Implementing a Reliable Digital Object Archive*

Brian Cooper, Arturo Crespo and Hector Garcia-Molina
Department of Computer Science
Stanford University
{cooperb,crespo,hector}@DB.Stanford.EDU

Extended version

ABSTRACT

An *Archival Repository* reliably stores digital objects for long periods of time (decades or centuries). The archival nature of the system requires new techniques for storing, indexing, and replicating digital objects. In this paper we discuss the specialized indexing needs of a *write-once* archive. We also present a *reliability algorithm* for effectively replicating sets of related objects. We describe an administrative user interface and a data import utility for archival repositories. Finally, we discuss and evaluate a prototype repository we have built, the Stanford Archival Vault, SAV.

KEYWORDS: archival storage, preservation of digital objects, replication, archival repository

1 INTRODUCTION

Information stored and managed by today's digital libraries can be lost within years or decades if special care is not taken. The causes include media and system failures, format obsolescence and bankruptcy of publishers. At Stanford we have implemented a prototype archival repository, the Stanford Archival Vault (SAV, pronounced "save"), for the long term preservation of digital objects. These objects may include documents, their metadata, and the programs for interpreting formats. Our repository does not entirely solve the preservation problem, but we believe it provides an extremely reliable storage infrastructure for preserving digital objects, even as hardware, software, and organizations evolve.

As we implemented and tested our SAV prototype, we identified some unexpected, important challenges that led us to modify our initial design, and to develop some new storage and replication techniques. We believe that the encountered challenges were not unique to our system, but represent some fundamental problems that will be faced in the design of any type of digital library preservation system.

For example, the nature of an archival repository implies that objects should be preserved and not erased. As a result, a repository should not allow users to arbitrarily delete or overwrite digital objects. This *write-once* policy, which is not present in most conventional data stores, forces us to manage data differently. For instance, consider a "set" object that contains pointers to the different materializations of a given document (e.g., the postscript version, the plain text version). The usual way of updating this set is to write a new pointer into the set object, or to delete a pointer from the object. Because the write-once policy forbids such changes, managing collections of objects using sets requires new storage structures. Furthermore, these new structures require specifically tailored indexes that can speed up common accesses to digital library sets.

A second area where we faced unexpected challenges was in the configuration of replication "agreements." Any archival repository must backup its digital objects to remote systems, and hence must enter into some type of agreement with the remote system regarding what objects to replicate. Agreements need to be flexible so that different arrangements can be described, e.g., a library L_1 may wish to backup all its technical reports (TRs) at library L_2, but in addition Physics TRs should be backed up at L_3. Library L_2 may in turn wish to replicate some of the TRs from L_1 at another site L_4, while simultaneously replicating some of its materials back at L_1. At the same time, it is important that new documents be automatically and fully incorporated into the proper agreements, without human intervention. For instance, suppose that a new Physics TR is created, consisting of two materializations, a postscript object, and a plain text object. As soon as the "root" digital object for this TR (e.g., the one that links to its components) is added to the set of Physics TRs, all the components should be implicitly added to the proper agreements and automatically backed up at L_2 and L_3. Achieving this flexibility and automation required the concepts of *replication sets* and *annotated links*, concepts that will be useful in any archival repository.

In this paper we discuss the challenges in implementing SAV and the lessons we learned. We describe the mechanisms that were developed and that could be used in any archival repository. Many of the problems we encountered have been described before in other domains; see Section 7 for related work. Here we build on previously developed techniques and, where necessary, present new techniques. Specifically, we make the following contributions:

- We identify the need for an index of the link structure between objects, or *pointer index*. We discuss other important

*This material is based upon work supported by the National Science Foundation under Award 9811992.

indexes and how, if necessary, these indexes can be built from the pointer index.

- We present a reliability algorithm that replicates digital objects, and detects and corrects corruption in these objects.
- We examine how to use annotated links that restrict traversals over a graph for the purpose of conveniently specifying replication sets. These sets are used by the reliability algorithm, and must grow implicitly for automatic operation of the system.
- We describe an administrative user interface that provides access to objects in a repository, with low system overhead.
- We introduce the InfoMonitor, an implemented software package for migrating real-world data (e.g., from a web site) into a repository.
- We present experimental performance results for SAV that illustrate the efficiency and costs of the techniques we describe. Our system scales well to large data sets.

This paper is organized as follows. First, we present a general model for an archival repository in Section 2. Then, in Section 3 we describe the object storage component of the system. Section 4 discusses the reliability layer, and Section 5 examines the user interface. Section 6 presents the InfoMonitor, while Section 7 discusses related work.

2 COMPONENTS OF AN ARCHIVAL REPOSITORY

Figure 1 shows the architecture of a prototypical archival repository. Our implemented SAV follows this basic design. However, here we address the general principles and features that would form the basis of *any* archival repository. (For specific details of the SAV architecture, refer to [11]).

The architecture in Figure 1 shows six distinct components of the system. The first component is the *object store*. This component stores and indexes digital objects so that they can be efficiently retrieved by other modules. In addition, the object store manages the assignment of object handles (Section 2.1), indexing, and caching of digital objects. The object store can be built on top of an existing storage system, such as a file system or DBMS.

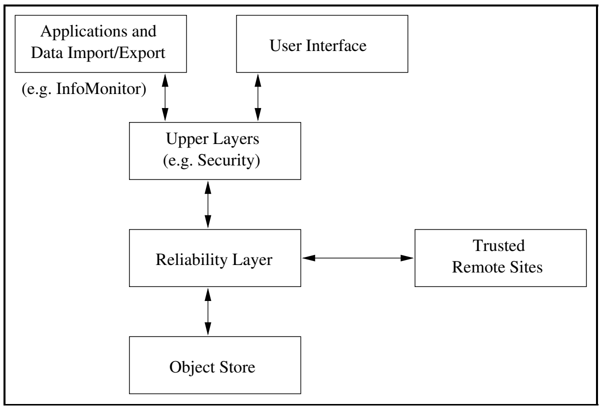

Figure 1: Architecture of the Archival Repository.

The long term archiving function of the repository is provided by the *reliability layer*, which manages object replication and corruption detection. This layer relies on different repository sites, usually geographically dispersed, to store copies of the objects. The reliability components at the various sites collaborate in detecting missing or corrupted information, and restoring it. We assume that remote reliability components are trusted. Communications among trusted reliability components can be encrypted and authenticated using standard techniques. The reliability layer can be configured in various ways (e.g., number of sites involved, number of copies needed for each object) to achieve different levels of reliability and system cost; the determination of appropriate configuration parameters is investigated in [12].

Upper layers on top of the reliability layer provide additional functionality, such as user security, intellectual property management, and query processing. The upper layers provide a programming interface (API) and appropriate information models so that various "applications" can access the repository. One application is a *user interface* component that allows users to view the contents of the repository and perform operations on it. Another important application is an import/export utility that provides batch migration of objects into and out of the repository, from digital libraries that do not provide the high reliability of the archive.

In this paper we focus on the lower system layers (object store and reliability), which are the ones that have been implemented in our initial SAV prototype. However, we do cover two important applications that must deal directly with the lower layers. One is a user interface for the system administrator (Section 5) that allows him to view the digital objects in the repository, create new objects, group semantically related items together, and construct agreements to replicate objects. A second application is the InfoMonitor (Section 6), which migrates information from a standard file system or web site into the repository.

2.1 Digital objects

A digital object in our system consists of a list of fields (name/value pairs), and is assigned an object handle. This model has the advantage of being both simple and powerful enough to store most types of information. An object handle is used by the system to efficiently locate an object. Handles are seldom seen by users. Users see human-readable names that are mapped by the system to one or more handles. For example, a user requesting the "postscript" version of "Tech Report #512" may be given access to the object with handle "62975." Our SAV system generates object handles by computing a signature of the object's contents. However, other mechanisms for assigning handles are possible. The work we describe here is equally applicable to any handle protocol.

The name/value pairs are defined by the creator of the object, who generates as many fields as necessary. These fields store content data, metadata, or any other useful values. Moreover, by storing another object's handle as the value of a field, an object creator can construct a relationship between objects. Such a *reference field* represents a "link" between two objects. To illustrate, a technical report object could include fields with names `AUTHORS`, `TITLE`, `CONTENT`, `PREVIOUS`, `HANDLE`, and `CHECK`. Field `PREVIOUS` could contain an object reference to the previous version of the technical report. In this way, a chain of report versions could be represented in the archive. More complex data structures (trees, sets, version graphs [40, 21], etc.) can be built using object references. Other data structures that may be useful are described in [11].

Two fields are required in all objects. Field `HANDLE` is a required reference field containing the handle of the object itself, while `CHECK` is an error detection code (e.g., CRC) computed over all remaining fields. These two fields make it possible to verify that a given object is not corrupted and is indeed the object one believes it to be.

2.2 AR Properties

In order to protect digital objects against loss over time, in general an archival repository must enforce certain properties. The *no deletions* policy specifies that users should not have the capability to delete objects once they are archived. A user can "take an object out of circulation" by changing its access rights, but this is different from physically erasing it from the repository. Allowing users to delete objects is dangerous in an archival system. Intentional deletions introduce ambiguous situations where it is not clear if a missing object was deleted by a user (and should not be restored) or lost due to some error (and should be restored). With no intentional deletions, the reliability layer simply restores any missing objects, leading to much better long term reliability.

Similarly, the *no modifications* policy prevents users from changing archived data. Modification are instead handled by creating version chains, with a newer object pointing to an older object via an object reference. No modifications again eliminates ambiguous states where it is unclear which is the "right" instance of a replicated object to restore. With version chains, any lost or corrupted version is restored to its original state. The no deletions and no modifications properties together define a *write-once* archive, where data, once written, is never erased. Write-once is a policy in SAV, not a requirement of the underlying media as in some other write-once schemes [14] (see Section 7 for related work on write-once storage).

The third property is *universal handles*. This property guarantees that an object retains its handle regardless of which repositories it is replicated to, and that the handle is unique within the repository network. Thus, a handle unambiguously identifies a single object. Without this property, the system would have to explicitly record what objects are copies of which, greatly increasing the chances of errors. Moreover, with universal handles, object references can be unambiguously resolved, allowing the structure of a graph of objects to be retained even as the objects are replicated to different sites. Universal handles also has important efficiency benefits; for example, two sites can quickly determine whether they have the same objects simply by comparing lists of handles.

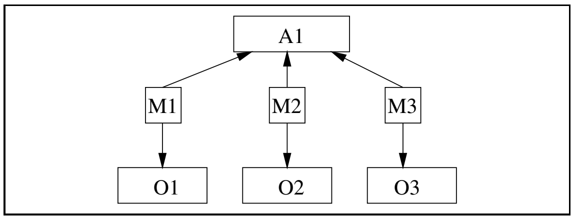

Figure 2: Structure of set $\{O_1, O_2, O_3\}$

3 OBJECT STORE

The write-once policy forces us to represent related objects in a way that is unlike traditional data stores. To illustrate, Figure 2 shows how a "set" can be represented. This set may represent a collection of technical reports, the set of materializations of one report, the set of replication agreements at one site (see Section 4), and so on. The set is initially created by generating a "set anchor" A_1 object. An object like O_2 is added to the set by creating a "set member" (represented by M_2 in the figure) which is an intermediate object pointing to both A_1 and O_2. A member O_2 could be deleted (not shown in the figure) by adding a "remove set member object" that links to A_1 and M_2. All changes are recorded by adding objects rather than by modifying objects.

A problem with write-once structures is that they are difficult to traverse. For instance, in order to find all of the members of A_1, it is necessary to identify the objects that point to A_1 (these objects would be the set member objects, e.g., M_1, M_2..., that also point to O_1, O_2...). One solution is to scan all repository objects, looking for objects that point to A_1. Clearly this traversal is very expensive, so we need auxiliary indexes to help us locate objects of interest. This "who points to me?" problem exists in other domains where objects are connected by directed links (e.g., hypermedia [22]). Our approach is described in Section 3.1, along with other important indexes. Indexes need to be modified, so they cannot be stored as digital objects, and do not enjoy the high reliability of digital objects. Section 3.2 discusses special mechanisms to ensure the correctness of indexes.

3.1 Indexing digital objects

A first critical index is the *handle index* that maps handles to the site-specific identifier (e.g., file name) that locates the object. This index is best implemented as a dictionary (e.g., hash table or balanced binary search tree) with universal handles as keys. This index, like the others we describe, is incrementally maintained. That is, as new objects are created, the index is notified so the appropriate handle-identifier pair is added. The handle index makes universal handles feasible. Without site-specific information in a handle, and without a handle index, one would be forced to find an object O_1

by scanning all repository objects looking for one with field `HANDLE` = O_1.

Another important index is the *pointer index* that gives the handles of all objects that link to a given object O_i. For example, for A_1 in Figure 2, the pointer index can quickly give us the handles for M_1, M_2 and M_3, from which we can find the members of set A_1. Note than in a traditional system a pointer index may be unnecessary if all references are "doubly linked." However, in an archival repository, A_1 cannot point to M_1 (which was created after A_1). Hence, a pointer index is essential. Again, a pointer index is best implemented as a dictionary. For convenience, the pointer index can be extended to list the outgoing links for each object. This makes it possible to traverse the repository's graph structure without retrieving the objects themselves.

To make a pointer index feasible, stored fields (Section 2.1) that contain references must be tagged as such. This allows the system to scan repository objects, extract references and build the index. The creator of an object must tag handle fields, either by indicating they are of "handle type" or by using field names that the system recognizes as containing handles (e.g., `PREVIOUS` in our earlier example).

The third type of index is an *object structure index*, designed to record the members of a particular object structure, e.g., a set or a version chain. For example, if we look up A_1 of Figure 2 in a set index, we would directly obtain the handles for O_1, O_2 and O_3. This same information could be obtained from the pointer index, but at a greater cost in execution time. (With a pointer index we would have to examine *all* objects that point to A_1, look for the set member objects, and then follow their links to the members.) Moreover, the set index can also give us a list of all sets in use, and (if properly inverted) the sets a given member participates in.

If space is more valuable than speed, some of these indexes can be eliminated. For example, an object structure index is a specialized view of the pointer index, and the SAV can query the pointer index rather than materialize a "set index". The handle index can be folded into the pointer index, especially if the site-specific identifier (filename) can be computed from the handle itself. For example, if the filename is the hexadecimal representation of the handle, then the list of handles indexed in the pointer index is equivalent to the handle index. Using the pointer index to emulate other indexes does not introduce significant efficiency overhead, but eliminating the pointer index is very expensive for reasons discussed above. Thus, the only index which must be materialized is the pointer index, and other indexes can be materialized to trade space for speed. The implications of this issue in the context of scalability are discussed in Section 3.3.

3.2 Maintaining index consistency

Indexes are important for the operation of the repository, yet they are inherently not as reliable as digital objects. First, it does not make sense to replicate indexes across sites to achieve reliability. (Indexes contain site specific information that is not useful at the remote sites, and since indexes change often, updating the remote copies would be too expensive). Second, since indexes are updated in place, they are much more prone to software errors than write-once digital objects.

There are two steps to ensure that index errors do not corrupt the underlying digital objects. The first step is to make indexes *disposable*. This means that no information that is critical for the long-term survival of the repository should be placed in an index. In other words, it should be possible to at any time throw away all indexes and reconstruct correct indexes from the underlying digital objects. As a corollary, all index information must be considered a hint only. For example, if a pointer index tells us that object O_1 points to O_2, we must verify this (by looking at the actual objects) before performing a critical operation based on this information.

With disposable indexes, a corrupted index will not adversely affect the digital objects, but can still be very inconvenient. For example, consider a set A_2 representing the three available recordings for a given song (e.g., MP3, wav, midi). If the index is corrupted, the index may tell us that only two recordings are available, or may give us a recording for a different song! A user query could check and ignore the bogus recording, but it will not easily discover that there is a missing recording. The information is not lost, since the recording objects are still in the repository, and are still linked to A_2. Yet, to avoid inconveniencing the user, it is very important to make every effort to ensure that the indexes are consistent with the uncorrupted digital objects.

There are two ways to ensure this consistency of indexes:

- *Rebuild from scratch*: Periodically discard an index, and completely rebuild it from the objects in the archive. The rebuild procedure is also useful when objects are added in bulk through a data import utility (see Figure 1).
- *Check and repair*: An index is checked and fixed incrementally.

To illustrate a check and repair process, consider checking the handle index. The object store iterates through each of the handles in the index, and loads the corresponding object from disk. Each object is then be examined to ensure that its `HANDLE` matches what the index reports. If not, the "bad" index entry referring to that object is deleted, and a new, correct index entry is added.

Note that index rebuilding easily discovers objects that are completely missing from the index, while a check and repair task can only verify existing entries in the index. On the other hand, check and repair allows the index to be available continuously, while the index created by the rebuild task is not available until the rebuild is complete. (Of course, the old, possibly corrupted index could still be used to serve requests while the new index is being built.)

In our implemented SAV system, indexes are kept in main memory and rebuilt from scratch at system startup. They are also rebuilt at the prompting of a user, or at predefined intervals. A check and repair mechanism could be added in the future.

3.3 Performance measurements

To evaluate the overhead of building indexes, we conducted experiments on our SAV prototype, running on an Gateway E-4200 (450 MHz Pentium III, 256 MB RAM, 128 MB swap, Red Hat Linux 6.0). The SAV is implemented in both in Java 2 and C++; the measurements presented here are from the C++ version. Digital objects containing real documents from the Stanford Database Group's web site were stored in the archive. Five object sets of different sizes were tested in order to assess scalability. The smallest set contained over 54,000 objects and 2 GB of total data, while the largest contained over 270,000 objects and 10 GB of total data. In each set, the average object size was 39 KB. For comparison, the largest data set (10 GB) represents the archived contents of approximately 25 average-sized web sites[1] [9].

The results are shown in Figure 3. The three lines in the figure represent the three tasks required to rebuild the handle, pointer and set indexes. These tasks are: read objects from disk (solid line), compute the CRCs to detect corruption (dotted line) and index the objects (dashed line). The times scale linearly with the size of the archived data set. The complete index building operation requires an average of 13 milliseconds per archive object (342 seconds per gigabyte), and this time is dominated by the disk read (77%) and CRC computation (21%). The high overhead of the disk read and CRC computation is mitigated by the fact that indexes are rarely rebuilt, and most SAV operations avoid these costs by using the indexes after they are already built. Moreover, any scheme that validates indexes by examining the actual objects on disk would incur these costs; our system is not unusual in this respect.

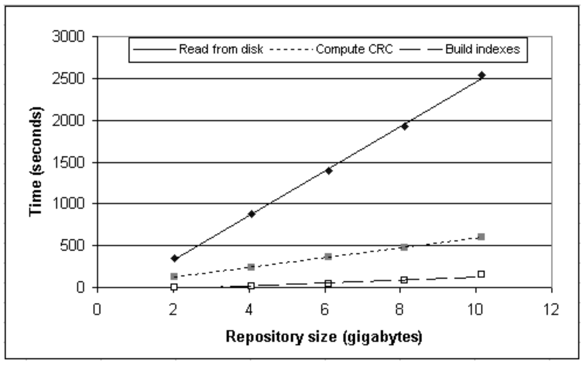

Figure 3: Performance of the object store.

Of course, it is very good that the cost to build indexes scales linearly, but such cost may still be significant for large archives. One solution is to rebuild each type of index at a different time. Another solution is to partition a repository into smaller sets that are reindexed at separate times. This would spread out the rebuilding over time. If this scheme is used, there must be some mechanism to deal with object references that cross partitions, perhaps by querying the indexes for both partitions simultaneously.

It is reasonable to ask how many objects can be indexed before the indexes no longer fit in main memory. We measured the per-object size of indexes as 57 bytes for the handle index, 76 bytes for the pointer index, and 9 bytes for the set index. We assume that, to save space, only the pointer index is materialized (as discussed in Section 3.1). If we dedicated 128 MB of RAM to indexes, the SAV could index over 1.7 million objects, or 65 GB of archived data[2]. For larger archives, more RAM could be purchased, or the index could be stored on disk and efficiently accessed using known techniques [16].

4 RELIABILITY LAYER

As described in Section 1, the replication layer backs up objects remotely, detects lost or corrupted objects, and restores them to their pristine state when necessary. The challenge is to develop flexible mechanisms for determining what sites participate in replication agreements, and what objects are backed up where. In addition, we need efficient mechanisms for checking and restoring information. In this section we describe the techniques and algorithms that were developed as the SAV prototype was implemented, but that we believe are well suited for any archival repository.

The example shown in Figure 4a illustrates the basic replication steps we follow. The replication process begins when a *replication agreement* R_1 is created at one of the three sites (Stanford in the example). Object R_1 identifies the sites that participate (Stanford, MIT, Berkeley) and the objects that are to be replicated. For now, let us assume that R_1 simply contains pointers to the objects to replicate, O_1 and O_2. Objects R_1, O_1 and O_2 initially exist only at Stanford, so Stanford conducts the first site check. The Stanford site contacts the MIT site and discovers that MIT does not yet know about the agreement, so that all three objects are replicated to MIT.[3] Similarly, all three objects are copied to Berkeley (Figure 4b).

Each of the three sites then begins a cycle of repeated site checks, connecting to the other two sites and comparing snapshots. As long as there are no errors, the snapshots will agree. However, consider the situation where O_1 is lost at Stanford due to a disk failure. The next site to perform a site check

[1] The Stanford Database Group's web site contains five times as many web pages as the average site as reported in [9]. The small data set (2 GB) contains the archived contents of our group's site, and the larger sets were produced by repeatedly archiving the group's site to produce slightly different objects.

[2] A large repository may also use compression [42] to save disk space.

[3] As described earlier, the reliability layers at each site trust each other, so they willingly take each others' agreements and objects. Clearly, before R_1 was created, Stanford checked with the other sites to see if there was enough storage capacity, or to arrange for payment for the service.

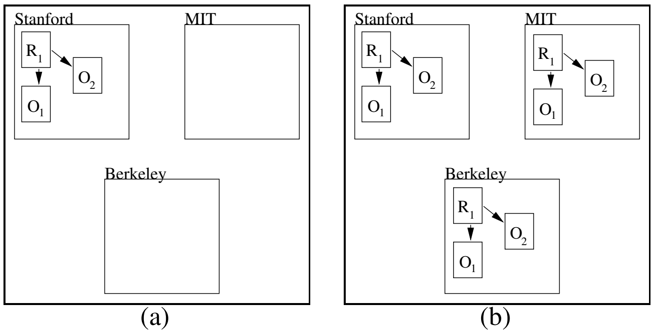

Figure 4: A replication network

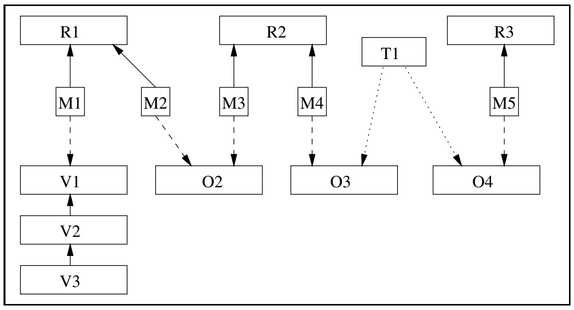

Figure 5: Example replication sets.

will notice that O_1 is missing, so O_1 will be copied back to the Stanford site.

4.1 Replication networks

Our example illustrates a *strongly connected* replication network. Each of the sites holding a copy of R_1 knows about the other sites, and each site contacts every other site during the site check. If there are N sites in the network, each site check must contact $N-1$ sites. This structure is recorded in R_1 by including a complete list of the sites in the agreement.

Other structures are possible. For example, in a *weakly connected* network, each site is connected to some, but not all, of the other sites. The topography of the structure could be a cycle, a tree, or some other structure. The strongly connected network has the advantage that each site check connects with every site, which means that new objects are quickly replicated to all sites. In contrast, the weakly connected network allows each site to connect to a fixed number of remote sites (e.g., two) even as the number of sites N in the network grows. Because fewer sites are contacted, site checks take less time and so they can be performed more frequently. This decreases the interval between the occurrence of a failure and the detection and correction of the error.

In a weakly connected network, links between sites are actually separate replication agreements, listing only the sites for that link. In order to construct weakly connected replication networks, it is therefore necessary for different agreements at the same site to include the same digital objects. This capability is one of the features of the snapshot construction algorithm described in the next section.

4.2 Constructing snapshots of the replication set

In Figure 4a we suggested that agreement R_1 point to the "covered" objects O_1 and O_2. This is clearly not a good idea since we could never add more objects to the agreement. (Digital object R_1 cannot be modified.) An alternative is to treat the agreement object as a set anchor, so that any object connected via a "set member" object is covered. For example, in Figure 5, R_2 would cover O_2 and O_3. (In this figure, please ignore for now the different types of pointers.) This is still not flexible enough, since new objects would have to be explicitly linked to R_2.

Our solution is to recursively define the covered objects in terms of the link structure of the repository. To illustrate, suppose we wish to cover all versions of a technical report under agreement R_1 in Figure 5. The different versions of the report, V_1, V_2 ... V_n, are related using a version chain, in which version V_i points to the previous version V_{i-1}. Initially, the first version V_1 is added to R_1 (through M_1). When V_2 is created, it need not be explicitly added to R_1. Our replication algorithm will implicitly include V_2 in R_1 because there is a path to it from R_1 (via M_1 and V_1). As more versions are created, they are also implicitly included. Thus, the *replication set* of R_1 includes all objects recursively reachable from R_1 ("backwards" links count).

There is a problem with this simple description of a replication set. To illustrate, consider agreements R_1 and R_2 in Figure 5. Their replication sets are connected by O_2, so if we blindly include everything that is linked to R_1 in its replication set, we would include all of R_2's set! Even if agreements do not overlap, other objects may act as *bridges* and connect them. For instance, in Figure 5, object T_1 is such a bridge object. (Object T_1 may be linking objects written by the same author, for example.)

Our solution is to annotate repository links to indicate when they should be traversed in building replication sets. Some links, like the ones out of T_1 in Figure 5, should never be traversed. Links such as these have nothing to do with replication, and are shown as dotted lines in the figure. Other links like the ones between M_2 and O_2, and between M_3 and O_2, should only be traversed in the direction of their "arrow" to avoid merging replication sets. Such links are shown as dashed lines in the figure. When computing the replication set for R_1 we would reach O_2 but would stop there. Similarly, when computing the R_2 set we would also reach O_2, but would again stop there.

In summary, we use the concept of a graph with *annotated links*. In such a graph, every link is annotated in one of three ways:

1. *two-way recursive*: The link should always be traversed during a replication set traversal.
2. *one-way recursive*: The links should only be traversed in the direction of the link during a replication set traversal.

3. *non-recursive*: The link should never be traversed when defining a replication set.

   The annotated type of a link is specified when the link (and thus the object containing the link) is created. The example shown in Figure 5 can serve as a template for determining how links should be marked. If it is desirable to change the annotation on a link after it is created, then the replication set traversal algorithm must be extended to allow the annotations on links to be modified by an administrator. Since modifications cannot be written to objects, these modifications can be represented as version chains, and the traversal algorithm would be designed only to consider the most current version of a link when deciding whether to traverse it. This is an example of the generally applicable strategy of representing modifications as version chains rather than modifying digital objects themselves.

Link annotations have been used in other domains; Halasz and Schwartz [22] describes their use in the hypertext domain. Our technique is similar in that our annotations restrict graph traversals. However, the goal in hypertext systems is to facilitate human navigation, whereas our goal is to automate the process of discovering subgraphs (replication sets) of larger graphs (the entire object structure of the repository.) Moreover, the annotations in [22] determine the direction of the link (e.g., by denoting which object is the parent and which object is the child) whereas our annotations describe how to interpret the direction. (See Section 7.)

4.3 Detecting object corruption

Each site periodically constructs a *snapshot* of the replication set of each known agreement.[4] A snapshot includes the handles of all non-corrupted objects that are part of the agreement. Snapshots are then compared with the corresponding ones at remote sites.

Sometimes it is easy to see that an object is corrupted. For example, if an attempt to read an object from disk results in an error, corruption is clearly present. In addition, the reliability layer also must detect less obvious corruption that exists when an object can be read from disk but nonetheless contains incorrect information. This type of corruption is detected by comparing an object's stored `CHECK` value (see Section 2) with a freshly recalculated error detection code based on the current contents of the object.

The snapshot construction algorithm is as follows:

1. A list (called *snapshot*) is created and is initially empty.
2. A search stack is created and initially contains only the handle of the replication agreement.
3. A handle is popped off of the search stack; the object it identifies is the *current* object.

[4] The objects representing replication agreements form part of an implicit agreement among all sites. Thus, if an agreement object is lost at a site, it will be recovered from another site.

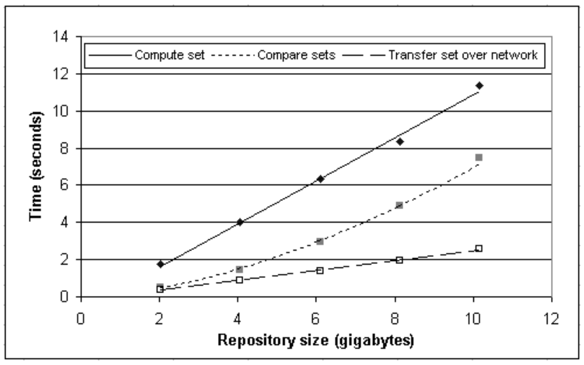

Figure 6: Performance of the reliability layer.

4. The *current* object is checked for corruption by comparing the recalculated error code with the value `CHECK` stored in the object. If *current* is corrupted, the object is ignored and the algorithm returns to step 3. If *current* is not corrupted, the algorithm continues.
5. The handle of the *current* object is added to the *snapshot* list.
6. Each of the links pointing to or from *current* are traversed (using the pointer index) if and only if such a traversal conforms to the annotation on the link. Traversing these links produces a set of objects. The handle of each of these objects is added to the search stack, unless the object has been seen before (infinite loops must be avoided).
7. If there are still handles on the search stack, the algorithm returns to step 3.

Once snapshots are created at both sites, the remote site sends the snapshot to the local site, and the local site performs a comparison. Any handles missing locally represent objects that must be retrieved from the remote site, and any handles missing remotely represent objects that must be sent to the remote site. Our current implementation performs the comparison by storing the local replication set in a red-black balanced binary tree and then searching this tree for each handle in the remote set. This process requires $nlog(n)$ time for n handles, and the performance ramifications of this growth are discussed in Section 4.4.

4.4 Performance measurements

In order to evaluate the performance of the reliability layer, we conducted experiments on our SAV prototype. We performed the reliability algorithm on the data sets described in Section 3.3, using two different SAV instances running on identical machines connected by 10 Mbit Ethernet. The measurements are shown in Figure 6. In the figure, the solid line represents the time to construct a snapshot at a particular repository site. This process must be repeated at both the local and remote sites for each site check; however, the snapshot construction at different sites can run concurrently. The snapshot construction time scales linearly with repository size, and represents an incremental duration of 40 microseconds per object (1.2 seconds per gigabyte).

The snapshot comparison time (dotted line in Figure 6) increases as $nlog(n)$ (as discussed above). This non-linearity is inconsequential since our implementation also examines each object on disk for corruption during the site check, and the I/O cost dominates the time to compare the snapshots by three orders of magnitude. If the examination for corruption is done lazily between (instead of during) site checks, then the comparison time would consume a larger fraction of the site check time (about $1/3$ in our experiments). If the non-linear growth of the comparison time hinders performance and scalability, we could substitute a scheme whose time grew linearly, for example by inserting handles into a hash table instead of a binary tree.

The amount of time to send a snapshot from one site to another was 10 microseconds per object (267 milliseconds/gigabyte), as shown by the dashed line in Figure 6. Various optimizations are possible for use with slow networks or very large repositories. For example, the remote site can compute a signature S (e.g., CRC) of all the handles in the snapshot. Instead of sending the entire snapshot, the remote site only sends S, a single number. The local site computes the signature of its snapshot, and compares both signatures. If the signatures match, then the snapshots are the same. If the signatures do not match, then the snapshots could be subdivided and signatures computed for each subdivision until the local site can determine what the differences are between the snapshots. This optimization is described in more detail in [10].

Another possibility is to perform the snapshot construction and comparison incrementally over a period of days. For example, both sites could start the traversal on the first day, but only descend a certain number of levels in a breadth first traversal of the replication set objects. This would produce partial snapshots, which the sites would compare. The sites would exchange any objects missing from the partial snapshots. On day two, both sites would descend further in the traversal to produce another partial snapshot. Eventually, both sites would reach the end of the traversal, at which point all of the partial snapshots that were produced would be equivalent to the complete snapshot. Then, the process would repeat at the first day again. In this way, only a small amount of bandwidth would be utilized each day. This scheme would require a mechanism for dealing with new objects added after the first day. Such objects could be included if they appear in a partial snapshot after they were added. Alternatively, they could be excluded until the snapshot process restarts.

5 USER INTERFACE

Our current SAV prototype includes an administrative user interface that lets a manager examine and modify the repository. In general, the goals for such an interface are as follows:

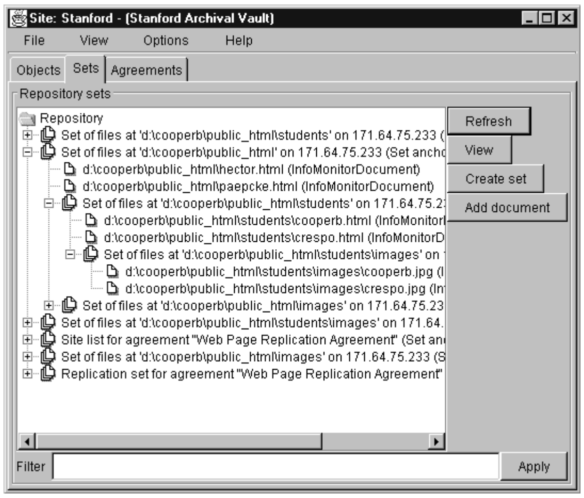

Figure 7: The *Sets* view

1. The user must be able to locate specific digital objects in the repository, even if the repository contains many objects.

2. The user must be able to easily perform structuring operations on objects, such as grouping related objects into sets, and to view the topology of object structures.

3. The user must be easily able to set configuration parameters of the system. This includes defining replication agreements.

4. The interface module must not significantly detract from the performance of the rest of the system.

The best way to achieve these goals is to provide specialized types of views into the repository:

- *objects view*: A general view which can display any object in the archive.
- *structure views*: Views that display common objects structures, such as sets or version chains.
- *configuration views*: Views which allow a user to configure the system and its replication agreements.

Our SAV prototype currently includes four different views, and will be extended to include others. Due to space limitations, in this section we only briefly illustrate two of the views. For a complete discussion, which covers performance issues related to the user interface, please see [6].

Figure 7 is a screen shot of our set interface (an example of a structure view) from the Java implementation. The objects that participate in sets can be viewed through a more generic interface (not shown here), but the set interface is especially tailored to show sets and their members clearly.

In the set view, only sets and their members are shown. A set is represented by the “stacked document” icon, and a set member is represented by the “single document” icon. The default view shows all of the sets in the repository and a

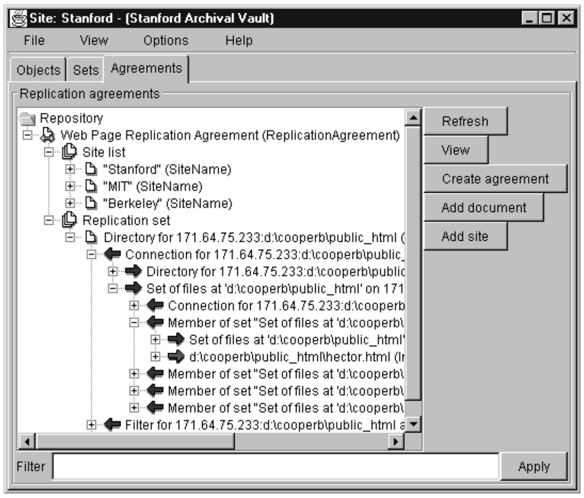

Figure 8: The *Agreements* view

descriptive string.[5] The filter window (bottom of Figure 7) can be used to restrict which sets are shown (using regular expressions). Set objects can be expanded (by clicking on the icon) to show the set members. If one of these members is another set, that set can be further expanded to show its members. The "View" button on the left lets one view the contents (label, value pairs) of a selected digital object. (A separate, specialized view window is opened.)

Because a structure view is specific to a particular object structure, it can also be used as an interface for constructing that particular structure. Figure 7 shows a "Create set" button, which can be used to create a new set, and an "Add document" button, which can be used to add an object to an existing set. The "Refresh" button is similar to a "reload" button on a web browser; it forces the interface driver to get fresh information from the repository. This decoupled interaction between the interface and the repository makes it unnecessary for the repository to continuously update the display. The menus at the top of the window provide additional functionality that is not discussed here.

An example of a configuration view is shown in Figure 8. This replication agreement interface lets administrators create and configure agreements. The default display of the replication agreement view is a list of the active agreements. Each agreement can be expanded to view the list of sites in the agreement as well as the replication set. Since replication sets are defined recursively (Section 4.2), our interface allows objects in the replication set to be expanded to reveal linked objects. In this way, a user can manually examine the graph that will be automatically traversed by the reliability algorithm. As before, individual objects can be viewed using the "View" button, and individual agreements can be found using the filter field. Finally, the "Create agreement," "Add site," and "Add document" buttons let the administrator enter new agreements, and add sites and objects to them.

6 THE INFOMONITOR

After developing SAV, we discovered a "sad fact" about archival repositories: Many users do not *want* to deposit their digital objects in an archival repository, or in any form of digital library for that matter! They are perfectly happy with their objects residing on conventional file systems or web servers, where they can use their favorite editors and tools to work on them. After all, it is not *their* job to ensure that their objects are available to future generations years from now. However, preservation *is* the job of a librarian, who needs tools to "capture" important objects in a way that does not require active participation by users (but of course requires user consent). The InfoMonitor we describe in this section represents one such tool; the goal is to provide an automated way to migrate data into the archive.[6]

The InfoMonitor serves as a "bridge" between a repository such as SAV and an existing environment where digital objects reside. Our example environment is a web site (but the InfoMonitor can be used in other scenarios too). Users continue to create, edit and access web pages using standard tools (e.g., text editors). The InfoMonitor carefully tracks the files representing the web pages, and decides what objects should be archived. In addition, it monitors changes to the files, *translating* those changes into repository updates.

One of the hardest challenges faced by the InfoMonitor is in deciding how to interpret the changes to the web site. For example, suppose that a web page is modified. Modifications are not allowed on the repository, so the action must be automatically translated into the creation of a new version of the corresponding digital object. If the web page is deleted, a "final" version is added in the repository, indicating that the web page was removed. Changes to the web site file structure must be carefully analyzed to determine how they impact the archived objects. For instance, if a web page is "moved" from one location to another, this action can be interpreted as a deletion followed by an insertion, or it can be interpreted as new version of the web page (where one of its properties, its file name, was changed).

The InfoMonitor offers an administrative user interface, analogous to the one described in Section 5. Through this interface, an administrator can define portions of the web site to archive (by setting "filters"), and can examine archived objects and how they map to web site files. The interface also offers a historical view, where archived objects can be viewed as of a given time. Finally, the administrator can also restore web site files based on the repository objects. Thus, the InfoMonitor offers a fairly automated way to archive a

[5] Currently, objects contain a `DISPLAYNOTES` field that describes their role or use. This field is used as the object description in the view. The filter window searches over these fields.

[6] This approach contrasts with some other tools that build linked object databases, such as the collaborative authoring tools in DeVise [19], which require and encourage human interaction.

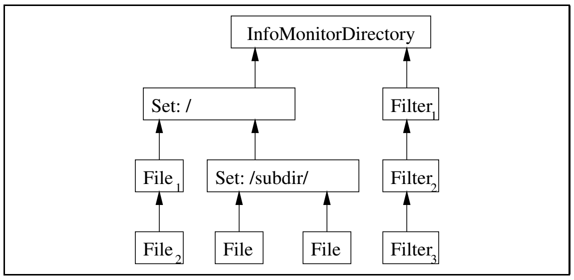

Figure 9: The InfoMonitor creates this data structure in the SAV.

web site. Web users do not need to perform explicit saves to the repository, yet their pages are safely archived.

Figure 9 illustrates how the InfoMonitor represents the web pages as digital objects. The left hand side structure mimics the target file structure, while the right side represents the selection filters and other data. If the top level InfoMonitor Directory is added to a replication agreement, then this entire structure will be replicated at other repository sites.

Initially, the structure of Figure 9 is created by a bulk load utility that scans the web site. (This same utility was used to acquire the data sets used for the experiments of Sections 3 and 4.) The InfoMonitor can perform two types of periodic checks to track the web site: a quick and a slow one. The quick scan compares the timestamps of files with those of the archived objects, to detect new or modified files. Timestamps can be unreliable, so the slow scan actually compares the contents of files to the archived content. In either case, as changes are observed, the appropriate objects are added to the archival repository.

The InfoMonitor has been implemented as part of our SAV prototype. It is currently being used to archive 26,000 files (2 GB) of our group's web site. Additional details and performance numbers are available in [7].

7 RELATED WORK

The digital library community has begun to focus on the problem of designing and implementing long term archives [17, 37, 20, 13, 15, 2, 36, 28]. Several projects have focused on building archives, including the Computing Research Repository [23], the Archival Intermemory Project [18, 4] and the Victorian Electronic Records Strategy [39]. These projects have focused on different archive architectures than the SAV design we discuss here, and information discovery, not preservation, has been the focus of many of the efforts. The San Diego Supercomputer Center [34] has examined indexing digital archives from the standpoint of metadata; such an infrastructure would be useful as a document discovery mechanism in the "upper layers" mentioned in Section 2. The Internet Archive [1] is building a collection of archived web pages, but so far has not addressed the problem of preservation.

The archiving problem is related to the problem of increasing the reliability of file systems. The traditional solution is data backup [5, 25, 29]. Several commercial producets use hierarchical replication systems to automatically backup and reliably store data [27, 8, 26]. The backup problem focuses on shorter durations than the archiving problem. Moreover, users of backup systems are usually interested in restoring the most current version of data, while archives are responsible for storing all versions. Another approach is to redesign the file system itself to incorporate more reliability features. One idea is to use Redundant Arrays of Inexpensive Disks (RAID) [32, 38], so that disk failures can be overcome. Others have suggested using logs to improve many aspects of the file system, including the reliability [35]. For example, the Clio Log File system [14] archives data to write once storage. Such systems could serve as the data storage component of our Object Store layer (Section 2).

Another related area is the problem of maintaining consistency between nodes in replicated databases. Much work has been done in designing algorithms for propagating data from one replicate to another [3, 33]. These systems focus on systems that allow updates and deletions of objects. Archival Repositories, which do not allow digital objects to be modified or erased, require different approaches. Similarly, filesystems using replication (such as LOCUS [41], Harp [30] or Zebra [24] among others) focus on providing high availability and fault tolerance for frequently accessed filesystems. Our focus is on long term reliability for data that may be archived for decades between accesses.

Finally, many of the issues we discuss here are also present in hypermedia systems [22, 19]. Although the problems are similar, hypermedia systems focus on presentation of objects as much as on the storage of objects, and also must cope with inconsistencies due to modifications and deletions. As a result, the general solutions tend to be similar (e.g., annotating links to restrict graph traversals) although the details and the implementation differ from what we present here.

8 CONCLUSIONS

In this paper we have discussed issues that arise when implementing a reliable archival storage system. Although we have discussed these issues from the perspective of our SAV design, they are relevant to the construction of any reliable archive. We have discussed solutions for defining and indexing digital objects and references between them in a write-once repository. We have discussed efficient algorithms for replicating objects to multiple sites using different replication networks, and for building and comparing snapshots of repository contents so that corruption can be detected. These algorithms allow the set of replicated objects to grow implicitly, rather than through the intervention of a human.

We have also described two "applications" that interface with SAV. One is an administrative user interface for monitoring and controlling SAV. The second is the InfoMonitor, a tool for automatically importing and tracking information outside the repository.

The SAV prototype demonstrates that a reliable archive can be built, that it can operate efficiently, and that it can interact effectively with the outside world.

REFERENCES

1. Internet Archive. The Internet Archive: Building an Internet library. http://www.rlg.org/longterm/index.html, 2000.

2. Howard Besser. Information longevity. http://-sunsite.berkeley.edu/Longevity/, 2000.

3. Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S. Seshadri, and Avi Silberschatz. Update propagation protocols for replicated databases. In *Proceedings of the ACM SIGMOD Conference*, 1999.

4. Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM DL Conference*, 1999.

5. Ann Chervenak, Vivekenand Vellanki, and Zachary Kurmas. Protecting file systems: A survey of backup techniques. In *Proceedings Joint NASA and IEEE Mass Storage Conference*, March 1998.

6. Brian Cooper, Arturo Crespo, and Hector Garcia-Molina. Implementing a reliable digital object archive. http://www-db.stanford.edu/pub/papers/-arpaperext.ps, 2000. Extended version of paper.

7. Brian Cooper and Hector Garcia-Molina. InfoMonitor: Unobtrusively archiving a World Wide Web server. http://www-db.stanford.edu/pub/papers/-fmpaper.ps, 2000. Technical Report.

8. IBM Corporation. Adstar distributed storage manager (ADSM) - distributed data recovery white paper. http://www.storage.ibm.com/storage/software/-adsm/adwhddr.htm, 1999.

9. Inktomi Corporation. Web surpasses one billion documents. http://www.inktomi.com/new/press/-billion.html, 2000.

10. Arturo Crespo and Hector Garcia-Molina. Awareness services for digital libraries. In *Lecture Notes in Computer Science*, volume 1324, 1997.

11. Arturo Crespo and Hector Garcia-Molina. Archival storage for digital libraries. In *Proceedings of the Third ACM DL Conference*, 1998.

12. Arturo Crespo and Hector Garcia-Molina. Modeling archival repositories for digital libraries. http://www-db.stanford.edu/pub/papers/archsim.ps, 1999. Technical Report.

13. Jean Deken. Writ in water? an exploration of the gap between archival construct and practice in the machine-readable environment. In *Working With Knowldge Conference*, May 1998. Accessible at http://www.slac.stanford.edu/pubs/slacpubs/-7000/slac-pub-7811.html.

14. Ross Finlayson and David Cheriton. Log files: An extended file service exploiting write-once storage. In *Proceedings of the 11th Symposium on Operating Systems Principles*, November 1987.

15. National Science Foundation. Workshop on Data Archival and Information Preservation: Executive summary. http://cecssrv1.cecs.missouri.edu/-NSFWorkshop/execsum.html, 1999.

16. Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom. *Database System Implementation*. Prentice Hall, Upper Saddle River, New Jersey, 2000.

17. John Garrett and Donald Waters. Preserving digital information: Report of the Task Force on Archiving of Digital Information, May 1996. Accessible at http://-www.rlg.org/ArchTF/.

18. Andrew Goldberg and Peter Yianilos. Towards an archival intermemory. In *Advances in Digital Libraries*, 1998.

19. Kaj Gronbaek and Randall Trigg. Design issues for a Dexter-based hypermedia system. *Communications of the ACM*, 37(2):40–49, February 1994.

20. Research Libraries Group. Long-term retention of digital research materials. http://www.rlg.org/longterm/-index.html, 2000.

21. Anja Haake and David Hicks. Verse: Towards hypertext versioning styles. In *Hypertext '96*, 1996.

22. Frank Halasz and Mayer Schwartz. The Dexter Hypertext Reference Model. *Communications of the ACM*, 37(2):30–39, February 1994.

23. Joseph Halpern and Carl Lagoze. The Computing Research Repository: Promoting the rapid dissemination and archiving of computer science research. In *Proceedings of the Fourth ACM DL Conference*, 1999.

24. John Hartman and John Ousterhout. The Zebra striped network file system. In *Proceedings 14th Symposium on Operating Systems Principles*, December 1993.

25. Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley. Logical vs. physical file system backup. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

26. Tivoli Systems Inc. Tivoli storage manager. http://-www.tivoli.com/products/index/storage_mgr/, 1999.

27. UniTree Software Inc. Unitree technical overview. http://www.unitree.com/overview/overview.htm, 1999.

28. Getty Conservation Institute. Time and Bits: Managing digital continuity. http://www.longnow.com/-10klibrary/TimeBitsDisc/, 1998.

29. Richard P. King, Nagui Halim, Hector Garcia-Molina, and Christos A. Polyzois. Management of a remote backup copy for disaster recovery. *TODS*, 16(2):338–68, 1991.

30. Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings 13th Symposium on Operating Systems Principles*, October 1991.

31. Stanford Conservation Online. Electronic storage media. http://palimpsest.stanford.edu/bytopic/electronic-records/electronic-storage-media/, 2000.

32. David Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Record*, 17(3):109–116, September 1988.

33. Michael Rabinovich, Narain Gehani, and Alex Kononov. Efficient update propagation in epidemic replicated databases. In *Proceedings of the 5th International Conference on Extending Database Technology*, 1996.

34. Arcot Rajasekar, Richard Marciano, and Reagan Moore. Collection-based persistent archives. http://www.sdsc.edu/NARA/-Publications/OTHER/Persistent/Persistent.html, 2000.

35. Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings 13th Symposium on Operating Systems Principles*, October 1991.

36. David Rosenthal and Vicky Reich. Permanent web publishing. http://lockss.stanford.edu/, 2000. To appear at Freenix, San Diego, CA, June 2000.

37. Jerome H. Saltzer. Technology, networks, and the library of the year 2000. In A. Bensoussan and J.-P. Verjus, editors, *In Future Tendencies in Computer Science, Control, and Applied Mathematics. Proceedings of the International Conference on the Occasion of the 25th Anniversary of INRIA*, pages 51–67, New York, 1992. Springer-Verlag.

38. G.A. Schloss and M. Stonebraker. Highly redundant management of distributed data. In *Proceedings of Workshop on the Management of Replicated Data*, pages 91–95. IEEE, IEEE Computing Society, November 1990.

39. Victorian Electronic Records Strategy. Victorian electronic records strategy final report. http://-home.vicnet.net.au/~ provic/vers/final.htm, 1999.

40. Walter Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.

41. Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings 9th Symposium on Operating Systems Principles*, October 1983.

42. Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.