing the query or the files in the file system).

5. **Independence from CBA mechanisms:** Our file system should not be tailored for a particular CBA mechanism or query language, because we believe that the type of query language that is most appropriate for a given set of objects depends on the information they contain. For example, if the information is well structured as in a data-base, then powerful query languages like SQL may be more appropriate. However, if the information is in the form of unorganized text, tools like Glimpse [28] may prove to be more useful. We therefore believe that the choice of the CBA mechanism should be left to the user — it should not be built into the design of the file system (see sections 2.4 and 3.14.3).

In the remainder of this dissertation, we describe the design and implementation of a new hierarchical file system, **HAC** (an acronym for **H**ierarchy **A**nd **C**ontent), into which it is possible to integrate any given CBA mechanism. We begin with a survey of some existing systems in Chapter 2, and show that there is a need for a new file system that satisfies the above goals. In Chapters 3 and 4 we discuss the principles behind the design of HAC. With the help of an illustrative example, we show that HAC allows us to organize both files and results of queries using a single file system. We also show that it satisfies all the above goals. In Chapter 5, we describe the implementation and performance of HAC. Finally, in Chapter 6, we discuss some important research issues raised by our work and mention some possible future improvements to HAC.

CHAPTER 2

SURVEY OF SOME EXISTING SYSTEMS

In this chapter, we present a brief survey of some existing systems that form the background for our work. In sections 2.1 to 2.3 we describe some systems that provide both name and content-based access to information. In sections 2.4 to 2.8, we discuss some file systems that are relevant to HAC.

We start with the survey of an information discovery tool that allows users to gather, search and share information distributed in repositories across a network. Then, we describe systems that allow users to both search and browse the information in a repository. Then, we discuss some semantic file systems that allow users to search, browse, and orgnize their files in a hierarchical directory structure. Towards the end of this chapter, we describe some virtual file systems that allow different users to build different personal classifications of the same underlying information, and share their classifications with others. During our discussion, we shall evaluate each system based on whether it satisfies the goals mentioned in section 1.3, and try to justify that there is a need for a new file system that satisfies these goals.

2.1 Harvest Information Discovery and Access System

Harvest [5, 3, 4] provides a set of customizable tools to gather information from diverse repositories across a wide-area network, possibly summarize this information, build topic-specific indices to search this information, replicate and share this information, cache information as it is retrieved from the network, and keep the cache consistent. Harvest allows users to construct diverse types of indices so that they can use search engines of their choice to retrieve information by content.

Harvest software is divided into two major parts: the *Gatherer* and the *Broker*. The Gatherer collects information from different sites in the network and possibly summarizes it. The Broker collects information from many Gatherers, indexes the collected information at the local site and provides a query interface to it. The Gatherer can feed the information it collects to more than one Broker, and thus save collection costs at these Brokers. A Broker can also collect information (and the corresponding index) from other Brokers in the network, and possibly summarize it. This allows users to focus their search on the information they filter from the network. Harvest also provides a top-level Broker called the Harvest Server Registry (HSR Broker), which registers information about each publicly accessible Harvest Broker and Gatherer in the network. The HSR Broker allows users to locate Brokers and Gatherers that deal with the topics they are interested in.

Harvest is an excellent tool to discover and share useful information in the network. As far as we know, it is the only tool that tries to provide a uniform interface to retrieve information from various autonomous repositories, and allows users to collect, summarize, index, search, replicate and share this information. None of the existing tools, including **Gopher**, **Archie**, **Netscape** and **WAIS**, have all these facilities. However, Harvest has one drawback: it does not allow users to organize information according to their personal tastes. Hence, it does not satisfy goals (2) and (3) mentioned in section 1.3.

2.2 Scatter/Gather Browsing

Document-clustering is a technique to partition documents into disjoint groups (*clusters*), where each cluster has documents whose contents are similar in some way. In earlier systems, clustering was used to make searching large repositories more efficient. However, these systems were not very successful since the clustering algorithms themselves were inefficient [11]. The Scatter/Gather system [11, 12], on the other hand, uses document-clustering as tool to make browsing large repositories easier. It automatically clusters (*scatters*) documents in the repository, and for each cluster, it outputs (i) the keywords that occur most frequently in the documents in that cluster, and (ii) the titles of some representative documents of that cluster. Users are free to browse the above output, chose (*gather*) one or more clusters of interest, and recursively apply the Scatter/Gather technique on the documents in these clusters, until they arrive at clusters that have one document each. Note that when users scatter the documents in a subset of a given set of clusters, they do not get back the clusters they started with. This property gives the recursive Scatter/Gather technique enormous power since users can discover fresh classifications of the same information as they narrow down (refine) their choice to the documents (clusters) they are really interested in.

The Scatter/Gather system applies different clustering algorithms at different scattering steps so that it can assign each document to the cluster that best describes its contents. These algorithms run in linear/near-liner time unlike quadratic time clustering algorithms of earlier systems. This is extremely important since the Scatter/Gather technique has to cluster documents "on-the-fly" while users are browsing these documents: it cannot afford to use slower algorithms even if they produce more accurate clusters. The paper contains a user session that clearly illustrates the merits of using this technique to browse large repositories.

However, the Scatter/Gather system has some drawbacks. It does not allow users to interact with the scattering process, i.e., it does not allow them to decide which clusters to create, or which documents to add (remove) to (from) these clusters. The only user-interaction is during the gathering step where they chose the clusters they want to refine. In other words, the Scatter/Gather system does not satisfy goals (2), (3) and (4) mentioned in section 1.3. In spite of these limitations, this work is commendable since the Scatter/Gather system was one of the first to recognize that searching and browsing should be provided together, and the first to use fast clustering algorithms to classify information on-the-fly.

2.3 GlimpseHTTP and WebGlimpse

GlimpseHTTP [27, 28, 47] combines browsing and searching in the context of WWW. It allows users to browse a hypertext-based [15] information base and to search from any point while browsing such that the search is limited to the area suggested by the current document. The information base is assumed to be a regular UNIX hierarchical file structure. This hierarchy is designed and built by the information provider. To allow users to search and browse this hierarchy, GlimpseHTTP first uses Glimpse [28] to index these documents and puts the index in the root directory of this hierarchy. Second, GlimpseHTTP constructs automatically, for each directory, an HTML query page that restricts the search to that directory and all the data below it. (The query page at the root provides search for the whole data.) In addition, each query page contains hypertext links to all the files and subdirectories of the corresponding directory to allow users to browse them. GlimpseHTTP uses Glimpse since it supports very flexible and efficient ways to limit searches to only parts of the indexed information.

WebGlimpse [30] is an extension of GlimpseHTTP. It does not assume that the the information base is a UNIX hierarchical file structure. Instead, it *interprets* the files in the information base as hypertext documents. For each document, WebGlimpse builds a *neighbourhood* file that contains the names of all the hypertext documents that are related to this document in some way. Users can browse hypertext documents in the usual way, and also search the documents in the neighbourhood of the current document at any point while browsing. WebGlimpse allows the information provider to define neighbourhoods in flexible ways. For example, the neighbourhood of a given document can be:

- All documents reachable within 2 levels of hyperlinks from this document,
- All documents reachable within 3 levels of hyperlinks but within the same host machine as this document,
- All documents within the same subtree in the directory hierarchy as this document,

and so on. WebGlimpse was designed so that the information provider can modify the neighbourhood of any document according to his/her personal taste. This allows him/her to restrict the searches to only those documents that are relevant to the current document in some way.

Note that though existing WWW browsers (like Netscape-2.0) provide search facilities (like Yahoo WWW directory) at every step of browsing, the searches themselves are always global. That is, users cannot chose a subset of the information base and restrict their search to that subset. The same is true for the McKinley Internet Directory [35], Excite [9], and others. GlimpseHTTP and WebGlimpse are the first packages we know of that combine browsing and searching of hypertext-based information. Note that GlimpseHTTP and WebGlimpse exploit the existing classification of information to combine browsing and searching, while the Scatter/Gather technique described above classifies information automatically to make browsing easier. The Scatter/Gather system does not allow users to search information by specifying queries of their choice, since it does not allow any user-interaction during the clustering process.

In spite of their advantages, GlimpseHTTP and WebGlimpse have two drawbacks. First, they do not allow users to classify the existing information according to their personal tastes and use this classification to restrict their searches. (WebGlimpse, however, tries to remove this drawback partially by allowing the information provider to modify the neighbourhood-files in any way he/she choses.) And second, though these packages construct HTML pages to display the results of searches, they do not allow users to organize and re-use these pages in any meaningful way. Hence, GlimpseHTTP and WebGlimpse do not satisfy any of the goals in section 1.3 except goal (1).

2.4 MIT Semantic File System

The Semantic File System (SFS) developed at MIT [23] is one of the first hierarchical file systems to provide both name and content based access to files. It makes this possible by introducing the concept of a *virtual directory.* The name of a virtual directory is a query, and the contents are symbolic links to files in the actual (physical) file system that satisfy its query. The only difference between a virtual and a physical directory is that users do not have to create a virtual directory to access its contents. That is, users can `cd` into a query and examine its result by executing the `ls` command "inside" the query. In other words, MIT SFS tries to map queries and their results to regular file system abstractions so that users can access and manipulate them using existing file system software.

MIT SFS assumes that queries are boolean **AND** combinations of "attribute-value" pairs, where an "attribute" is a typed field in the file system and the "value" is a value this field can have. For example, attributes can be "author", "date", "title", "type" "text", etc., and the corresponding values can be "Gopal", "6/13/96", "Survey of Semantic File Systems", "LaTeX-source" and some piece of ASCII text respectively. MIT SFS considers even a single attribute name like `author:` to be a query, and interprets it as the name of a virtual directory that contains one virtual subdirectory for each author in the file system.

Here, `author:` is called a *field* virtual directory. A name containing an attribute-value pair like `author:/Gopal` contains a set of symbolic links to all the files in the file system whose author is "Gopal". Here, `Gopal` is called a *value* virtual directory. To combine attribute-value pairs, users can create field virtual subdirectories in value virtual directories. In this case, MIT SFS interprets the / path name separator as a conjunction operation. Hence, the virtual directory `author:/Gopal/type:/LaTeX-source` will contain all the "LaTeX" files in the file system that are written by "Gopal".

MIT SFS has the concept of *transducer* programs that extract attributes and their values from the files in the file system. MIT SFS uses these attribute-value pairs to index files for query processing. Though it has some default transducers that can process most files in the file system, it allows users to write their own transducers for their personal files. To avoid re-evaluating queries of virtual directories whenever users access their contents (say, by doing a `cd` followed by an `ls`), MIT SFS maintains a cache that maps virtual-directory names to the results of their queries. It tries to keep the contents of virtual directories in its cache up-to-date (consistent) when there are changes to the files in the file system by periodically re-indexing the files, and re-evaluating the queries of the virtual directories in its cache.

Though MIT SFS has many novel features, it has some disadvantages. First, it assumes that queries are always conjunctions of attribute-value pairs, i.e., users cannot integrate arbitrary CBA mechanisms into the SFS. Second, it is built on top of an existing underlying physical file system that contains files — it is not integrated into the physical file system. That is, a virtual directory can contain only queries or results of a queries: it cannot contain, say, files or mount points. Hence, users must use virtual directories to organize results of queries, but use real directories in the underlying file system to organize the actual information. And last, MIT SFS does not allow users to modify the results of queries without modifying queries or files in the file system. That is, users cannot use their judgement to tune the results of queries to suit their personal tastes: they are forced to restrict themselves to the query language to achieve their objectives. Hence, MIT SFS does not satisfy goals (2), (3) and (5) in section 1.3.

2.5 Multistructured Naming

Multistructured naming [45] tries to remedy some problems in MIT SFS. It tries to blend hierarchical or graph structured naming (e.g., naming in UNIX) with flat attribute or set based naming (e.g., naming in MIT SFS). The authors of this work argue that though attribute based naming allows users to retrieve files using any combination of information about them, users lose the "sense of *place*" that hierarchical naming gives. In other words, a path name like `/u/projects/multi-struct/papers/icdcs/final.tex` in a hierarchical name space tells users (i) how the information is organized, and (ii) where the information `final.tex` is located. However, in an attribute-based name space, the above path name (or a permutation like `/u/projects/papers/multi-struct/icdcs/final.tex`) describes the contents of `final.tex`, i.e., that this file is a paper about the project on multi-structured naming. This is very useful since it allows them to retrieve `final.tex` without knowing where it is actually located. However, they lose all information about how the files are organized and where they can be found.

Multistructured naming allows users to build personal name spaces to classify information in an underlying physical file system. It assumes that each file satisfies some "properties", where a property of a file is some condition about that file that can be evaluated: if the condition is true, then the file has that property; otherwise it does not. Multistructured naming also assumes that given a "property", it can be evaluated on all the files in the file system (by contacting a CBA system), and the result is a set of files that satisfy the property. The main idea behind multistructured naming is that it allows users to impose ancestor-descendent relationships on the properties of files, and selectively loosen these relationships, so that they can name files by specifying (i) either their exact locations, or (ii) a list of properties they satisfy in arbitrary order. This is a two step process: in the first step, users assign aliases (*labels*) to the properties they want to use in their path names. A label represents a directory that contains the set of files that have the corresponding property. A path name is a series of labels separated by /. In the second step, they specify a set of rules that determine the relationships among the labels, and thus the structure of the name-space. These rules can be one of the following:

1 *Scoping Rules:* These rules tell the naming system which labels are related by an

ancestor-descendent relationship, and which are not. In a path name, users must specify the former in the ancestor-descendent order, while they can permute the latter in any way they chose. Scoping rules also allow a property **P** with label $\mathbf{l}_p$ to be inherited in the subtree rooted at the label $\mathbf{l}_q$. In this case, users have the option of using $\mathbf{l}_p$ as a component of all the path names that enter this subtree, i.e., the path name "/l_q/foo" and "l_q/l_p/foo" will refer to the same file "foo".

2 *Attribute-Value Constraint Rules:* Suppose two labels $\mathbf{l}_p$ and $\mathbf{l}_q$ can be specified in any order in a path name after the label $\mathbf{l}_r$. Now, suppose users say that all files in $\mathbf{l}_r$ that satisfy the property corresponding to $\mathbf{l}_q$ must also satisfy the property corresponding to $\mathbf{l}_p$, then they can impose a hierarchical relationship between $\mathbf{l}_p$ and $\mathbf{l}_q$ without imposing any order restriction among them. That is, users must use either both $\mathbf{l}_q$ and $\mathbf{l}_p$ in path names of objects inside $\mathbf{l}_r$, or none, but when they use both labels in a path name, they can specify them in any order.

3 *Implicit Value Rules:* These rules allow users to specify the values that are automatically assigned to properties. Suppose there is a property **P** with label $\mathbf{l}_p$, and that $\mathbf{l}_p$ is a descendent of another label $\mathbf{l}_q$ in the label-hierarchy. Now, if users assign the implicit negative value to **P**, then $\mathbf{l}_q$ will contain only those files that do not satisfy **P**. And if they do not assign an implicit value to **P**, then $\mathbf{l}_q$ will contain files whether or not they satisfy **P** (this is the default and this is what one would normally expect). In this case, $\mathbf{l}_p$ can be used as an optional property for files in $\mathbf{l}_q$. Implicit value rules allow users to disable sets of files from a label. This in turn allows them to build some structure into into the label hierarchy which makes browsing easier.

(Note: users can also specify Aliasing Rules that allow them to link up their personal name spaces with that of other users. See [45] for details.) By using these rules intelligently, users can build a hierarchical structure into any attribute based naming system. Note that multistructured naming does not address the issue of how the contents of labels are kept consistent when users modify files in the file system. However, consistency is easy to ensure if each user runs a "background process" that traverses his/her personal name space and contacts the CBA system to re-evaluate the properties of each label on all the files in the file system. Hence, multistructured naming provides a novel way to combine searching and browsing in the context of hierarchical file systems.

The only drawback of multistructured naming is that it does not allow users to delete or add a set of files **f** in a label $\mathbf{l}_q$ with property **Q** without specifying some property **P** with label $\mathbf{l}_p$ of **f** such that **P** is not satisfied by any other file in the file system. Once they come up with such a property, they can (i) delete **f** from $\mathbf{l}_q$ by giving an implicit negative value to **P** and making $\mathbf{l}_p$ a child of $\mathbf{l}_q$, and (ii) add **f** to $\mathbf{l}_q$ by changing the property corresponding to $\mathbf{l}_q$ from **Q** to "**Q OR P**". However, to find such a property **P**, users must solve the much harder problem of coming up with a query (a list of properties) that describes a given set of files (**f**) uniquely, rather than finding out which files satisfy a given query. (This is related to the clustering problem mentioned in section 2.2.) We believe that this limitation can prove to be serious if users choose files in **f** based on some context-dependent information or by some reasoning process which cannot be easily expressed as a property that can be evaluated by a CBA mechanism. However, if we assume that each file has a special property (like a *name*) that describes it uniquely, then **P** would simply be a list of names of files in **f**. But this can prove to be cumbersome if **f** is huge. Moreover, the (unique) names of files will have no relation to their *path names* in the file system (which are labels separated by /), since users can define hierarchical relationships between labels independently of the properties the labels represent. We believe that this distinction is counter-intuitive and makes this system difficult to use. Hence, multistructured naming only partially satisfies goal (3) mentioned in section 1.3. However, it satisfies all the other goals.

2.6 Nebula and Synopsis File Systems

The Nebula file system [6] allows name and content based access to files by adding an object oriented interface that allows a hierarchical file system to interact with a CBA mechanism. Nebula assumes that files are stored in an underlying file system and are collections of attribute-value tuples (like MIT SFS), and queries that retrieve files by content are expressions involving these tuples. Nebula encapsulates each file in a *file-object* that contains the attribute-value tuples that describe the file. Unlike multistructured naming, Nebula assumes that every file has a special attribute that identifies it uniquely.

This is its path name in the underlying file system. Nebula replaces the traditional idea of a fixed directory hierarchy by dynamic *views* of this hierarchy that can automatically classify files in the underlying file system. A view is similar to a virtual directory in MIT SFS and has a query associated with it. It contains file-objects corresponding to the files in the files system that satisfy its query. These files are said to be visible in the corresponding view. (Note that a file can be visible in more than one view.) Like a directory, a view is a namable object in the Nebula file system.

Every view **v** has a *scope* that contains a set of views. When Nebula evaluates **v**'s query, it searches only those files that are visible from the views in **v**'s scope. In this case, **v** is said to depend on all the views in its scope. Nebula therefore allows users to tune the results of a query by changing the scope of the corresponding view, or by refining the query itself. Whenever the set of file-objects in a view **v** changes, Nebula automatically evaluates the queries of all the views that depend on **v**, and thereby ensures that the queries of views and the file-objects they contain are always consistent. Like MIT SFS, Nebula periodically re-indexes files in the underlying file system, and re-evaluates the queries of all the views in the Nebula file system. In this way, Nebula allows users to build a personal collection of views and organize files according to their tastes.

The Synopsis file system [8] is an extension of Nebula that addresses the issue of scale in a wide-area information system by encapsulating a file in the underlying file system within an object called a *synopsis*. A synopsis maintains attribute-value tuples that *summarize* the contents of the underlying file. The interface to a synopsis defines operations to update attributes with their new values, generate new attributes and resolve references to related synopses. A special type of synopsis called a *digest* defines operations to query the attribute-value tuples that correspond to a *collection* of synopses. Like Harvest, the Synopsis file system has an extensive set of procedures to collect diverse pieces of information from a wide-area network, summarize this information, update the summaries when there are changes to the actual information, and allow users to search or browse this information using an interface similar to that of Nebula. We shall not discuss the Synopsis file system any further since it is outside our scope.

Nebula has two advantages over MIT SFS. First, Nebula does not depend on the query language that used for content based access. Though Nebula has built-in functions to handle arbitrary boolean expressions involving relational operators like =, < and >, it has an extension language in which functions to handle more complex expressions can be written (see [6] for details). And second, it tries to go beyond a strict hierarchical classification of files by allowing users to chose the scope of a view to be a set of views rather than a single view. In MIT SFS, the set of symbolic links in a virtual directory is always a subset of the set of symbolic links in its parent.

However, Nebula still suffers from the following limitations: (i) Nebula does not allow users to modify the contents of a view without modifying its scope, its query, or the files in the file system, and (ii) Nebula is not integrated into the underlying file system, i.e., users must use views to organize results of queries but use actual directories to organize data. Hence, Nebula does not satisfy goals (2) and (3) of section 1.3.

2.7 Jade File System

The Jade file system [38] provides a uniform way to name and access files in an internet environment. Jade is a logical (virtual) file system that integrates a collection of existing file systems that provide different ways to name files, share them, cache their contents, maintain cache-consistency, etc. Because each existing file system is autonomous, Jade is built with the restriction that none of these file systems can be modified. Jade uses the concept of a *Uniform File System Interface* to provide transparent access to files in different file systems. Jade users build their personal virtual file systems by chosing the physical file systems they want to access, and gluing these systems together using *mount points*. All the information required to "mount" existing file systems onto a user's virtual file system is maintained by Jade — none of these file systems are aware that they have been mounted by a user. Jade also allows users to create files, directories, etc. within their personal name spaces in the usual way. Hence, a Jade file name, rather than being global (like a name in UNIX or AFS [41]), has a *scope* relative to the virtual name space of a particular user. That is, different users can name the same physical file in different ways.

Jade's virtual name space has two novel features: it allows multiple file systems to be mounted under one directory, and it permits one virtual name space to mount other virtual name spaces. To facilitate this, Jade has the concept of a *skeleton* directory. A skeleton directory defines a boundary between one Jade file system and other physical or Jade file systems. A Skeleton directory **d** within the virtual file system of a user **u** supports three main types of mounts:

- *Simple Mounts:* This allows a directory in a physical file system **F** to be mounted on **d** — **u** can access files in this directory without referring to **F**, i.e., *transparently* through **d**. This is similar to mount points in UNIX.
- *Logical Mounts:* This allows a directory in another user's virtual (Jade) file system to be mounted on **d** — **u** can access files in this directory transparently through **d**.
- *Multiple Mounts:* This allows directories in more than one physical or virtual file system to be mounted on **d**. The contents of **d** are the set of files and directories that are present in all the directories that are mounted. Figure 4.1 in section 4 shows a multiple mount point. Note, however, that multiple mounts can lead to name conflicts if two mounted directories have same name, say **/bin**. When more than one object has the same name in a multiple mount point, to avoid name conflicts, Jade choses one of these objects based on an internal order of priority, and discards the rest (see [38] for details and illustrative examples).

Jade also addresses other issues like caching, cache-consistency, path name resolution and access control that are relevant in an internet-wide file system. We will not go into them here since they are outside our scope. To conclude, Jade allows users to organize their personal information, and the information that exists in different autonomous physical and virtual (Jade) file systems, according to their individual tastes. Though Jade does not allow users to retrieve information by content, it has many features including per-user virtual name spaces, and logical and multiple mount points, that provide novel ways to retrieve information by name.

2.8 Prospero File System

The Prospero file system [32] is based on the Virtual System Model [33]. In this model, each user creates a virtual file system which is a directed graph built on top of a physical graph-structured information repository. In these graphs, *nodes* are used to store files and *links* are used to connect nodes with each other. That is, nodes are similar to UNIX-directories and links are similar to UNIX-symbolic links. In a user's virtual file system, virtual nodes can also contain pointers to the information present in one or more physical nodes. Users can specify (virtual or physical) path names that go from one node to another by traversing links. Users can also include other virtual name spaces in their personal (virtual) name spaces by using *union-links*. These are similar to UNIX-mount points. Hence, like Jade, Prospero allows users to build personal virtual name spaces that use the information present in the physical name space or other virtual name spaces.

However, Prospero has an additional feature that makes it very different from Jade. Prospero allows users to associate *filters* with links. A Filter is a program that can alter users' perception of the contents of the directory the link points to (this is called the *target* directory of that link). The input of the filter is the target directory and the files and links it contains, while the output is a set of links that point to new directories whose contents are derived from the contents of the target directory. This output is called a *view* of the target directory. Note that since a filter is an arbitrary program, it can access not only the files in the target directory, but also follow the links in this directory and access the files in other virtual and physical directories as well. Prospero also allows users to compose the filter associated with one link with the filter associated with another link. This allows users to specify the view of the directory pointed to by the first link as a function of the view of the directory pointed to by the second link. Users can execute filters whenever they wish and thereby derive views that classify information in existing directories according to their personal tastes. Hence, Prospero's filters are powerful tools for information retrieval.

The only drawback of Prospero is that filters *must* be written and executed by the user. Prospero does not ensure that the views of target directories are up-to-date when there are changes to (i) the contents of these directories, (ii) the filters associated with links to these directories, or (iii) the filters of other links that are composed with the filters mentioned in (ii). That is, Prospero does not offer consistency guarantees of any kind — users must execute the appropriate filters at the appropriate time to ensure consitency. Hence, Prospero meets all the goals mentioned in section 1.3 except goal (4).

In this chapter, we surveyed some existing systems that provide novel ways to access information. There are many others systems that have interesting ideas and discuss related issues [7, 10, 14, 16, 18, 24, 37, 42]. In general, systems that are very flexible and powerful like Prospero do not have a consistency model, and systems that are intuitive and simple like MIT SFS offer consistency guarantees but are not as powerful and do not allow users to organize the information retrieved by name and content using the same file system. Other systems like Harvest allow users to retrieve information using different naming schemes, but they do not allow them to organize this information in any meaningful way. Note that we did not discuss any database systems in this chapter since they concentrate on how to provide powerful paradigms to retrieve information by content — they do not discuss how to integrate these paradigms with mechanisms to retrieve information by name. We therefore conclude that there is a need for a new file system that provides both name and content based access to information, and satisfies all the goals mentioned in section 1.3.

CHAPTER 3
DESIGN OF THE HAC FILE SYSTEM

In this chapter, we examine various approaches to the design of a file system that provides CBA, and justify why we made certain choices in HAC. Towards the end of the chapter, we discuss a detailed example of an actual user-session that illustrates the important features of HAC. We shall also compare HAC with existing systems that provide name and content based access to information, and try to show that HAC offers a good alternative to these systems.

To simplify our discussion, we shall adopt the following convention in this dissertation: we shall use (i) bold upper-case **F** to denote a HAC file system, (ii) bold upper-case **Q** to denote a query, (iii) bold lower-case **f** to denote a file, (iv) bold lower-case **d** to denote a directory, **c** to denote a child (sub) directory and **p** to denote a parent directory, and (v) bold lower-case **l** to denote a (UNIX-like) symbolic link.

3.1 Basic Assumptions

To begin with, we mention the assumptions about hierarchical file systems and CBA mechanisms that HAC uses.

3.1.1 Hiearchical File Systems

HAC assumes a hierarchical file system with the following properties:

1. There is a notion of a *file*, a unit of data that can be identified uniquely and manipulated independently of other files in the file system.

2. There is a notion of a *directory*, a mechanism to group files and other (sub) directories together in a hierarchical fashion, so that files and directories can be identified using

path names that "traverse" directories one by one until they "reach" the appropriate file or directory. The files and sub-directories grouped within a directory are known as its *contents*. There is also a notion of a *root* directory from which all path name traversals must begin. (A traversal is a logical notion: we may or may not need to examine the contents each directory in a path name while we are traversing it. For example, see *Prefix-caching* in the Sprite file system [36].)

3.1.2 CBA Mechanisms

HAC also assumes a CBA mechanism with the following properties:

1. There is a *query-language* to express queries.
2. The result of a query is a set of *pointers* to objects that satisfy the corresponding query. (A "pointer" is an object that contains the identity or name of another object.)
3. Given a pointer to an object, it is possible to retrieve from that object the information that satisfies a given query.

Note that we do not assume that hierarchical file systems have the notion of a "pointer" to an object. (In the UNIX file system, pointers are nothing but *symbolic links*.) It is sufficient that such a notion is present in CBA mechanisms. We believe that these assumptions are reasonable since there are file systems without the notion of pointers to objects (e.g., MS-DOS file system), but we have not come across any search engine that does not allow us to retrieve the names of objects that satisfy a query (without returning the actual information itself).

3.2 One Name Space per User

HAC is designed so that users can extract and access the files they find relevant. That is, users are allowed to name the same files in different ways based on their individual tastes. Hence, in HAC, the *name space* within which path names are resolved is different for different users. This is similar to the Jade file system [38]. We believe that a single (global) name space for all users makes it harder to find relevant information (Chapter 14 of [31]). From this point onwards, we shall use the term "a HAC file system" or "an instance of the HAC file system" to refer to the personal name space of a single user. Note that HAC manages the file name space — it does not manage file data. Hence, we can think of HAC as a "layer" built on top of an "underlying file system" (which may or may not be hierarchical) that manages file data.

3.3 One CBA Mechanism per HAC File System

Each instance of the HAC file system has one content-based access (CBA) mechanism associated with it. HAC uses this CBA mechanism to process queries about all files in the corresponding file system. HAC also allows users to associate different CBA mechanisms with a given set of files. However, they must explicitly specify the set of files and the CBA mechanism they want to associate with it, and create a new name space (i.e., "file system") using the *semantic mount* operation. We shall discuss this in detail in Chapter 4. For now, it is sufficent to know that HAC uses the same CBA mechanism for all files in one file system. We believe that this design is simple because users can ask queries in one query language within one file system, and flexible because they can chose different CBA mechanism for different parts of their file system if they so desire. Hence, HAC satisfies Goal (5) mentioned in section 1.3.

3.4 UNIX-like Interface to CBA Mechanisms

We designed HAC's user and application programmer interfaces to CBA mechanisms using the UNIX file system as our model. This was because:

- UNIX was the most readily available hierarchical file system.
- We wanted to test if HAC could be used meaningfully when we mapped queries and query-results onto file system abstractions, and at the same time, allowed complex UNIX-like manipulations of the hierarchy (e.g., by using commands like: `mv`, `mkdir`, `creat`, `rm`, `rmdir`, `ln`, `mount`, etc.).

- Many hierarchical file systems that exist today are based on UNIX, and UNIX is in wide use: this meant that HAC could be extended easily to many of these file systems and its development could also be faster.

3.5 Syntactic and Semantic Commands

To integrate CBA mechanisms into file systems, our approach is to associate queries and their results with existing file system abstractions (Goal (1) in section 1.3), and provide new operations to manipulate queries and query-results. We call these new operations *semantic commands*. We designed semantic commands to be intuitive extensions of existing UNIX file system commands, which we call *syntactic commands*. In HAC, we can use both semantic and syntactic commands independently on the same file system abstractions.

We also define a lower-level application programmer interface for accessing and manipulating queries and their results. The interface consists of intuitive extensions to existing UNIX file system calls. We call them *semantic functions*. In this chapter, we shall introduce some important semantic commands as we go through HAC's design. We shall not discuss any semantic functions here since they can be derived easily from semantic commands. We give a detailed explanation of the semantic functions and commands in HAC in Appendices A and B.

3.6 Queries Associated with Directories

We know that CBA mechanisms can return either the information that matches a query, or the names of files that satisfy the query, or both (see section 1.1). We also know that some CBA mechanisms [11, 12] can group similar files together in *clusters* and return the set of clusters that satisfy the query. In other words, queries in CBA mechanisms allow us to group related objects together. Note that directories in hierarchical file systems also serve the same purpose. Hence, we associate queries with directories in the HAC file system. (This approach is similar to MITSFS [23] and Nebula [8].) We call the resulting directories *semantic directories*. The command to create a semantic directory is called `smkdir`. When a user creates a semantic directory, he/she must specify not only its name, but also a query to be associated with it. Note that a semantic directory is exactly like an ordinary directory in all other ways. That is, users can access a semantic directory by specifying its path name, and create, delete, and rename files and sub-directories, etc., within a semantic-directory in the usual way.

3.7 Query Results Associated with Sets of Symbolic Links

In this dissertation, we assume that when a CBA mechanism evaluates a query, the result is a set of pointers to files that satisfy the query. The UNIX file system has the notion of "symbolic links" which are pointers to (path names of) files, directories, or other symbolic links. We extend this notion in HAC and associate the result of a query with a *set* of symbolic links. In other words, when a user creates a semantic directory **d** with a query **Q**, HAC automatically creates symbolic links to files that match **Q** within the new directory **d**. (There is one symbolic link in **d** for each "pointer to a file" returned by the CBA mechanism.) To access the actual files that match **Q**, the user can simply traverse the appropriate symbolic links from **d** (i.e., de-reference the appropriate "pointers"). From now on, we shall use the term "symbolic link" to mean both a pointer to an object in a HAC file system, and a "pointer to a file" returned by a CBA mechanism. HAC supports symbolic links to files whether or not the underlying file system supports them.

Note that to map query-results onto a file system abstractions, there are two other alternatives. We discuss them below.

1. Instead of creating symbolic links, HAC can create UNIX-like *hard* links to files that match a query. (A hard link is an explicit reference to an object maintained by the file system. An object cannot be deleted from the file system unless all hard-links to it are explicitly removed.) We know from section 1.1 that it is possible for more than one query to match a given file. This means that users cannot delete a file unless they explicitly remove the hard links to that file from all semantic directories in the file system whose query-results refer to that file! This restricts the users' freedom unnecessarily. On the other hand, if we want HAC to delete the appropriate hard links automatically whenever a file is deleted, HAC must distinguish between the hard-links that are created explicitly by users and those that are created automatically by HAC when users ask queries, so that

it can delete a file if there are no user-created hard-links that point to that file. But this means that the links that HAC creates when users ask queries are not explicit references at all! They are just "pointers" to files, i.e., symbolic links. Another major problem with hard-links is that they cannot be maintained if they point to files in remote file systems over which we do not have any control (see Chapter 4 for more details).

2. Given a semantic directory **d**, HAC can create new files in **d** such that each new file **f'** corresponds to one existing file **f** that matches **d**'s query **Q**, and contains all the information in **f** that satisfies **Q**. That is, HAC does not have to bother about symbolic links, and users can consider query-results to be just regular files in the underlying file system. We rejected this idea since we wanted to clearly distinguish the data in the file system from the results of queries. (The former can be searched by a CBA mechanism, while the latter cannot.) This allows users to manipulate and tune the results of a query according to their personal tastes, *without* (inadvertently) modifying the results of *other* queries. They can do so by simply creating and deleting symbolic links in semantic directories. This is one of our main goals (see (3) and (4) in section 1.3, and sections 3.9 and 3.10 below). This is not possible if query-results are sets of files because queries can then match both the "original" files (like **f**), and their "subsets" (like **f'**). (It is *possible* to distinguish these two kinds of files, but that is extremely cumbersome and counter-intuitive.)

The idea of creating new files automatically in **d** is also not very practical since the query-result in **d** can contain too much information — users can get overwhelmed by it and even ignore most of it. HAC can also end up unnecessarily replicating huge amounts of data whenever users create new semantic directories, and this data can occupy too much space in the underlying file system. Sets of symbolic links, on the other hand, occupy very little space in the underlying file system (see Chapter 5 for more details). And sometimes, even the names of files that match a query (i.e., the symbolic links) can contain enough information to allow users to refine the query. That is why, instead of creating new files in **d** that contain information that matches **Q**, given a symbolic link **l** in **d**, we decided to provide a special user-command `scat` to retrieve only the information that matches **Q** from the file **l** points to. That is, HAC extracts the context of a match from a file only if users ask it to do so.

3.8 Sub-directories and Scope of Queries

We now discuss what we mean by creating a semantic sub-directory within an existing semantic directory. To do so, we must define the *scope* of a query: every query (and the corresponding semantic directory) has a *scope* which is the set of files over which the query is evaluated. That is, a query does not return symbolic links to files outside its scope even if those files match the query. The scope of a query depends on the parent of the corresponding semantic directory. If a semantic directory **d** is created within an existing semantic directory **p**, then the scope of **d** is defined to be the files pointed to by the existing set of symbolic links in **p**. This set of symbolic links is also known as the scope *provided* by **p**. However, if **d** is created within the root (/) of a HAC file system, then the scope of **d** is defined to be all the files in that file system.

By this definition, the scope provided by a newly created child semantic directory is always a *refinement* of the scope provided by its parent (this is similar to [23] and [6]). When a user creates a semantic subdirectory, the set of symbolic links in that directory is always a subset of the existing set of symbolic links in its parent. In other words, HAC treats the sets of symbolic links in different semantic directories as *separate* entities whose contents depend on how these directories are related to each other hierarchically. We chose this definition since we wanted to use semantic directories to organize both files and results of queries, and classify results of queries in a hierarchical fashion. The example towards the end of this chapter demonstrates that this is indeed desirable.

When a user creates a semantic subdirectory **d** within a parent directory **p**, we have three options for representing the query result associated with **d**:

1. We can explicitly copy the appropriate symbolic links from **p** into **d**.
2. We can move the symbolic links in **p** that satisfies **d**'s query from **p** into **d** and maintain additional information so that we can compute the query-result associated with **p** whenever necessary (say, if a user wants to create another semantic subdirectory within **p**). This information can be a list of names of sub-directories of **p** whose query-results are also a part of **p**.
3. Instead of moving symbolic links from **p** into **d**, we can maintain additional infor-

mation in **d** that tells us what part (range) of set of symbolic links in **p** also belong to **d**.

We chose the first approach since (i) it is easy to implement, and (ii) the sets of symbolic links in HAC occupy very little space compared to files (see Chapter 5). The other two approaches can been used if it is expensive to store sets of symbolic links. The only problem with our approach is that whenever a user creates a new semantic directory, HAC always creates a new set of symbolic links in it. That is, HAC may end up creating too many symbolic links and these might keep recurring in different subdirectories. Symbolic links can be a nuisance to users when they are browsing a directory hierarchy, since symbolic links are path names of objects in the file system, and path names can be very long if the file system is large. We get around this problem by providing a special command `sls` to display only those symbolic links in a directory that do not occur in any of its children. Using `sls`, users can hide the information they are not interested in and concentrate on what is relevant.

3.9 Editing Query Results and Scope of Queries

We mentioned earlier that users can treat a semantic directory exactly as an ordinary (syntactic) directory after they create it. This means that in particular, they can modify the set of symbolic links in a semantic directory by (i) removing some irrelevant links returned by the query, or (ii) creating new symbolic links to files that have related information but were missed by the query. This helps users tune the contents of a semantic directory according to their personal tastes.

For example, given a database of businesses in Tucson, users can create the directory `/restaurants/chinese` and decide to add a symbolic link `Min-Thai` inside it since they might feel that `Min-Thai`, a Thai restaurant, has a good selection of Chinese entreés. (Here, for simplicity, we assume that the query of a semantic directory is the same as its name.) Of course, users might also try creating `/restaurants/{chinese OR thai}`, but that will return symbolic links to all Thai restaurants, not just Min-Thai. As another example, the user might delete the symbolic link to `Seri-Melaka` in `/restaurants/chinese` since they mostly serve Malaysian food, not Chinese.

3.9.1 Scope Restrictions and Scope Consistency

Though it is desirable to allow users to edit the set of symbolic links in a semantic directory, we are now faced with an additional problem: it is possible for users to create a hierarchy of directories that violates the restriction that the scope provided by a semantic directory should be a subset of the scope provided by its parent. For example, users might explicitly add a link to `El-Regis Bar` within `/restaurants/mexican` since they serve excellent margaritas, but `El-Regis Bar` may not qualify as a "restaurant" according to the CBA mechanism. Similarly, they might explicitly delete the link to `Taco-Bell` within `/restaurants` since it is a fast-food joint and they want `/restaurants` to contain symbolic links to all restaurants where they can host formal dinners. However, they might still consider the symbolic link `Taco-Bell` to be very much relevant within `/restaurants/mexican`.

We resolve this issue by defining what we mean by a *scope-restriction* and describing an algorithm to enforce it. To begin with, we classify the set of symbolic links present in every semantic directory as follows:

1. The symbolic links that were explicitly added to the directory, i.e., its *permanent* symbolic links. (Symbolic links in existing hierarchical file systems are of this type.)
2. The symbolic links that were obtained by evaluating its query and were not explicitly added or deleted by users, i.e., its *transient* symbolic links. (All the freshly created symbolic links in a new semantic directory are transient.)

Among the symbolic links *not* present in the directory, we keep track of those links (whether transient or permanent) that were once present in the directory but were later explicitly deleted from it. We call them the *prohibited* symbolic links of the directory. Hence, given a semantic directory, we associate three disjoint sets of symbolic links with it: the transient, permanent and prohibited links. Note that given any symbolic link in the file system, we can determine if it is in one of these three sets. Now we define the following: given a semantic directory **d** that is not the root of a HAC file system, the "scope restriction" on the set of symbolic links in **d** is the invariant that

1. *The set of transient symbolic links in* **d** *is always a subset of the scope provided by*

its parent **p**, and

2. **d** *should have transient symbolic links to all the files in the scope provided by* **p** *that satisfy* **d***'s query, unless symbolic links to these files are explicitly prohibited in* **d**.

HAC must maintain this invariant for **d** whenever there is a change in its scope, since there can be transient symbolic links in **d** that point to files that are no longer in its new scope, and there can be files in **d**'s new scope that match its query but transient symbolic links to these files may not be present in **d**. In these cases, **d**'s query and its result are said to be inconsistent with its scope. We call this a *scope-inconsistency*. This can happen whenever there is a change in a query, the result of a query, or a change in the structure of the directory hierarchy. That is, when:

1. A user finishes editing the set of symbolic links in **d**'s parent **p**,
2. A user renames **d** as the child of a semantic directory other than **p**,
3. There is a change in the scope of **p**, or
4. A user changes the query of **d** after he/she creates it. (The user can do this using the special semantic command `smv -q`. He/she can also read the current query associated with a semantic directory using the semantic command `sreadln`.)

In these cases, HAC re-computes the set of transient symbolic links in **d** as follows: first, HAC uses the CBA mechanism to re-evaluate **d**'s query on all the files in its current scope. Then, from this result, HAC discards the links that occur in **d**'s set of permanent and prohibited symbolic links. The links that remain are the new transient symbolic links of **d**. Note that HAC does not add a prohibited symbolic link to the above result even if that link points to a file that is in **d**'s scope and matches its query. Similarly, HAC does not delete a permanent symbolic link from **d** even if that link points to a file that is no longer in **d**'s scope or does not match its query. Also note that HAC re-computes only the set of transient symbolic links of **d** — HAC does not change the set of permanent or prohibited symbolic links associated with **d** [1].

[1] HAC has special API routines `eunlink`, `elink`, `iunlink`, `link` and `symlink` to directly modify the set of permanent and prohibited symbolic links in semantic directories. Sophisticated users can use these routines to control the behaviour of the scope consistency algorithm. These are described in Appendix A.

To conclude: we allow users to edit and fine-tune the results of queries without modifying the query since we feel that the query of a semantic directory is not as important as the set of symbolic links in it: the query is just a quick first step to obtain more or less the information users are looking for. On the other hand, the set of symbolic links in a semantic directory may be the result of many (possibly time-consuming) browsing and editing steps. Hence, HAC does not modify this set unless it is explicitly asked to do so. Moreover, with this design, HAC is responsible only for the transient symbolic links in the file system, while users are responsible for all the permanent and prohibited symbolic links.

3.9.2 Advantages of the Scope Consistency Algorithm

The above definition of scope-consistency and the corresponding algorithm have some important consequences:

1. The above algorithm allows users to organize both files and results of queries in a hierarchy, and at the same time, takes care of the cases where strict hierarchical classification of information is not possible.

2. If users modify the query-result associated with a directory **d**, the scope consistency algorithm restricts all changes to the query-results of the directories that are within the subtree rooted at **d**. (These directories are said to *depend* on **d**.) Hence, users can modify the query-result associated with **d** without changing the query-result of its ancestors. This design is intuitive since even in hierarchical file systems, the changes within a directory do not effect the contents of its parent in any way. This is also simple to implement due to our decision in section 3.8 to create fresh symbolic links whenever a user creates a new semantic directory. If we had used one of the other techniques, we would have to use complex *copy-on-write* mechanisms to allow users to modify the query-result of a child semantic directory without effecting its parent.

3. The above definition of scope consistency and our decision to distinguish between syntactic and semantic commands gives users a lot of flexibility. For example, if they want to restrict the scope of a query **Q** to the set of files reachable from a given directory **d**, they can: (i) create a (syntactic) directory **p**, (ii) traverse the subtree rooted at **d** and

create permanent symbolic links in **p** to all the files encountered within **d**, and (iii) create a new semantic directory **p/c** with query **Q**. Then, **p/c** will contain symbolic links to all files reachable from **d** that satisfy **Q**. In this way, users can exploit both the existing organization of information, and the power of the CBA mechanism to classify information. Note that in this example, users can generate the set of symbolic links in **p** using any means they like without effecting the behaviour of `smkdir` or other semantic commands. This does not mean that semantic directories can have arbitrary sets of symbolic links in them, since HAC always enforces scope consistency. We believe that our design achieves a good balance between users' freedom on the one hand, and scope-consistency restrictions on the other.

3.9.3 Other Definitions of Scope

We define the set of transient symbolic links in a semantic directory **d** to be a refinement of the scope provided by its parent. We rejected the idea of defining this set to be, say, the union of the transient symbolic links in **d** and the scope provided by all its children. (In this case, **d** will depend on its children, not the other way round.) If we use this definition, users can never add a symbolic link **l** to a child of **d** such that **l** does not automatically belong to the scope provided by **d**. In other words, we cannot take care of the possibility that some information cannot be classified in a strict hierarchical fashion. This is unacceptable. We also rejected the idea of defining the set of transient symbolic links in **d** to be the union of the transient symbolic links in **d** and all its children, since in that case, changes to the set of transient links in a child semantic directory can effect the set of transient links in a parent. This is counter-intuitive since in hierarchical file systems, changes to the contents of a subdirectory do not effect the contents of its parent in any way. (Users of hierarchical file systems are in general more familiar with information propagating downward to the leaves of a directory hierarchy rather than upward to the root.)

3.9.4 Scope Provided by Directory Independent of its Children

If users remove a semantic directory **d**, the set of symbolic links in it is lost forever: however, the set of symbolic links in its parent **p** is not affected. Similarly, if users modify the set of symbolic links in **d** or change its parent (i.e., move **d** to a different position in the directory hierarchy), the scope provided by **p** is not affected.

Note that due to the above design decision and the decision in section 3.9.3, the scope provided by a semantic directory remains independent of the scope provided by its children. This allows users to modify and rename the children without changing their parent in any way. However, if they modify the set of symbolic links in a semantic directory, HAC updates all its children. This is intuitive since even in hierarchical file systems, changes within a directory are not visible to its parent, but for instance, if users rename (disable permissions to access) a directory, they implicitly rename (disable access to) all its descendents.

3.10 Modifying Files and Consistency of Query Results

HAC is a complete file system in its own right and allows all regular file system operations on its objects. Hence, users can not only modify results of queries, they can also create, remove, rename (move), or modify data in actual files in the file system. Then, there is a possibility that the transient symbolic links in a semantic directory **d** may not represent the true result of evaluating its query. That is, there can be a file **f** such that the transient symbolic link **l** to **f** is not present in **d**, and **l** is not prohibited in **d**, but **f** now matches **d**'s query. Similarly, there may be a file **f′** such that the transient symbolic link **l′** to it is present in **d**, but **f′** no longer exists or no longer matches **d**'s query. In these cases, **d**'s query and its result are said to be inconsistent with the data in the file system. We call this a *data-inconsistency.*

Though HAC removes *scope*-inconsistencies from the file system as soon as possible, when there are changes to the data (files) in the file system, HAC does not remove data inconsistencies instantly since this can be very expensive — HAC may have to invoke the CBA mechanism to re-index the whole file system and re-evaluate the queries of all semantic directories in it. (We assume that the CBA mechanism needs an index of the information to speed up its information retrieval [28]. We also assume that the CBA mechanism has a query-processor that answers queries about the files that were indexed.) Instead, HAC invokes the CBA mechanism to re-index the file system periodically (say, twice a day). Then, HAC traverses all semantic directories in the file system in a top-down fashion (from the root down to the leaves) and invokes the CBA mechanism to re-evaluate the query of each semantic directory it encounters (see section 3.11 for more details). That is, HAC makes sure that it updates the scope of every semantic directory before the CBA mechanism re-evaluates the corresponding query. This mechanism is simple and efficient, but data-inconsistencies in the file system can persist for many hours. That is why HAC also allows users to re-index specific parts of the file system and re-evaluate the queries of a specific set of semantic directories instantly. We are now trying to explore more sophisticated event-driven/lazy mechanisms to handle data-inconsistencies.

We now wish to mention a subtle point that can arise when we handle data inconsistencies. Suppose a user creates a file **f** within an existing semantic directory **d** with query **Q**. When the CBA mechanism re-evaluates **Q**, it can return a symbolic link **l** to **f** if it matches **Q**. Though there is no real need to "display" **l** when users are browsing the contents of **d** (using commands such as `ls` and `cd`, say), it is necessary to retain it in **d** since **l** will be in the scope of the semantic sub-directories of **d**, and **f** can match their queries. In general, we believe that there is no need to display symbolic links within a semantic direcotry that point to objects within the sub-tree rooted at this directory. We take care of this problem by modifying the semantic command `sls` (see section 3.8). When `sls` is used to browse the contents of **d**, it will not display symbolic links such as **l** that point to objects within **d**.

3.11 Using Existing Query Results in New Queries

So far we considered a query to be an attribute of a semantic directory: we did not care about its internal structure. However, if we assume that:

1. A query is an expression of some kind involving terms, where a term is defined to be the simplest form of a query (e.g., a keyword, an attribute, a value, etc.),

2. A term must be *evaluated* by the CBA mechanism, and the result of this evaluation is (also) a set of symbolic links to files that satisfy the term,

3. Sets of symbolic links (whether they are obtained by evaluating terms and queries, or from existing semantic directories) can be combined using operators (like AND, OR, $<$, $\geq$, etc.),

4. HAC knows how to use the query language parser of the CBA mechanism to split queries into its constituent terms and operators, and

5. HAC knows how to use the query language interpreter of the CBA mechanism to combine sets of symbolic links using operators,

Then, it is possible to use the name of an existing directory instead of a term in a query **Q**. Let us see how: whenever HAC wants to evaluate **Q**, it first invokes the query-language parser of the CBA mechanism to construct some form of intermediate code (say, a parse tree) for **Q**. We assume that it is possible to modify the parser to distinguish between terms and names of directories that appear in **Q**. (This can be a very simple modification, for instance, we can use back-quotes (')s to enclose names of directories in queries. See the example in section 3.12.) HAC then invokes the query language interpreter on this intermediate code. We assume that when the interpreter comes across a term, it can contact the query-processor of the CBA mechanism to evaluate it, and when it comes across the name of an existing directory **d**, it can call an external HAC-function to "evaluate" **d**. In this function, HAC directly accesses the existing set of symbolic links in **d** and returns that as the result of "evaluating" **d**. Note that HAC does not (recursively) invoke the interpreter to evaluate the query associated with **d** — that is similar to aliasing and not very interesting.

In the above discussion, it is possible for **Q** to be composed solely of directory names, i.e., **Q** need not have any terms at all! Moreover, **d** can be an ordinary (syntactic) directory without an associated query — HAC can still access its existing set of symbolic links to files, and combine it with search-expressions (terms) or edited query-results in semantic directories in the file system. (Note that HAC ignores symbolic links that point to files that were not indexed. It also ignores symbolic links that point to other file system abstractions.) The resulting "hybrid" query language can be more expressive than the query language of the CBA mechanism alone. For example, we implemented HAC using Glimpse [28] which has a simple query language but the result was sufficiently powerful for many purposes, including the example in section 3.12.

3.11.1 Consistency of Query Results Revisited

We now address the issue of consistency of queries and their results when queries are allowed to contain names of other semantic directories. Suppose the query of a directory **c** contains the name of a semantic directory **d** (that is not necessarily an ancestor of **c**). Now if there is a change in the scope provided by **d**, we must re-evaluate **c**'s query to reflect this change. In this case, we say that **c** *depends* on **d** or **c** *refers* to **d**. That is, the scope of **c** contains the scope provided by **d**. (Note that as we mentioned in section 3.9.1, the children of **d** always depend on **d** since the transient symbolic links in the children are always within the scope provided by **d**.) This set of dependencies gives rise to a directed graph which we call the dependency graph. (Note that if we did not allow queries to refer to names of other semantic directories, this graph would be an 'up' tree starting at the root of the file system [2].) We do not allow cycles to exist in this graph. The order in which we must update the set of symbolic links in different directories is given by a topological sort of the dependency graph. Note that the root of a HAC file system always occurs first in this order since all directories depend on it and the root does not depend on any other directory (the root does not have a query associated with it). HAC uses the top-sort order to re-evaluate queries of semantic directories whenever there are changes to the file system that can effect the scope provided by these directories. That is, whenever:

1. HAC wants to re-index the file system,
2. A user wants to re-index a specific set of files and/or re-evaluate a specific set of queries,
3. A user finishes editing the set of symbolic links in a semantic directory,
4. A user renames a directory in the naming hierarchy, or

[2] Each node in an up-tree has one link pointing to its parent, except the root which has no links.

5. A user changes the query of a semantic directory after he/she creates it using the `smv -q` command.

(Note that (1) and (2) handle data inconsistencies, and (3), (4) and (5) handle scope inconsistencies.) This shows that HAC has a consistency-model and satisfies goal (4) in section 1.3.

3.11.2 Uses of Path name Based Scoping

Before we end this section, there is one important point we must mention. Suppose a user creates a semantic directory **/c** whose query is a boolean AND of the path names of two other directories **/d** and **/p**. (For simplicity, we assume that all these directories are children of the root of a HAC file system.) Then, the set of symbolic links in **/c** will be identical to the set of symbolic links in the semantic directory **/p/c**, say, with the query **/d** (the path name of this directory). So, it is possible to argue that there is no need to define the scope of queries based on path names if it is possible to specify directory names in queries. However, we feel that query-refinement based on path names is important for two reasons. First, it is simple to understand and it can be used even if HAC does not know anything about the query-language parser. And second, the user might have created all the links in two directories **/p/c** and **/p** explicitly, and decided to "keep" **c** within **/p** (instead of **p** within **/c**, say) for reasons that may not be easy to express using a query. This also allows the user to derive a new directory hierarchy **p′** from the existing hierarchy **p** by using the names of directories in **p** in queries of semantic directories in **p′** — he/she does not have to modify the structure of **p** in any way. We believe that the structure of a directory hierarchy contains important organizational information, and that HAC should not consider two different structures that give rise to identical query results as equivalent.

3.12 Illustrative Example

We now illustrate the important features of HAC through an interactive user-command session [3] over a HAC file system containing summaries of about 1000 personal computer (PC) software programs. (This information was gathered from the World Wide Web.) We wish to mention that the reader can skip the entire example without any loss of continuity since we do not present any new information here. However, we believe that the example clarifies some important issues discussed earlier in this chapter, and shows that we can use HAC to access and organize information in a better way.

In this example, user-commands are shown in the `normal-typewriter` font, while HAC-output is shown in the `small-typewriter` font. The root of our HAC file system is in `/export/home /local/bgopal /phd/debug`. We shall use "..." as an abbreviation for this path name. In this file system, the summary-database is in the directory `.../pcindex` [4]. We use the Glimpse search engine [28] to provide content based access to files in HAC. Queries in Glimpse are boolean (AND, OR) combinations of keywords. A keyword is any set of alphanumeric characters. The AND operation is denoted by ';' and an OR operation by ','. Glimpse also handles regular expressions and wild-cards. To store the files HAC needs to interact with Glimpse, HAC automatically creates internal data structures that are accessible from the root of the file system.

```
[kno] pwd
```

```
/export/home/local/bgopal/phd/debug
```

```
[kno] ls -F
```

```
pcindex/
```

3.12.1 Queries and Query Results

The command to create a semantic directory is called `smkdir`. It has two arguments: a query and a name. If a user do not specify the name of the new semantic directory, `smkdir`

[3] We shall discuss the application-programmer interface in Appendix A.

[4] We did not copy the database into the directory `pcindex` but "mounted" it there (see Chapter 4).

assumes that it is the same as the query. `smkdir` automatically creates symbolic links to files that satisfy the specified query. Note that these symbolic links are different from those in UNIX in an important way. In UNIX, a symbolic link always has (i) a "name" that is given by a user who created the link, and (ii) a "content", which is the path name of the object the symbolic link points to. In HAC, however, a symbolic link that is created automatically in a new semantic directory **d** does not have any name [5]. It only has a content, which is the path name of the file the symbolic link points to, relative to the root of the HAC file system, `/export/home/local/bgopal/debug`. Hence, when a user does an `ls` operation on **d**, HAC directly displays the contents of all freshly created (i.e., unnamed) symbolic links to the user. (That is, `ls` on results of queries in HAC behaves like ls -l in UNIX).

This raises an important issue — since the contents of symbolic links (i.e., path names of files) can have "/"s in them, HAC must distinguish between path names that actually traverse the directory hierarchy rooted at **d** from path names that represent results of queries in **d**. To do so, HAC prepends a pair of ':'s to all path names that represent results of queries. That is, HAC defines the name of an automatically generated symbolic link to be two ':'s followed by its content. (These :s can be considered to be an abbreviation for the root of the HAC file system, i.e., equivalent to "...". This notation will become clearer in Chapter 4.) Note that instead of modifying the UNIX file-naming convention in this way, we could have:

1. Defined the name of an automatically generated symbolic link to be the same as its content except that every / is replaced with another character, say a #, or
2. Automatically generate a mnemonic name (that does not have a / in it) for an unnamed symbolic link.

We rejected the first solution since the character # itself can appear anywhere in the content of a symbolic link, and we could not find a clear-cut way to deal with this situation. Though our solution modifies the UNIX naming convention, we believe that it is intuitive since even in a hypertext-document, hyper-links (names of hypertext-documents)

[5] Users can give a mnemonic name to such a symbolic link if they wish.

are enclosed within the well known delimiters (< and > respectively), but the hyper-link itself is not modified in any way. We rejected the second solution because we could not find an easy way to generate mnemonic names automatically so that no two symbolic links in **d** have the same name. We plan to look for a better solution to this problem in future.

We now return to our example and show how to use the `smkdir` command.

```
[kno] smkdir network net-files
[kno] ls -F

net-files/      pcindex/

[kno] ls net-files

::pcindex/ftp.cdrom.com/network/README
::pcindex/ftp.uml.edu/apogee/1rott13.zip
::pcindex/ftp.uml.edu/3drealms/tvdwango.zip
::pcindex/ftp.uml.edu/epic/tyrdemo.zip

[kno] ls -F net-files

::pcindex/ftp.cdrom.com/network/README@
::pcindex/ftp.uml.edu/apogee/1rott13.zip@
::pcindex/ftp.uml.edu/3drealms/tvdwango.zip@
::pcindex/ftp.uml.edu/epic/tyrdemo.zip@

[kno] rmdir net-files

[kno] smkdir children
[kno] ls children

::pcindex/ftp.uml.edu/MVP/gb30.zip
::pcindex/ftp.uml.edu/apogee/1math.zip
::pcindex/ftp.uml.edu/apogee/1rescue.zip
::pcindex/ftp.uml.edu/epic/amath.zip
::pcindex/ftp.uml.edu/epic/tddemo.zip
::pcindex/info.rutgers.edu/games/children/README
::pcindex/info.rutgers.edu/programming/children/README
```

```
[kno] cat children/::pcindex/ftp.uml.edu/MVP/gb30.zip

File: gb30.zip          Size: 410763    14-Mar-95

        GameBuilder Lite v3. Create graphics adventure games with no
        programming.  Your games can include 256 color graphics, main
        character and background animation, masking, object inventory and
        detection, music and sound effects, and more, with a powerful point
        'n click interface just like the pros use!  Easy enough for
        children; sophisticated enough for savvy gamers. Supports CGA, EGA,
        VGA; mouse recommended.

[kno] scat children/::pcindex/ftp.uml.edu/MVP/gb30.zip

        children; sophisticated enough for savvy gamers. Supports CGA, EGA,
```

Now, we might realize that the existing program `/home/bgopal/pcgames/hangman` is suitable for children, and therefore might like to move it into the new semantic directory `.../children`.

```
[kno] mv /home/bgopal/pcgames/hangman children
[kno] cat > children/README
This directory has some programs suitable for growing children.
[kno] ls -F children

::pcindex/ftp.uml.edu/MVP/gb30.zip@
::pcindex/ftp.uml.edu/apogee/1math.zip@
::pcindex/ftp.uml.edu/apogee/1rescue.zip@
::pcindex/ftp.uml.edu/epic/amath.zip@
::pcindex/ftp.uml.edu/epic/tddemo.zip@
::pcindex/info.rutgers.edu/games/children/README@
::pcindex/info.rutgers.edu/programming/children/README@
README
hangman*
```

From these examples, it is clear that semantic directories allow us to access information by both name and content.

3.12.2 Sub-directories and Scope of Queries

Now, we show how to create subdirectories with `smkdir`. The set of symbolic links in a new semantic directory is always a subset of the set of symbolic links in its parent.

```
[kno] ls -F

children/      pcindex/

[kno] smkdir VGA
[kno] ls VGA

::pcindex/ftp.uml.edu/MVP/1arcy.zip
::pcindex/ftp.uml.edu/MVP/1cf2.zip
::pcindex/ftp.uml.edu/MVP/1cru116.zip
::pcindex/ftp.uml.edu/MVP/1derby.zip
::pcindex/ftp.uml.edu/MVP/1flash.zip
... and so on ...

[kno] rmdir VGA
[kno] cd children
[kno] smkdir VGA
[kno] ls VGA

::pcindex/ftp.uml.edu/MVP/gb30.zip
::pcindex/ftp.uml.edu/epic/amath.zip
::pcindex/ftp.uml.edu/epic/tddemo.zip

[kno] smkdir math
[kno] ls math

::pcindex/ftp.uml.edu/apogee/1math.zip
::pcindex/ftp.uml.edu/epic/amath.zip
```

Figure 3.1 shows the semantic directory hierarchy corresponding to the above examples. Since `.../pcindex/ftp.uml.edu/epic/amath.zip` matches the queries "math" and "VGA" the symbolic link to it occurs in both `.../children/math` and `.../children/VGA`. The following example illustrates how we can use the `sls` command to hide irrelevant symbolic links.

```
[kno] cd /export/home/local/bgopal/phd/debug/children
[kno] ls .

::pcindex/ftp.uml.edu/MVP/gb30.zip
::pcindex/ftp.uml.edu/apogee/1math.zip
::pcindex/ftp.uml.edu/apogee/1rescue.zip
::pcindex/ftp.uml.edu/epic/amath.zip
::pcindex/ftp.uml.edu/epic/tddemo.zip
::pcindex/info.rutgers.edu/games/children/README
::pcindex/info.rutgers.edu/programming/children/README
README
VGA
hangman
math

[kno] sls .

hangman
README
VGA
math
::pcindex/ftp.uml.edu/apogee/1rescue.zip
::pcindex/info.rutgers.edu/games/children/README
::pcindex/info.rutgers.edu/programming/children/README

[kno] ls VGA

::pcindex/ftp.uml.edu/MVP/gb30.zip
::pcindex/ftp.uml.edu/epic/amath.zip
::pcindex/ftp.uml.edu/epic/tddemo.zip
```

```
[kno] ls math

::pcindex/ftp.uml.edu/apogee/1math.zip
::pcindex/ftp.uml.edu/epic/amath.zip
```

3.12.3 Editing Query Results and Scope of Queries

In this subsection, we show how we can edit the query-results associated with semantic directories. For example, we can delete the symbolic link to `children/::/ pcindex/ ftp.uml.edu/ MVP/gb30.zip` since we are not interested in superficial adventure games. We can also add a symbolic link to `::pcindex/ ftp.uml.edu/ epic/heart.zip` in `.../children` since though Glimpse does not find it (it does not have the word "children" in it), we realize that it is clearly a good game for children. (We can continue to edit the set of symbolic links in `.../children` until we are satisfied.)

```
[kno] cd /export/home/local/bgopal/phd/debug
[kno] cat children/::pcindex/ftp.uml.edu/MVP/gb30.zip

File: gb30.zip          Size: 410763    14-Mar-95

        GameBuilder Lite v3. Create graphics adventure games with no
        programming.  Your games can include 256 color graphics, main
        character and background animation, masking, object inventory and
        detection, music and sound effects, and more, with a powerful point
        'n click interface just like the pros use!  Easy enough for
        children; sophisticated enough for savvy gamers. Supports CGA, EGA,
        VGA; mouse recommended.

[kno] rm children/::pcindex/ftp.uml.edu/MVP/gb30.zip
[kno] ls children

::pcindex/ftp.uml.edu/apogee/1math.zip
::pcindex/ftp.uml.edu/apogee/1rescue.zip
::pcindex/ftp.uml.edu/epic/amath.zip
::pcindex/ftp.uml.edu/epic/tddemo.zip
```

```
::pcindex/info.rutgers.edu/games/children/README
::pcindex/info.rutgers.edu/programming/children/README
README
VGA
hangman
math
```

```
[kno] cat ::pcindex/ftp.uml.edu/epic/heart.zip
```

```
File: heart.zip        Size: 563774

        Heartlight, part of Epic's Puzzle Pack. Guide and elf through a
        realm of puzzles. Requires 386+. Supports SB/Adlib.
```

```
[kno] ln ::pcindex/ftp.uml.edu/epic/heart.zip children
[kno] ls children
```

```
::pcindex/ftp.uml.edu/apogee/1math.zip
::pcindex/ftp.uml.edu/apogee/1rescue.zip
::pcindex/ftp.uml.edu/epic/amath.zip
::pcindex/ftp.uml.edu/epic/heart.zip
::pcindex/ftp.uml.edu/epic/tddemo.zip
::pcindex/info.rutgers.edu/games/children/README
::pcindex/info.rutgers.edu/programming/children/README
README
VGA
hangman
math
```

Now, suppose we come across the file `.../pcindex/ftp.uml.edu/epic/brix.zip` while browsing the PC-database.

```
cat pcindex/ftp.uml.edu/epic/brix.zip
```

```
File: brix.zip         Size: 308577

        Brix, am arcade style game involving moving various blocks. Requires
        VGA and 286 or better. Supports SoundBlaster.
```

Then, we can add the symbolic link to `.../pcindex/ftp.uml.edu/epic/brix.zip` since it involves geometrical skills, which are related to math skills. However, `brix` may not be suitable for small children, i.e., the symbolic link to `.../pcindex/ftp.uml.edu/epic/brix.zip` in `.../children/math` can be outside the scope of the parent directory `.../children`.

```
[kno] ls children/math

::pcindex/ftp.uml.edu/apogee/1math.zip
::pcindex/ftp.uml.edu/epic/amath.zip

[kno] ln ::pcindex/ftp.uml.edu/epic/brix.zip children/math
[kno] ls children/math

::pcindex/ftp.uml.edu/apogee/1math.zip
::pcindex/ftp.uml.edu/epic/brix.zip
::pcindex/ftp.uml.edu/epic/amath.zip

[kno] ls children

::pcindex/ftp.uml.edu/apogee/1math.zip
::pcindex/ftp.uml.edu/apogee/1rescue.zip
::pcindex/ftp.uml.edu/epic/amath.zip
::pcindex/ftp.uml.edu/epic/heart.zip
::pcindex/ftp.uml.edu/epic/tddemo.zip
::pcindex/info.rutgers.edu/games/children/README
::pcindex/info.rutgers.edu/programming/children/README
README
VGA
hangman
math
```

At this point, we can discover that the link `.../pcindex/ftp.uml.edu/epic/tddemo.zip` is not meant for young children, so we delete it from `.../children`. The command to update (synchronize) the sets of symbolic links in the sub-directories of a semantic directory **d** once we finish editing the set of symbolic links in **d** is called `ssync -q`. If we use the `ssync`

`-q` command on `.../children`, the link to `.../pcindex/ftp.uml.edu/epic/tddemo.zip` disappears from `.../children/VGA`. Also, if we create a new subdirectory `freetime` that has the query "game OR play" in `children`, `freetime` will not have the symbolic link to `.../pcindex/ftp.uml.edu /epic/tddemo.zip`.

```
[kno] smkdir '{game,play}' children/freetime
[kno] ls children/freetime

::pcindex/ftp.uml.edu/MVP/gb30.zip
::pcindex/ftp.uml.edu/apogee/1math.zip
::pcindex/ftp.uml.edu/apogee/1rescue.zip
::pcindex/ftp.uml.edu/epic/amath.zip
::pcindex/ftp.uml.edu/epic/tddemo.zip
::pcindex/info.rutgers.edu/games/children/README

[kno] rmdir freetime
[kno] cat ::pcindex/ftp.uml.edu/epic/tddemo.zip

File: tddemo.zip        Size: 293826

        Traffic Department 2192 Demo. An intense journey through a new world
        of corruption and deceit. Not for young children.  Great graphics
        and plenty of playing enjoyment. Requires VGA, 286+. Supports
        SB/SBPro.

[kno] rm children/::pcindex/ftp.uml.edu/epic/tddemo.zip
[kno] ssync -q children
[kno] ls children

::pcindex/ftp.uml.edu/apogee/1math.zip
::pcindex/ftp.uml.edu/apogee/1rescue.zip
::pcindex/ftp.uml.edu/epic/amath.zip
::pcindex/ftp.uml.edu/epic/heart.zip
::pcindex/info.rutgers.edu/games/children/README
::pcindex/info.rutgers.edu/programming/children/README
```

```
README
VGA
hangman
math

[kno] ls children/VGA

::pcindex/ftp.uml.edu/MVP/gb30.zip
::pcindex/ftp.uml.edu/epic/amath.zip

[kno] smkdir '{game,play}' children/freetime
[kno] ls children/freetime

::pcindex/ftp.uml.edu/MVP/gb30.zip
::pcindex/ftp.uml.edu/apogee/1math.zip
::pcindex/ftp.uml.edu/apogee/1rescue.zip
::pcindex/ftp.uml.edu/epic/amath.zip
::pcindex/info.rutgers.edu/games/children/README
```

We can see the symbolic links which were explicitly added (**-i** option) or deleted (**-e** option) in a semantic directory using the `sls` command.

```
[kno] sls -e children

::pcindex/ftp.uml.edu/epic/tddemo.zip

[kno] sls -i children/math

::pcindex/ftp.uml.edu/epic/brix.zip
```

3.12.4 Modifying Files and Consistency of Query Results

In an earlier example in section 3.12.1, new files `README` and `hangman` were created in the directory `.../children`. In the following example, we re-index the files in the file system using the semantic command `smount` (see Chapter 4 and Appendix B for more details), and re-evaluate `.../children`'s query using `ssync -q`). When we check the new set of files in `.../children` using `ls`, we see that the link `::children/README` to `.../children/README` appears in `.../children`. This tells us that `.../children/README` will be within the scope of all directories in the file system that depend on `.../children`. However, if we are only interested in the information present in the subtree rooted at `.../children`, then this link is actually redundant since it points to a file within this subtree. By using the `sls` command, we can make sure that such links are not displayed.

```
[kno] cd /export/home/local/bgopal/phd/debug
[kno] ls -F

children/       pcindex/


[kno] ls -F children

::pcindex/ftp.uml.edu/apogee/1math.zip@
::pcindex/ftp.uml.edu/apogee/1rescue.zip@
::pcindex/ftp.uml.edu/epic/amath.zip@
::pcindex/ftp.uml.edu/epic/heart.zip@
::pcindex/info.rutgers.edu/games/children/README@
::pcindex/info.rutgers.edu/programming/children/README@
README
VGA/
freetime/
hangman*
math/


[kno] smount -u -n -f `pwd`/ `pwd` `pwd`
... output of CBA mechanism as it indexes files ...
[kno] ssync -q children
[kno] ls children

::children/README
::pcindex/ftp.uml.edu/apogee/1math.zip
```

```
::pcindex/ftp.uml.edu/apogee/1rescue.zip
::pcindex/ftp.uml.edu/epic/amath.zip
::pcindex/ftp.uml.edu/epic/heart.zip
::pcindex/info.rutgers.edu/games/children/README
::pcindex/info.rutgers.edu/programming/children/README
README
VGA
freetime
hangman
math

[kno] scat ::/children/README

This directory has some programs suitable for growing children.

[kno] sls .

hangman
README
VGA
math
freetime
::pcindex/ftp.uml.edu/epic/heart.zip
::pcindex/info.rutgers.edu/programming/children/README
```

3.12.5 Using Existing Query Results in New Queries

In the following example, we create the semantic directories `space` and `wargame`, and then create the semantic directory `startrek` whose query contains the names of directories `space` and `wargame` [6]. We also create the semantic directory `qstartrek` whose query is a combination of the queries of `space` and `wargame` to show that the set of symbolic links in `qstartrek` and `startrek` are not identical.

[6]The file `::pcindex/ftp.uml.edu/MVP/1flash.zip` satisfied our query since it has the words "Soft**ware**" and "das**tar**dly", which have nothing to do with "war" and "star"! This is one of the reasons why we believe that the power to edit the result of a query is very important.

```
[kno] pwd

/export/home/local/bgopal/phd/debug

[kno] smkdir '{star,space,planet,galaxy}' space
[kno] ls space

::pcindex/ftp.uml.edu/3drealms/tvshow.zip
::pcindex/ftp.uml.edu/MVP/1cru116.zip
::pcindex/ftp.uml.edu/MVP/1flash.zip
::pcindex/ftp.uml.edu/MVP/3point.zip
::pcindex/ftp.uml.edu/MVP/pickle.zip
... and so on ...

[kno] cat space/::pcindex/info.rutgers.edu/mswindows/README

 msdos/mswindows/aplt11.arc                                    88202 bytes
 msdos/mswindows/applet11.zip                                  80941 bytes
 msdos/mswindows/apps.arc                                       7164 bytes
 msdos/mswindows/balloon.arc                                    9591 bytes
 msdos/mswindows/blank2.arc                                    15280 bytes
... and so on ...

[kno] rm space/::pcindex/info.rutgers.edu/mswindows/README
[kno] smkdir '{war,fight,battle};game' wargame
[kno] ls wargame

::pcindex/ftp.uml.edu/MVP/1cf2.zip
::pcindex/ftp.uml.edu/MVP/1cru116.zip
::pcindex/ftp.uml.edu/MVP/1flash.zip
::pcindex/ftp.uml.edu/MVP/1rapid.zip
::pcindex/ftp.uml.edu/MVP/1sea2.zip
... and so on ...

[kno] ls
```

```
children        space
pcindex         wargame

[kno] smkdir '{`wargame`;`space`}' startrek
[kno] ls startrek

::pcindex/ftp.uml.edu/MVP/1cru116.zip
::pcindex/ftp.uml.edu/MVP/1flash.zip
::pcindex/ftp.uml.edu/MVP/pickle.zip

[kno] cat startrek/::pcindex/ftp.uml.edu/MVP/1flash.zip

File: 1flash.zip        Size: 578710    25-Jul-95

        Jack Flash by MVP Software. Jump into the action with Jack and his
        Succ-O-Matic as you try to stop Evil Eddie and his dastardly
        "things" from destroying the universe. Parallax-scrolling,
        side-splitting humor. More zest than a Microsoft press release. More
        action than a Stallone flick, and cuter too! Relive the good old
        days when mad scientists threatened mankind and games were fun! Req
        386, VGA; supports most sound cards.

[kno] smkdir '{war,fight,battle};game;\
    {star,space,planet,galaxy}' qstartrek
[kno] ls qstartrek

::pcindex/ftp.uml.edu/MVP/1cru116.zip
::pcindex/ftp.uml.edu/MVP/1flash.zip
::pcindex/ftp.uml.edu/MVP/pickle.zip
::pcindex/info.rutgers.edu/mswindows/games/README
```

The following example illustrates that HAC re-evaluates the queries of semantic directories whenever we rename a directory using `mv` or change its query using `smv -q`. We can use the `sreadln` command to read the query associated with a semantic directory.

```
[kno] ls children/math
```

```
::pcindex/ftp.uml.edu/apogee/1math.zip
::pcindex/ftp.uml.edu/epic/brix.zip
::pcindex/ftp.uml.edu/epic/amath.zip

[kno] mv children/math ./math
[kno] ls math

::pcindex/ftp.cdrom.com/math/README
::pcindex/ftp.uml.edu/apogee/1math.zip
::pcindex/ftp.uml.edu/epic/amath.zip
::pcindex/ftp.uml.edu/epic/brix.zip
::pcindex/info.rutgers.edu/math/README
... and so on ...

[kno] sreadln math
}}
\vspace{-.175in}
\begin{verbatim}
math

[kno] smv -q mathematics math
[kno] ls math

::pcindex/ftp.uml.edu/epic/brix.zip
```

This concludes our example. We believe that it shows how we can tackle important practical problems using HAC's user-command interface. We will refer to the PC software database again in Chapter 4.

3.13 Discussion: Does HAC Satisfy Our Goals?

So far, we discussed the basic design of the HAC file system with the help of an illustrative example. We now show that HAC satisfies all the goals we mentioned in section 1.3.

In our implementation of HAC, though we use one particular search engine (Glimpse) to provide content based access, HAC itself is independent of it. This is because the only interaction between HAC and Glimpse is through queries, and queries are evaluated by Glimpse, not HAC. Though HAC allows us to use the name of a directory in a query, HAC does not restrict the query language in any way. The query language parser and interpreter are provided as external inputs to HAC. Hence, it is possible to integrate any CBA mechanism into HAC, i.e., it satisfies goal (5).

In the above example we saw that HAC allows us to retrieve summaries of interesting programs by both name and content, and organize these summaries according to our taste. This can help us choose the PC software we want intelligently. (Note that in order to do this, we do not have to make a copy of the summary database. The `pcindex` directory is a *mount point* — see Chapter 4 that describes how we can access other CBA mechanisms and file systems from one HAC file system using mount points. For example, we can access other summary-databases, access other people's organizations of a given summary database, or access databases that have related information, e.g., the database of software vendors or PC-repair shops in the vicinity.) HAC also allows us to use the above organization of summaries to classify the actual PC-software itself, i.e., it allows us to store the actual software (`hangman`) "somewhere near" the summaries of related software (`.../children`). In other words, we can use the same file system to organize both data and results of queries, and use regular file system operations to manipulate them. Hence HAC satisfies goals (1), (2) and (3).

HAC enforces data consistency whenever there are changes to the files in the file system. This feature, for example, allows us to get to know: (i) if new versions of existing programs are available (i.e., summary-files in the database are modified), (ii) if any new software is available (new summary-files are added to the database), and (iii) if existing software will no-longer being supported or sold (summary-files are deleted from the database). HAC also enforces scope consistency whenever there are changes to queries or their results. Hence HAC satisfies goal (4). To conclude, it meets all the goals we had in mind when we started to design a file system that provides both name and content based access to files.

3.14 Comparison with Other Systems

Now, we would like to compare some important features of HAC with similar features in other systems that provide both name and content based access together.

3.14.1 Browsing and Searching in GlimpseHTTP, WebGlimpse, HAC

WWW browsers like **Netscape** offer facilities to search documents pointed to by URLs. These search facilities are ad-hoc since the scope of the search, i.e., the set of URLs over which the search is conducted, is always global. In other words, there is no facility for users to restrict the scope of their search to a subset of the global collection of URLs. GlimpseHTTP goes one step further: it assumes that the set of URLs at a WWW site are classified in a UNIX directory hierarchy by the information provider, and allows users to restrict the scope of their search to all the URLs reachable from the UNIX directory that corresponds to the current URL they are browsing. Note that if users search from the root of this directory hierarchy, the scope of the search is all the URLs in that WWW site.

WebGlimpse does not assume that the set of URLs are classified in a UNIX directory hierarchy. Instead, it interprets each file in the WWW site as a hypertext document. For each document, WebGlimpse builds a neighbourhood file that contains the names of all the hypertext documents that are related to this document in some way. Users can restrict the scope of their search to the neighbourhood of the current document they are browsing. WebGlimpse allows the information provider to modify the neighbourhood file of each document so that he/she can restrict the scope of the search according to his/her personal taste.

Both GlimpseHTTP and WebGlimpse have a drawback: they do not allow users to classify the existing information according to their choice. They can only use the existing organization of information to restrict the scope of their search. These packages also do not allow users to organize and re-use the results of queries in any meaningful way. However, semantic directories in HAC allow users to retrieve information by both name and content, and allow them to organize this information to suit their individual tastes.

It is important to note that HAC does *not* interpret files like WebGlimpse does. That is, WebGlimpse's idea of building neighbourhoods for files based on their content can be used even when the files are stored in the HAC file system. We believe that these systems can complement each other in a very useful way.

3.14.2 Scatter/Gather Browsing and HAC

The Scatter/Gather technique [11, 12] applies clustering algorithms to approximately summarize any given set of documents by grouping related documents together. This provides a good overall picture of the kind of information the documents contain. The only drawback of this system is that users have no control over the clustering process — the system makes all important decisions. In particular, it decides what clusters to create and what documents to assign to each cluster. However, summaries generated by the Scatter/Gather technique can be extremely useful when used in conjunction with HAC since HAC does not have facilities to classify files automatically based on their content. HAC users can use the Scatter/Gather technique as the first step to decide which queries to ask, and then create semantic directories of their choice using these queries. Thereafter, users can use HAC's query-result manipulation power to tune the contents of these directories according to their preferences. They can also use the Scatter/Gather technique *recursively* on their semantic directories to discover good ways to classify the information the directories contain. That is, they can use HAC to "gather" the documents they are interested in, and use the Scatter/Gather system to discover good ways to "scatter" these documents into related groups. In other words, though Scatter/Gather Browsing and the HAC file system attempt to sove different problems, we believe that their combination can result in a very powerful information retrieval system.

3.14.3 Queries and Path Names in MITSFS and HAC

The query of a semantic directory in HAC is independent of its path name [7]. Only the query-result of a semantic directory depends on its parent, since the transient symbolic links in the directory are always within the scope provided by its parent. This is intuitive

[7] However, the query can refer to other paths in the file system.

since the most important use of directories in hierarchical file systems is to classify files. Semantic directories in HAC are an extension of the same idea: they allow users to classify both files and symbolic links. Moreover, this design allows them to name semantic directories in any way they choose. In contrast, MIT SFS defines the query of a child to be "the query of its parent" **AND** "the name of the child", i.e., it interprets the / path name separator as the boolean **AND** operator and does not distinguish between the name and the query of a directory. It also does not allow users to edit the result of the child's query. Though this design ensures that the scope provided by a child directory is always a refinement of that of its parent (i.e., it is trivial to enforce scope-consistency — it is a no-op), the query language supported by MIT SFS is very restrictive. Moreover, users have no freedom to modify the names of directories after creating them. (In particular, they cannot move directories to different points in the directory hierarchy.) HAC does not have any of these limitations.

3.14.4 Multistructured Naming and HAC

Multistructured naming tries to blend hierarchical and content-based naming of files. In HAC parlance, if we do not allow users to modify the results of a queries, then the query-result stored in a directory **d** with query $\mathbf{Q}_d$ will always be a subset of the query-result stored in its parent **/p** with query $\mathbf{Q}_p$. In this case, the query-result in the semantic directory **/p/d** would be the same as the query-result in some other semantic directory with a query "$\mathbf{Q}_p$ **AND** $\mathbf{Q}_d$", or "$\mathbf{Q}_d$ **AND** $\mathbf{Q}_p$". Hence, there is no point in forcing users to specify path names in the ancestor-descendent order, i.e., we can allow users to specify both **/p/d** and **/d/p** to locate the same set of files by content if they wish. With multistructured naming, users can do exactly what we described above by specifying rules that define relationship between queries (see section 2.5). These rules allow users to: (i) permute some components of a path name, omit some components or add extra components to a path name, (ii) assign implicit values to query-results, and (iii) specify which queries must be satisfied simultaneously, so that they can build a hierarchical structure into an otherwise flat content-based naming scheme. This can help them search and browse the information in the file system.

Though multistructured naming allows users to retrieve and organize files by both name and content, it does not allow them to modify the results of queries. Hence, they cannot tune these results according to their personal tastes. This is one of HAC's main goals. Continuing with the above example, if there are two other directories **/d** and **/d/p** with queries $\mathbf{Q}_d$ and $\mathbf{Q}_p$ respectively, since we allow users to modify results of queries, the set of symbolic links in **/d/p** and **/p/d** (above) may be different. Hence, we sacrifice the ability to specify complex structural constraints on the components of path names, but gain the power to customize query results and use modified results in future queries. However, if users want both **/p/d** and **/d/p** to refer to the same directory, say **/d/p**, they can store the path names of all the directories in the file system (separated by newlines) in a special file, invoke a search engine like Glimpse to search and output all the lines in that file that match the pattern **p;d** (the ";" is a *commutative AND* operation in Glimpse), and pass on the result of this search to HAC. The above procedure works equally well on any hierarchical file system, and we did not feel it was necessary to include it as a part of HAC's design.

Also note that multistructured naming does not come for "free" — users must specify rules that describe relationships between the aliases (labels) of queries (attributes), and this can be a complex task. On the other hand, HAC users just have to organize their semantic directories in a hierarchical fashion, which is trivial if they are familiar with hierarchical file systems. For the above two reasons, we believe that HAC offers a good alternative to multistructured naming.

3.14.5 Views in Nebula and Semantic Directories in HAC

Nebula provides content based access to files in an existing (underlying) syntactic file system by creating *views* of the file system. A view in Nebula has a query associated with it. A view consists of a set of pointers to files (i.e., *file-objects* that encapsulate files) in the underlying file system that match the query. These are called its *contents*. The query of a view **v** cannot refer to names of other views, but users can choose the views within which the query is evaluated. These views form the *scope* of **v**. In this case, **v** is said to depend on all the views in its scope. A user can refine the contents of a view by refining the query associated with the view or by changing its scope. When the contents of a view change due to changes in the files in the underlying file system, Nebula makes sure that the queries of all views that depend on this view are re-evaluated. This mechanism is similar to what happens in HAC. Note however that the contents of a view **v** change only if the files in the file system or views that **v** depends on change. Nebula does not allow users to edit the contents of a view, i.e., they cannot change the result of any query to suit their personal tastes. Also note that unlike HAC, Nebula's views are built on top of an existing file system — they are not integrated into the underlying file system. That is, views help users organize the information they retrieve by content, but not the information present in the underlying file system. For these two reasons, we believe that HAC compares favorably with Nebula.

3.14.6 User-defined Filters in Prospero and Queries in HAC

The Prospero file system also allows users to associate *filters* with links that point to directories. These are called the target directories. A Filter is an arbitrary program (not a query in a fixed query language) that can alter the users' perception of the contents of a target directory. The input of a filter is a target directory and the output is a set of links that point to new directories whose contents are derived from the target directory. The output is called a *view* of the target directory. Prospero allows users to compose the filter associated with a link that points to a directory **d** with the filter associated with a link pointing to another directory $\mathbf{d}'$. This allows users to specify a view of **d** as a function of a view of $\mathbf{d}'$. In this case, **d** is said to depend on $\mathbf{d}'$. (Note that these dependencies can give rise to an arbitrary graph.) Users can execute filters whenever they wish and thereby derive views that classify information in existing directories according to their tastes.

Note, however, that Prospero does not guarantee that the view of a directory **d** remains consistent when there are changes to the contents of **d**, the filters associated links pointing to **d**, or the filters associated with the directories **d** depends on. Users have to manually execute the appropriate filters at the appropriate time, or write their own programs to do so. Prospero also does not have the notion of a CBA mechanism: users have to write programs that interact with different CBA mechanisms on their own and use them as filters when necessary.

HAC differs from Prospero in many ways. First, HAC associates a query with each semantic directory, not with symbolic links or hard links that point to that directory. Though both forms are equivalent (we can assume that a link in Prospero is a semantic directory, and the filter associated with the link is the query of this directory; we can also assume that the contents of the target directory form the scope of the above query, and so on), our design allows users to treat a semantic directory as a unit whose contents do not depend on who refers to it. We feel that this is intuitive since directories in hierarchical file systems are also are interpreted in the same way. Second, queries of semantic directories in a HAC file system are like filters, but their power is restricted to the query-language of the CBA mechanism associated with the HAC file system. However, unlike Prospero, HAC offers consistency guarantees when there are changes to the information in the file system (see above) and allows users to use arbitrary directory names in queries. Finally, unlike Prospero, HAC has a well defined user-command and application-programmer interfaces to CBA mechanisms. Due to these reasons, we believe that HAC offers a good alternative to Prospero.

In this chapter, we discussed the principles behind the design of the HAC file system, and showed that it is conceptually simple, and at the same time, achieves all our goals. We also showed that HAC has an interface that is intuitive and easy-to-use. In the next chapter, we shall discuss how to access different file systems and CBA mechanisms from within one HAC file system.

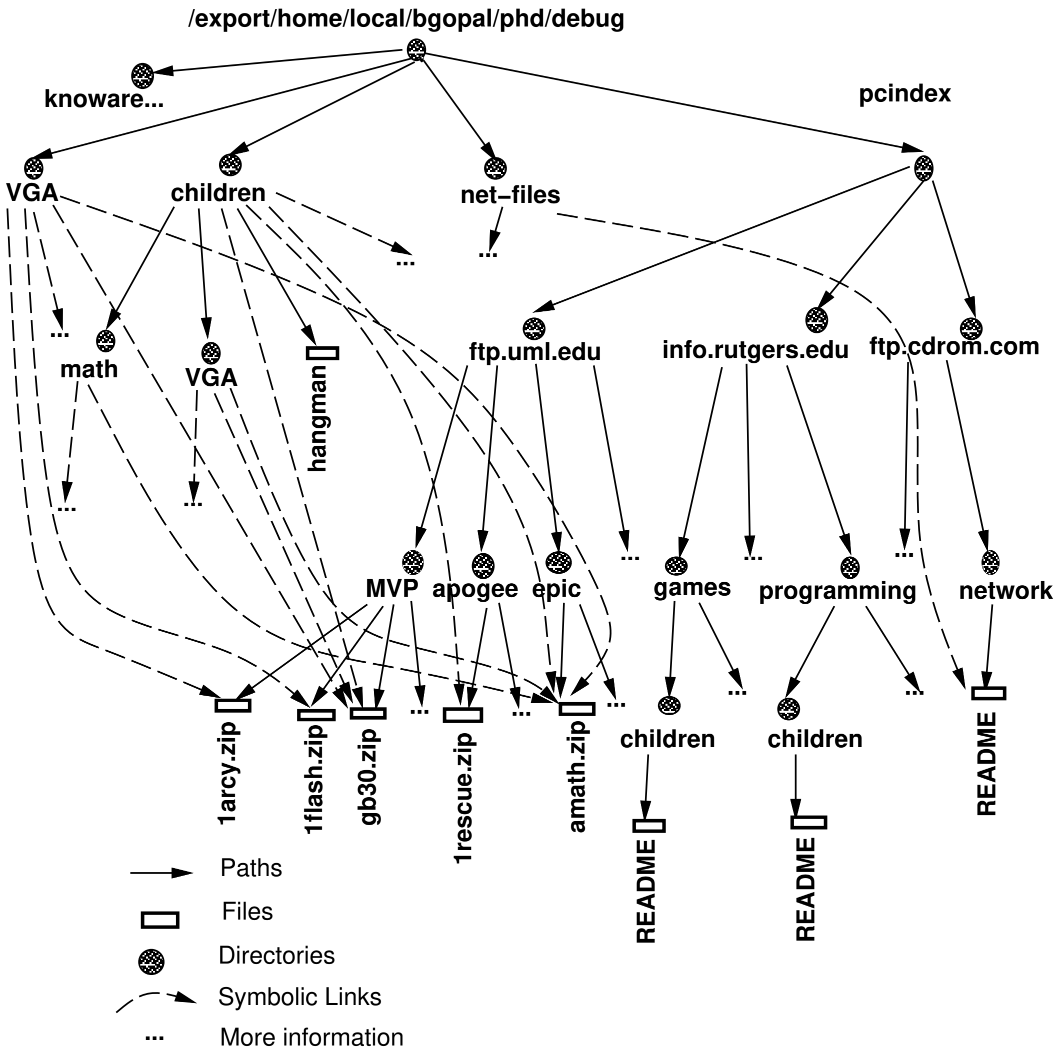


Figure 3.1: Files, Symbolic Links and Directory Hierarchies

CHAPTER 4

ACCESSING NAME-SPACES AND CBA-MECHANISMS

So far, we discussed how one content-based access (CBA) mechanism can be integrated into a HAC file system. In this chapter, we discuss how users on one HAC file system can access other HAC file systems and CBA mechanisms. To motivate this discussion, we first explain how users on one file system can access objects in other file systems using *mount points*.

Mount points exist in many file systems including UNIX, Prospero and Jade. A mount point is a directory that is interpreted in a special way by the file system in which it exists. A mount point provides an interface for this file system (the *source*) to access objects in other file systems (the *targets*). A mount point also allows the file-naming mechanism in the source file system to scale since the source file system knows exactly what target name-spaces it must access to resolve path names that "cross" the mount point. The act of creating a mount point in the source file system is called *mounting* the target file systems *on* the source file system *at* the mount point. To create a mount point, the source file system must know how to identify (name) the target file systems, i.e., there must be a well-understood convention for naming file systems.

Figure 4.1 shows a mount point in UNIX and a *multiple* mount point in Jade. Details about different kinds of mount points can be found in [38]. We shall not discuss them here.

We call mount points in existing file systems *syntactic mount points* since they define the name-spaces within which path names must be resolved. In HAC, we introduce a new kind of mount point, called a *semantic mount point*, that defines the name-spaces of CBA mechanisms within which queries must be evaluated. We now describe how semantic and syntactic mount points in HAC allow users to share their information with each other. In this chapter, we shall use bold lower-case **m** to denote a mount-point. We shall also use the term "an instance of a CBA mechanism" to refer to a CBA mechanism that physically exists and has a query-processor to answer queries about the files that were indexed.

4.1 Identifying HAC File Systems and CBA Mechanisms

We assume that every instance of a HAC file system and a CBA mechanism is unique and has an identifier associated with it. As we mentioned above, identifiers are necessary if we want one HAC file system to be able to access other HAC file systems and CBA mechanisms. In our implementation, we use a common naming scheme for these identifiers. In the naming scheme, each identifier has three components:

1. The *host-name* or *network-address* of the machine where the file system or CBA mechanism exists,
2. The *port-number* that should be used to communicate with the file system or CBA mechanism, and
3. The *type-identifier* that describes the kind of file system (MS-DOS, UNIX, SUN NFS [40], AFS [41], etc.) or CBA mechanism (Glimpse [28], WAIS [26], etc.) it is.

Note that the above naming scheme is specific to HAC. It may or may not allow us to identify other types of file systems (e.g., Jade, SUN NFS, etc.). However, a HAC file system can always identify and access objects (files, directories, etc.) in its underlying file system, by specifying the path names of these objects. We believe that this simple naming scheme is sufficient to illustrate the main ideas in this chapter.

Also note that in this dissertation, we do not discuss how to maintain the database that maps HAC file systems and CBA mechanisms to their identifiers, since it is outside our scope. However, we do wish to mention that any distributed Naming Service (e.g., Grapevine [31]) can be used for this purpose. In our implementation, users can specify these identifiers via the UNIX environment, or as options to various semantic commands. (See Appendices A and B for details.)

4.2 Accessing one File System from Another

To access a HAC or underlying file system $\mathbf{F}'$ from another HAC or underlying file system $\mathbf{F}$, a user must mount $\mathbf{F}'$ at a syntactic mount point $\mathbf{m}$ in $\mathbf{F}$. All file system operations within $\mathbf{m}$ are redirected to $\mathbf{F}'$. Once a user creates $\mathbf{m}$, he/she can access resources on $\mathbf{F}'$ transparently by using path names that begin at the root of $\mathbf{F}$ (not the root of $\mathbf{F}'$). If $\mathbf{F}'$ is a HAC file system, users on $\mathbf{F}$ can access the CBA mechanism of $\mathbf{F}'$. Figure 4.2 shows a syntactic mount point $\mathbf{m}$ in a file system $\mathbf{F}$: $\mathbf{m}$ points to the root of the file system $\mathbf{F}'$. This is similar to syntactic mount points in UNIX (see Figure 4.1). Note that in HAC, all mount information is maintained in the source file system $\mathbf{F}$: the target file system $\mathbf{F}'$ does not have to be modified in any way. This is true for all mount points in HAC.

We now explain in brief how we resolve a path name that begins in a HAC file system $\mathbf{F}$ and crosses $\mathbf{m}$ into another HAC file system $\mathbf{F}'$. When this happens, $\mathbf{F}$ passes on the unresolved portion of the path name to $\mathbf{F}'$. $\mathbf{F}'$ interprets this path name and returns the result, which can either be a *handle* to a file (or directory, etc.) in $\mathbf{F}'$, or yet another unresolved path name along with the identity of the HAC file system $\mathbf{F}''$ which can resolve it. If the result is a handle, $\mathbf{F}$ considers the path to be resolved and contacts $\mathbf{F}'$ for all file system operations on the file (or directory, etc.) that corresponds to the handle. On the other hand, if the result is another unresolved path, $\mathbf{F}$ continues to contact different HAC file systems until the path is completely resolved. That is, HAC uses *remote iterative resolution* [38] to handle path names that cross syntactic mount points. We plan to use *local* iterative resolution and *directory-entry caching* [38] in future to make path name lookups in remote file systems more efficient. Figure 4.3 shows how we can link up different HAC and underlying file systems with each other using syntactic mount points.

In the PC-software example (see section 3.12), we mounted a HAC file system at `/export/home/local/bgopal/phd/debug` (abbreviated "...") by creating a syntactic mount point in the underlying UNIX file system at this directory. (See section 5.1.1 for more details.) We also originally created `.../pcindex` using HAC's `smount` command. The **-y** option of this command tells HAC that this `.../pcindex` is a syntactic mount point, and the **-h** and **-p** options tell HAC about the host-name "`kno`" and port-number "`5423`" that identify the the target HAC file system. (To learn more about these options, please refer to Appendix B.) The following shows how we created `.../pcindex`:

```
[kno] smount -y -h kno -p 5423 \
    /export/home/local/bgopal/phd/debug/pcindex
[kno] ls -F /export/home/local/bgopal/phd/debug

pcindex/
```

Once a user mounts the target HAC file system at `.../pcindex`, HAC treats it as an ordinary directory. That is, HAC uses the local CBA mechanism (not the CBA mechanism of the mounted file system) to provide CBA to files within `.../pcindex`. After HAC invokes the local CBA mechanism to re-index the file system at "`...`", if a user creates a semantic directory within "`...`", HAC will return symbolic links to files in `.../pcindex` (we illustrated this in the previous chapter). However, if the user enters (`cd`-s into) `pcindex`, he/she is no longer in the local file system, but in the mounted file system. Hence, if he/she creates a semantic directory within `pcindex`, HAC will re-direct the `smkdir` operation to the the mounted file system. This file system will access its CBA mechanism to search its files (which are exactly the files visible from `pcindex` in "`...`)", and create the new directory in its name-space. The name-space of the local file system at "`...`" will not be effected in anyway. Therefore, syntactic mount points in HAC make it possible for a user to use different CBA mechanisms to retrieve the same information by content.

However, to access a CBA mechanism via a syntactic mount point, a user *must* have permissions to modify the name-space of the mounted file system, since he/she must be able to create a semantic directory there. This is not desirable since each HAC file system corresponds to the name-space of a single user, and this user may not want others to modify his/her name-space (though he/she may want others to access some of his/her files). Another undesirable property of this scheme is that a user cannot access an existing CBA mechanism (that provides CBA to some files on SUN NFS, or some hypertext documents, say), unless the CBA mechanism is associated with an existing HAC file system. The main reason for these problems is that there is no way we can "decouple" query-processing from path name resolution if we use only syntactic mount points. That is why, we introduce semantic mount points in HAC. We discuss them in detail below.

4.3 Accessing Different CBA Mechanisms and File Systems

A semantic mount point **m** in a HAC file system **F** allows users to automatically change the scope of all queries asked within **m** from the set of files that exist within **F** to the set of files in the name-spaces mounted on **m**. Queries asked within **m** can return symbolic links to the files mounted on **m**, but queries asked outside **m** cannot. We must distinguish the files *mounted* on **m** from the files (and directories, etc.) *created* within **m** — the latter are always reachable from the root of **F**, i.e., are within the scope provided by the root of **F**, but the former need not. The same CBA mechanism that is used to retrieve files in **F** is also used to retrieve files created in **m**. However, it is possible to use a different CBA mechanism to retrieve files mounted on **m**. Figure 4.4 makes these notions clear.

4.3.1 Local and Remote Semantic Mount Points

Semantic mount points come in two flavors, *local* and *remote*. A local semantic mount point **m** in a HAC file system **F** is one which allows a user on **F** to mount file systems of his/her choice on **m**, and associate a CBA mechanism of their choice to index and search the files in these file systems. A remote mount point $\mathbf{m}'$ in **F**, on the other hand, allows a user on **F** to mount an instance of a CBA mechanism on $\mathbf{m}'$. It does not allow the user to specify what files should be accessed by that CBA mechanism or when those files should be indexed. When a user creates a local semantic mount point **m**, he/she specifies the CBA mechanism that should be used for files mounted on **m**, and a unique identifier (host-name or network address, and port-number) for the instance of that CBA mechanism that will process queries about these files. When the user creates a remote semantic mount point $\mathbf{m}'$, he/she just specifies the identifier for the instance of the CBA mechanism that should be mounted on $\mathbf{m}'$. Local semantic mount points allow a user to associate a set of files with a CBA mechanism and "export" this tuple as a new "name-space". Due to this definition, the root of every HAC file system is also a local semantic mount point, since every HAC file system has a CBA mechanism associated with it that answers queries about the files in that file system. Note that a syntactic mount point allows a user to "import" a whole name-space, while a remote semantic mount point allows him/her to import only the CBA mechanism associated with a name-space. Hence, a remote semantic mount point allows only content based access to information — for path name based access to that inormation, the user must create an appropriate syntactic mount point. Also note that a syntactic mount point allows a user to browse through an existing classification of information (i.e., semantic directory hierarchy) in a name-space, while a remote semantic mount point allows him/her to create a personal classification of this information, without modifying the name-space itself in any way. These ideas are summarized in Table 4.1:

Type of Mount	searching	browsing	Own mounted information
Syntactic (target = HAC)	YES	YES	NO
Syntactic (target ≠ HAC)	NO	YES	NO
Local Semantic	YES	YES	YES
Remote Semantic	YES	NO	NO

Table 4.1: Types of Mount Points

4.3.2 Multiple Semantic Mount Points

At semantic mount points such as **m**, users have the freedom to mount as many file systems as they want. The scope of queries asked within **m** will include all the files in the file systems that are mounted on **m**. It is possible to use different CBA mechanisms to process queries about different files in file systems mounted on **m**. In this case, **m** is called a multiple semantic mount point. In fact, it is possible to mount both CBA mechanisms and HAC file systems on **m**, since the results of queries asked within both remote and local semantic mount points are always sets of symbolic links. It is important to note that the CBA mechanisms that are associated with the file systems mounted on **m**, and the CBA mechanisms directly mounted on **m**, must have the same query language. This is because each semantic directory in HAC has one query associated with it. It is possible to implement "translators" that convert queries from one language to another, but we do not discuss them here.

When more than one CBA mechanism is used to process queries within **m**, i.e., **m** is a multiple mount point, semantic directories within **m** will contain two types of symbolic links: (i) those that point to to files in the file systems mounted on **m**, and (ii) those that are returned by the CBA mechanisms mounted on **m**. Users can directly traverse the symbolic links of type (i) and access the files they point to, but they must create an appropriate syntactic mount point to be able to traverse the symbolic links of type (ii). We believe that this interface is adequate since it is simple to create different kinds of mount points using the `smount` command (see the example in section 4.3.3 below).

Note that if another mount point $\mathbf{m}'$ exists within **m**, then the scope of queries asked within $\mathbf{m}'$ does not include the files mounted on **m**, unless of course these files are also mounted on $\mathbf{m}'$. (This is shown in Figure 4.5.) Even if the same file systems are mounted on both **m** and $\mathbf{m}'$, the query result of a semantic directory **m/d** cannot be influenced by the query result of any semantic directory in $\mathbf{m}'$ (and vice versa). For example, the query of **m/d** query cannot contain names of directories that are not within **m**, or directories whose path names cross other mount points (like $\mathbf{m}'$) within **m**. We chose this design since mount points in HAC define new name-spaces where queries can be evaluated, and we wanted these name-spaces to be independent of each other. For example, changes to the semantic directory hierarchy within one name space does not affect the other name spaces in any way. This restriction, of course, does not apply to the *data* (i.e., files) accessed through these name spaces, since the same set of files can be mounted on more than one mount point in HAC. This is similar to existing hierarchical file systems where each syntactic mount point changes the set of files that are accessible from the source file system independently. We also do not allow a user to move a directory across a mount point since this involves changing the file system in which the query-result of that directory make sense. The user has to create a new directory within that mount point, explicitly copy the contents of the source directory into the new directory, and then delete the source directory. This is exactly what the user must do even in existing hiearchical file systems like UNIX. (Note that we do allow directories to be moved anywhere within a mount point, and we discussed this in detail in section 3.12.1.)

Figure 4.6 shows how we can mount multiple file systems at the same mount point. One particular case of multiple mounts is when the file system **F** in which the mount point **m** exists is *also* mounted on **m**. In this case, the same query (**F:/m/d** in the figure) can be used to access files in both **F** and $\mathbf{F}'$ by content.

Figure 4.7 shows how we can combine semantic and syntactic mount points. This allows us to browse remote file systems, and use both the local and remote CBA mechanisms to query remote files. Note that if we have already mounted the file system (or CBA mechanism) $\mathbf{F}'$ on $\mathbf{m}$, and created a hieararchy $\mathbf{m/d}$ of semantic directories within $\mathbf{m}$, and if we now mount a new file system (or CBA mechanism) $\mathbf{F}''$ on $\mathbf{m}$, HAC will automatically contact $\mathbf{F}''$ to re-evaluate the queries of all the semantic directories within $\mathbf{m/d}$. HAC will then create the appropriate symbolic links to files in $\mathbf{F}''$ in $\mathbf{m/d}$ without modifying the existing set of symbolic links in $\mathbf{m/d}$. That is, HAC considers the results of queries evaluated within different file systems (or CBA mechanisms) to be independent of each other. (This is shown in the example in section 4.3.3.) For the same reson, if we unmount $\mathbf{F}''$ from $\mathbf{m}$, HAC will automatically remove symbolic links to files in $\mathbf{F}''$ from directories in $\mathbf{m/d}$, without effecting the other symbolic links in these directories.

Now, it is possible to argue that there is no need to create a multiple mount point such as $\mathbf{m}$ if HAC considers the results of queries evaluated within $\mathbf{F}'$ and $\mathbf{F}''$ to be independent of each other: creating $\mathbf{m}$ is conceptually no different from creating two different mount points $\mathbf{m}'$ and $\mathbf{m}''$ which mount $\mathbf{F}'$ and $\mathbf{F}''$ respectively. However, we believe that multiple mount points are important because they allow us to treat an existing semantic directory hierarchy $\mathbf{m/d}$ within a multiple mount point $\mathbf{m}$ (that mounts $\mathbf{F}'$) as a *filter* that automatically classifies information in a new file system (such as $\mathbf{F}''$) when it is mounted on $\mathbf{m}$. A filtering operation automatically groups related information in two different file systems together, and makes it easier to access this information. For example, multiple semantic mount points can make it easier to find files that are similar to each other, if these files exist in different file systems.

4.3.3 Illustrative Example

The following is an example of how a remote semantic mount point (`garbo.uwasa.fi`) changes the scope of queries to the files in the name space mounted on it. These files are the contents of the underlying (UNIX) directory at `$target`, i.e., `/export/home/ local/bgopal/ phd/experiments/ pcindex.remote/ garbo.uwasa.fi`. The CBA mechanism that is used to process queries about these files runs on the machine `kno` at the port number `5555`. When HAC changes the scope of queries to this CBA mechanism, and a user creates a new semantic directory, say, `garbo.uwasa.fi/astronomy`, HAC prepends the host-name `kno` and port-number 5555 of the CBA mechanism (separated by ':'s) to the contents of each automatically generated (i.e., "unnamed") symbolic link in `astronomy`, and generates a unique name for that link. As explained in section 3.12.1, the content of an unnamed symbolic link is the path name of the file the symbolic link points to, relative to the root of the file system in which that file exists (the root is `$target` in this case). In section 3.12, for simplicity, we omitted the host-name and port-number in the names of all "unnamed" symbolic links. In the example below, note that the semantic directory `.../garbo.uwasa.fi/ astronomy` is within the HAC file system at "...", but the symbolic links with the prefix `kno:5555` are not. These links, however, can be accessed as usual using commands like `cat`, `scat`, etc.

```
[kno] set myroot = '/export/home/local/bgopal/phd/experiments'
[kno] set target = \
    "$myroot"'/pcindex.remote/garbo.uwasa.fi'
[kno] smount -q -p 5555 -h kno -f $target/ garbo.uwasa.fi
[kno] cd garbo.uwasa.fi
[kno] smkdir astronomy
[kno] ls astronomy

kno:5555:pc/INDEX
kno:5555:pc/astronomy/00index.txt
kno:5555:pc/astronomy/README
kno:5555:pc/astronomy/starwrk2.zip
kno:5555:pc/gifunid/README
kno:5555:windows/WINDEX
kno:5555:windows/astronomy/00index.txt
kno:5555:windows/astronomy/README

[kno] scat kno:5555:pc/astronomy/starwrk2.zip

starwrk2.zip    SVGA/Herc astronomy planetarium simulator, G.Lee
```

As another example, suppose an instance of a CBA mechanism exists on the machine named smallalo at port number `2001`, and suppose we mount it on the existing mount point `garbo.uwasa.fi`. This would make `garbo.uwasa.fi` a multiple semantic mount point. Then, after an `ssync`, the semantic directory `astronomy` within `garbo.uwasa.fi` will also have symbolic links to files indexed by the CBA mechanism at `alo:2001` that satisfy the corresponding query [1].

```
[kno] set myrroot = '/usr1/bgopal/phd/experiments'
[kno] set rtarget = \
    "$myrroot"'/pcindex.remote/oak.oakland.edu'
[kno] smount -q -p 2001 -h alo -f $rtarget/ garbo.uwasa.fi
[kno] cd garbo.uwasa.fi
[kno] ssync -q astronomy
[kno] ls astronomy
```

```
alo:2001:SimTel/msdos/astrnomy/README
alo:2001:SimTel/msdos/astrnomy/brungsc1.zip
... and so on ...
kno:5555:pc/INDEX
kno:5555:windows/astronomy/00index.txt
... and so on ...
```

4.4 Index Caching

Most CBA mechanisms need an "index" of the information to speed up information retrieval. The index can be a special set of files that contain "hints" which tell us where we can find a given piece of information [28, 26]. In the Glimpse CBA mechanism, the size of the index is a small fraction of the size of the data that is indexed. (In our illustrative example, the index is just 8% of the actual data.) Glimpse can also answer some queries by accessing only the index of the information. For example, Glimpse can return the names of files that satisfy case-insensitive queries that contain boolean combinations of keywords (not phrases), using only the index. (See [27] for details.) However, the set of files that satisfy the Glimpse phrase-query 'word1 word2' is always a *subset* of the set of files that satisfy the Glimpse boolean-query 'word1ANDword2'. Hence, the boolean-AND-query

[1] The spelling error `astrnomy` exists in the original data collected from `oak.oakland.edu`.

'word1ANDword2' is an *approximation* to the phrase-query 'word1 word2' in Glimpse. In other words, Glimpse can give exact answers to a some queries, and approximate answers to others, by using only a small index of the information. In this section, we will generalize these ideas and introduce the concept of index-caching.

Suppose **m** is a semantic mount point in **F** that changes the scope of queries to the files in another file system $\mathbf{F}'$. Also suppose that:

1. It is possible for the CBA mechanism associated with $\mathbf{F}'$ to provide "approximate" answers to all (or a restricted subset) of queries by using only the index of the files in $\mathbf{F}'$. This assumption is reasonable not only in Glimpse, but also in WAIS [26] and Nebula [8].

2. The user wants to get a rough idea of the amount and type of information that a query will return so that he/she can refine the query and narrow down the result to a manageable level before looking for more accurate results. From our experience with GlimpseHTTP [27] and Harvest [3] systems, we found that users tend to ignore very large query-results, especially if the database itself is huge (e.g., a digital library of books on Science or Engineering, or a medical records of patients in a large hospital, etc.). Users also get a fair idea of the information contained in files by just looking at their path names, sizes, modification dates, and other attributes. Hence, we believe that this assumption is also reasonable.

3. The CBA access mechanism can provide the user a rough idea of the information that his/her query will return by using only a *small* index of the information. This assumption is valid for the Glimpse search engine.

Then, there is no need for the CBA mechanism of $\mathbf{F}'$ to access the actual files in $\mathbf{F}'$ to answer queries. Not only that, if **F** and $\mathbf{F}'$ have the same CBA mechanism (and hence the same query language), then **F** can borrow and cache the index of the files in $\mathbf{F}'$, store it locally, and use this index to provide the same answers as above — i.e., the CBA mechanism of **F** does not need to access the files in $\mathbf{F}'$ either. This method of borrowing an index and using it to answer queries approximately is called *index-caching.* Index caching is useful when the size of the index is a small fraction of the size of the files that were indexed [28], and **F** and $\mathbf{F}'$ are separated by a network that has large round-trip communication delays which make it inefficient for **F** to contact $\mathbf{F}'$ to evaluate all queries. In this case, a user on $\mathbf{F}$ can use the local copy of $\mathbf{F}'$'s index to narrow down his/her queries, and then send the queries he/she is really interested in over to $\mathbf{F}'$. That is, index-caching allows a user to trade accuracy for scalability and performance.

In HAC, a user can specify index-caching as an option when he/she creates a remote semantic mount point in $\mathbf{F}$ to mount a CBA mechanism $\mathbf{F}'$. (This is the `-r` option in the `smount` command.) Index caching is obviously not useful if the user has already created a local semantic mount point for the file system associated with $\mathbf{F}'$, since there is no need to use just the index of the files in this file system to answer queries approximately, when it is possible to access these files directly and answer queries accurately. It is also not useful if the index is a significant fraction of the actual information — in that case, $\mathbf{F}$ can simply copy the files in $\mathbf{F}'$ and access them locally. But this may not be feasible if the amount of data in these files is huge. This "solution" is certainly not scalable as the amount data in different file systems increases. However, $\mathbf{F}$ can *summarize* the information in $\mathbf{F}'$ and copy the summaries into $\mathbf{F}$. The technique is not new. It has been adopted in wide-area information retrieval systems like Essence [24], Indie [16, 18], Harvest [3] and the Synopsis file system [8]. Some of these systems also have mechanisms to keep the summary of the information consistent with the actual information. These systems allow users to browse the summary ($\mathbf{F}$) and filter-out the information they are really interested in, and then access the actual repository ($\mathbf{F}'$) to retrieve this information. This not only reduces the processing load on the actual repository, but also the communication load on the network. This idea is very similar to index-caching, since both summaries and indices give us hints about the actual information — indices allow us to search information more quickly, whereas summaries allow us to browse it more quickly. We shall not discuss the above technique any more since it is beyond our scope.

4.5 Consistency in the Presence of Mount Points

A syntactic mount point $\mathbf{m}$ in a HAC file system $\mathbf{F}$ gives users on $\mathbf{F}$ a way to access the contents of the mounted file system $\mathbf{F}'$. $\mathbf{F}$ redirects all operations on objects within $\mathbf{m}$ to $\mathbf{F}'$. Hence $\mathbf{F}'$ enforces scope and data consistency of directories within $\mathbf{m}$, not $\mathbf{F}$. That is, in this section, we only need to discuss how to enforce scope and data consistency in the presence of semantic mount points in **F**.

We mentioned earlier in section 4.3.2 that the query-results within a directory in a semantic mount point cannot influcence the query-results of directories within other semantic mount points. Hence, it is sufficient to focus on consistency in the presence of one semantic mount point **m** in **F**.

We also mentioned in section 4.3.2 that given a semantic directory **m/d** in a multiple semantic mount point **m**, HAC operates on the sets of symbolic links in **m/d** corresponding to each mounted file system and CBA mechanism of **m** independently. Hence, the algorithms to enforce scope consistency remain the same as before, except that they enforce the consistency of each set of symbolic links independently.

However, we must modify the algorithms to enforce data consistency as follows: whenever **F** reindexes its files, it also re-indexes the files in file systems mounted locally on **m**. However, **F** cannot force the CBA mechanisms that are mounted remotely on **m** to re-index their files. This is because **F** can abuse this power and bring down their performance by repeatedly asking them to re-index their files. Hence, when **F** mounts CBA mechanisms remotely, **F** has no control on whether their indices are up to date and accurate.

Once **F** re-indexes the appropriate files, it can re-evaluate the queries of semantic directories that exist within **m** in the usual way.

In this chapter, we described how syntactic mount points allow users to link up their HAC file systems with other file systems, and thereby access the corresponding CBA mechanisms. We also described how semantic mount points allow users to change the scope of their queries to files outside their file systems, and thereby link up their file systems with other CBA mechanisms. We then showed how users can combine syntactic and semantic mount points, and share the information they retrieve by name and content with each other. Towards the end of the chapter, we mentioned how HAC maintains consistency in the presence of mount points. This concludes our discussion of HAC's design. We shall discuss its implementation and performance next.

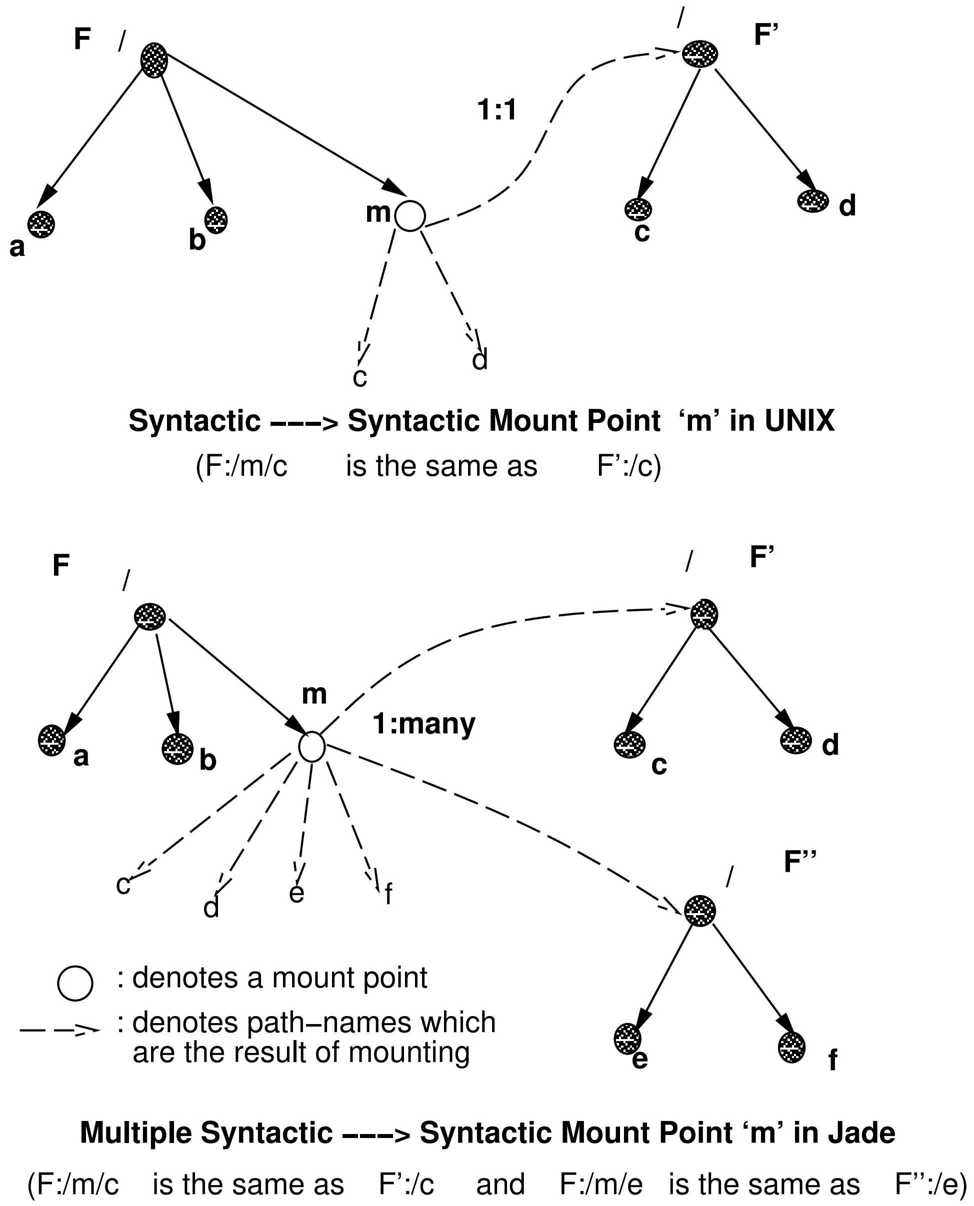


Figure 4.1: Syntactic Mount Points in UNIX and Jade

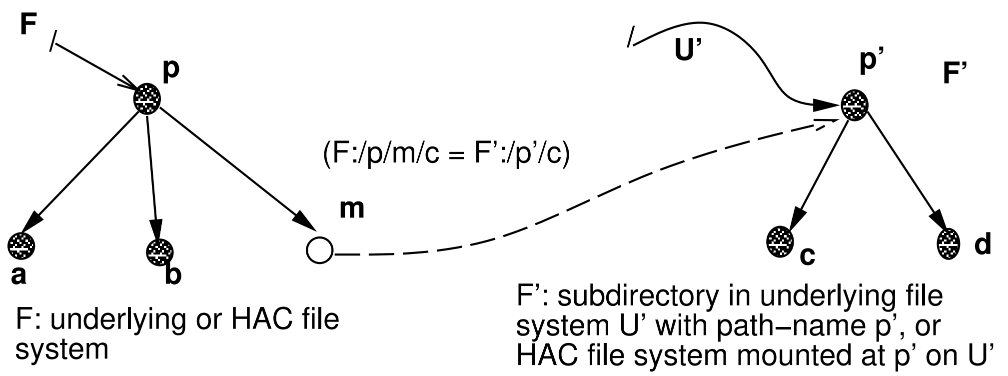


Figure 4.2: Syntactic Mount Point in HAC

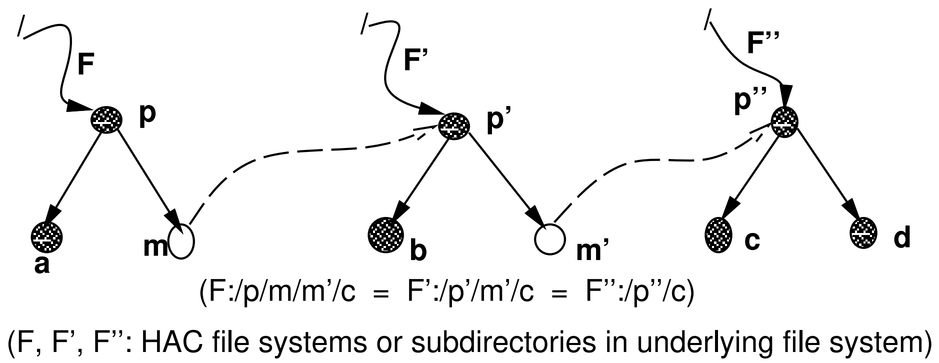
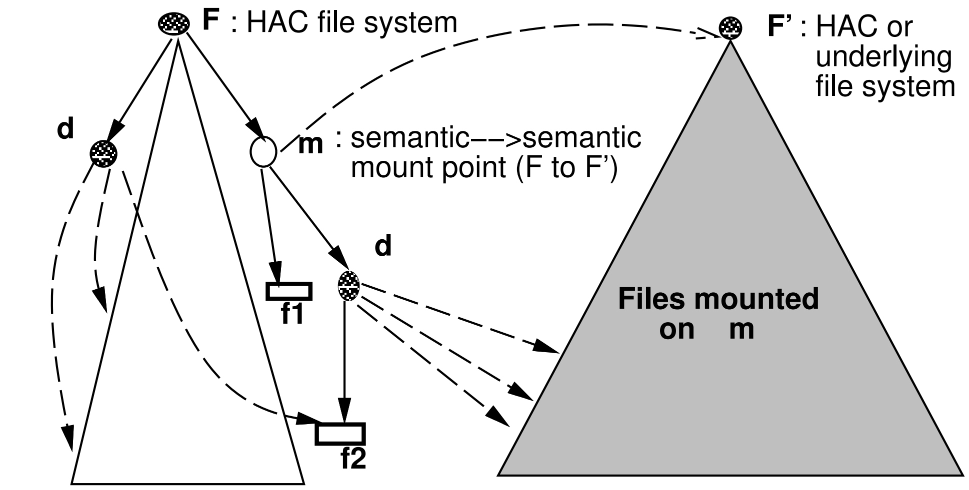


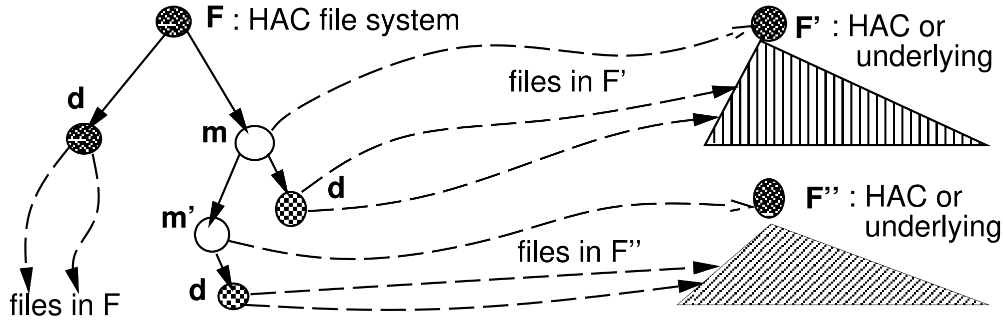
Figure 4.3: Linking up Different HAC File Systems



Note: F:/m/d, F:/m/f1 and F:/m/d/f2 exist within F
Query F:/m/d in m returns links to files in F', while
F:/d outside m returns links to files in F

F:/d and F:/m/d can use different query languages.

Figure 4.4: Semantic Mount Points



F:/d has links to files in F, F:/m/d has links to files in F',
F:/m/m'/d has links to files in F''

Figure 4.5: Scope of Semantic Mount Points

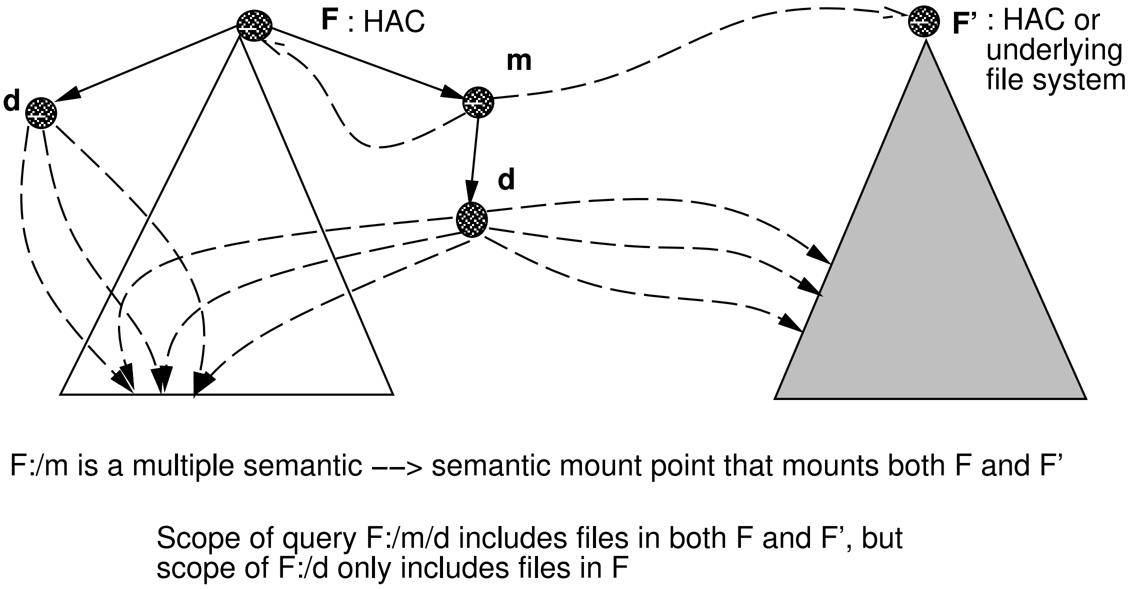


Figure 4.6: Multiple Semantic Mounts

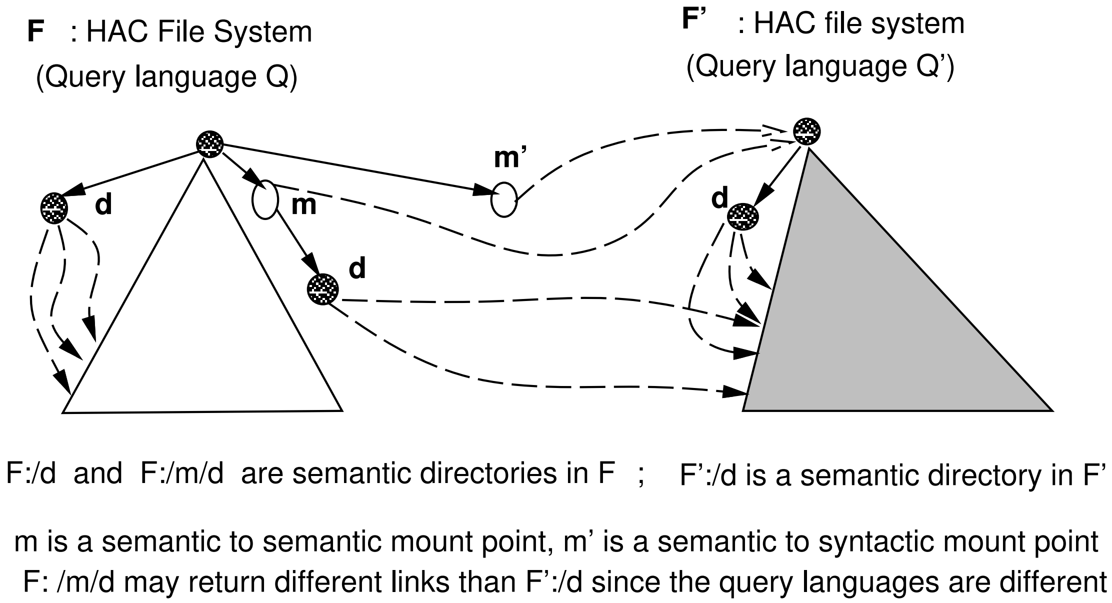


Figure 4.7: Using Both Semantic and Syntactic Mount Points

CHAPTER 5
IMPLEMENTATION AND PERFORMANCE

HAC has been implemented on `SUNOS4.1.3_U1 sun4m`, a sparc 5 running SUNOS. The prototype implementation runs at the user level and requires no kernel modification. The advantage of a user-level implementation is that it is easier to test different design options and debug the code. Moreover, the code itself is more portable since it is a UNIX application program. The obvious disadvantage is that the performance of HAC will suffer compared to that of UNIX. We shall discuss HAC's performance in more detail later in this chapter. First, we shall describe the modules that make up HAC.

5.1 The Modules in HAC

HAC consists of three distinct modules: (i) the C System Call Library, (ii) the Semantic File System, and (iii) the Interface to CBA Mechanisms. The interaction between the user, these modules and UNIX is shown in Figure 5.1.

5.1.1 C System Call Library

This module implements all UNIX file system calls that access objects (file system abstractions) by name. It also provides additional calls that access objects by content (see Appendix A for details). All the semantic commands (`smv, smkdir, ssync`, etc.) described earlier (see Chapter 3) were built using this module. The module is in the form of a library that can be linked *dynamically* with all UNIX applications that use the SUN dynamic-linking facility and the C-Library [1]. This facility allows us to treat a directory in the UNIX file system as the root of a HAC file system, i.e., it allows us to (syntactically) "mount" a user-level HAC file system on UNIX. Hence, all applications that use UNIX file

[1]We were not able to dynamically link the UNIX `ln` and `mv` commands on our system with the HAC library. Hence we wrote our own versions of these commands.

system calls can access the HAC file system without any modification or re-compilation. Note that HAC uses UNIX only as an *object store*, i.e., HAC uses UNIX to store all its internal data structures and the data in all the files in the HAC file system, but provides its own interface to locate files and directories, and manipulate queries and their results.

This module uses *shared memory* (currently 0.5 megabytes) to store the current state of each process that is accessing the library. This state includes the set of open file descriptors, the current directory of the process, a cache of attributes of all recently accessed files and directories, and so on. The state contains all the information that is necessary for this module to interact with the local UNIX operating system, and the remote HAC file systems that are mounted at syntactic mount points. The module also uses shared memory as a cache for its data structures so that it does not have to access remote HAC file systems or UNIX (hence possibly the network and the local disk) during every file system call.

HAC is a user level file system. This means that every instance of the file system is created by a user on an *existing* operating system, not by the operating system itself. HAC provides facilities for this user to access, manipulate and organize his/her information. Hence, it is up to this user to decide whether he/she wants others to access or manipulate this information. That is, HAC has no access control mechanism of its own — HAC borrows it from the underlying operating system.

5.1.2 Semantic File System

This is the most important part of the HAC file system. It implements all operations that access and manipulate objects by name and content. It assumes that objects are stored in a hierarchical file system, but it is otherwise independent of UNIX. It interprets queries and their results and enforces scope consistency (see Chapter 3).

We mentioned earlier in section 3.7 that the result of a query is a set of symbolic links to files that satisfy the query. We also saw how users could manipulate a whole set of symbolic links using syntactic and semantic commands. To speed up set operations like intersection, union, set-difference, and so on, the Semantic File System module uses a compact representation for the query result of a semantic directory. Note that this module does not have to create full-fledged directory entries [2] for the symbolic links. It just needs to keep track of the path-names of the files that match the query. Also note that these files must have been indexed by the CBA mechanism, Glimpse. When Glimpse indexes files, it stores the path-names of these files in an ordered set. The Semantic File System module exploits this fact and represents a set of symbolic links using a bit map. If the i-th bit in a particular bit map is 1, it means that the i-th file that was indexed is present in the corresponding query result. This module transparently converts path-names to bits and vice-versa so that users need not bother about the internals of Glimpse when they are manipulating query results. (For efficiency, this module uses hash tables during these translations.)

The Semantic File System module also implements syntactic and semantic mount points, interacts with the Content Based Access mechanism of the HAC file system, and decides when to re-index files and re-evaluate queries of semantic directories. That is, it also enforces data consistency.

5.1.3 Interface to CBA Mechanisms

This module provides an application programmer interface (API) to Glimpse. The interface was designed in such a way that the the rest of the implementation can be independent of Glimpse. In future, when we integrate other CBA mechanisms into HAC, all we need to do is re-write this module. The module has only about 2000 lines of C code — less than 10 % of the total. It has routines to parse a query, to do the union, intersection and set difference operations on bit maps, to communicate with CBA mechanism using RPCs and to start the indexing process. Note that the module assumes that the indexer and the query processor of the CBA mechanism (see section 3.10) are already available.

5.2 Experiments

HAC is built on top of UNIX and uses Glimpse as the CBA mechanism. HAC adds more power to both (i) the file-naming facility in UNIX, and (ii) the indexing and searching capabilities of Glimpse. In this section, we describe two experiments that determine the

[2] with corresponding UNIX-like *inodes*

price we have to pay for this additional power. In the first experiment, we measured the overhead when we used HAC as a syntactic file system like UNIX, and did the same operations on the same data in both file systems. In the second experiment we measured the overhead when we ran Glimpse to index and search the same data in both HAC and UNIX.

5.2.1 Syntactic File System Overhead

In this experiment, we ran the Andrew Benchmark [25] on HAC and UNIX. The Andrew Benchmark has been used as a standard to evaluate the performance of many new file systems [38]. The input to the benchmark is a read-only source directory hierarchy of an application program and contains about 70 files and occupies about 200 KB. The benchmark has 5 distinct phases:

1 **Makedir**: Constructs a destination directory hierarchy that is identical to the source directory hierarchy.

2 **Copy**: Copies each file in the source hierarchy to the destination hierarchy.

3 **Scan**: Recursively traverses the whole destination hierarchy and examines the status of every file in the hierarchy without reading the actual data in the file.

4 **Read**: Reads every byte of every file in the destination hierarchy.

5 **Make**: Compiles and links the files in the target hierarchy.

Table 5.1 compares the performance of HAC and UNIX for each phase of the benchmark. From this table, we see that phases 1 and 2 have the maximum overhead. It can be as high as 80 - 100 %. This is because in phase 1, when HAC creates a new directory, it also creates and initializes (to "empty") the data structures that store its query, its query-result, and its set of permanent and prohibited symbolic links. And in phase 2, when HAC creates a new file, it also initializes the open file-descriptor and the attribute-cache for that file. This helps to speed up Scan and Read operations on that file. Phases 3 and 4 have a 60 - 75 % overhead. This is because in both phases, HAC must map the shared memory region into every UNIX process that is running. In phase

3, HAC accesses the attribute-cache to retrieve the appropriate status information, and in phase 4, HAC accesses and update the per-process file-descriptor table to implement the read-operation. Phase 5 has the least overhead since it is computationally intensive. On the whole, HAC is about 46 % slower than UNIX. From Table 5.2 HAC is only slightly slower than the Jade [38] and Pseudo [46] file systems (both of which are user-level file systems like HAC). We believe that HAC's performance is quite reasonable since unlike the other two file systems, HAC must create and maintain additional data-structures that provide content-based access to files. We believe that we can improve HAC's implementation by using better caching strategies (e.g., prefix-caching like the Sprite file system [36]) and more compact data-structures (e.g., sparse-arrays instead of bit-maps to represent query-results).

File System	Makedir	Copy	Scan	Read	Make	Total
UNIX	2s	5s	5s	8s	19s	39s
HAC	4s	9s	8s	14s	22s	57s

Table 5.1: Results of Andrew Benchmark

File System	% Slowdown
Jade FS	36
Pseudo FS	33-41
HAC FS	46

Table 5.2: Comparison with other File Systems

We also calculated the extra disk-space HAC needs to store its data structures when both HAC and UNIX are used to store the source code of the Andrew Benchmark. We found that (i) HAC has a fixed space overhead of 10 KB, and (ii) the Andrew Benchmark occupies 212 KB when stored in HAC while it occupies 210 KB when stored in UNIX. Hence the space overhead in HAC is around 5 %, which is negligible.

5.2.2 CBA Mechanism Overhead

In the first part of this experiment, we used Glimpse to index a database consisting of over 17000 files that occupy about 150 MB. We ran the indexing mechanism directly over UNIX to get an estimate of the time taken to index the database and the space needed to store the index. We then indexed a different copy of the same database by using the HAC file system library instead. The results are shown in Table 5.3.

No. of files	17154
Size of files	149 Megabytes
Size of UNIX index	10 Megabytes
Size of HAC index	11.5 Megabytes
Time taken in UNIX	25 min
Time taken in HAC	31 min 48 sec

Table 5.3: Results of Indexing

We see that HAC has needs 6 min 48 sec more than UNIX, and UNIX takes 25 min to index these files. Hence, HAC has a 27 % time overhead. We also calculated the extra space HAC needs to store the hash tables that help in translating bit-maps to path-names and vice-versa (see above). These are not required for a regular UNIX index. We found that HAC needs about 1.5 MB more than UNIX and the UNIX-index is about 10 MB. Hence, HAC has a 15 % space overhead. We believe that both these overheads are reasonable.

In the second part of this experiment, we used the `smkdir` command in HAC to create a semantic directory with a query **Q**. We also ran Glimpse through UNIX to search the above database for the same query **Q**. We chose three kinds of queries: (i) those that matched very few files, (ii) those that matched a lot of files, and (iii) those that matched an intermediate number of files. (We believe that queries of type (iii) correspond to real usage, since most semantic directories in the example in section 3.12 have query results that point to about 1/10-th to 1/20-th of the total number of files in the `pcindex` database, and 98 is just about 1/20-th of 17154.) The results are shown in Table 5.4.

We see that for queries that matched very few files, Glimpse running on UNIX is more

No. of files that matched	1	6556	98
Time taken in UNIX	.45 sec	4 min 23 sec	7 sec
Time taken in HAC	2 sec	4 min 28 sec	8 sec

Table 5.4: Results of Searching

than 4 times as fast as HAC. This is because to interact with the CBA mechanism in HAC, we must create a semantic directory. We do not incur this overhead when we run Glimpse on UNIX. The overhead of creating a semantic directory, however, reduces as the number of files that match the query increases. For queries that match an intermediate number of files, the overhead is about 15 %. For queries that match a lot of files, the overhead is only 2 %, which is negligible.

The drawback of our implementation is that we need to store the result of the query of each semantic directory. The extra space we need per semantic directory is 1/8-th of the number of files that are indexed, i.e., about 2 KB in this experiment. We plan to improve this in future by using sparse-array representations.

To summarize, the HAC file system has small overheads compared to UNIX. Its performance is actually quite good when we consider the fact that it combines the flexibility of hierarchical file systems with the power of CBA mechanisms, and allows users to access and organize their information in a better way (see section 3.12).

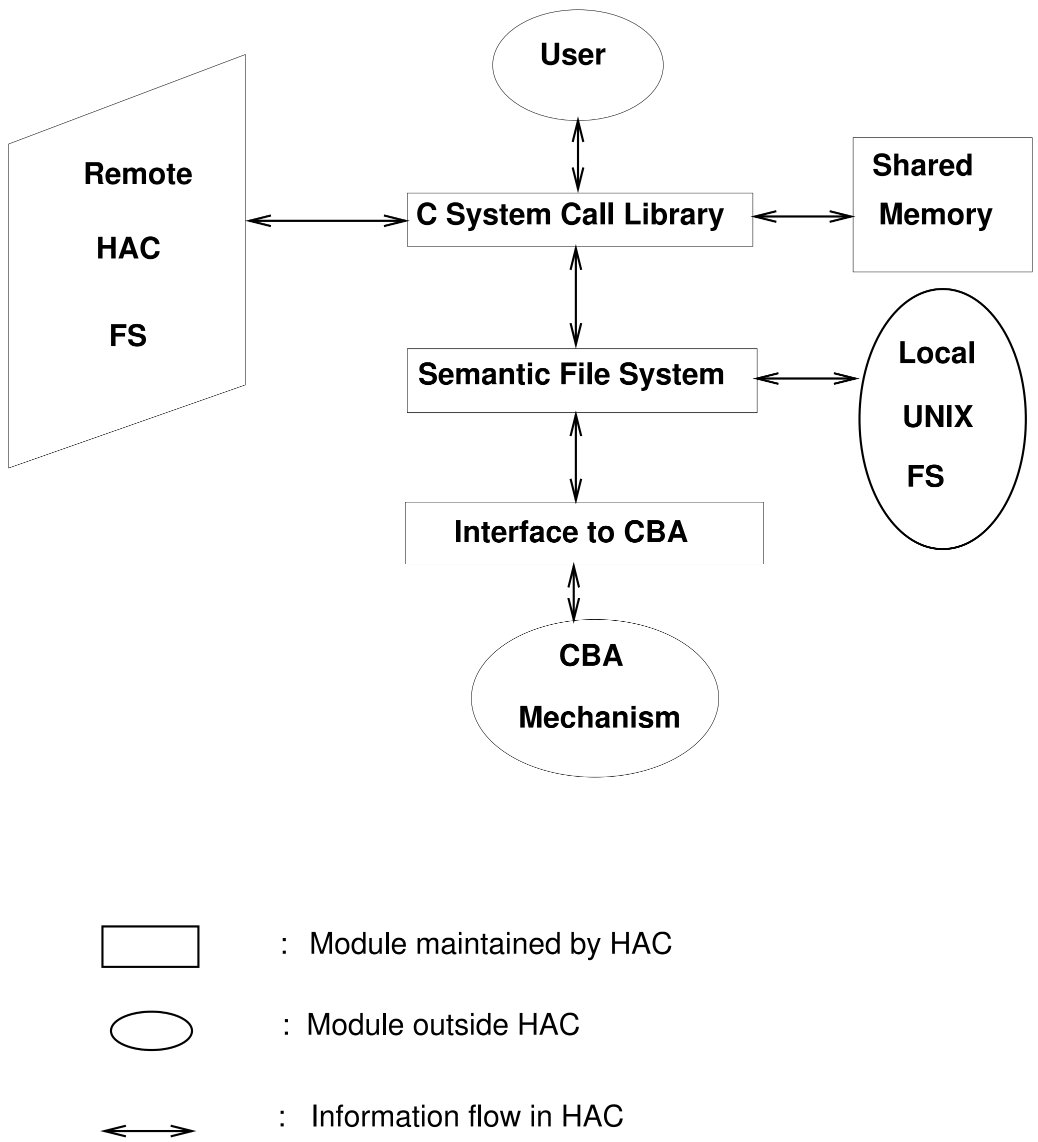


Figure 5.1: Interaction Between Modules in HAC

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

HAC demonstrates that it is possible to integrate any content-based access mechanism into a hierachical file system in such a way that the full power of hierarchical and content-based naming is always available. The notions of semantic directories, semantic commands and semantic mount points in HAC show that hierarchical and content-based naming can not only complement each other, but also enhance each other's capabilities in useful ways. This is the main contribution of our research.

Our work on HAC, however, is far from complete. HAC has some limitations and there are many interesting research questions yet to be answered. Some of them are mentioned below:

1. HAC does not have a mechanism to match a given query against the queries of existing semantic directories in the file system, and retrieve those directories whose queries are similar (or identical) to the given query. This would be an interesting extension since search engines of today just retrieve files based on their content: not other file system abstractions.

2. HAC does not automatically classify a set of files based on their content. Classification must be done by the user or the information provider. Hence, it would be very rewarding to integrate automatic classification schemes such as Scatter-Gather (see section 2.2) into HAC.

3. HAC integrates CBA mechanisms into hierarchical file systems. Nowadays, however, emphasis is shifting toward graph-structured information repositories like the World Wide Web. To integrate CBA mechanisms with these repositories, we *can* use Prospero [34], but we must remember that it provides no consistency guarantees. Hence, the first step in this direction would be to define a useful consistency model for queries and their results in a graph-structured information repository.

4. On the other end of the spectrum, Relational data-base systems [17] have more complex set-manipulation operations than *selection* (`smkdir`), *intersection* (`smv, ssync`) and *union* (using names of directories in queries) operations defined in HAC. To allow more sophisticated content-based information retrieval, the first step would be to define what we mean by operations like *projection* and *join* in a file system. Intuitively, a "projection" of a file is similar to the extraction of a *summary* of the file, and a "join" of two files is similar to the derivation of a new file from the contents of these files. Some related ideas are discussed in [13].

5. *Dynamic Sets* [43] is a new operating system abstraction that can be used to optimize access to sets of files. We can use this idea to speed up *semantic* operations (like `ssync`, `smkdir`) that search groups of files within the scope of different semantic directories.

We believe that the principles behind HAC's design are universal and can be applied in many contexts. HAC is simple, flexible and extensible. Though the prototype implementation is very good for a user-level file system, it also has some limitations and we believe we can improve it even further. For example, we can:

1. Integrate other search engines besides Glimpse into HAC,

2. Integrate HAC and Jade file systems together so that we can combine various kinds of syntactic mount points in Jade with *semantic* directories and mount points in HAC, and use Jade's file-data and directory-entry caching mechanisms to improve HAC's performance,

3. Use event-driven or lazy evaluation to keep queries and their results consistent,

4. Use sparse-array representations to reduce the amount of space required to store query results, and

5. Prune query-results (say, in `sls`) by identifing symbolic links to files that are very similar to each other, and removing all but one of those links before display.

We plan to make the source code for HAC available to the general public very soon. We hope that our work will continue to simulate further research in information systems.

APPENDIX A
APPLICATION PROGRAMMER INTERFACE

Below are the C-prototypes for the file-system library functions in HAC. A brief explanation is given for the new functions added in HAC in the form of a C-comment above the function prototype definition. The arguments and return values of all functions are similar to the corresponding functions in UNIX.

```
int exit(int status);

int execve(char *path, char *argv[], char *envp[]);

int fork();

caddr_t mmap(caddr_t addr, size_t len, int prot, int flags,
int fd, off_t off);

int close(int fd);

int read(int fd, char *buf, int nbytes);

int getdents(int fd, char *buf, int nbytes);

/* Return set of permanent symbolic links */
int igetdents(int fd, char *buf, int nbytes);

/* Return set of prohibited symbolic links */
int egetdents(int fd, char *buf, int nbytes);

off_t lseek(int fd, off_t offset, int whence);

int write(int fd, char *buf, int nbytes);
```

```
int ftruncate(int fd, off_t length);

int dup(int fd);

int dup2(int fd, int newfd);

int fchdir(int fd);

int fchroot(int fd);

int fchmod(int fd, int mode);

int fchown(int fd, int owner, int group);

int fstat(int fd, struct stat *buf);

int fstatfs(int fd, struct statfs *buf);

int open(char *path, int flags, int mode);

/* Open the "path" to access its summary instead
   of the actual data;
   The summary is computed based on the query of
   the semantic directory in which "path" exists;
   If there is no query, then the summary will be
   empty. */
int sopen(char *path, int flags, int mode);

int creat(char  *path, int mode);

int chdir(char *path);

char *getwd(char *s, int size);

int chroot(char *path);
```

```
int access(char *path, int mode);

int chmod(char *path, int mode);

int chown(char *path, int owner, int group)

int mkdir(char *path, int mode);

/* Creates a semantic directory with query = "query",
   and name = "path" */
int smkdir(char *query, char *path, int mode);

/* Re-evaluates the queries of all semantic directories that
depend on the directory "path" including "path" */
int qsync(char *path);

int rmdir(char *path);

int truncate(char *path, off_t length);

int unlink(char *path);

/* Remove the symbolic link "path" from the set of permanent
symbolic links of its parent: do not change its set of
transient symbolic links in any way */
int iunlink(char *path);

/* Remove the symbolic link "linkname" from the set of
prohibited links of the directory "dirname" */
int eunlink(char *linkname, char *dirname);

/* Add the symbolic link "linkname" to the set of
prohibited links of the directory "dirname" */
int elink(char *linkname, char *dirname);
```

```
/* If the realname is a symbolic link of the form "::path",
add a permanent symbolic link localname to the target of
"::path" in the parent of localname; Otherwise, create a
regular UNIX symbolic link */
int symlink(char  *realname, char  *localname);

/* If the realname is a symbolic link of the form "::path",
add a permanent symbolic link localname to the target of
"::path" in the parent of localname; Otherwise, create a
regular UNIX hard link */
int link(char *realname, char *localname);

int rename(char *realname, char *localname);

int readlink(char *path, char *buf, int bufsiz);

/* If path is the name of a directory, return its query;
   it it is a mount point, return the type of mount point,
   i.e., whether syntactic/semantic, and
   whether local/remote */
int sreadlink(char *path, char *buf, int bufsiz);

int utimes(char *path, struct timeval *tvp);

int stat(char *path, struct stat *buf);

int statfs(char *path, struct statfs *buf);

int lstat(char *path, struct stat *buf);

int mount(char *type, char *dir, int flags, caddr_t data);

/* Create a mount point dir in the HAC file system: if the mount
   point already exists, create a multiple mount point.
   The "flags" specify whether the mount point is
   semantic/syntactic and whether it is local/remote.
```

```
   The "data" specifies the address of the target file system
   in a well-known format (see header-files in the source code
   for more details).  */
int smount(char *type, char *dir, int flags, caddr_t data);

int unmount(char *dir, int flags, char *data);

/* Unmount a mount point created by smount above */
int sunmount(char *dir, int flags, char *data);
```

APPENDIX B

BRIEF DESCRIPTIONS OF USER COMMANDS

```
USAGE: smkdir [query] name
**************************
description: creates a new semantic directory with alias=‘name’
              and query=‘query’ --- the default query is the alias itself;
note: the query is evaluated within the scope of the parent of name;
       the scope of the root of the file system / mountpts extend to
       all the indexed files in the corresponding file systems

USAGE: imv realname localname
*****************************
description: the directory entry realname is renamed as localname
       if realname is a directory, its contents are changed so that they
       are now within the scope of the parent of localname;
       'imv file directory/.' won't work; you must: 'imv file
       directory/file';
       if localname was not previously present in realname, it is added
       to the include list of realname;
       localname is always removed from the include list of its parent
       and added to its exclude list.
note: imv does not support any options now, and works on 2 arguments only.
note: we wrote imv since the system 'mv' is statically linked

USAGE: sls [-e/-i] directory
****************************
description: you can use the -i option (show contents of include list)
       or the -e option (show contents of exclude list);
       if you don't use any option, it acts as 'ls' except that it
       outputs the list of links in directory that don't occur in any of
       its children.
```

```
note: sls does not support other options of 'ls' options now: use 'ls'.

USAGE: ssync [-q] directory
***************************
description: no -q: synchronizes links in directory and all
       sub directories below it in the hierarchy: stops at mountpts --
       does not reevaluate queries;
with -q: synchronizes links in directory and all sub directories that
       depend on it by reevaluating queries: stops at both semantic and
       syntactic mount pts --- must call ssync explicitly in each mount
       point to sync whole hierarchy.

USAGE: iln [-s] realname localname
**********************************
description: creates a hard link between realname and localname
       if the -s option is used, it creates a symbolic link instead;
       if realname is a system generated link (beginning with *:*:*),
       then localname must be the target directory where the link
       should be added.
note: we wrote iln since the system 'ln' is statically linked

USAGE: smv [-q] realname localname
**********************************
description: no -q: move all links in realname to localname: realname
       will now have only those links that are in its include list;
with -q: move the query=realname into localname: contents of localname
       are sync-ed automatically

USAGE: sreadln localname
************************
description: output semantic info corr. to localname; if localname is a:
       directory: output its query
       mountpoint: output command line options that describe it
       file: print absolute path name of file
       link: print complete path name of link
       symbolic link: print path name it leads to
```

```
USAGE: smount [-ryqun] [-h host] [-p port] [-f prefix] [-s server]
       [-i indexer] [-H host] [-P port] directory {mountptname}*
*****************************************************************
description: mounts a semantic file system onto a an existing / new
       mount point
no -r, no -y: local semantic (indexing, full-text search)
-r, no -y: remote semantic (no local control of indexing,
       do index-search only)
-y: syntactic mount (RPCs to server: -r flag not interpreted here)
-q: don't do indexing locally: use existing query processor
-h host: host for server process
-p port: port for server process
-f prefix: part of indexed filenames that is stripped before display
       (no -y)
-s server: path name of server program (no -y)
-i indexer: path name of indexer program (no -y)
-H: host of remote server that should be mounted (no -y, -r)
-P: port of remote server that should be mounted (no -y, -r)
{mountptname}*: dirs that should be indexed (no -y, no -r):
       empty=just recompute index for existing files / directories
       in the index
-n: start local server process again
-u: reindex all files (no -y, no -r) / obtain index afresh (no -y, -r)
Commonly used forms (root is root of sfs: $bg/phd/try, say):
       SYNTACTIC: smount -p PORT -h HOST -u -n -y mtptname
       SEMANTIC LOCAL: smount -p PORT -h HOST -u -n -f root/ root root
       SEMANTIC REMOTE (index caching): smount -p PORT -h HOST -P RPORT
              -H RHOST -u -n -r dir

USAGE: sumount [-ryun] [-h host] [-p port] [-f prefix] [-s server]
       [-i indexer] [-H host] [-P port] directory {mountptname}*
******************************************************************
description: mounts a semantic file system onto a an existing / new
       mount point
no -r, no -y: local semantic (indexing, full-text search)
```

```
-r, no -y: remote semantic (no local control of indexing,
       do index-search only)
-y: syntactic mount (RPCs to server: -r flag not interpreted here)
-q: don't do indexing locally: use existing query processor
-h host: host for server process
-p port: port for server process
-f prefix: part of indexed filenames that is stripped before display
       (no -y)
-s server: path name of server program (no -y)
-i indexer: path name of indexer program (no -y)
-H: host of remote server that should be mounted (no -y, -r)
-P: port of remote server that should be mounted (no -y, -r)
{mountptname}*: dirs that should be indexed (no -y, no -r):
       empty=just recompute index for existing files / directories
       in the index
-n: start local server process again
-u: reindex all files (no -y, no -r) / obtain index afresh (no -y, -r)
Commonly used forms (root is root of sfs: $bg/phd/try, say):
       SYNTACTIC: sumount -p PORT -h HOST -u -n -y mtptname
       SEMANTIC LOCAL: sumount -p PORT -h HOST -u -n -f root/ root root
       SEMANTIC REMOTE (index caching): sumount -p PORT -h HOST -P RPORT
      -H RHOST -u -n -r dir

USAGE: scat links
*****************
description: outputs the parts of the files pointed to by the
       indicated links that match the query of the directory in
       which the links exist (their parent)
```

REFERENCES

[1] M. Andreessen. NCSA Mosaic technical summary. Technical report, National Center for Supercomputing Applications, May 1993.

[2] T. Berners-Lee, R. Calliau, and B. Pollermann. World-wide web: The information universe. *Electronic Networking: Research, Applications, and Policy*, 2:52–58, Spring 1992.

[3] C. Bowman, P. Danzig, D. Hardy, U. Manber, and M. Schwartz. The Harvest information discovery and access system. In *Proc. 2nd Intl. World Wide Web Conference*, pages 763–771, Chicago, Illinois, October 1994. (Earlier version; Later version to appear in a special issue of Computer networks and ISDN Systems).

[4] M. Bowman, P. Danzig, D. Hardy, U. Manber, and M. Schwartz. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, University of Colorado, Boulder, Dept. of Computer Science, July 1994.

[5] M. Bowman, P. Danzig, D. Hardy, U. Manber, and M. Schwartz. Scalable internet resource discovery: Research problems and approaches. *Communications of the ACM*, 37(8):98–107, August 1994.

[6] M. Bowman, Chanda Dharap, Mrinal Baruah, B. Camargo, and Sunil Potti. A file system for information management. In *Proc. Conf. on Intelligent Information Management Systems*, Washington, DC, June 1994.

[7] M. Bowman, L. Peterson, and A. Yeatts. Univers: An attribute-based name server. *Software - Practice and Experience*, 20(4):403–424, April 1990.

[8] M. Bowman, M. Spasojevic, and A. Spector. File system support for search. Technical report, Transarc Corporation, Pittsburgh, November 1994.

[9] Excite by Architext. http://www.excite.com/Subject/.

[10] V. Cate. Alex: A global file system. In *Proc. Workshop on FIle Systems*, May 1992.

[11] D. Cutting, D. Karger, O. Pedersen, and J. Tukey. Scatter/Gather: A Cluster-based Approach to Browsing Large Document Collections. In *Proc. 15th Annual ACM SIGIR Conf. on Research and Development in Information Retrieval*, Denmark, 1992.

[12] D. Cutting, D. Karger, and O. Pedersen. Constant Interaction-time Scatter/Gather Browsing of Very Large Document Collections. In *Proc. 16th Annual ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 126–134, 1993.

[13] M.P. Consens and T. Milo. Optimizing Queries on Files. Department of Computer Science, University of Toronto, Toronto, Canada M5S 1A4, pages 1-17, December 1993.

[14] P. Clark and U. Manber. Developing a personal internet assistant. In *Proc. ED-Media 95, World. Conf on Multimedia and Hypermedia*, pages 372–377, Graz, Austria, June 1995.

[15] J. Conklin. Hypertext: An introduction and survey. *Computer*, 20(9):17–41, September 1987.

[16] P. Danzig, J. Ahn, J. Noll, and K. Obraczka. Distributed indexing: A scalable mechanism for distributed information retrieval. Technical Report USC-TR 91-06, University of Southern California, Computer Science Dept., 1991.

[17] C.J. Date. An Introduction to Database Systems, 6th ed. Addison-Wesley Publishing Company, 1995, Reading, Massachusetts.

[18] P. Danzig, S. Li, and K. Obraczka. Distributed indexing of autonomous internet services. *Computing Systems*, 5(4):433–459, Fall 1992.

[19] Yahoo Directory. http://www.netscape.com.

[20] Netscape. http://www.excite.com/Subject/.

[21] A. Emtage and P. Deutsch. Archie: An electronic directory service for the internet. In *Proc. Winter 1992 Usenix Conference*, pages 93–110, January 1992.

[22] G. Fowler. cql - A Flat File Database Query Language. *Proc. Winter 1992 Usenix Conference*, pages 11-21, San Fransisco, CA, January 1994.

[23] D. Gifford, P. Jouvelot, M. Sheldon, and Jr J. O'Toole. Semantic file systems. In *Proc. 13th ACM Symposium on Operating Systems Principles*, pages 16–25, Pacific Grove, Ca, October 1991. ACM.

[24] D. Hardy and M. Schwartz. Essence: A resource discovery system based on semantic file indexing. In *Proc. USENIX Winter Conference*, pages 361–374, San Diego, January 1993.

[25] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[26] B. Khale and A. Medlar. An information system for corporate users: Wide area information servers. In *Connexions — The Interoperability Report*, 5(11). November 1991.

[27] GlimpseHTTP Server. http://glimpse.cs.arizona.edu.

[28] U. Manber and S. Wu. Glimpse: A tool to search through entire file systems. In *Usenix Winter 1994 Technical Conference*, pages 23–32, San Fransisco, January 1994.

[29] M. McCahill. The internet Gopher: A distributed server information system. *ConneXions - The Interoperability Report*, 6(7):10–14, July 1992.

[30] U. Manber, M. Smith, and B. Gopal. WebGlimpse – Combining Browsing and Searching. To appear in: Usenix Annual Technical Conference. In Jan 6-10, Anaheim, California, 1997.

[31] S. Mullender, editor. Distributed Systems, 2nd ed. ACM Press, 1993, New York.

[32] B. Neuman. Prospero: A tool for organizing internet resources. *Electronic Networking: Research, Applications, and Policy*, 2(1):30–37, Spring 1992.

[33] B. Neumann. The virtual system model: A scalable approach to organizing large systems. Technical Report 90-05-01, University of Washington, CS Dept., May 1990.

[34] B. Neumann. The Prospero file system: A global file system based on the virtual system model. In *Proc. Usenix Workshop on File Systems*, May 1992.

[35] McKinley Group Directory of the Internet. http://www.mckinley.com.

[36] J. Ousterhout, A. Chrenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 28(2):23–36, February 1988.

[37] L .Peterson. The profile naming service. *ACM Transactions on Computer Systems*, 6(4):341–364, November 1988.

[38] H. Rao and L. Peterson. Accessing files in an internet: The Jade file system. *IEEE Transactions on Software Engineering*, 19(6):613–624, June 1993.

[39] D. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[40] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, Design and implementation of the Sun Network File System In *Proc. Summer Usenix*, pages 119-130, June 1985.

[41] M. Satyanarayanan Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5):9-21, May 1990.

[42] M. Schwartz, A. Emtage, B. Kahle, and B. Neuman. A comparison of internet resource discovery approaches. *Computing Systems*, 5(4):461–493, Fall 1992.

[43] D. Steere and M. Satyanarayana. A Case for Dynamic Sets in Operating Systems. Technical Report TR CMU-CS-94-216, Carnegie Mellon University, Pittsburg, PA 15213, Nov. 1994.

[44] InfoSeek Search. http://www2.infoseek.com/.

[45] S. Sechrest and M. McClennen. Blending hierarchical and attribute-based file naming. In *Proc. 12th Intl. Conf. on Distributed Computer Systems*, June 1992.

[46] B. Welch and J. Ousterhout. Pseudo-File-Systems. Technical Report UCB/CSD 89/499, University of California, Berkeley, CA 1989.

[47] S. Wu and U. Manber. Agrep — a fast approximate pattern-matching tool. In *Usenix Winter 1992 Technical Conference*, pages 153–162, San Fransisco, January 1992.