Semantic File Systems

David K. Gifford, Pierre Jouvelot [1],
Mark A. Sheldon, James W. O'Toole, Jr.

Programming Systems Research Group
MIT Laboratory for Computer Science

Abstract

A *semantic file system* is an information storage system that provides flexible associative access to the system's contents by automatically extracting attributes from files with file type specific *transducers*. Associative access is provided by a conservative extension to existing tree-structured file system protocols, and by protocols that are designed specifically for content based access. Compatibility with existing file system protocols is provided by introducing the concept of a *virtual directory*. Virtual directory names are interpreted as queries, and thus provide flexible associative access to files and directories in a manner compatible with existing software. Rapid attribute-based access to file system contents is implemented by automatic extraction and indexing of key properties of file system objects. The automatic indexing of files and directories is called "semantic" because user programmable transducers use information about the semantics of updated file system objects to extract the properties for indexing. Experimental results from a semantic file system implementation support the thesis that semantic file systems present a more effective storage abstraction than do traditional tree structured file systems for information sharing and command level programming.

1 Introduction

We would like to develop an approach for information storage that both permits users to share information more effectively and provides reductions in programming effort and program complexity. To be effective this new approach must be used and thus an approach that provides a transition path from existing file systems is desirable.

In this paper we explore the thesis that *semantic file systems* present a more effective storage abstraction than do traditional tree structured file systems for information sharing and command level programming. A semantic file system is an information storage system that provides flexible associative access to the system's contents by automatically extracting attributes from files with file type specific *transducers*. Associative access is provided by a conservative extension to existing tree-structured file system protocols, and by protocols that are designed specifically for content based access. Automatic indexing is performed when files or directories are created or updated.

The automatic indexing of files and directories is called "semantic" because user programmable transducers use information about the semantics of updated file system objects to extract the properties for indexing. Through the use of specialized transducers, a semantic file system "understands" the documents, program object code, mail, images, name service databases, bibliographies, and other files contained by the system. For example, the transducer for a C program could extract the names of the procedures that the program exports or imports, procedure types, and the files included by the program. A semantic file system can be extended easily by users through the addition of specialized transducers.

Associative access is designed to make it easier for users to share information by helping them discover and locate programs, documents, and other relevant objects. For example, files can be located based upon transducer generated attributes such as author, exported or imported procedures, words contained, type, and title.

A semantic file system provides both a user interface and an application programming interface to its associative access facilities. User interfaces based upon browsers [Inf90, Ver90] have proven to be effective for query based access to information, and we expect browsers to be offered by most semantic file system implementations. Application programming interfaces that permit remote access include specialized protocols for information retrieval [NIS91], and remote procedure call based interfaces [GGS87].

It is also possible to export the facilities of a semantic file system without introducing any new interfaces. This can be accomplished by extending the naming semantics of files and directories to support associative access. A benefit of this approach is that all existing applications, including user interfaces, immediately inherit the benefits of associative access.

A semantic file system integrates associative access into a tree structured file system through the concept of a *virtual directory*. Virtual directory names are interpreted as queries and thus provide flexible associative access to files and directories in a manner compatible with existing software.

For example, in the following session with a semantic

This research was funded by the Defense Advanced Research Projects Agency of the U.S. Department of Defense and was monitored by the Office of Naval Research under grant number N00014-89-J-1988.

[1] Also with CRI, Ecole des Mines de Paris, France.

file system we first locate within a library all of the files that export the procedure `lookup_fault`, and then further restrict this set of files to those that have the extension `.c`:

```
% cd /sfs/exports:/lookup_fault
% ls -F
virtdir_query.c@        virtdir_query.o@
% cd ext:/c
% ls -F
virtdir_query.c@
%
```

Semantic file systems can provide associative access to a group of file servers in a distributed system. This distributed search capability provides a simplified mechanism for locating information in large wide-area file systems.

Semantic file systems should be of use to both individuals and groups. Individuals can use the query facility of a semantic file system to locate files and to provide alternative views of data. Groups of users should find semantic file systems an effective way to learn about shared files and to keep themselves up to date about the status of group projects. As workgroups increasingly use file servers as shared library resources we expect that semantic file system technology will become even more useful.

Because semantic file systems are compatible with existing tree-structured file systems, implementations of semantic file systems can be fully compatible with existing network file system protocols such as NFS [SCK+85, Sun88] and AFS [Kaz88]. NFS compatibility permits existing client machines to use the indexing and associative access features of a semantic file system without modification. Files stored in a semantic file system via NFS will be automatically indexed and query result sets will appear as virtual directories in the NFS name space. This approach directly addresses the "dusty data" problem of existing UNIX file systems by allowing existing UNIX file servers to be converted transparently to semantic file systems.

We have built a prototype semantic file system and run a series of experiments to test our thesis that semantic file systems present a more effective storage abstraction than do traditional tree-structured file systems for information sharing and command-level programming. We tried to locate various documents and programs in the file system using unmodified NFS clients. The results of these experiments suggest that semantic file systems can be used to find information more quickly than is possible using ordinary file systems, and add expressive power to command-level programming languages.

In the remainder of the paper we discuss previous research (Section 2), introduce the interface and semantics for a semantic file system (Section 3), review the design and implementation of a semantic file system (Section 4), present our experimental results (Section 5) and conclude with observations on other applications of virtual directories (Section 6).

2 Previous Work

Associative access to on-line information was pioneered in early bibliographic retrieval systems where it was found to be of great value in locating information in large databases [Sal83]. The utility of associative access motivated its subsequent application to file and document management. The previous research we build upon includes work on personal computer indexing systems, information retrieval systems, distributed file systems, naming models for file systems, and wide-area naming systems.

- Personal computer indexing systems such as On Location [Tec90], Magellan [Gri], and the Digital Librarian [Mos89b, Mos89a] provide window-based file system browsers that permit word-based associative access to file system contents. Magellan and the Digital Librarian permit searches based upon boolean combinations of words, while On Location is limited to conjunctions of words. All three systems rank matching files using a relevance score. These systems all create indexes to reduce search time. On Location automatically indexes files in the background, while Magellan and the Digital Librarian require users to explicitly create indexes. Both On Location and the Digital Librarian permit users to add appropriate keyword-generation programs [Cla90, Mos89b] to index new types of files. However, Magellan, On Location, and the Digital Librarian are limited to a list of words for file description.

- Information retrieval systems such as Basis [Inf90], Verity [Ver90], and Boss DBMS [Log91] extend the semantics of personal computer indexing systems by adding field-specific queries. Fields that can be queried include document category, author, type, title, identifier, status, date, and text contents. Many of these document relationships and attributes can be stored in relational database systems that provide a general query language and support application programmaccess. The WAIS system permits information at remote sites to be queried, but relies upon the user to choose an appropriate remote host from a directory of services [KM91, Ste91]. Distributed information retrieval systems [GGS87, DAN91] perform query routing based upon database content labels to ensure that all relevant hosts are contacted in response to a query.

- Distributed file systems [Sun89, Kaz88] provide remote access to files with tree-structured names. These systems have enabled file sharing among groups of people and over wide geographic areas. Existing UNIX tools such as `grep` and `find` [Gro86] are often used to perform associative searches in distributed file systems.

- Naming models for file systems include the Portable Common Tool Environment (PCTE) [GMR86], the Property List DIRectory system (PLDIR) [Mog86], Virtual Systems [Nic90] and Siris Network Software Environment (NSE) [SS88]. PCTE provides an entity-relationship database that models the attributes of objects including files. PCTE has been implemented as a compatible extension to UNIX. However, PCTE users must use specialized tools to query the PCTE database, and thus do not receive the benefits of associative access via a file system interface. The Property List DIRectory system implements a file system model designed around file properties and offers a Unix front-end user interface. Similarly, Virtual Systems permit users to hand-craft customized views of services, files, and directories. However, neither system provides automatic attribute extraction (although [Mog86] alludes to it as a possible extension) or attribute-based access to their contents. NSE is a network transparent software development tool that allows different views of

a file system hierarchy called *environments* to be defined. Unlike virtual directories, these views must be explicitly created before being accessed.

- Wide-area naming systems such as X.500 [CCI88], Profile [Pet88], and the Networked Resource Discovery Project [Sch89] provide attribute-based access to a wide variety of objects, but they are not integrated into a file system nor do they provide automatic attribute-based access to the contents of a file system.

Key advances offered by the present work include:

- Virtual directories integrate associative access into existing tree structured file systems in a manner that is compatible with existing applications.
- Virtual directories permit unmodified remote hosts to access the facilities of a semantic file system with existing network file system protocols.
- Transducers can be programmed by users to perform arbitrary interpretation of file and directory contents in order to produce a desired set of field-value pairs for later retrieval. The use of fields allows transducers to describe many aspects of a file, and thus permits subsequent sophisticated associative access to computed properties. In addition, transducers can identify entities within files as independent objects for retrieval. For example, individual mail messages within a mail file can be treated as independent entities.

Previous research supports our view that overloading file system semantics can improve system uniformity and utility when compared with the alternative of creating a new interface that is incompatible with existing applications. Examples of this approach include:

- Devices in UNIX appear as special files [RT74] in the `/dev` directory, enabling them to be used as ordinary files from UNIX applications.
- UNIX System III named pipes [Roc85, p. 159f] appear as special files, enabling programs to rendezvous using file system operations.
- File systems appear as special directories in Automount daemon directories [CL89, Pen90, PW90], enabling the binding of a name to a file system to be computed at the time of reference.
- Processes appear as special directories in Killian's process file system [Ken84], enabling process observation and control via file operations.
- Services appear as special directories in Plan 9 [PPTT90], enabling service access in a distributed system through file system operations in the service's name space.
- Arbitrary semantics can be associated with files and directories using Watchdogs [BP88], Pseudo Devices [WO88], and Filters [Neu90], enabling file system extensions such as terminal drivers, network protocols, X servers, file access control, file compression, mail notification, user specific directory views, heterogeneous file access, and service access.
- The ATTIC system [CG91] uses a modified NFS server to provide transparent access to automatically compressed files.

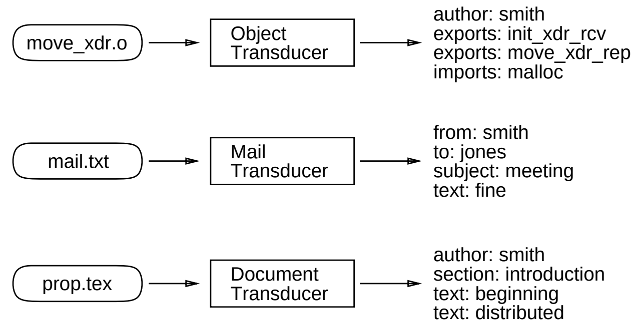

Figure 1: Sample Transducer Output

3 Semantic File System Semantics

Semantic file systems can implement a wide variety of semantics. In this section we present one such semantics that we have implemented. Section 6 describes some other possibilities.

Files stored in a semantic file system are interpreted by file type specific transducers to produce a set of descriptive attributes that enable later retrieval of the files. An *attribute* is a *field-value* pair, where a *field* describes a property of a file (such as its author, or the words in its text), and a *value* is a string or an integer. A given file can have many attributes that have the same field name. For example, a text file would have as many `text:` attributes as it has unique words. By convention, field names end with a colon.

A user extensible *transducer table* is used to determine the transducer that should be used to interpret a given file type. One way of implementing a transducer table is to permit users to store subtree specific transducers in the subtree's parent directory, and to look for an appropriate transducer at indexing time by searching up the directory hierarchy.

To accommodate files (such as mail files) that contain multiple objects we have generalized the unit of associative access beyond whole files. We call the unit of associative access an *entity*. An entity can consist of an entire file, an object within a file, or a directory. Directories are assigned attributes by directory transducers.

A transducer is a filter that takes as input the contents of a file, and outputs the file's entities and their corresponding attributes. A simple transducer could treat an input file as a single entity, and use the file's unique words as attributes. A complex transducer might perform type reconstruction on an input file, identify each procedure as an independent entity and use attributes to record their reconstructed types. Figure 1 shows examples of an object file transducer, a mail file transducer, and a TeX file transducer.

The semantics of a semantic file system can be readily extended because users can write new transducers. Transducers are free to use new field names to describe special attributes. For example, a CAD file transducer could introduce a `drawing:` field to describe a drawing identifier.

The associative access interface to a semantic file system is based upon queries that describe desired attributes of entities. A *query* is a description of desired attributes that permits a high degree of selectivity in locating entities of interest. The result of a query is a set of files and/or directories that contain the entities described. Queries are

boolean combinations of attributes, where each attribute describes the desired value of a field. It is also possible to ask for all of the values of a given field in a query result set. The values of a field can be useful when narrowing a query to eliminate entities that are not of interest.

A semantic file system is *query consistent* when it guarantees query results that correspond to its current contents. If updates cease to the contents of a semantic file system it will eventually be query consistent. This property is known as convergent consistency. The rate at which a given implementation converges is administratively determined by balancing the user benefits of fast convergence when compared with the higher processing cost of indexing rapidly changing entities multiple times. It is of course possible to guarantee that a semantic file system is always query consistent with appropriate use of atomic actions.

In the remainder of this section we will explore how conjunctive queries can be mapped into tree-structured path names. As we mentioned earlier, this is only one of the possible interfaces to the query capabilities of a semantic file system. It is also possible to map disjunction and negation into tree-structured names, but they have not been implemented in our prototype and we will not discuss them.

Queries are performed in a semantic file system through use of virtual directories to describe a desired view of file system contents. A virtual directory is computed on demand by a semantic file system. From the point of view of a client program a virtual directory is indistinguishable from an ordinary directory. However, unlike ordinary directories, virtual directories do not have to be explicitly created to be accessed.

The query facilities of a semantic file system appear as virtual directories at each level of the directory tree. A *field virtual directory* is named by a field, and has one entry for each possible value of its corresponding field. Thus in `/sfs`, the virtual directory `/sfs/owner:` corresponds to the `owner:` field. The field virtual directory `/sfs/owner:` would have one entry for each owner that has written a file in `/sfs`. For example:

```
% ls -F /sfs/owner:
jones/          root/           smith/
%
```

The entries in a field virtual directory are value virtual directories. A *value virtual directory* has one entry for each entity described by a field-value pair. Thus the value virtual directory `/sfs/owner:/smith` contains entries for files in `/sfs` that are owned by Smith. Each entry is a symbolic link to the file. For example:

```
% ls -F /sfs/owner:/smith
bio.txt@        paper.tex@      prop.tex@
%
```

When an entity is smaller than an entire file, a view of the file can be presented by extending file naming semantics to include view specifications. To permit the conjunction of attributes in a query, value virtual directories contain field virtual directories. For example:

```
% ls -F /sfs/owner:/smith/text:/resume
bio.txt@
%
```

A pleasant property of virtual directories is their synergistic interaction with existing file system facilities. For example, when a symbolic link names a virtual directory the link describes a computed view of a file system. It is also possible to use file save programs such as `tar`, on virtual directories to save a computed subset of a file system. It would be possible also to generalize virtual directories to present views of file systems with respect to a certain time in the past.

A semantic file system can be overlaid on top of an ordinary file system allowing all file system operations to go through the SFS server. The overlaid approach has the advantage that it provides the power of a semantic file system to a user at all times without the need to refer to a distinguished directory for query processing. It also allows the server to do indexing in response to file system mutation operations. Alternatively, a semantic file system may create virtual directories that contain links to the files in the underlying file system. This means that subsequent client operations bypass the semantic file system server.

When an overlaid approach is used field virtual directories must be invisible to preserve the proper operation of tree traversal applications. A directory is *invisible* when it is not returned by directory enumeration requests, but can be accessed via explicit lookup. If field virtual directories were visible, the set of trees under `/sfs` in our above example would be infinite. Unfortunately making directories invisible causes the UNIX command `pwd` to fail when the current path includes an invisible directory. It is possible to fix this through inclusion of unusual `..` entries in invisible directories.

The distinguished `field:` virtual directory makes field virtual directories visible. This permits users to enumerate possible search fields. The `field:` directory is itself invisible. For example:

```
% ls -F /sfs/field:
author:/     exports:/   owner:/     text:/
category:/   ext:/       priority:/  title:/
date:/       imports:/   subject:/   type:/
dir:/        name:/
% ls -F /sfs/field:/text:/semantic/owner:/jones
mail.txt@        paper.tex@      prop.tex@
%
```

The syntax of semantic file system path names is:

```
<sfs-path>    ::=  /<pn> | <pn>
<pn>          ::=  <name> | <attribute>
                   <field-name> | <name>/<pn>
                   <attribute>/<pn>
<attribute>   ::=  field: | <field-name>/<value>
<field-name>  ::=  <string>:
<value>       ::=  <string>
<name>        ::=  <string>
```

The semantics of semantic file system path names is:

- The universe of entities is defined by the path name prefix before the first virtual directory name.
- The contents of a field virtual directory is a set of value virtual directories, one for each value that the field describes in the universe.

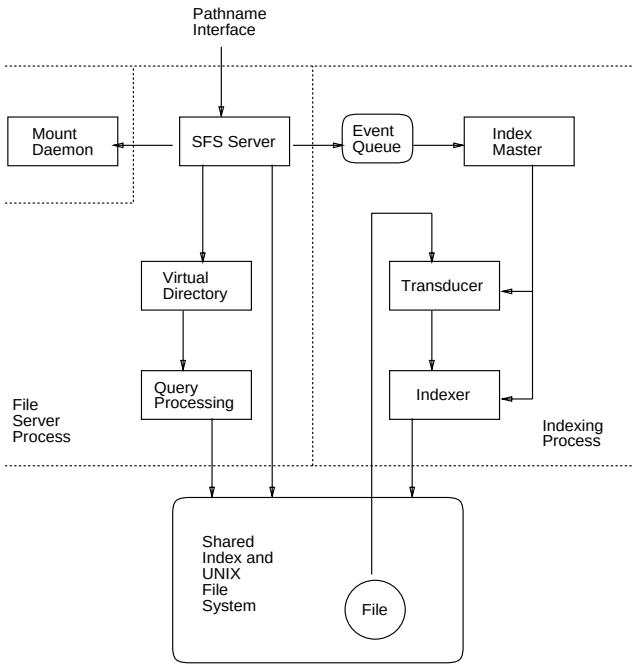

Figure 2: SFS Block Diagram

- The contents of a value virtual directory is a set of entries, one for each entity in the the universe that has the attribute described by the name of the value virtual directory and its parent field virtual directory. The contents of a value virtual directory defines the universe of entities for its subdirectories. In the absence of name conflicts, the name of an entry in a value virtual directory is its original entry name. Entry name conflicts are resolved by assigning more names to entries.

- The contents of a `field:` virtual directory is the set of fields in use.

4 Semantic File System Implementation

We have built a semantic file system that implements the NFS [SGK+85, Sun89] protocol as its external interface. To use the search facilities of our semantic file system an Internet client can simply mount our file system at a desired point and begin using virtual directory names. Our NFS server computes the contents of virtual directories as necessary in response to NFS `lookup` and `readdir` requests.

A block diagram of our implementation is shown in Figure 2. The dashed lines in the figure describe process boundaries. The major processes are:

- The *client* process is responsible for generating file system requests using normal NFS style path names.

- The *file server process* is responsible for creating virtual directories in response to path name based queries. The SFS Server module implements a user level NFS server and is responsible for implementing the NFS interface to the system. The SFS Server uses *directory faults* to request computation of needed entries by the Virtual Directory module. A faulting mechanism is used because the SFS Server caches virtual directory results, and will only fault when needed information is requested the first time or is no longer cached. The Virtual Directory module in turn calls the Query Processing module to actually compute the contents of a virtual directory.

  The file server process records file system modification events in a write-behind log. The modification log eliminates duplicate modification events.

- The *indexing process* is responsible for keeping the index of file system contents up-to-date. The Index Master module examines the modification log generated by the file server process every two minutes. The indexing process responds to a file system modification event by choosing an appropriate transducer for the modified object. An appropriate transducer is selected by determination of the type of the object (e.g. C source file, object file, directory). If no special transducer is found a default transducer is used. The output of the transducer is fed to the Indexer module that inserts the computed attributes into the index. Indexing and retrieval are based upon Peter Weinberger's PStore package [Wei] and an adapted version of the `refer` [Les] software to maintain the mappings between attributes and objects.

- The *mount daemon* is contacted to determine the root file handle of the underlying UNIX file system. The file server process exports its NFS service using the same root file handle on a distinct port number.

- The *kernel* implements a standard file system that is used to store the shared index. The file server process could be integrated into the kernel by a VFS based implementation [Kle86] of an semantic file system. We chose to implement our prototype using a user level NFS server to simplify development.

Instead of computing all of the virtual directories that are present in a path name, our implementation only computes a virtual directory if it is enumerated by a client `readdir` request or a `lookup` is performed on one of its entries. This optimization allows the SFS Server to postpone query processing in the hope that further attribute specifications will reduce the amount of work necessary for computation of the result set. This optimization is implemented as follows:

- The SFS Server responds to a `lookup` request on a virtual directory with a `lookup_not_found` fault to the Virtual Directory module. The Virtual Directory module checks to make sure that the virtual directory name is syntactically well formed according to the grammar in Section 3. If the name is well formed, the directory fault is immediately satisfied by calling the `create_dir` procedure in the SFS Server. This procedure creates a placeholder directory that is used to satisfy the client's original `lookup` request.

- The SFS Server responds to a `readdir` request on a virtual directory or a `lookup` on one of its entries with a `fill_directory` fault to the Virtual Directory module. The Virtual Directory module collects all of the attribute specifications in the virtual directory path

name and passes them to the Query Processing module. The Query Processing module uses simple heuristics to reorder the processing of attributes to optimize query performance. The matching entries are then materialized in the placeholder directory by the Virtual Directory module that calls the `create_link` procedure in the SFS Server for each matching file or directory.

The transducers that are presently supported by our semantic file system implementation include:

- A transducer that describes New York Times articles with `type:`, `priority:`, `date:`, `category:`, `subject:`, `title:`, `author:`, and `text:` attributes.
- A transducer that describes object files with `exports:` and `imports:` attributes for procedures and global variables.
- A transducer that describes C, Pascal, and Scheme source files with `exports:` and `imports:` attributes for procedures.
- A transducer that describes mail files with `from:`, `to:`, `subject:`, and `text:` attributes.
- A transducer that describes text files with `text:` attributes. The text file transducer is the default transducer for ASCII files.

In addition to the specialized attributes listed above, all files and directories are further described by `owner`, `group`, `dir`, `name`, and `ext` attributes.

At present, we only index publicly readable files. We are investigating indexing protected files as well, and limiting query results to entities that can be read by the requester. We are in the process of making a number of improvements to our prototype implementation. These enhancements include 1) full support for multi-host queries using query routing, 2) an enhanced query language, 3) better support for file deletion and renaming, and 4) integration of views for entities smaller than files. Our present implementation deals with deletions by keeping a table of deleted entities and removing them from the results of query processing. Entities are permanently removed from the database when a full reindexing of the system is performed. We are investigating performing file and directory renames without reindexing the underlying files.

5 Results

We ran a series of experiments using our semantic file system implementation to test our thesis that semantic file systems present a more effective storage abstraction than do traditional tree structured file systems for information sharing and command level programming. All of the experimental data we report are from our research group's file server using a semantic file system. The server is a Micron 3 running UNIX version 4.3bsd. The server indexes all of its publicly readable files and directories.

To compact the indexes our prototype system reconstructs a full index of the file system contents every week. On 23 July 1991, full indexing of our user file system processed 68 MBytes in 7,771 files (Table 5).[1] Indexing the resulting 1 million attributes took 1 hour and 36 minutes (Table 2). This works out to an indexing rate of 712 KBytes/minute.

File system mutation operations trigger incremental indexing. In update tests simulating typical user editing and compiling, incremental indexing is normally completed in less than 5 minutes. In these tests, only 2 megabytes of modified file data were reindexed. Incremental indexing is slower than full indexing in the prototype system because the incremental indexer does not make good use of real memory for caching. The full indexer uses 10 megabytes of real memory for caching; the incremental indexer uses less than 1 megabyte.

The indexing operations of our prototype are I/O bound. The CPU is 60% idle during indexing. Our measurements show that transducers generate approximately 30 disk transfers per second, thereby saturating the disk. Indexing the resulting attributes also saturates the disk. Although the transducers and the indexer use different disk drives, the transducer-indexer pipeline does not allow I/O operations to proceed in parallel on the two disks. This, we feel that we could double the throughput by improving the pipeline's

Total file system size	326 MBytes
Amount publicly readable	230 MBytes
Amount with known transducer	68 MBytes
Number of distinct attributes	173,075
Number of attributes indexed	1,042,832

Type	Number of Files	KBytes
Object	871	8,503
Source	2,735	17,991
Text	1,871	20,638
Other	2,274	21,187
Total	7,771	68,319

Table 1: User File System Statistics for 23 July 1991

Part of index	Size in KBytes
Index Tables	6,621
Index Trees	3,398
Total	10,019

Phase	Time (hr:mn)
Directory Enumeration	0:07
Determine File Types	0:01
Transduce Directory	0:01
Transduce Object	0:08
Transduce Source	0:23
Transduce Text	0:23
Transduce Other	0:24
Build Index Tables [2]	1:22
Build Index Trees	0:06
Total	1:36

Table 2: User IFS Indexing Statistics on 23 July 1991

[1] The 162 MBytes in publicly readable files that were not processed were in files for which transducers have not yet been written: executable files, PostScript files, DVI files, tar files, image data, etc.

[2] in parallel with Transduce

structure.

We expect our indexing strategy to scale to larger file systems because indexing is limited by the update rate to a file system rather than its total storage capacity. Incremental processing of updates will require additional read bandwidth approximately equal to the write traffic that actually occurs. Past studies of Unix file system activity [OCH+85] indicate that update rates are low and that most new data is deleted or overwritten quickly; thus, delaying slightly the processing of updates might reduce the additional bandwidth required by indexing.

To determine the increased latency of overlaid NFS operations introduced by interposing our SFS server between the client and the native file system we used the `nhfsstone` benchmark [Leg89] at low loads. The delays observed from an unmodified client machine were smaller than the variation in latencies of the native NFS operations. Preliminary measurements show that `lookup` operations are delayed by 2 msec on average, and operations that generate update notifications incur a larger delay.

The following anecdotal evidence supports our thesis that a semantic file system is more effective than traditional file systems for information sharing.

- The typical response time for the first `ls` command on a virtual directory is approximately 2 seconds. This response time reflects a substantial time savings over linear search through our entire file system with existing tools. In addition, subsequent `ls` commands respond immediately with cached results.

  We ran a series of experiments to test how the number of attributes in a virtual directory name altered the observed performance of the `ls` command on a virtual directory. Attributes were added one at a time to arrive at the final path name

  ```
  /sfs/text:/virtual/
       text:/directory/
       text:/semantic/
       ext:/tex/
       owner:/gifford
  ```

  The two properties of a query that affect its response time are the number of attributes in the query and the number of objects in the result set. The effect of an increase in either of these factors is additional disk accesses. Figure 3 illustrates the interplay of these factors. Each point on the response time graph is the average of three experiments. In a separate experiment we measured an average response time of 5.4 seconds when the result set grew to 545 entities.

- We began to use the semantic file system as soon as it was operable to help coordinate the production of this paper and for a variety of other everyday tasks. We have found the virtual directory interface to be easy to use. (We were immediately able to use the GNU Emacs directory editor DIRED [Sta87] to submit queries and browse the results. No code modification was required.) At least two users in our group reexamined their file protections in view of the ease with which other users could locate interesting files in the system.

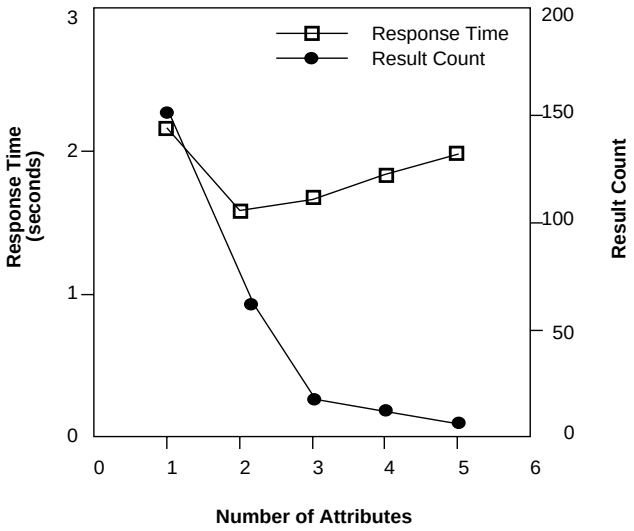

Figure 3: Plot of Number of Attributes vs. Response Time and Number of Results

- Users outside our research group have successfully used the query interface to locate information, including newspaper articles, in our file system.

- Users outside our research group have failed to find files for which no transducer had yet been installed. We are developing new transducers in response to these failed queries.

The following anecdotal evidence supports our thesis that a semantic file system is more effective than traditional file systems for command level programming.

- The UNIX shell pathname expansion facilities integrate well with virtual directories. For example, it is possible to query the file system for all `dvi` files owned by a particular user, and to print those whose names begin with a certain sequence of characters.

- Symbolic links have proven to be an effective way to describe file system views. The result of using such a symbolic link as a directory is a dynamically computed set of files.

6 Conclusions

We have described how a semantic file system can provide associative attribute-based access to the contents of an information storage system with the help of file type specific transducers. We have also discussed how this access can be integrated into the file system itself with virtual directories. Virtual directories are directories that are computed upon demand.

The results to date are consistent with our thesis that semantic file systems present a more effective storage abstraction than do traditional tree structured file systems for information sharing and command level programing. We plan to conduct further experiments to explore this thesis in further detail. We plan also to examine how virtual directories can directly benefit application programs.

Our experimental system has tested one semantics for virtual directories, but there are many other possibilities. For example:

- The virtual directory syntax can be extended to support a richer query language. Disjunctive queries would permit users to use "or" in their queries, and would also offer the ability to search on multiple network semantic file systems concurrently.
- Users could assign attributes to file system entities in addition to the attributes that are automatically assigned by transducers.
- Transducers could be created for audio and video files. In principle this would permit access by time, frame number, or content [Nee91].
- The data model underlying a semantic file system could be enhanced. For example, an entity-relationship model [Cat83] would provide more expressive power than simple attribute based retrieval.
- The entities indexed by a semantic file system could include a wide variety of object types, including I/O devices and file servers. Wide-area naming systems such as X.500 [CCI88] could be presented in terms of virtual directories.
- A confederation of semantic file systems, possibly numbering in the thousands, can be organized into a *semantic library system*. A semantic library system exports the same interface as an individual semantic file system, and thus a semantic library system permits associative access to the contents of its constituent servers with existing file system protocols as well as with protocols that are designed specifically for content based access. A semantic library system is implemented by servers that use content based routing [GLB85] to direct a single user request to one or more relevant semantic file systems.

  We have already completed the implementation of an NFS-compatible query processing system that forwards requests to multiple hosts and combines the results.
- Virtual directories can be used as an interface to other systems, such as information retrieval systems and programming environment support systems, such as PCTE. We are exploring also how existing applications could access object repositories via a virtual directory interface. It is possible to extend the semantics of a semantic file system to include access to individual entities in a manner suitable for an object repository [GO91].
- Relevance feedback and query results could be added by introducing new virtual directories.

The implementation of real-time indexing may require a substantial amount of computing power at a semantic file server. We are investigating how to optimize the task of real-time indexing in order to minimize this load. Another area of research is exploring how massive parallelism [SK86] might replace indexing.

An interesting limiting case of our design is a system that makes an underlying tree structured naming system superfluous. In such a system all directories would be computed upon demand, including directories that correspond to traditional tree structured file names. Such a system might help us share information more effectively by encouraging query based access that would lead to the discovery of unexpected but useful information.

Acknowledgments

We would like to thank Doug Grundman, Andrew Myers, and Raymie Stata, for their various contributions to the paper and the implementation. The referees provided valuable feedback and concrete suggestions that we have endeavored to incorporate into the paper. In particular, we very much appreciate the useful and detailed comments provided by Mike Burrows.

References

[BP88] Brian N. Bershad and C. Brian Pinkerton. Watchdogs: Extending the UNIX file system. In *USENIX Association 1988 Winter Conference Proceedings*, pages 267–275, Dallas, Texas, February 1988.

[Cat83] R. G. G. Cattell. Design and implementation of a relationship-entity-datum data model. Technical Report CSL-83-4, Xerox PARC, Palo Alto, California, May 1983.

[CCI88] CCITT. The Directory - Overview of Concepts, Models and Services. Recommendation X.500, 1988.

[CG91] Vincent Cate and Thomas Gross. Combining the concepts of compression and caching for a two-level filesystem. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 200–211, Santa Clara, California, April 1991. ACM.

[CL89] Brent Callaghan and Tom Lyon. The automounter. In *USENIX Association 1989 Winter Conference Proceedings*, 1989.

[Cla90] Claris Corporation, Santa Clara, California, January 1990. News Release.

[Cor] Lotus Corporation. Lotus Magellan: Quick Launch. Product tutorial, Lotus Corporation, Cambridge, Massachusetts. Part number 35115.

[DANO91] Peter B. Danzig, Jongsuk Ahn, John Noll, and Katia Obraczka. Distributed indexing: A scalable mechanism for distributed information retrieval. Technical Report USC-TR 91-06, University of Southern California, Computer Science Department, 1991.

[GCS87] David K. Gifford, Robert G. Cote, and David A. Segal. Walter user's manual. Technical Report MIT/LCS/TR-399, M.I.T. Laboratory for Computer Science, September 1987.

[GLB85] David K. Gifford, John M. Lucassen, and Stephen T. Berlin. An architecture for large scale information systems. In *10th Symposium on Operating System Principles*, pages 161–170. ACM, December 1985.

[CMM86] Ferdinando Gallo, Regis Minot, and Ian Thomas. The object management system of PCTE as a software engineering database management system. In *Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 12–15. ACM, December 1986.

[Gif91] David K. Gifford and James W. O'Toole. Intelligent file systems for object repositories. In *Operating Systems of the 90s and Beyond*, Saarbrücken, Germany, July 1991. Internationales Begegnungs- und Forschungszentrum für Informatik, Schloss Dagstuhl-Geschäftsstelle. To be published by Springer-Verlag.

[Gro86] Computer Systems Research Group. UNIX User's Reference Manual. 4.3 Berkeley Software Distribution, Berkeley, California, April 1986. Virtual VAX-11 Version.

[Inf90] Information Dimensions, Inc. BASISplus. The Key To Managing The World Of Information. Information Dimensions, Inc., Dublin, Ohio, 1990. Product description.

[Kaz88] Michael Leon Kazar. Synchronization and caching issues in the Andrew File System. In *USENIX Association 1988 Winter Conference Proceedings*, pages 31–43, 1988.

[Kil84] T. J. Killian. Processes as files. In *USENIX Association 1984 Summer Conference Proceedings*, Salt Lake City, Utah, 1984.

[Kle86] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Association 1986 Winter Conference Proceedings*, pages 238–247, 1986.

[Kah91] Brewster Kahle and Art Medlar. An information system for corporate users: Wide area information servers. Technical Report TMC199, Thinking Machines, Inc., April 1991. Version 3.

[Leg89] Legato Systems, Inc. NFSstone. Software package. Legato Systems, Inc., Palo Alto, California, 1989.

[Les] M. E. Lesk. Some applications of inverted indexes on the UNIX system. UNIX Supplementary Document, Section 30.

[Log91] Ross Logic, Inc. Ross DBI development specification. Technical documentation. Ross Logic, Inc., Fairfield, IA, February 1991.

[Mog86] Jeffrey C. Mogul. Representing information about files. Technical Report 86-1103, Stanford Univ. Department of CS, March 1986. PhD Thesis.

[NS90a] NeXT Corporation. 1.0 release notes: Indexing. NeXT Corporation, Palo Alto, California, 1990.

[NS90b] NeXT Corporation. Text indexing facilities on the NeXT computer. NeXT Corporation, Palo Alto, California, 1990. from 1.0 Release Notes.

[Nel91] Roger Needham, 1991. Personal communication.

[New90] B. Clifford Neuman. The virtual system model: A scalable approach to organizing large systems. Technical Report 90-05-01, Univ. of Washington CS Department, May 1990. Thesis Proposal.

[NIS91] Ansi z39.50 version 2. National Information Standards Organization, Bethesda, Maryland, January 1991. Second Draft.

[OCE+85] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the unix 4.2bsd file system. In *Symposium on Operating System Principles*, pages 15–24. ACM, December 1985.

[Per90] Jan-Simon Pendry. Amd — an automounter. Department of Computing, Imperial College, London, May 1990.

[Pet88] Larry Peterson. The Profile Naming Service. *ACM Transactions on Computer Systems*, 6(4):341–364, November 1988.

[PPTT90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. UK UG proceedings, 1990.

[PW90] Jan-Simon Pendry and Nick Williams. Amd: The 4.4 BSD automounter reference manual, December 1990. Documentation for software revision 5.3 Alpha.

[Rue85] Marc J. Rochkind. *Advanced UNIX Programming*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[RT74] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Comm ACM*, 17(7):365–375, July 1974.

[Sal83] Gerard Salton. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[Sun88] Sun Corporation. The Network Software Environment. Technical report, Sun Computer Corporation, Mountain View, California, 1988.

[Sch89] Michael F. Schwartz. The Networked Resource Discovery Project. In *Proceedings of the IFIP XI World Congress*, pages 827–832. IFIP, August 1989.

[SCK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *USENIX Association 1985 Summer Conference Proceedings*, pages 119–130, 1985.

[Sta86] C. Stanfill and B. Kahle. Parallel Free-Text Search on the Connection Machine System. *Comm ACM*, pages 1229–1239, December 1986.

[Sta87] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, Cambridge, MA, March 1987. Sixth Edition, Version 18.

[Ste91] Richard Mellon Stein. Browsing through terabytes: Wide-area information servers open a new frontier in personal and corporate information services. *Byte*, pages 157–164, May 1991.

[Sun88] Sun Microsystems, Sunnyvale, California. *Network Programming*, May 1988. Part Number 800-1779-10.

[Sun89] NFS: Network file system protocol specification. Sun Microsystems, Network Working Group, Request for Comments (RFC 1094), March 1989. Version 2.

[Tec90] ON Technology. ON Technology Inc. announces On Location for the Apple Macintosh computer. News Release ON Technology, Inc., Cambridge, Massachusetts, January 1990.

[Ver90] Verity, Topic. Product description, Verity, Mountain View, California, 1990.

[Vog] Peter Weinberger. CB1 Program documentation. Bell Laboratories.

[Wel88] Robert B. Welch and John K. Ousterhout. Pseudo-devices: User-level extensions to the Sprite file system. In *USENIX Association 1988 Summer Conference Proceedings*, pages 37–49, San Francisco, California, June 1988.