Integrating Content-based Access Mechanisms with Hierarchical File Systems

Burra Gopal
Microsoft Corp
Udi Manber
University of Arizona

Integrating Content-Based Access Mechanisms with Hierarchical File Systems

Burra Gopal[*]
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052
burrag@microsoft.com

Udi Manber [†]
Dept. of Computer Science
University of Arizona
Tucson, AZ 85721
udi@cs.arizona.edu

ABSTRACT

We present a new file system that combines name-based and content-based access to files at the same time. Our design allows both methods to be used at any time, thus preserving the benefits of both. Users can create their own name spaces based on queries, on explicit path names, or on any combination interleaved arbitrarily. All regular file operations – such as adding, deleting, or moving files – are supported in the same way, and in addition, query consistency is maintained and adapted to what the user is manually doing. One can add, remove, or move results of queries, and in general handle them as if they were regular files. This creates interesting new consistency problems, for which we suggest and implement solutions. Remote file systems or remote query systems (e.g., web search) can be integrated by users into their own coherent name spaces in a clean way. We believe that our design can serve as the basis for the future information-rich file systems, allowing users better handle on their information.

1 Introduction

One of the most important challenges to current operating systems is to provide convenient access to vast amounts of information. By convenience, we mean not only the ability to quickly transfer information from one place to another, but the ability to find the right information and deal with it. This is arguably a new problem due to the scale of available information. File systems that were designed when typical users had few Magabytes and hundreds of files to contend with are getting inadequate when Gigabytes and hundreds of thousands of files are the norm.

The way we access file systems has not changed much in the last 30 years. Most file systems are based on a hierarchical arrangement with access by explicit path names or browsing (i.e., going down and up the tree). Hierarchical file systems have been successful because they provided everything we needed. They were extended to network file systems (trying to keep this transparent to the users), and widely distributed file systems. Numerous added features – such as quick search for file names, symbolic links or shortcuts, and automatic compression and backup, to name a few – make file system access even more convenient.

However, current file systems are hard pressed to deal with the vast amount of available information that is already upon us. Not in the physical sense – it is still relatively easy to store and access information. But being able to make effective use of that information is becoming harder and harder. For example, although a lot of information is obtained through searches, integrating this information into a file system is still done mostly by hand with little support. We present in this paper a new method of attacking this problem. We introduce a new paradigm,

[*]Work done while at the University of Arizona.

[†]Supported in part by DARPA contract N66001-96-C-8615. New-address: Yahoo! Inc., 3420 Central Expressway, Santa Clara, CA 95051.

actually a combination of old paradigms, and report on a successful implementation of a file system that follows that paradigm.

Our starting point is the *semantic file system* (SFS) paradigm introduced by Gifford et al [7]. Semantic file systems provide access by queries. They support the creation of *virtual directories*, each pointing to files that satisfy a query. Virtual sub-directories can be built using pointers from the parent, making a hierarchy based on query refinement. Semantic file systems allow users to organize their files by content and provide means to do that conveniently. This is sorely needed, because beyond a certain scale limit, people cannot remember locations by explicit path names. After so many years, it is still amusing to see even experienced UNIX system administrators spend time trying /usr/lib, or was it /usr/local/lib, maybe /opt/local/etc/lib, or /opt/unsupported/lib? There are, of course, many search tools available, but organizing large file systems is still too hard. The web, of course, has raised this problem to new heights.

So why haven't semantic file systems caught on? Clearly, as in any innovation, it takes a long time for people to change paradigms, especially if this directly involves everyday's tasks. It is essential to provide a smooth transition, which is currently not available. In addition, hierarchical file systems offer strong features that are not supported by semantic file systems. So the natural question is "can we combine the two paradigms?" Can we build a file system that will have the benefits of both hierarchical *and* semantic file systems, and allow users to choose among their features at any time?

We want to allow the use of the file system as a regular traditional hierarchical file system with no need to change anything. The added features of a content-based access (**CBA**) should be optional under the control of the user. They can cover the whole file system, any part of it, or none at all. They can be discarded and added at any time. Consequently, we base our design on a hierarchical file system and add content-based access, rather than extend a given content-based mechanism [12].

The main contribution of this paper is to show that combining name and content-based access is possible and that it can be implemented efficiently and reasonably cleanly. Our main goal is convenient and intuitive integration of information, without tying ourselves into any one special model. We maintain the full power of hierarchical file systems, allow users to automatically or manually modify and refine query results, preserve consistency of results even under manual changes, and provide integrated flexible access to remote file systems or query systems.

The paper is organized as follows. In section 2 we introduce our new file system, **HAC**, which stands for **Hierarchy And Content**. We discuss the major design problems, and suggest solutions and tradeoffs. Section 3 discusses how HAC connects to remote file systems and query systems through our notion of *semantic mount points*. Section 4 describes the implementation of HAC and gives performance measures, and section 5 discusses related work. A lot more work is needed to make such a system a mainstream general-purpose file system. We believe that this paper makes a significant step towards this goal.

2 The Design of the HAC File System

2.1 A Running Example

To describe the design of HAC we will use one running example. Suppose that the user is working on a project involving the use of *fingerprints* (as one of the authors had). Information about the project may be found in email with its participants, in notes, articles, source code files, etc. Typically each of these will be stored in a different place, possibly on a different computer (e.g, a laptop or a network file server). The user may also have relevant information from previous projects or from other sources which the user may not even remember. Furthermore, important information can be obtained through a search of remote facilities. HAC allows to combine all relevant material in one *semantic* directory; let's call it **fingerprint**. We'll see how to build it, maintain it, and use it later on.

2.2 Queries, Query-Results and Semantic Directories

Current and suggested file systems that provide query support treat the "name space" associated with the query-based access to files as logically different from the name space associated with path name-based access. This makes it very difficult (if not impossible) to offer both forms of naming within the same system. For example, it is not possible to create new files within the *virtual directories* of SFS [7], and it is not possible to combine *views* of Nebula with directories in the "underlying" file system [5]. The *Multi-structured Naming* system [12] comes close. It allows users to specify certain relationships between queries (or "labels") so that users can organize queries and their results in a hierarchy. (Unlike SFS, if two queries in this system are related to each other in a hierarchy, their query-results do not necessarily have to be related in any way.) However, they still do not have the freedom to group files of their choice together within a label: they must also think of a query that matches the contents of exactly these files (and no others), and associate the query with this label.

Our approach to this problem is radically different: instead of starting with a query-based naming system and imposing a hierarchy or other relationships on queries, we start with a hierarchical naming system and extend it to support query (content) based naming. We show that this approach has many advantages: it gives users a lot of flexibility and power, and at the same time it makes the system easy and intuitive to use.

The first step is to map queries and their results onto file system abstractions. For obvious reasons, we decided to map queries into directories in the HAC file system. We call such directories *semantic directories*. When users create a new semantic directory, they specify both its path name and its query. HAC then creates a new directory, associates it with the query, and contacts the CBA mechanism to evaluate the query. In the new directory, HAC automatically creates new symbolic links to all files that satisfy the query. These symbolic links can co-exist with other information in the semantic directory, including other symbolic links or other regular files. The symbolic links can also point to files in other semantic directories in the file system, or even to remote file systems. HAC also provides a mechanism by which the user can easily extract the results of the query from these files. Semantic directories provide the abstraction and utility of virtual directories, but in HAC they are also *regular* hierarchical directories for all purposes. Users can add files to them, modify them, run applications from them, and so on.

HAC allows both ordinary *syntactic* directories to co-exist in the same file system. Directories (whether semantic or syntactic) can be accessed by specifying path names, and they can contain files, sub-directories, symbolic links, etc., as usual. Semantic directories contain additional information that helps HAC to maintain them and keep them consistent with whatever the user is doing. The consistency problem is a new non-trivial problem that we discuss next.

2.3 Scope of Queries and Scope Consistency

In HAC, every query – and its corresponding semantic directory – has a *scope* which is the set of files over which the query is evaluated. A query does not return symbolic links to files that are 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 **..../parent/child** is a path such that both **parent** and **child** are semantic directories, then the scope of **child** is defined to be the existing set of symbolic links in **parent**. This set of symbolic links is also called the scope "provided" by **parent**. The scope provided by the root of a HAC file system is defined to be all the files in that file system. A change in the scope provided by **parent**, for example, will also change the scope of **child**. In this case, we say that **child** *depends* on **parent**. Note that all directories in the file system directly or indirectly depend on the root.

By these definitions, the scope provided by a newly created child semantic directory is always a *refinement* of the scope provided by its parent [5, 7]. When a user creates a new semantic subdirectory, HAC guarantees that the new set of symbolic links in that directory is always a subset of the set of the existing 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. Hence, semantic directories allow users to organize both files and results of queries in a hierarchical fashion.

Semantic directories also allow users to tune the results of queries according to their personal tastes. HAC interprets the existing set of symbolic links in a semantic directory as its existing ("current") query result. Since each query-result is a separate entity, users can *modify* the result of any query by (i) deleting some irrelevant links returned by the query, (ii) creating new links to files that have related information, but were missed by the query, or (iii) adding regular files to that directory. In our fingerprint example, users may want to add a set of C programs implementing fingerprints, email messages from a certain user or about a certain topic, and/or image files to the **fingerprint** semantic directory, even though these files do not match **fingerprint**'s query. They may also decide that news stories about a certain crime should be removed from **fingerprint** even though they do match the query. They can do that by making the query more complex (e.g., "fingerprint AND NOT murder"), but often it is easier to remove a few files manually. Users can also build email semantic directories, allowing a message to be in more than one directory (e.g., by sender, receipient, topic, and/or a combination).

No query system is perfect, and currently most are not even close. HAC gives users more power to customize and adjust. It allows users to refine queries by using either the query language *or* the file system directly. Both methods are valid and being able to apply both at any time makes HAC very powerful and intuitive. But there's a major problem with this freedom. Since HAC allows users to edit the results of queries, it is now possible for them to create a hierarchy of semantic directories that makes sense to them intuitively, but still violates the scope restrictions discussed earlier. We call this the *Scope Consistency Problem.*

As far as we know, no existing file system has addressed the scope consistency problem. The Semantic File System [7] and Nebula [5] do not allow users to modify results of queries without modifying the query or the files in the file system. Prospero [9] allows users complete freedom to define and manipulate queries (or "filters") and their results, but does not talk about enforcing any kind of consistency when results of queries are arranged in a hierarchy (or a graph). Search systems like Harvest [4] and various WWW search engines are geared to bring search results to users, but not to organize results in any meaningful way.

Our approach to this problem in one of the main contributions of this paper. We now describe our solution in detail and show that it gives rise to a powerful new paradigm. To begin with, we classify symbolic links in a semantic directory in three ways (this distinction, for the most part, is hidden from users):

Permanent symbolic links: links that were explicitly added by the user to the directory.

Transient symbolic links: links that were obtained by evaluating a query.

Prohibited symbolic links: links (whether transient or permanent) that were once present in the directory but at some point were explicitly deleted from it by the user. HAC will ensure that these links will not be implicitly added later without a direct action by the user.

Given a semantic directory **sd**, which is not the root of a HAC file system, we define the *scope restriction* on the set of symbolic links in **sd** as the following invariant:

1. *The set of transient symbolic links in* **sd** *is*

always a subset of the scope provided by its parent **parent**, and

2. **sd** *should have transient symbolic links to all the files in the scope provided by* **parent** *that satisfy* **sd***'s query, except for links that are explicitly prohibited in* **sd**.

Changes to **sd**'s scope can lead to a breakup of these invariants, a situation we call *scope-inconsistency*. This can happen, for example, whenever

1. a user modifies the set of symbolic links in **sd**'s parent **parent**,
2. a user moves **sd** toa different part of the file system,
3. there is a change in the scope of **parent**, or
4. a user changes the query of **sd** after he/she creates it. (HAC allows users to access and modify the query associated with a semantic directory.)

A major part of HAC is an algorithm to maintain scope consistency, which is briefly described below. First, HAC uses the CBA mechanism to re-evaluate **sd**'s query on all the files in its current scope. Then, from this result, HAC discards the links that occur in **sd**'s set of permanent and prohibited symbolic links. The links that remain are the new transient symbolic links of **sd**. 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 **sd**'s scope and matches its query. Similarly, HAC does not delete a permanent symbolic link from **sd** even if that link points to a file that is no longer in **sd**'s scope or does not match its query. Also note that HAC re-computes only the set of transient symbolic links of **sd** — HAC does not change the set of permanent or prohibited symbolic links associated with **sd**[1]. When the algorithm modifies the set of transient symbolic links in **sd**, it changes the scope provided by **sd**. Hence, the algorithm will also re-evaluate the queries of all the directories which directly or indirectly depend on **sd**. These are the descendents of **sd** and are present in the sub-tree rooted at **sd**. Any top-down traversal of this sub-tree (e.g., a breadth-first search) gives us the order in which we must re-evaluate the queries.

[1]HAC has special API routines to directly modify the set of permanent and prohibited symbolic links in semantic directories. Sophisticated users can use these routines to control the behavior of the scope consistency algorithm.

We decided to define the set of transient symbolic links in **sd** to be a refinement of the scope provided by its parent **parent**. We rejected the idea of defining this set to be, say, the union of the transient symbolic links in **sd** and the scope provided by all its children. (In this case, **sd** will depend on its children, not the other way round.) If we use this definition, users can never add a symbolic link **sl** to a child of **sd** such that **sl** does not automatically belong to the scope provided by **sd**. 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 **sd** to be the union of the transient symbolic links in **sd** 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.

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. HAC gives advice and help — users decide how to organize their file system.

2.4 Data Consistency

Users can create, remove, rename (move), or modify any data in the file system at will. There is therefore a possibility that the set of transient symbolic links in a semantic directory **sd** may not represent the current result of evaluating its query. This gives rise to a *data-inconsistency* problem. Data inconsistencies manifest themselves in the following ways: (i) A query result can contain an invalid link to a file that no longer exists, has been renamed, or has been modified so that it no longer satisfies the query, (ii) A query result may not contain a link to a new or modified file when it actually should. For example, new email that match the fingerprint query should be added to that semantic directory, and at the same time if a certain matching file has been moved to an area outside the scope of the query (e.g., it was deemed old and moved to archive), it should be removed from the semantic directory.

Though HAC removes *scope*-inconsistencies from the file system as soon as possible, HAC does not remove data-inconsistencies instantly. We could have adopted such a policy, similar to databases, but we believe that file systems typically do not require it, and the extra cost (determining when files have changed, re-indexing files automatically, etc.) will not warrant it. At present, HAC invokes the CBA mechanism to reindex the file system periodically (say, once a day or once an hour), determined by the user. At reindexing time, all scope and data consistencies are settled. HAC also allows users to initiate reindexing at any time, and for any part of the file system. So, for example, users can decide to update certain semantic directories as soon as new mail comes in, but not when an application modifies some files in the file system. In future, we plan to explore more sophisticated mechanisms to enforce data consistency in file systems.

2.5 Using Existing Results in New Queries

In addition to dependencies based on the hierarchy, HAC allow users to define arbitrary dependencies by adding directories names to queries. This gives users the power to combine query-language expressions (searching) with edited query results (browsing) in a very powerful way by specifying path names of existing directories (syntactic or semantic) as part of their queries. If the query of a semantic directory **new** contains the name of another semantic directory **old**, then we say that **new** *depends* on **old**, or **new** *refers* to **old**. Notice that dependencies are transitive. That is, if **old** depends on **ancient**, then **new** depends on **ancient**. When the CBA mechanism evaluates such a query, it can use HAC's API to determine which parts of the query contain path names of directories, and which parts contain search expressions. When it encounters the path name of a directory, it can use HAC's API to obtain the *existing* query-result (set of "pointers" to files) stored in that directory. Then, the CBA mechanism can operate on this set of pointers exactly as if it is the result of evaluating a search expression. In this way, users can easily augment the search capabilities of the CBA mechanism with their ability to customize information to their tastes.

One complication that arises here is that path-names can change when users rename directories. In the above example, if **old** is renamed as **old′**, **new**'s query is inconsistent since it refers to the old path name **old**. To solve this problem, HAC maintains a global mapping of unique identifiers to directory path-names, and stores only the identifers in actual queries. So, instead of updating the queries of all directories like **new** that depend on **old**, HAC simply updates the global map when **old** is renamed as **old′**.

When directory names are parts of queries, scope consistency becomes harder and trickier. For example, we must re-evaluate **new**'s query whenever the scope provided by **old** changes, even if **new** is **not** in the subtree rooted at **old**. We start with the definition of the *dependency graph* – it is the directed graph of dependencies between semantic directories. We do not allow cycles to exist in this graph for obvious reasons. Updating the query-results cannot be done in an arbitrary order. We must use the order obtained from a *topological sort* of the dependency graph. There is always a valid topological sort since this is a *Directed Acyclic Graph* (DAG). 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). Also note that there is no need for HAC to explicitly restrict the query result of a child directory to the scope provided by its parent. If users want this behavior, all they need to do is modify the query of the child directory to be: "<old query> **AND** <path-name of parent>" — HAC takes care of everything else. In fact, underneath the covers, this is exactly how we implement parent-child dependencies in the strict hierarchical scope consistency algorithm above. HAC gives users the freedom to chose whether they want strict hierarchical dependencies, DAG based dependencies, or both, interleaved arbitrarily.

Since we allow explicit scope consistency definitions along with implicit hierarchical scope consistency, one may argue that there is no need for the latter; we could leave it up to the user to specify the parent of a directory in its query. However, we feel that query-refinement based on path names is important for two reasons. First, tree-based classification of information is intuitive, and is sufficient for many real world scenarios. And second, most users may find tree-based classification simpler to understand because they do not have to worry about two structures — one based on path-names and one based on the dependency graph — at the same time.

3 Accessing Remote File and Query Systems

Another major benefit of HAC is its ability to cleanly access other HAC file systems, and other CBA mechanisms (possibly remote). In this section, we shall use *name space* to denote either a traditional file system (which provides path name-based access), a CBA mechanism, or a HAC file system. Connecting different file systems across a distributed system can be done with *mount points* [9, 10]. Mount points define new name spaces within which path names can be resolved. They allow different file systems to share certain directories so that they can access each other. HAC supports such mount points, which we call *syntactic mount points.* But we want to do more. We want to connect "semantically" so that we can evaluate queries against different name spaces, even if these name spaces *do not* allow us to organize information hierarchically (e.g., commercial search engines on the web). We want to allow users to use data from anywhere, create semantic directories anywhere, and in general treat the remote file systems and CBA mechanisms as if they were local. To achieve such rich, transparent connection, we must "decouple" the part of HAC that provides path name based access from the part that provides content based access, so that both can be used independently of each other. This is close to impossible to do if we restrict ourselves to syntactic mount points. We therefore introduced *semantic mount points* in HAC.

3.1 Semantic Mount Points

Let **Remote** be a remote file or query system. A semantic mount point **s.Remote** in a HAC file system **Local** connects queries within **Local** to results from **Remote**. Specifically, if the scope of a query within **Local** includes **s.Remote**, then it imports all the results asked within **Remote** with whatever query mechanism is used there. **s.Remote** provides an interface for *content-based* access to files in **Remote**. The power of a semantic mount point lies in the fact that the semantic directories created in it belong to the user's personal HAC file system, even though the symbolic links in these directories point to other (possibly remote) file systems. This allows users to create their own personal content-based classification of remote information. Furthermore, users can create physical files, semantic and syntactic directories, symbolic links, etc., as usual within semantic mount points. For instance, the physical files within a semantic mount point are indexed by HAC, and they can match queries of semantic directories created outside the subtree rooted at the mount point. This level of integration of name and content based access gives users a tremendous amount of power – they can extract exactly what they want and organize it in exactly the way they like. Previous semantic file systems, such as SFS or Nebula, do not allow such rich integration of query results and physical files, semantic directories and mount points like HAC.

3.2 Multiple Semantic Mount Points

Just as it is possible to mount more than one file system on a syntactic mount point [10], it is also possible to mount more than one name space on a semantic mount point. HAC treats each such name space as an independent entity. The scope of queries asked within a *multiple* semantic mount point is simply a union of the scope provided by each mounted name space. Queries are evaluated independently in each name space and their results are treated as disjoint sets of symbolic links. The only restriction is that all name spaces mounted on a multiple semantic mount point must be accessible via the same query language. (Currently, HAC does not deal with overlapping name spaces or data, i.e., it does not resolve cases where two symbolic links might actually point to the same remote file, or to similar files.)

For example, suppose that we want to cover **Remote** and **Local** at the same time. All we need to do is create another semantic mount point **s.Local** as a multiple semantic mount point! There is no problem of cyclic reference here, because **s.Local** is just an interface to a CBA mechanism; it does not provide CBA on its own.

Syntactic and semantic mount points can be combined in various ways to share information by both name and content. Getting back to our fingerprint example, we may have access to a digital library with scientific articles. We can add a semantic mount point associated with a query for "fingerprint" (or a more complex query), thus ensuring that our knowledge of the subject is up to date (at least with the library). There will probably be other sources as well, and we may need to form different queries depending on the source. We may want to have syntactic mount points to all these sources and search there manually once in a while, but in addition HAC allows a user to build remote semantic directories for each source (or one for all of them), and have a better access to and better integration with this information. For example, one can "remove" certain results of no interest, add comments, add results from other places, etc. Other users (e.g., coworkers on the same project) can use syntactic mount points to browse through one user's personal classification (instead of doing the searches themselves) and retrieve relevant information. It is also possible to collect the names, queries and query-results of many semantic directories of many users in a central database that *itself* can be indexed and searched. Users can browse and search this database and find others who have similar tastes as they have. This may help them find what they are looking for even more quickly. Finally, users can add their favorite books, articles, memoirs, shortcuts to other information, etc., to their personal HAC file systems, index and search them, and export their file systems as mini-digital libraries to others. To conclude: semantic mount points give us a powerful new way to access the semantic aspect of information. They can be combined with syntactic mount points to yield a rich set of primitives for sharing information in a distributed system.

4 Implementation and Performance

We implemented HAC on top of a UNIX file system (SunOS) using *Glimpse* as the default CBA mechanism [8]. Our prototype contains about 25,000 lines of C code. It was implemented as a dynamically linked library (DLL) which can be accessed by all user-level applications. *No kernel modifications were used.* This made the design easier to experiment with, easier to port, and easier to convince people to use it. The obvious disadvantage is a penalty in performance compared to a native UNIX file system, but as we will show, it is a small penalty. We start with a brief overview of the implementation.

HAC allows users to define their own personal name spaces (i.e., a personal file system). HAC uses this name space to resolve the users' path names and evaluate their queries. This name space exists within a directory in the UNIX file system. HAC intercepts all file system calls that access this directory or its contents, and provides a transparent interface to all applications. Well-known file system commands, such as **`cd, ls, mkdir, mv, rm`**, etc., can be used to access and manipulate objects in the file system in the usual way. HAC also provides additional commands that manipulate queries and semantic directories. These are for the most part intuitive extensions of regular file system commands. For example, `smkdir` creates a semantic directory, `smv` modifies the query of a directory and `sreadln` retrieves it, `scat` accepts a symbolic link in a semantic directory and returns the information in the corresponding file that matches the query of the directory, `smount` defines new syntactic and semantic mount points, and `ssync` re-evaluates the queries of all the directories that directly or indirectly depend on a given directory.

HAC interacts with UNIX using a well defined API which assumes very little about the native file system — HAC can be used even on "flat" file systems and file systems that do not support symbolic links. HAC interacts with Glimpse using another simple, well defined API. We believe that this API is general enough to integrate any CBA mechanism into HAC. Since HAC is a user-level file system, it does not contain any security and access-control features of its own: it borrows them from the underlying operating system.

We ran several experiments to determine the overhead to extended file system operations compared with regular file systems and/or regular glimpse queries. In the first experiment, we measured the overhead when we used HAC as a syntactic file system like UNIX and ran the Andrew Benchmark [11] on both systems. The Andrew Benchmark has been used as a standard to evaluate the performance of many new file systems. The benchmark has 5 phases: (i)**Makedir**: constructs a destination directory hierarchy that is identical to the source directory hierarchy, (ii)**Copy**: copies each file in the source hierarchy to the destination hierarchy, (iii)**Scan**: recursively traverses the whole destination hierarchy and examines the status of every file in the hierarchy without reading the actual data in the files, (iv)**Read**: reads every byte of every file in the destination hierarchy, and (v)**Make**: compiles and links the files in the destination hierarchy. The results for HAC are shown in tables 1 and 2 below:

From table 1, we see that phases 1 and 2 have the maximum overhead. 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. HAC keeps track of the name of this directory in a global map so that it can track changes to the structure of the file system. Finally, HAC creates a new (empty) node for the directory in the dependency graph. (All of these are stored in the disk and require extra I/O operations.) In phase 2, when HAC creates a new file, it also initializes the open file-descriptor and the attribute-cache for that file. (This is stored in UNIX *shared memory* so that different processes can access it.) This helps to speed up Scan and Read operations on that file. Phases 3 and 4 have a medium overhead. In phase 3, HAC accesses the attribute-cache to retrieve the appropriate status information, and in phase 4, HAC accesses and updates 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 2 HAC is only slightly slower than the Jade [10] and Pseudo [13] file systems (both of which are user-level file systems like HAC). We also calculated the space overhead to store HAC's data structures (the extra information needed for each directory mentioned above, along with a fixed amount of book-keeping information). In the example we used, HAC required 222 KB while UNIX needs 210 KB. This is about 5 % more. The average amount of shared memory needed per process

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

Table 1: Results of Andrew Benchmark

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

Table 2: Comparison with other File Systems

(including the attribute cache and the descriptor table) is about 16KB. Both these overheads are negligible. We believe that HAC's performance is quite reasonable since unlike the other two file systems, HAC must also create and maintain data-structures that provide content-based access to files.

In the second experiment, we first 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 Glimpse takes 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 3. We see that HAC has a 27 % time overhead and a 15 % space overhead: we believe that both of these 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. 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 (i) and (iii) are the most realistic — and the most useful.) The results are shown in Table 4.

For queries that matched very few files, Glimpse running on UNIX is more 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 to search files. While this may seem like a large overhead, in absolute terms it is very small. The overhead of creating a semantic directory 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 %.

Regarding the space overhead, note that we need to store, with each semantic directory, the list of files matching the query. Instead of storing actual file names, which could add quite a bit of space, we use a compact representation of the list of all file names. This is part of HAC's API for the CBA mechanism. We currently use bitmaps since it is simple to implement and has speed advantages for Glimpse. The extra space we need per semantic directory is therefore N/8 Bytes, where N is the number of indexed files. This comes out to be about 2 KB in this experiment. We plan to improve this in future by using better sparse-set representations, so that it is possible to index a very large number of files.

5 Related Work

The first hierarchical file system to provide both name and content based access to files was the MIT Semantic File System (SFS) [7]. SFS introduced the concept of a *virtual directory*. The name of a virtual directory in SFS is a query, and the contents are symbolic links to files that

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 3: Results of Indexing

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 4: Results of Searching

satisfy it. SFS assumes that queries are boolean **AND** combinations of "attribute-value" pairs, where an "attribute" is a typed field in the file system (e.g., "author:", "date:", etc.) and the "value" is a value this field can have (e.g., "John Doe", "3/12/97"). SFS always interprets the / path name separator between virtual directories as a conjunction operation. This feature can be used for query refinement.

SFS has many other novel features: (i) it caches the contents of different virtual directories to save query processing costs, (ii) it has special *transducer* programs that extract attributes and values from files in the file system to help with the indexing process (it also allows users to define their own transducers if necessary), and (iii) it has mechanisms to keep queries and their results consistent when there are changes to the files in the physical file system. However, SFS has some disadvantages. First, it assumes that queries are always conjunctions of attribute-value pairs, which makes it difficult to integrate arbitrary CBA mechanisms into the SFS. Second, virtual directories do not reside in the physical file system. Hence, users must use virtual directories to organize results of queries, but use real directories in the underlying file system to organize files. Third, SFS does not allow users to customize the results of queries according to their tastes without modifying queries or files in the file system. And finally, SFS does not provide a mechanism by which users can share their content-based classification of information with each other.

Other file systems follow in the footsteps of SFS. The Nebula File System [5] also assumes that files can be viewed as collections of attribute-value tuples. Queries in Nebula, however, can be arbitrary search expressions, not just boolean ANDs as in SFS. Nebula replaces the traditional idea of a fixed directory hierarchy by dynamic *views* of this hierarchy that can classify files in the underlying file system. A view is similar to a virtual directory: it has a query associated with it and contains pointers to files that satisfy the query. However, every view also has a "scope" which is defined to be a set of views. When Nebula evaluates the query of a view, it searches only those files which are referred to by the views in its scope. Nebula allows users to organize views in a DAG instead of a tree like SFS. Users can also alter the structure of this DAG by changing the scopes of views without changing their queries. This allows users to customize the contents of their views. Nebula has means to keep the contents of views consistent when there are changes to the data in the file system. It also allows users to share their views with each other. Though Nebula has many advantages, note that views are not a part of the underlying physical file system and cannot be used to organize data. Also note that Nebula does not allow users to group pointers to arbitrary files together and put them in a view: the files must satisfy the query associated with the view. Hence, users cannot modify results of queries to customize them according to their tastes.

Another example is the Multistructured naming system [12]. It tries to blend hierarchical or graph structured naming (e.g., the UNIX file system) with flat attribute or set based naming (e.g., SFS). It attempts to combine the "sense of *place*" present in graph-based naming with the ability of set-based naming to retrieve files using any combination of information about them. In this system, every query has a *label*, which is simply an alias for the query. Users can then impose "ancestor-descendent" (and other) relationships on labels, and selectively loosen these relationships, so that users can name files by specifying (i) either path names that contain labels, or (ii) a list of queries the files satisfy, or both, in arbitrary order. Multistructured naming allows users to access each others' personal name spaces and share information. Note, however, that it is not possible to group arbitrary files together and assign them a label. Like views in Nebula, a label must always have a query associated with it and can refer to only those files that satisfy this query. Hence, labels are not as powerful as directories in a regular file system. An important limitation of the above systems is that they do not provide a way to decouple name based access from content based access. This makes it difficult for a user to gather information from different CBA mechanisms (with possibly different query languages), and create a personal classification of this information using a *single* file system.

The Prospero file system [9] uses another approach: it allows each user to create his/her own personal graph-structured name space (called a *virtual file system*) that can refer to files in one or more existing graph-structured physical file systems. Users can also access the name spaces of other users. In both virtual and physical file systems, "nodes" are directories and contain files or pointers to other (virtual or physical) files, while "links" are used to connect nodes with each other. The novelty of Prospero is that users can associate *filters* with links in their virtual file systems. A Filter is an arbitrary program that can alter users' perception of the contents of the directory (node) 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 its input, but 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, so that they can 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 and derive views that classify information according to their personal tastes. Prospero's filters, therefore, are powerful tools for information retrieval. Their only drawback 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 consistency.

The Synopsis File System ([2], [3]) provides a secure access mechanism to retrieve and manipulate large amounts of data within a wide-area file system. It hides the heterogeneity of data behind a logical interface to information based on typed *synopses*. Each synopsis is an *object* that encapsulates information about a single file in the form of *attributes* indexed for fast search and retrieval. The extensible type system allows users to define *methods* on each synopsis for customized display, access and manipulation of the synopsis content and the associated file. Since a synopsis is an object, its attributes and methods can also be inherited (composed) from other those of other synopses. A collection of synopses can be combined into a *digest*, that provides topic-based searches. Synopses and digests together can make content based access over very large heterogenous file systems more meaningful. Together, they can form the basis for locating and organizing information.

Like Nebula, the Synopsis File System introduces new abstractions to encapsulate information based on content. And like Prospero, it allows users to define how they want to organize and manipulate this information. However, it does not define how a user's hierarchical organization of information is kept consistent when the structure of the hierarchy changes (e.g., what happens if you interchange child and parent synopses in the synopsis hierarchy). That is, consistency criteria are specific to each synopsis object – not to the Synopsis File System as a whole. HAC, on the other hand, defines and enforces a "global" consistency criteria based on the hierarchy, and fully integrates path-name and content-based access in a file system. Though their basic approaches are different, we believe that HAC and the Synopsis File System can be used in conjunction to yield a very powerful tool for information retrieval.

There are several other systems that address related issues [9, 4, 6]. 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 the 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. We believe that the HAC file system meets both these needs.

6 Conclusions

We have shown in this paper that it is possible, with reasonable overhead, to combine name-based and content-based access to files at the same time, while preserving the main benefits of both methods. We identified several scope and consistency problems, suggested solutions, and described an implementation. This is obviously not the last word on this topic. More work needs to be done to convince people to add a major paradigm to their daily arsenal of tools. A major missing piece is usability testing, which we have not performed to date. Only when a working version is widely distributed and used can we determine how beneficial is this approach.

The implementation described in the paper was geared towards personal file systems, and as is it is not scalable to very large file systems (e.g., Internet wide). This is true for many existing file systems. In particular, our decision not to modify anything at the kernel level (so the system will be easier to distribute) adds quite a bit to the overhead. Implementing our ideas in the context of a large scale Intranet, for example, will be a major next step. We believe it is possible and very desirable. The current situation of server-based search facilities and user-based file systems with almost no connection between them can be and should be improved.

References

[1] 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.

[2] M. Bowman and R. John, "The Synopsis File System: From Files to File Objects," In *Workshop on Distributed Object and Mobile Code*, Boston, MA, June, 1996.

[3] Mic Bowman, "Managing Diversity in Wide-Area File Systems". *Second IEEE Metadata Conference*, Silver Spring, Maryland. September 1997.

[4] M. Bowman, P. Danzig, D. Hardy, U. Manber, and M. Schwartz, The Harvest information discovery and access system, *Computer Networks and ISDN Systems*, **28** (1995), pp. 119-125.

[5] 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.

[6] 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.

[7] D. Gifford, P. Jouvelot, M. Sheldon, and Jr J. O'Toole. Semantic file systems. In *Proc. 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA (October 1991), pp. 16-25.

[8] U. Manber and S. Wu. Glimpse: A tool to search through entire file systems. In *Usenix Winter 1994 Technical Conference*, San Francisco (January 1994), pp. 23-32.

[9] 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.

[10] 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.

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

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

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