Towards a Semantic, Deep Archival File System

Mallik Mahalingam, Chunqiang Tang, Zhichen Xu
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2002-199
July 12th , 2002*

E-mail: {mmallik, chunqian, zhichen} @hpl.hp.com

semantic
archival, deep
archival
storage, P2P,
Sedar,
semantic
retrieval

We advocate the need for integrating semantic information into a file system. We demonstrate the benefits of this in *Sedar*, a deep archival file system. Sedar is the first archival file system 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 snapshots of the namespace, and conceptual deletions of files and directories. Sedar exposes a semantic catalogue that allows other semantic-based tools (e.g., visualization and statistical analysis) to be built. It uses a decentralized P2P storage utility enabling horizontal scalability.

* Internal Accession Date Only
Approved for External Publication

Towards a Semantic, Deep Archival File System

Mallik Mahalingam, Chunqiang Tang, Zhichen Xu †

Hewlett-Packard Laboratories
1501 Page Mill Rd, Palo Alto, CA94304, USA
{mmallik, chunqian, zhichen}@hpl.hp.com

Abstract

We advocate the need for integrating semantic information into a file system. We demonstrate the benefits of this in *Sedar*, a deep archival file system. Sedar is the first archival file system 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 snapshots of the namespace, and conceptual deletions of files and directories. Sedar exposes a semantic catalogue that allows other semantic-based tools (e.g., visualization and statistical analysis) to be built. It uses a decentralized P2P storage utility enabling horizontal scalability.

1. Introduction

Fundamentally, computers are tools to help people with their everyday activity. CPU cycles are the extension to our reasoning capability and disks are the extension to our memory. But the gap between the human memory and the simple hierarchical namespace of existing file systems makes it hard to use. 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 the round face and shinny smile. We call these features *semantics*. Semantic information can be derived from various types of data. For instance, people use Singular Value Decomposition to extract features from text documents and images; and use various extractors to derive frequency, amplitude, and tempo feature vectors from the music data.

To bridge this gap, people have used either separate tools or file systems that integrate rudimentary search capabilities. Tools such as `grep` and other local search engines have to exhaustively search every document and match the pattern. File systems such as SFS [1] and HAC [2] provide only simple keywords-based searches and they do not maintain any indices to speedup the retrieval. According to Gartner [3], 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 much better likelihood to find the content people are interested in.

We argue that the semantic information should be directly embedded into the file system itself. This not only makes it easier to use but also can improve efficiency with respect to storage usage and data access. If the documents stored in the system are already organized according to their semantics, we only need to look at the documents that are close in semantics. Moreover, embedding a single flexible semantic index in the file system can remove the redundancy among indices kept by separate tools.

In this paper, we focus on demonstrating the benefits of integrating semantic information in a deep archival file system. With the cost and density of the random access devices approaching those of the magnetic tapes, it is affordable to archive each 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 backed up version is to find the right document and the right version. Currently the only way to locate the version is by remembering the date that the version was produced. In many cases, 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 may make many variations to the clips; to produce a variant the artist goes through several tries to get the right “look and feel”. In the process the artist may go back to previous versions (may not be the latest version). Also, the artist may need to incorporate scenes produced by other artists, but the only thing she may know is that these files have certain semantics. Such things also happen in other environments e.g. universities, research laboratories, and medical institutions.

† Author names in alphabetical order

Integrating semantic information into a deep archival system offers the following advantage over current techniques:

- Easy to locate the appropriate versions using semantic-based retrieval capabilities and provides the basis for understanding the semantic evolution of the documents.
- Clustering documents that are semantically close for the purposes of finding related materials and purging.
- A novel use of semantic information is to eliminate duplicate information through the notion of *semantic hashing* (see section 3.2). 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 personal edits, different releases).

In this paper we propose a semantic-based deep archival system *Sedar*. To our knowledge, Sedar is the only system that provides the following features. (i) It uses semantic information to organize and retrieve files. (ii) It uses semantic hashing to reduce the storage consumption that is robust against misalignments. (iii) Storage is provided over completely decentralized Peer-to-Peer storage utility allowing horizontal scalability. (iv) It provides high availability using erasure-coding techniques.

2. Related Work

Semantic files systems [1] and *HAC* [2] provides support for maintaining orthogonal namespace by executing queries and constructing the namespace to organize the query results. These systems provide support only for simple keyword-based queries and requires some level of support from the applications. Also, these two systems do not address deep archival capability and availability.

venti [4] provides versioning capability through a block level interface. It uses block level hashing to avoid storing redundant copies of block data. *SnapMirror* [5] provides versioning by taking advantage of meta-data stored in the underlying file system. The *Elephant file system* [6] provides versioning capability and retention policies that can be applied at a file level. However, none of the above techniques provide semantic storage and retrieval capabilities. Besides, approaches based on the block-level hashing does not handle misalignments in the objects.

SFSRO [8] uses block level hashing by applying SHA-1 recursively to build data-structure to support distribution of read-only contents. This system focuses on security not on providing deep archival capability.

OceanStore [7] provides versioning of objects and reliability using erasure-coding technique. Like the other systems, it does not provide any capability to store, retrieve and manipulate files using semantic information.

3. Architecture

Sedar is a semantic-based deep archival system. In Sedar, each time a file is modified and closed, a new version of the file is produced. Different instances of the same file will be given a different version number. The metadata, however, is not versioned, but we support a notion of virtual snapshot using timestamps that allows accessing the namespace arbitrary back in time. The system provides a semantic-based interface that allows clients to locate files according to the semantics in the files. The system can create materialized views of the results by presenting them through the traditional file system abstraction.

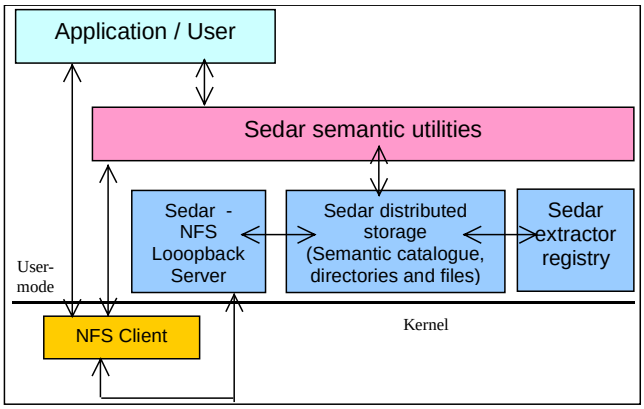

Figure 1: Major system components in Sedar

3.1 Major System Components

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

NFS loop-back server: To access the file system, client mounts Sedar, which presents a standard NFS interface.

Catalogue: Catalogue in Sedar contains an index of the files based on their semantic vectors (SV) derived from the contents of the files. A semantic vector is a vector of file type specific features extracted from file contents. For instance, the vector space model [9] extracts the term frequency information from text documents and latent semantic indexing [10] use matrix decomposition and truncation to discover the semantic underlying terms and documents. Welsh et.al [11] derive frequency, amplitude, and tempo features from encoded music data. In the catalogue, the index for a file contains the Inode number of the file and a version number. 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 (e.g, the difference between 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 statistical analysis to derive features from the bit stream. Similarly, for each data type, a different diff function can be introduced. The extractor registry provides an extensible interface that allows new extractor and diff functions to be added.

Sedar distributed storage (SDS): SDS provides basic support for storing and retrieving files, directories and the catalogue. We are implementing SDS on top of distributed peer-peer storage utility, CAN [12], which provides a logical abstraction to aggregate physical storage resources. The "root" of Sedar is kept at a well-known location in the storage utility. When a SDS module starts, it contacts the node that contains the "root" of the system and presents the namespace to the client during the mount time.

Sedar semantic utility: 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 catalogue for query. This is analogous to query-by-example (QBE) in the database system, but is much more powerful.

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 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 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 "Inode" that references the object. An Inode can be derived, for example, by hashing the name of the object.

In Sedar, an Inode of a directory file does not contain version information. However, we use timestamp to handle namespace changes (see section 4.3 for more details). Inode of a file contains entry for each version of the file, Inode of the base document and the ID of the "diff" (Figure 2). Applying the diff to the base document will produce the whole document. One way to compute an ID for a diff is to hash its content using consistent hash such as SHA-1. The Inode is responsible for computing the SV of a document by invoking the appropriate extractor depending on the file type. It is

also responsible for reassembling the documents. Once a SV is computed, the Inode contacts the catalogue to locate the semantically closest files to compute the diff.

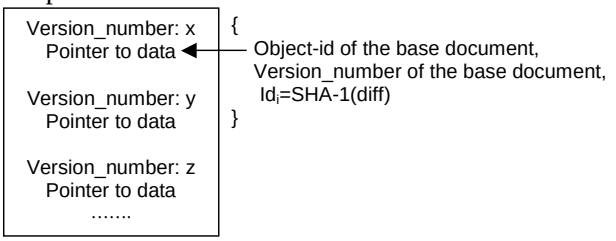

Figure 2: Sedar Inode layout

Sedar employs a novel technique to condense storage utilization based on the semantic vector of a document. The basic idea is to use 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*.

Sedar produces 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 version behind the most recent version, to get the maximum benefit in terms of storage utilization ideally it should be diffed with the version it was produced from. Even if we do an exhaustive-search using RCS to reduce the diff, it could still be a time consuming process as it may need to go through several hundreds of versions. 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.

The catalogue in Sedar is a distributed index that provides functionality such as retrieving IDs of objects that are semantically close given a semantic vector. One way to implement and store the catalogue is to partition the vector space into multiple regions, each region is assigned to a node in SDS 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 [13], we describe a technique that can place SVs that are semantically close to each other logically close in CAN (Figure 3 illustrates this).

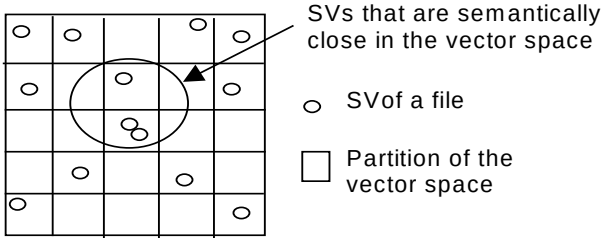

Figure 3: An example implementation of Sedar catalogue using CAN

3.3 Important File System Operations

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

3.3.1 Mount

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

3.3.2 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.

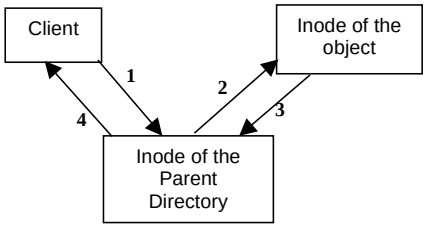

Figure 4: Illustration of Create/Mkdir protocol. *(1) The client contacts the parent directory (2) The parent directory checks to see if the entry already exists, if so it contacts the "Inode" to assign a version number. Otherwise, a new Inode is created. (3) Inode assigns a new version and returns it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

3.3.3 Lookup

Lookup is typically done before reading or writing to a file. 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. For example, to perform lookup on "/etc/hosts", the client must first resolve "/etc". 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.

3.3.4 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.

3.3.5 Write

Parameters for the write are the same as in read. Writes in Sedar are buffered at the Inode until the client "closes" the file. 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 catalogue service. Once the base document is located, Sedar compares them to create a diff. The diff is stored in SDS by performing 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 catalogue 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 the availability of Sedar, we plan to apply erasure code(s) to store the data to provide fault-resiliency and availability of the system. It has been shown that erasure code(s) provides better availability then simple replication with the same amount of space overhead [14].

4.2 Virtual Snapshot of Namespace

In Sedar, directories are not versioned. This is done to avoid recursive update leading all the way to the root (as is the case with *venti* and *SnapMirror*) when any namespace change occurs. As a result, it is 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. 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. For a file, it 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 a major issue. To make accessing a snapshot efficient, we employ some caching scheme to improve the access latency.

4.3 Conceptual Deletion

Not being able to modify the name space would be too rigid a requirement. In Sedar, 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.

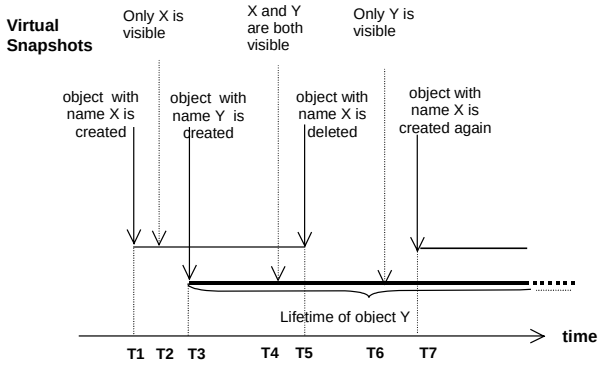

Figure 5: Example showing the conceptual deletion of objects and snapshots of name spaces

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 the similar SVs, how to find the file that can produce a small diff is a challenge. There are several possibilities to tackle this. 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: (i) we sample several files with close SVs; (ii) 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, reassembling the documents, and the distributed metadata operations.

6. Conclusion

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

7. References

1. Gifford, D.K., et al. *Semantic File Systems*. in *13th ACM Symposium on Operating Systems Principles (SOSP)*. 1991.
2. Gopal, B. and U. Manber. *Intergrating Content-based Access machanisms with Hierarchical File Systems*. in *Usenix OSDI*. 1999. New Orleans, Louisiana, USA.
3. Gartner, *Data Management Trends: Growth Drivers and New Technology Requirements.* Planet Storage 2002.
4. Quinlan, S. and S. Dorward. *Venti: a new approach to archival storage*. in *First USENIX conference on File and Storage Technologies*. 2002. Monterey, CA, USA.
5. Patterson, R.H., et al. *SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery*. in *First USENIX conference on File and Storage Technologies*. 2002. Monterey, CA, USA.
6. Santry, D.S., et al. *Deciding When to Forget in the Elephant File System*. in *17th ACM Symposium on Operating Systems Principles (SOSP)*. 1999.
7. Kubiatowicz, J., et al. *OceanStore: An Architecture for Global-Scale Persistent Storage*. in *ASPLOS 2000*. 2000. MA, USA.
8. Fu, K., F. Kaashoek, and D. Mazieres. *Fast and Secure Distributed Read-only File System*. in *4th Symposium on Operating Systems Design and Implementation (OSDI)*. 2000. San Diefo, California, USA.
9. Berry, M.W., Z. Drmac, and E.R. Jessup. *Matrices, Vector Spaces, and Information Retrieval*. in *Society for Industrial and Applied Mathematics Review*. 1999. San Diego, CA, USA.
10. Deerwester, S.C., et al., *Indexing by Latent Semantic Analysis.* Journal of the American Society of Information Science, 1990. **41**: p. 391-407.
11. Welsh, M., et al., *Querying Large Collections of Music for Similarity*. 1999: Berkeley, CA, USA.
12. Ratnasamy, S., et al. *A Scalable Content-Addressable Network*. in *ACM SIGCOMM*. 2001. San Diego, CA, USA.
13. Tang, C., Z. Xu, and M. Mahalingam, *PeerSearch: Efficient Information retrieval in Peer-Peer Networks*. 2002, Hewlett-Packard Labs: Palo Alto.
14. Weatherspoon, H. and J.D. Kubiatowicz. *Erasure Coding vs. Replication: A Quantitative Comparison*. in *1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*. 2002. MA, USA.