A File System for Information Management

C. Mic Bowman*
Transarc Corporation
mic@transarc.com

Chanda Dharap, Mrinal Baruah, Bill Camargo, and Sunil Potti
The Pennsylvania State University

Abstract

Nebula is a file system that explicitly supports information management. It differs from traditional systems in three important ways. First, Nebula implements files as sets of attributes. Each attribute describes some property of the file such as owner, protection, functions defined, sections specified, project, or file type. The content of the file is represented by a special `text` attribute. Second, Nebula supports associative access of files within a scoped index. A scoped index restricts associative access to a subset of the files in one or more file systems. Finally, Nebula replaces traditional directories with database views. A *directory* implements a naming closure; i.e. the name of a file is assigned by the directories in which it resides. A *view* is a query that selects file objects from a scoped index. Thus, the name of a file is a property of the file, not the view. Like directories, views can be named and included in other views. When a file (or view) is created, it is placed in an index, not a directory. The file appears in the appropriate set of views when the file system is refreshed. In this way, views dynamically classify files. The unique properties of the Nebula file system enable users to locate, classify, and identify information quickly and easily as demonstrated by several Nebula-based applications.

Keywords: Resource Discovery, Information Management, Object-Oriented Database.

1 Introduction

Traditional file systems support a limited set of operations for information management: user-oriented file names identify simple semantic properties; directories group related files; and links allow files and directories to be classified in several different ways. There are many user-level tools, services and systems designed for the purpose of managing information. Most, like Archie [1], Gopher [2], Prospero [3], Wais [4], and WWW [5] use an existing file system as a storage repository. Some, like Propsero, Wais and WWW, extend the functionality of the file system interface to include information management operations. However, these operations are available only to applications of the particular information system and are not shared among different information systems. Further, the separation of information management and data storage introduces inefficiencies; e.g. parsing path names is a major source of overhead for Unix-based file location utilities such as the Glimpse personal information system [6].

The Nebula file system described in this paper merges information management and data storage in a single file system. Essentially, Nebula combines the functionality of a file system with information management operations similar to those provided by an object-oriented database system. It extends the notion of a file from an unstructured sequence of bytes to a typed, structured set of attributes. Structured files facilitate information hiding in the sense that a bibliographic citation hides the details of a paper's content with a concise summary. This limits the information that users must manipulate when browsing large collections of files. In addition, file types constrain the structure of files to improve the quality and efficiency of information access.

Nebula replaces traditional file naming by associative access to file objects using a combination of file attributes. The "name" of a file is one attribute that can be used to identify the file. However, a file can also be named by any query that uniquely identifies the file. Associative access simplifies file location which is a key problem in any information management system. Instead of browsing through a directory structure, a user identifies a file with a query—also called a *descriptive name* [7]—that describes the file's characteristics.

Finally, Nebula replaces the fixed structure created by static directories with dynamic views that automatically classify information. A view is the set of objects identified by a particular descriptive name. For example, a view may contain all file objects with the path attribute "/usr/include", or the set of documents that contain references to distributed computing. Like a directory, a view is a nameable object; i.e., a Nebula view is a file object represented by a structured set of attributes with a type, and may be contained in other views. In this way the information space can be organized for efficient browsing. Further, descriptive name resolution can be *scoped* on a set of views to augment and customize the structure of an information space.

2 File Objects

A Nebula file object consists of a set of attributes—an attribute is a tag/value pair like `(type C-source)`—that represents properties of the file as shown in Figure 1. The set of attributes is dynamic, but generally consists of the protection and storage attributes required by traditional file systems, plus the file type, a name, a list of aliases, and a unique identifier. Some file objects have type-specific attributes; e.g. a

*This work is supported in part by the Advanced Research Projects Agency (DABT64-93-C-0052) and an equipment grant from Sun Microsystems, Inc.

```
( (uid          !ab09)
  (name         hello.c)
  (file-path    HOME/source/C)
  (type         C-source)
  (description  The canonical first C program that
                prints "hello world" on standard
                output.)
  (functions    main)
  (includes     stdio.h)
  (text         #include "stdio.h"
                main() {
                        printf("Hello world");
                }))
```

Figure 1: A Nebula file object.

file that contains C source code might have attributes for included files, functions and variable definitions, development project, and modification history. Typically, the `text` attribute contains the body of the file. Alternatively, it may contain pointers to the actual location of the file on the disk or the URL [8] of the file. The `uid` attribute uniquely identifies the file and points to a fixed record that contains storage and protection information.

2.1 Structured File Types

The type of a file object specifies the domain of acceptable attributes. In particular, it constrains the set of applicable attributes tags and enforces a structure on the corresponding attribute values. For example, Nebula requires that files of type `C-source` have attributes that list the functions that are defined and the header files that are included. It also constrains the `text` attribute to be a C code fragment.

The Nebula type system classifies attribute tags as: *mandatory*, *recommended*, and *optional*. Mandatory attributes are those that must be registered for a file object. Recommended attributes are special attributes which maintain "hints" to facilitate efficient, global file classification and location. The quality of information in mandatory and recommended attributes is high. Nebula emphasizes the need to maintain accurate values through automatic and forced manual update of mandatory and recommended attributes. In contrast, optional attributes are user defined. There is no consensus for the set of optional attributes that may be defined for file type. For this reason, optional attributes are less useful for classifying files—one user may define a set of optional attributes that have no meaning to other users—and generally contain low quality information.

In addition to improving and classifying the quality of information, Nebula uses the type system for hints that improve the efficiency of file access. The type of a file object often implies expected access patterns. This information can be used to improve performance by increasing the effectiveness of caching and replication. For example, binary executables can be replicated with a simple call-back mechanism since they are never opened for concurrent write sharing.

Nebula is not unique in using typed files. Traditional file systems maintain a minimal amount of type information. For example the mode field in the inode serves to identify ordinary files, directories, block special files (devices), named communication channels and links [9]. However, Nebula types are more expressive than traditional file types and can be extended like types in the Saguaro operating system [10]. Nebula types contain more information than traditional file types and encode implicit knowledge that can be used for better information management.

2.2 Gathering Attributes

File attributes can be explicitly created by a user or implicitly generated by the system. The user explicitly creates or modifies certain file attributes such as `text` and `name`. Attributes such as `includes` and `functions` can be derived by the system from the `text` attribute.

Generating representative values for derived attributes is a non-trivial task. To this end, Nebula enforces type-based constraints on attribute values. In general, attribute value constraints merely enforce consistent structure on files that are typically structured anyway. For example, the Nebula document type enforces a constraint that the `text` attribute must have a title page, some sections or chapters, and a reference section. Similar structure exists in program source files. For example, a C source file contains function and variable declarations in the format of the C language. Understanding the structure of files facilitates the collection of high quality information for derived attributes.

The information gathering interface is handled by two modules which integrate files into a Nebula context. The first is the **enforcer** module which handles creation of files in Nebula. The enforcer module helps create files with a well-defined structure. The second module—the **collector**—imports files from other file systems. The collector uses a type specific grammar to parse the file, converting it into a list of tokens. A user-specified template identifies the relationship between tokens and attributes of the corresponding Nebula file object. A detailed description of the enforcer and collector can be found in [11].

3 Primitives for Information Management

Existing systems use two paradigms for managing information: searching and browsing. To search for a file, the user provides a description or a query. The system resolves the query and returns to the user a list of files. To search for a file, the user must possess enough information about the file to formulate a query. Without a sufficiently precise query, the user is left with the task of sorting through extraneous matches. Archie is the canonical example of an information system based on the searching paradigm [1].

Browsing is a human guided activity. The system does not resolve queries. Instead it organizes information so that a user can navigate it easily. A well organized information space is the key to effective browsing. However, deciding what constitutes well organized is very difficult. For example, a file system organized by role—e.g. into classes for binaries, libraries, man pages, and include files—complicates the problem of locating all files that belong to a certain software package since pieces of the package will reside in all classes. The fundamental problem is that static organization reflects the requirements of the system designer, not the varied requirements of a diverse user-base. Gopher [2], Prospero [3], and WWW [5] represent the browsing paradigm.

Nebula combines browsing and searching in a novel way that allows dynamic organization and user customization. The set of file objects is partitioned into contexts that index the attributes. Within a context, files are organized by a set of views. Each context has a distinguished view called the "Objects" view that contains all file objects in the context. To locate a file, a user resolves a descriptive name within the scope of a view. If the query is sufficiently precise, a single file object is identified. Otherwise, the organization of information is augmented by a temporary view that contains all file objects named by the query. Since views are themselves file objects, the new view may contain pointers to the existing information structure. Using the temporary view, the user can navigate the augmented information space, or can refine the query and construct a new view. This combination of search and browse makes the Nebula paradigm more flexible to user requirements and more expressive than other systems.

3.1 Descriptive Names

Traditional name resolution in file systems is a string translation problem [12]. The name of a file is a sequence of tokens that identify a path through a name space. Nebula extends the functionality of file name resolution to include associative access through descriptive names [7]. For example, the following name identifies the text version of a notes file for a project called "plan2":

format=text & project=plan2 & name=notes2.txt

The full expressiveness of descriptive names is most useful for finding files whose location in the name space is unknown. This is particularly important for information management in very large file systems. Traditional names are convenient when the file's location in the name space is known. The traditional name of a file, however, is just a descriptive name that selects files with a particular path and name.

Associative access is provided through resolution functions implemented in each file server. A set of well known resolution functions are available on all servers. These include attribute operations—e.g. equal, less than, greater than, and regular expression match—and set operations—e.g. union, intersection, and set difference. Our experience suggests that these operations are sufficient to resolve most descriptive names. However, to accommodate domain specific requirements, Nebula provides a Scheme-based extension language for constructing new resolution functions [13].

3.2 Views

Unlike directories that implement a naming closure, a view is the set of objects within a scope that are identified by a descriptive name. In a traditional file system, a file is created within a directory. The name of the file is a property of the directory. A link to the file from another directory may give the file a different name. When a file object is created in Nebula, it is placed in an index, not a directory. The name of a file is a property of the file object and is the same no matter what view contains the file object. The file object appears in the set of views whose constructors name the new file. That is, views dynamically classify objects.

Like other Nebula file objects, a view is an object and is represented by a set of attributes. Specifically, the view type specifies mandatory attributes for `constructor`, `refresh-time`, and `scope`. The `constructor` attribute is the descriptive name used to identify the objects in the view. The `refresh-time` attribute ensures that the information contained within a view is up-to-date. The user specifies a value for it at view creation and the server provides operations to refresh the views as necessary. The `scope` attribute lists views in which the constructor should be evaluated. The context in Figure 2 is organized into views that classify documents by project and format. A temporary view has been constructed from objects in the "project=plan2" view with the descriptive name—stored as the constructor attribute—"format=text". Scoping the name within the "project=plan2" view obviates the need to evaluate the query over the entire object space and thus improves the performance of query resolution.

3.3 Organizing Information with Views

Nebula views provide a powerful and flexible mechanism for logically organizing an information space. Views relate file objects in two important ways: containment and scope. Containment defines an *abstraction* relation among a set of file objects that are contained in a view. Intuitively, a view contains a set of similar objects. The properties shared by those objects are abstracted in the `constructor` attribute of the view. Thus, the view serves as a class specification for the objects it contains. In Figure 2 the "project=plan2" view contains file objects that are part of the project "Plan2". All file objects in the view are members of the class of objects defined by the view.

Containment in directories is the only tool available for organizing information in traditional file systems. Directories force a rigid structure on information that is difficult to customize. Some systems like the Tilde naming scheme [14] increase the flexibility of organization. Tilde users choose individual name spaces (called *trees*) for their naming environment (called the *forest*). Jade [15] and Plan 9 [16] also provide mechanisms that facilitate the construction of personal name spaces. Like Nebula, Prospero [3] provides personalized name spaces that can dynamically classify information through filters. However, the fundamental building block of a Prospero name space is closure. This makes dynamic classification difficult and expensive.

Containment in Nebula, however, is distinct from other customizable file systems because it is based on object sets (i.e. views) not naming closure (i.e. directories). Nebula file objects exist in a flat space of contexts. Unlike other file systems, the structure that organizes objects—a collection of views—is distinct from the objects being organized. Information can be given a global structure—or several global structures—by a set of views. A user, however, can structure the information within a context using a completely different

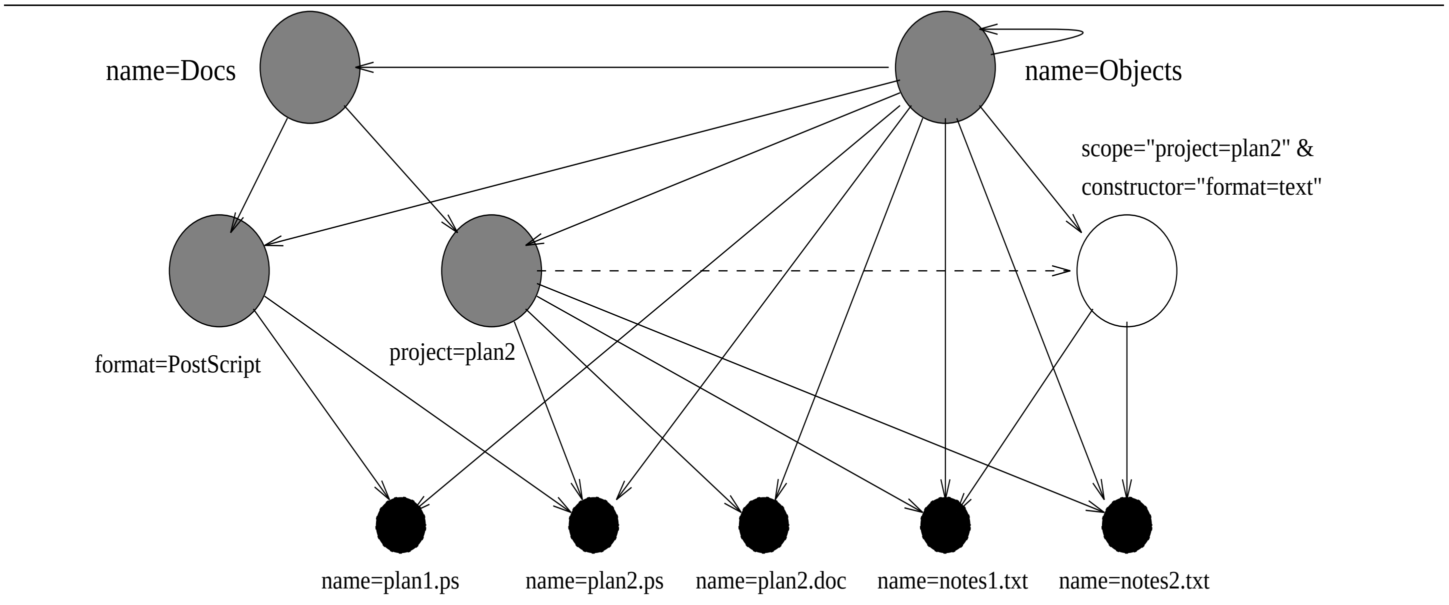

Figure 2: A sample organization for documents.

set of views. If necessary, the user can personalize the structure of some information and still incorporate pieces of the global structure.

Scope defines a *specialization* relation between classes. A view contains a subset of the file objects from the views that scope it. That is, it constrains the set of objects to be a distinguished subset of the objects in its scope. For example, the temporary view created in Figure 2 is a specialization of the "project=plan2" view; it contains only those documents that are stored with a textual format. It therefore defines a new class that specializes the class defined by the "project=plan2" view.

The scope relationship exists exclusively between views. It is most useful for defining taxonomies that build complex relationships between classes. More general classes in a taxonomy contain many file objects. Specialized classes are scoped on general classes and contain fewer files. Note that scope does not imply containment. On the contrary, a view is rarely contained in its scoping view. For this reason, Nebula provides primitives for examining the set of views that scope a view—the classes that are more general—and the set of views that the view scopes—the classes that are more specialized. These primitives allow users to browse a taxonomy to understand how file objects are classified.

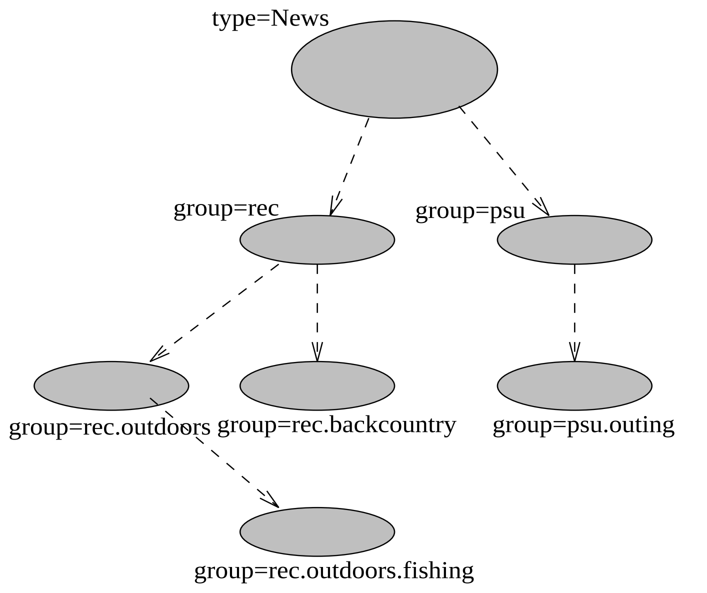

Figure 3: The server structure defined by scope.

4 Applications

The practical usefulness of Nebula for organizing information is demonstrated by NebNNTP, a threaded news reader that supports sophisticated article classification. NebNNTP consists of an NNTP server, a Nebula News context, and a number of client contexts. The NNTP server stores article bodies and headers. The News context builds file objects from header information provided by the NNTP server using a collector for files of type "article".

The Nebula News context is organized according to a newsgroup taxonomy such as the one shown in Figure 3. The most general class in the taxonomy is a view that contains all file objects of type "News". At the next level, classes such as "rec" and "psu" contain articles from the rec and psu news hierarchies respectively. The views for "rec" and "psu" are scoped on the News view. The rest of classes in the newsgroup taxonomy are implemented in the same way. A client can browse the newsgroup taxonomy or search it for interesting newsgroup classes. To enhance the search, each newsgroup view collects keywords from the articles it contains. These keywords simplify the process of finding pertinent newsgroups.

Each NebNNTP client constructs a set of views that identify interesting articles. A simple structure supports traditional newsgroup subscriptions: the client constructs a view that contains views from the newsgroup taxonomy. In Figure 4, the client constructs a view with the descriptive name "keywords=fishing,trout" and scopes it on the newsgroup views "group=rec.outdoors", "group=rec.backpacking", and "group=psu.outing". The view contains news articles that pertain to trout fishing from several recreational newsgroups. In this way the client has customized the classification news articles. Further, NebNNTP treats each article as a view that contains all articles sent in response to it; i.e. each article points to a thread of articles on a particular topic. When the client removes an article from the context, it kills all messages in the thread.

Although the effectiveness of NebNNTP depends on the structure created by the client's views, our experience suggests that NebNNTP removes as many as 90% of the articles that would be browsed by a traditional, subscription-based news reader. Given the quantity of articles posted to Usenet newsgroups, this represents a significant savings in time and effort.

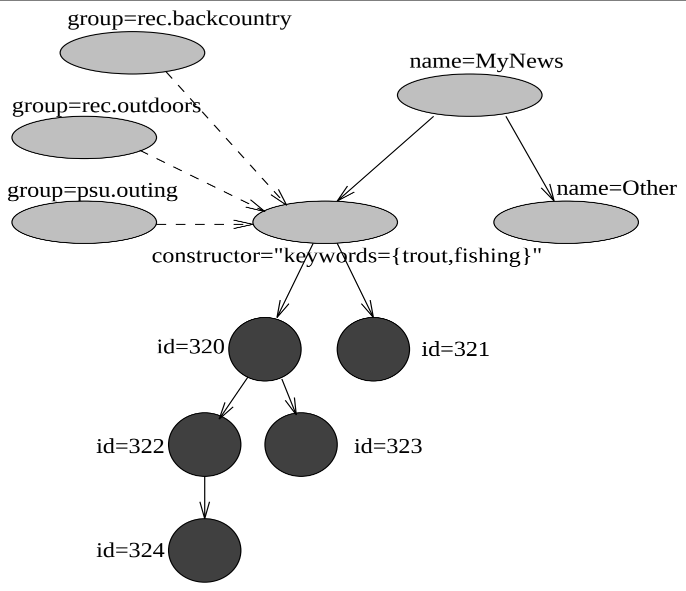

Figure 4: The client structure defined by containment.

Several other applications have been constructed in addition to NebNNTP. These include:

- StarGazer: A database of bibliographic citations that can be accessed by resolution functions for keyword search.
- NebArchie: A gateway to the Archie file location service; the results of an Archie query are cached and manipulated locally.
- NebArchive: A simple gateway to several common MS-DOS archives; index files from a set of archives are converted to Nebula objects.

5 Implementation and Performance

The Nebula prototype is implemented as a user level file system on top of Univers, an attribute-based name server [7]. The interface to Nebula consists of a shared library that intercepts system calls invoked by the application and handles name resolution. If the name resolves to a Unix file, then the call is passed to the underlying operating system. Otherwise the name is sent to the Nebula server via an RPC call. The Nebula server returns information about the object depending on the type of system call.

A file object is stored on disk with an object descriptor. The object descriptor contains protection/security fields, type information, and pointers to the file's attributes. Each attribute is stored in a descriptor with fields for tag and value. Depending on its size, the value is stored directly in the descriptor or indirectly through a table of pointers to data blocks.

File attributes are generated via the information gathering interface, viz. the collector. The collector is a suite of programs with a module for each file type. A collector module extracts appropriate information from files that are incorporated into a Nebula context. Currently, collector modules exist for C, pascal, latex, mail and news files. Once collected, attributes are indexed via a wide range of indexing mechanisms including btrees, hash tables and sorted lists. A uniform interface simplifies the construction of new indexing mechanisms. When an index is created the user chooses an indexing mechanism based on the expected index size, data type, and query type. For example, file names are placed in a dynamic hash table for exact lookups while modification times are kept in sorted lists to simplify the location of most recently modified files.

The disk is managed using a change log similar to the log structured file system described in [17]. The motivation for logging changes is that most reads are handled by the cache, making writes the major producer of disk traffic. Disk head movement is reduced since all disk writes are performed contiguously at the "frontier" of the disk. This reduces overall disk latency and improves bandwidth. A compression algorithm is used to "squeeze" the set of active blocks into a contiguous region. The premise is that stable long-lived blocks will work themselves to the front of the disk. This creates large free extents at the end of the disk. Currently the compression algorithm is a brute-force copy and compact technique that can only reasonably be used when the file system is idle. A more efficient approach would be to organize the disk as segments as done in [18].

Since Nebula uses well established techniques for storing bulk data, the performance depends primarily on the time required to resolve descriptive names. Table 1 shows the results for resolving descriptive names within a collection of Nebula servers implemented on four Sun IPC Sparcstations connected by a 10Mbps ethernet. Two different queries were sent to the servers. The numbers reflect the time taken to transmit the query, search the index, package the resulting set of objects (about 80 objects were returned with roughly 200 bytes of data for each), and send it back to the client. The simple query is an exact match on the "file-type" attribute. The complex query includes a regular expression comparison, an equality check, and the intersection of the resulting sets. The queries were scoped on views in three different servers, each with 100 (or 1000) objects. For comparison, similar queries using the Unix `find` utility took approximately 18 seconds to locate 145 files by name in a directory tree that contained 1300 files. Nebula clearly outperforms `find` for search operations.

	No Cache		Local Cache	
Query	300	3000	300	3000
Simple Query	101	347	71	91
Complex Query	1126	4266	472	658

Table 1: Time in milliseconds for simple and complex queries.

6 Conclusion

This paper describes the design and implementation of the Nebula File System. The goal was to integrate information management with traditional file system operations. With this prototype it is possible to provide explicit information management within a wide-area file system. Nebula introduces three new concepts: types to structure file objects, associative access to name files, and views to dynamically classify information.

- Structured types contain more information than traditional file types and encode implicit knowledge that can be used for better information management. Specifically the type indicates the domain of attributes that can be collected for a given file. Further, by enforcing type-based constraints on attribute values, Nebula facilitates the collection of high quality information for derived attributes.
- Associative access to files improves file location in Nebula. Effective file location is achieved by a rich set of attribute selection primitives and facilities for constructing complex resolution functions from these primitives.
- Views facilitate explicit, logical organization of the information in the file system repository. By replacing the traditional directory implementation with dynamic views we provide automatic and flexible classification of information/files.

We plan to further extend Nebula by using wide-area file access patterns to improve caching policies, develop stronger consistency policies for views and incorporate gateways to other information services.

References

[1] Alan Emtage and Peter Deutsch. Archie - An Electronic Directory Service for the Internet. In *Proceedings of the Winter 1992 Usenix Conference*, pages 93–110, Montreal, Canada, January 1992. McGill University.

[2] Bob Alberti, Farhad Anklesaria, Paul Linder, and Daniel McCahill, MarkTorrey. Exploring the Internet Gopherspace. *Internet Society News*, 1(2), 1992.

[3] B. Clifford Neuman. The Prospero File System: A global file system based on the Virtual System Model. *Computing Systems*, 5(4), Fall 1992.

[4] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In Stephen Cook, editor, *ecoop89*, pages 89–106. Cambridge University Press, july 1989.

[5] 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.

[6] Udi Manber and Sun Wu. GLIMPSE: A Tool to Search Through Entire File Systems. In *Proceedings of the 1994 Usenix Winter Conference*, pages 23–32, January 1994.

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

[8] Tim Berners-Lee. Uniform Resource Locaters. Technical report, Internet Engineering Task Force (IETF Draft), October 1993.

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

[10] Gregory R. Andrews, Richard D. Schlichting, Robert Hayes, and Titus D. M. Purdin. The Design of the Saguaro Distributed Operating System. *IEEE Transactions on Software Engineering*, 13(1):104–118, January 1987.

[11] Chanda Dharap and Mic Bowman. Structure in file systems. Technical Report CSE-94-021, Department of Computer Science and Engineering, The Pennsylvania State University, Univesity Park, Pennsylvania, January 1994.

[12] Douglas E. Comer and Larry L. Peterson. Understanding naming in distributed systems. *Distributed Computing*, 3(2):51–60, May 1989.

[13] David Michael Betz. *Xscheme: An Object-oriented Scheme*. Peterborough, NH, 1989.

[14] D. Comer and T.P Murtagh. The Tilde File Naming Scheme. *IEEE Transactions on Software Engineering*, May 1985.

[15] H.C Rao and L.L Peterson. Accessing files in an internet: The Jade File System. Technical report, The University of Arizona, 1991.

[16] Dave Presotto, Rob Pike, Ken Thompson, and Howard Trickey. Plan9, A Distributed System. Technical report, AT and T Bell Laboratories, Murray Hill, New Jersey 07974, 1990.

[17] M. Rosenblum and J.K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[18] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-Structured File Systen for UNIX. In *Proceedings of the Winter 1993 Usenix Conference*, pages 307–326, January 1993.