Towards a Semantic, Deep Archival File System

Mallik Mahalingam
HP Laboratories
1501 Page Mill Rd.
Palo Alto, CA 94304
mallik@vmware.com

Chunqiang Tang *
Computer Science Department
University of Rochester
Rochester, NY, 14627-0226
sarrmor@cs.rochester.edu

Zhichen Xu
HP Laboratories
1501 Page Mill Rd.
Palo Alto, CA 94304
zhichen@hpl.hp.com

Abstract

In essence, computers are tools to help us with our daily lives. CPUs are extensions to our reasoning capability whereas disks are extensions to our memory. But the simple hierarchical namespace of existing file systems is inadequate in managing files today that have rich semantics. In this paper, we advocate the need for integrating semantic information into a storage system. We propose "Sedar", a deep archival file system. Sedar is one of the the first archival file systems that integrates semantic storage and retrieval capabilities. In addition, Sedar introduces several novel features: the notion of "semantic-hashing" to reduce the storage consumption that is robust against misalignment of documents; "virtual snapshot" of namespace, and "conceptual deletions" of files and directories. It exposes a semantic catalog that allows other semantic-based tools (e.g., visualization and statistical analysis) to be built. It uses a decentralized peer-to-peer storage utility enabling horizontal scalability.

1. Introduction

In essence, computers are tools to help us with our everyday activity. CPU cycles are extensions to our reasoning capability whereas disks are extensions to our memory. But there exists a gap between the human memory and the simple hierarchical namespace of existing file systems. The hierarchical namespace was invented decades ago and is no longer adequate in managing files today that have rich semantics. Human brains remember objects based on their *contents* or *features*. When you run into a friend in elementary school, you may not remember her name, but you can recognize her by features like round face and shinny smile. We call these features *semantics*. Semantic information can be derived from various types of data. For instance, people employ vector space model to extract features from text documents and images [7, 1]; and derive frequency, amplitude, and tempo feature vectors from music data [19, 21].

To bridge the gap between the simple hierarchical namespace and our memory system, people have used various utilities to manipulate files based on their semantics. For instance, search tools like grep can exhaustively examining every document in a file system to locate all documents that contain certain phrase. File systems such as SFS [9] and HAC [10] provide content-based retrieval capability that is compatible with existing file system interface. However, search in these systems is limited to simple keyword match, unable to conduct advanced retrieval, such as searching for songs by humming or whistling a tune, or searching for images by submitting a sample of patches or texture [7]. According to Gartner [6], there is a shift from simple search (e.g, keyword) to more-complex techniques that leverage natural language processing and cognitive concepts. These advanced approaches have a better likelihood to find the content people are interested in.

In this paper, we advocate the need for directly embedding semantic information into file systems. This not only makes file systems more friendly to end users but can also improve efficiency with respect to storage usage and data access. If documents stored in the system are already "organized" according to their semantics, content-based retrieval operations can be performed efficiently. In addition, the storage system can take advantage of the similarity among files by avoiding storing redundant information to improve the storage utilization. Further, embedding common and flexible semantic indices in file system removes the redundancy among indices kept by separate tools.

In particular, we focus on demonstrating the benefits of integrating semantic information into an archival system. With the cost and density of the random access devices approaching those of the magnetic tapes [11], it is practically affordable for a *deep* archival system to backup every individual version of a file. With disks, old versions of a document can be recovered instantly without much of human intervention in contrast to traditional tape-based solutions.

The biggest headache in restoring a backup is to find the right document and the right version. Currently a common way to do this is by remembering the date the version

[1]This work was done when Chunqiang Tang worked as an intern at HP Labs during 2002.

was produced. In many cases, however, people are interested in files produced by other people, and are interested in versions with certain features. For example, in a digital movie studio, an artist makes many changes to clips. To produce a variant, the artist goes through several tries to get the right "look and feel". In this process the artist may go back to previous versions (not necessarily the latest one). Sometimes the artist needs to incorporate scenes produced by other artists, but the only thing this artist knows is that these files should have certain semantics. Scenarios of this kind also happen in other environments, e.g., universities, research laboratories, and medical institutions, but they are not well supported by existing archival systems.

Integrating semantic information into a deep archival system offers several advantages over current techniques.

- It allows easy access to the appropriate versions using advanced semantic-based retrieval. To recover a file without such support, a user would have to remember the name of the file and time at which a particular version was produced.
- It provides a basis for understanding the semantic evolution and relationship of documents. Information of this kind can be used to guide system optimization (e.g., data placement and prefetching).
- It can cluster documents that are semantically close for the purposes of, for instance, finding related materials, and purging unused old files.
- A novel use of semantic information is to reduce storage consumption by eliminating redundancy among files through the notion of *semantic hashing* (see Section 3.2).

In this paper we propose a semantic-based deep archival system *Sedar* that has the following features: (i) It uses semantic information to organize and retrieve files. (ii) It uses semantic vector to guide file "compression" to reduce the storage consumption that is robust against misalignments. (iii) Storage is provided over completely decentralized peer-to-peer (P2P) storage utility that allows horizontal scalability. (iv) It provides high availability using erasure-coding techniques.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 describes the architecture of Sedar in details. Section 4 discusses advanced issues, and Section 6 concludes the paper.

2. Related Work

Semantic files systems [9] and HAC [10] provide support for maintaining the orthogonal namespace by executing queries and constructing the namespace according to the query results. These systems provide support only for simple keyword based queries and require some level of support from the applications. Also, these two systems do not have deep archival capability and high availability.

Venti [15] provides versioning capability through a block level interface. It uses block-level content hashing to identify and avoid storing redundant copies of the same block. However, Venti cannot handle files with nearly identical but misaligned contents. SnapMirror [14] provides versioning by taking advantage of the metadata stored in the underlying file system. Elephant file system [17] provides versioning capability and retention policies that can be applied to each individual file. None of the above systems provides semantic storage and retrieval capabilities.

SFSRO [8] employs SHA-1 to hash blocks recursively, building data structures that facilitates the sharing of distributed, read-only contents. Its focus is on security rather than advanced search or deep archival.

OceanStore [12] uses erasure-coding to provide reliability for versioned objects and rely on Tapestry [25] for horizontal scalability. Like the other systems, it does not have any support for semantic-based file storage, retrieval or manipulation.

3. Architecture

Sedar is a semantic-based deep archival system. Every version of a file is written only once. Any update to an existing file will create a completely new version. This is commonly referred as the *write once and read many* (WORM) semantics. Different instances of the same file will be given a different version number. The metadata, however, is not versioned. We provide a mechanism known as *virtual snapshot* for freely accessing a namespace arbitrary back in time. In addition to the conventional file system interface, Sedar also has a semantic-based interface that allows users to locate files according to the semantics (contents) of the files. For backward compatibility, the system can create materialized views of the search results by presenting them through a hierarchical namespace. In the following, we will use "file", "object", or "document" to refer to a particular version of a file, provided that the context does not cause any confusion.

3.1. Major System Components

The architecture of Sedar is shown in Figure 1. The major components of Sedar include a NFS loopback server, a semantic catalog, a registry of semantic *extractors*, a distributed storage, and the Sedar semantic utilities. We describe each of the core components below:

NFS loop-back server: This component allows users to mount Sedar through a standard NFS interface.

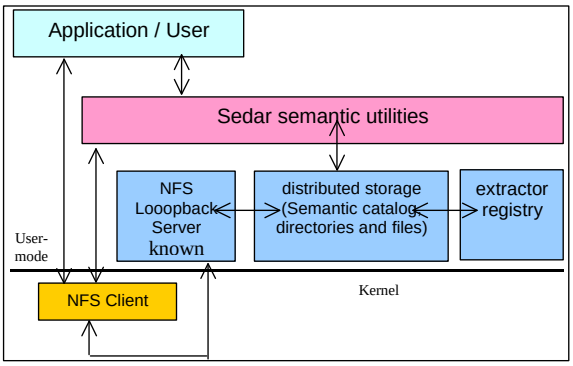

Figure 1. Major system components in Sedar.

Catalog: For each version of a file (object), the catalog stores an index that contains, among other things, an inode number and a version number that uniquely identify the object, and semantic information (e.g., semantic vector) of the object. Semantic vector (SV) is a vector of file-type-specific features extracted from file contents. For instance, vector space model (VSM) [2] extracts the term frequency information from text documents and latent semantic indexing [5] uses matrix decomposition and truncation to discover the semantics underlying terms and documents. Welsh et al. [21] derive frequency, amplitude, and tempo features from encoded music data.

As an optimization, rather than storing the SV for each individual version of a file, we store only representative SVs. Each representative SV will be associated with files whose SV are very close to the representative SV (i.e., the difference among them is below a threshold).

Extractor registry: For each known data type, Sedar uses an external plug-in called *extractor* to derive the semantic vector; for data of unknown types, it uses statistic analysis to drive features from the bit stream. Similarly, for each data type, a different diff function (or differential compression technique [4]) can be introduced (see Section 3.2). The extractor registry provides an extensible interface that allows new extractors and diff functions to be added incrementally.

Sedar distributed storage (SDS): SDS provides basic support for storing and retrieving files, directories, and the catalog. We plan to implement Sedar distributed storage on top of a distributed hash table (DHT) based overlay network (e.g., CAN [16]). DHTs aggregate physical storage resources in a scalable fashion and provide a hash table abstraction that maps `keys` to `objects`. The "root" of the Sedar file system is assigned a well-known key in the storage utility. Should the node that stores the root object fail, another node in the storage utility that holds a replica of the root object will take over transparently. A newly started Sedar client contacts the root node to mount the Sedar file system.

Sedar semantic utilities: The Sedar semantic utility offers semantic-based retrieval capabilities. It interacts with the file system to generate materialized views of query results and users can access these materialized views as regular file system objects. For example, a user can issue commands to create results of a query into a directory.

sdr-mkdir cn

sdr-cp "similar to 'hawaii.jpg'" cn

The directory cn contain links to files that that are semantically close to the sample file, hawaii.jpg. Directories like 'cn' are called "semantic directories"; they can be accessed as the "regular" directories. Sedar supports semantic based retrieval capability. Queries themselves can be arbitrary text of bit stream whose features will be extracted by the appropriate extractor to produce SVs to be used by the catalog for query. This is analogous to query-by-example in the database system, but is much more flexible.

Similar to database queries, queries in Sedar can be constrained. The typical constraints include time and namespace. When a query is not time constrained, it provides the capability to restore contents that are deleted "conceptually" (see Section 4.3).

To give the readers a flavor of how the constrained queries look like, we show few examples. A user can specify that she is only interested in documents that are created after 1/1/1999, by issuing a command like sdr-ls "after 1/1/1999". Similarly, she can specify that she is only interested in documents that are under a list of directories (e.g., sdr-ls "'computer networks' under /etc, cn/, before 1/1/1999".) The directories themselves can be "semantic directories".

3.2. Semantic Hashing

Sedar employs a novel technique to improve storage utilization based on the semantic vector of a document. The basic idea is to use the SV of a document to locate a document that is closest to the current document in the semantic space. We hypothesize that documents that are close in the semantic space will also be very close in the actual contents, i.e., they will produce a small diff. We call this *semantic hashing*.

A naïve block-level content hashing using, e.g., SHA-1 can remove duplicates of only identical blocks, but does not work well when there are misalignments in the documents (e.g. source code ports, video and audio clips with

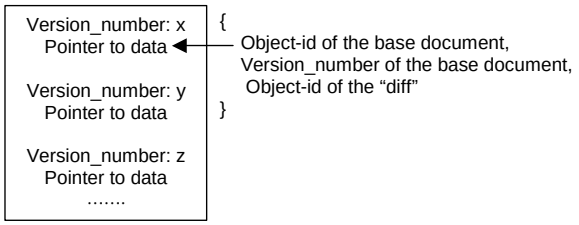

Figure 2. Sedar inode layout.

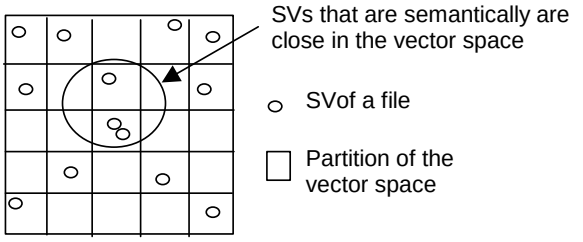

Figure 3. A partition of Sedar's Semantic Catalog.

personal edits, different releases). Techniques such as that is described by Manber [13] do remove the limitations of naïve block-level content hashing. These techniques produce *approximate fingerprints* over a slide window of a document and use anchors to "synchronize" the equal parts in two different documents without priori knowledge regarding which files (parts) are similar (identical). Using this technique for locating a file that is syntactically similar to a given file, however, requires matching the fingerprints of a given file with all fingerprints. This matching process can be expensive for a large document collection.

The effectiveness of semantic hashing remains to be validated with experimentation. We envision several ways to make it work.

- If existing document ranking algorithms, such as VSM (or LSI) cannot be directly used for this purpose, we believe that they can be extended to rank the similarity between contents.
- It is conceivable to combine semantic hashing with syntactic similarity matching techniques such as those described by Manber [13] and Broder [3]. That is, to use semantic hashing as a pre-selection process and then use syntactic similarity matching techniques for refinement. As a result, we can reduce the size of the base fingerprints set when trying to locate the file that is syntactically closest to a given file.
- Using semantic hashing to cluster similar documents has the potential to make the traditional compression techniques work better by taking advantage of higher frequencies of occurrences of common patterns.

We assume efficient technique based on the above basic ideas do exist, and refer it as semantic hashing without the loss of generality.

Sedar can produce better storage utilization than techniques such as RCS that produces diff between current version and the most recent version. For example, you might work on a copy that is several versions behind the most recent version, to get the maximum benefit in terms of storage utilization ideally it should be diffed with version it was produced from. Even if we can do an exhaustive-search using RCS to reduce the diff, it could be a time consuming process when it has to go through several hundreds of versions, whereas in our case, we only need to locate the version that has the closest semantic vector. Besides, comparing two semantic vectors can be much more efficient than comparing two documents because the dimension of a semantic vector is typically only 200-300.

3.3. Important Data Structures

The directory structure in Sedar is similar to that of a traditional UNIX based file system. Directory entries contain the name of the object, type of the object and a unique identifier that consists of a "inode" and a version number. Inode number can be derived, for example, by hashing the name of the object.

More specifically, inode in Sedar contains version number for each version of the file. For each version of the file, the inode includes a data pointer that includes inode of the base document and the inode of the "diff" (see Figure 2). Please note that we use "diff" in a generic sense. "diff" here refers to techniques such as differential or delta compression [4]. For example, with delta compression, the "diff" can be represented a sequence of `copy` and `add` instructions using the base document as a reference. The inode structure for the base document and for the "diff" is the same as that is used in the traditional file system, that is, each of them consists of indirect blocks and direct blocks. The only difference between a base document and a "diff" lies in the contents. In fact, the base document can be regarded as a diff to an empty document.

Applying the diff to the base object will produce the whole document. The inode is responsible for computing the SV of a document by invoking appropriate extractor depending on the type of the file. It is also responsible for reassembling the documents. Once an SV is computed, the inode contacts the Catalog to locate the semantically closest files to compute the diff.

The Catalog is a distributed index that provides the functionality such as retrieving IDs of objects that are semantically close, given the semantic vector. One way to implement and store the Catalog is to partition the vector space into multiple regions, each region assigned to a node in Sedar distributed storage in a way that indices that are semantically close to each other are also close to each other in network distance. In a related paper [18], we describe a

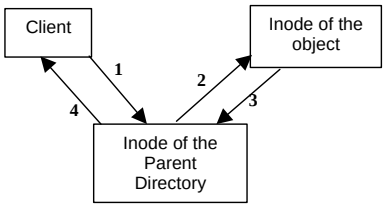

Figure 4. Illustration of create/mkdir protocol. (1) The client contacts the parent directory. (2) The parent directory checks to see if the entry already exists, if so it contacts the "inode" to assign a version number. Otherwise, a new inode is created. (3) Inode assigns a new version and returns it back to the parent directory. (4) The parent directory returns the inode number and the version of the object to the client.

technique that can place SVs that are semantically close to each other also logically close in a structured overlay network. Figure 3 illustrates this.

3.4. Discussion

It should be noted that represent documents as the "diff" to a base document does created a dependency among files. This not only can make read operation less efficient, but also make purging of unwanted files difficult. To deal with the efficiency of read operations some kind of caching scheme can be employed. Consistency is not a problem due to the write once semantics of Sedar.

As one of the reviewer of the paper points out, keeping unwanted files as base documents in the system can pose a trust threat as the user may want to remove confidential data from the system physically. One solution to this problem is to materialize all the files that directly dependent upon the files that is to be physically removed.

3.5. Important File System Operations

In this section we describe few important file operations performed in Sedar.

Mount: When the mount is performed at the client, the Sedar NFS Server receives request from the client through the loop-back interface. It then contacts a node at a "well-known" location in the SDS.

Create/ Mkdir: `Create` or `Mkdir` is done using the inode of the parent object and the name of the file or directory that needs to be created. Figure 4 illustrates the protocol of create/Mkdir operation.

Lookup: To perform a lookup, the client must first obtain the inode of the parent object and then perform the lookup using the parent inode and the name of the component to perform lookup on. Lookup returns the inode of the object and the latest version number of the object. User or application can override the version number by specifying any valid version number when the file is accessed.

Read: For reading a file, client passes the inode of the file, the version number, offset and the number of bytes to read. When Inode receives the request, it assembles the whole version of the file using the base document and the corresponding diff if it does not exist locally. Inode returns the requested number of bytes back to the client from the assembled file.

Write: Once the file is closed, Sedar computes the diff between the current version of the file and the version from which it is derived. If the size of the diff is above a threshold, Sedar passes the whole file to an *extractor* that derives the semantic information from the document and generates a semantic vector. This semantic vector is used to locate the best base document that is "closest" in the semantic space using the Catalog service. Once the base document is located, Sedar compares them to create a diff. The diff is stored in SDS by performing, e.g., content hashing on the diff. Sedar then stores the inode of the base document and the ID of the diff under the entry for the new version and creates a new catalog entry for that version of the file.

In the case of concurrent writers, multiple versions of the same file are created. Assigning non-conflicting version numbers is done at the Inode of that file. To prevent concurrent access, lease or lock service can be used.

4. Advanced Features

In this section we touch upon some advanced issues to improve the usability of Sedar.

4.1. Erasure-Coding for Fault-Resiliency and Availability

To improve availability of Sedar, we plan to apply erasure-coding to store the fragments of the data to provide fault-resiliency and availability of the system. It has been shown that erasure coding provides better availability then simple replication with the same amount of space overhead [20].

4.2. Virtual Snapshot of Namespace

In Sedar, directories are not versioned. As a result, it is very difficult to restore a snapshot of the entire file system at any particular instance of a time. To remedy this effect, we introduce the notion of virtual snapshot using timestamps.

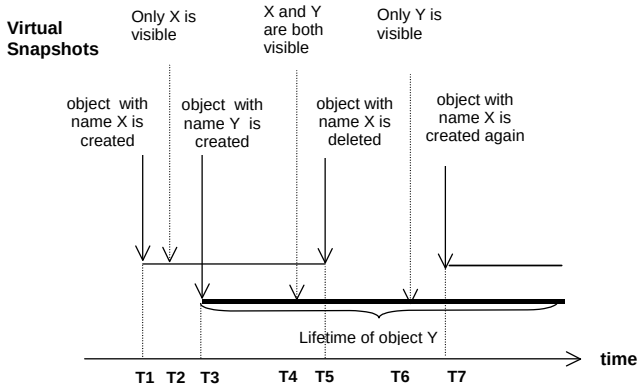

Figure 5. Example showing the conceptual deletion of objects and snapshots of name spaces.

The basic idea is to use timestamps to identify directory entries that are created before the requested snapshot time. For a directory, its timestamp is the time it was created, and the timestamp will never change. A file may have multiple versions, therefore multiple timestamps. Rather than making a copy of the metadata during snapshot, we only need to record the timestamp of the snapshot.

To access any particular snapshot, the file system will show only the entries that were created before the timestamp of the snapshot. In Sedar, it is sufficient to just keep the timestamp of the snapshot time because all versions of the files are kept in the system, and unique IDs are used to identify each individual version of each file.

For this scheme to work, we require time to be loosely synchronized at the boundary of seconds. We believe this should not be an issue. To make accessing a snapshot efficient, we employ some caching scheme to improve the access latency.

4.3. Conceptual Deletion

In Sedar, besides the capability to physically purge a unwanted object, we introduce the notion of conceptual deletion that make directories or files invisible to the users and applications without permanently removing them from the system.

To implement conceptual deletion, we introduce an additional timestamp for an entry to be removed. This timestamp is called *invisible_after*. The file system will hide these entries and items beneath them in the name space, if the timestamp specified by the request is later than the invisible_after timestamp. We also change the object name by appending the invisible_after timestamp to it. In this way, we allow object names to be reused without permanently deleting old objects. To make these files or directories visible, the users use the semantic utility.

Figure 5 explains how the timestamps are used to view of the system at any point of time.

4.4. Distributed Peer-to-Peer Storage Utility

The distributed storage model of Sedar is built on top of a P2P Storage utility.

Recent P2P systems represented by OceanStore [12] and CAN [16], offer an administration-free and fault-tolerant storage utility. Nodes in these systems collectively contribute towards a storage space, in a self-organizing fashion. In these systems, there is a consistent binding between objects to nodes. Locating an object is reduced to the problem of routing to the destination node from the node where the query is submitted. The logical overlay of these systems provide some guarantee with respect to the number of logical hops that need to be traversed to locate an object.

We have built a structured overlay called eCAN [24, 22, 23]. eCAN is a hierarchical version of CAN. It preserves CAN's abstraction of a Cartesian space, and employs proximity neighbor selection using global soft-state to take advantage of the underlying network proximity. In addition, for systems where node population is relatively stable, eCAN uses an auxiliary network that employs IP-like routing protocol to achieve performance comparable to IP routing. The Cartesian space abstraction of CAN and eCAN makes them more attractive in places where the application directly demands such an abstraction, e.g., document ranking using latent semantics [18].

5. Open Issues

There are still quite a few open issues to be addressed. One is to understand the benefit of using "semantic hashing". There can be many files that have similar SVs, how to find the file that can produce a small diff is a challenging task. There are several possibilities to attack this. One possibility is to use semantic hashing only as a pre-selection process and then use syntactic similarity matching techniques for refinement. In fact, we do not need to find the *best* base document to produce the smallest diff. Storing a reasonable size diff is still a win over storing the entire document. We can use a two-step process to find a base document. First, we sample several files with close SVs. Second, we randomly sample fragments of these files and compare the sample fragments with the those of the new document. The file that produces the smallest total diff is picked as the base document. If the size of the diff is big, we can repeat this process. This approach only compares a small number of fragments, and can be efficient. Another way is to increase the dimension of the SVs or devise special extractors that can capture the differences in contents. As a final resort, we can use block-level content hashing.

Another issue is to understand the overhead involved in the system, especially that involves in computing diff, re-

assembling the documents, and the distributed metadata operations.

A third issue related to the access control and protection of data and metadata. For example, a content-based search submitted by a particular user should only return the files that this user has the right to access.

Yet another open issue related to device more advanced searching capabilities that make use not only the contents of the files but also other information such as interrelationships among the files.

6. Conclusion

In this paper, we argued the need for integrating semantic information into a storage system. We painted a vision of Sedar, a semantic deep archival file system. Sedar provides several novel features such as virtual snapshot, conceptual deletion, semantic hashing and horizontal scalability in addition to semantic retrieval capability.

Acknowledgment: We thank Magnus Karlsson, Christos Karamanolis and the anonymous reviewers for their comments.

References

[1] K. Aas and L. Eikvil. A survey on: Content-based access to image and video databases.

[2] M. Berry, Z. Drmac, and E. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 1999.

[3] Broder. On the resemblance and containment of documents. In *SEQS: Sequences '91*, 1998.

[4] R. Burns. Differential compression: A generalized solution for binary files, 1997.

[5] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[6] C. DiCenzo. Data management trends: Growth drivers and new technology requirements. In *Gartner Planet Storage 2002*, 2002. http://www.gartner.com.

[7] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3(3/4):231–262, 1994.

[8] K. Fu, F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. In *4th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diefo, California, USA, 2000.

[9] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O. Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[10] B. Gopal and U. Manber. Intergrating content-based access machanisms with hierarchical file systems. In *Usenix OSDI*, New Orleans, Louisiana, USA, 1999.

[11] J. Gray and P. Shenoy. Rules of Thumb in data Engineering. In *the Proceedings of ICDE*, 2000.

[12] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[13] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Fransisco, CA, USA, 17–21 1994.

[14] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. Snapmirror: File-system-based asynchronous mirroring for disaster recovery. In *First USENIX conference on File and Storage Technologies*, Monterey, CA, USA, 2001.

[15] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, Monterey, CA, USA, 2002.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM'01*, August 2001.

[17] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *Symposium on Operating Systems Principles*, pages 110–123, 1999.

[18] C. Tang, Z. Xu, and M. Mahalingam. pSearch: Information Retrieval in Structured Overlays. In *HotNets-I*, Princeton, NJ, October 2002.

[19] G. Tzanetakis and P. Cook. Audio information retrieval (air) tools. 2000.

[20] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, MA, USA, 2002.

[21] M. Welsh, N. Borisov, J. Hill, R. von Behren, and A. Woo. Querying large collections of music for similarity. Technical Report UCB/CSD-00-1096, UC Berkeley, November 1999.

[22] Z. Xu, M. Mahalingam, and M. Karlsson. Turning Heterogeneity into an Advantage in Overlay Routing. In *INFOCOM'03*, 2003.

[23] Z. Xu, C. Tang, and Z. Zhang. Building Topology-Aware Overlays using Global Soft-State. In *Proceedings of ICDCS'03*, May 2003.

[24] Z. Xu and Z. Zhang. Building low-maintenance expressways for p2p systems. Technical Report HPL-2002-41, HP Laboratories Palo Alto, 2002.

[25] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001.