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

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, digital objects, object replication, object indexing, user interface, 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 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 new 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. Specifically, we make the following contributions:

- We identify the most useful indexes for a write-once archival repository.
- We present a reliability algorithm that replicates digital objects, and detects and corrects corruption in these objects.

- We introduce the concept of a replication set for the automatic replication of digital objects.
- We define the concept of annotated links that restrict traversals over a graph, for the purpose of conveniently specifying replication sets.
- 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 costs and efficiencies of some of the techniques we describe.

The rest of 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 additional details on the specific SAV architecture, refer to [8]).

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.

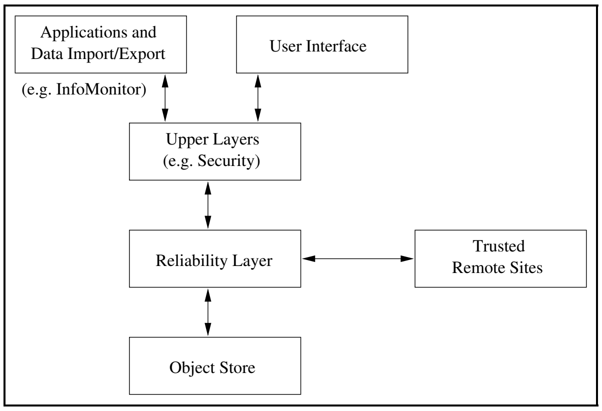


Figure 1: Architecture of the Archival Repository.

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 [9].

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 such 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 archival repository.

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

We suggest that a stored digital object be given an object handle, and consist of a list of fields (which are name/value pairs). 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 in this paper 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. Other data structures that may be useful are described in [8].

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 out of circulation" an object 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.

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.

3 OBJECT STORE

The write-once policy forces us to represent related objects in a way that is unique to archival repositories. 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.

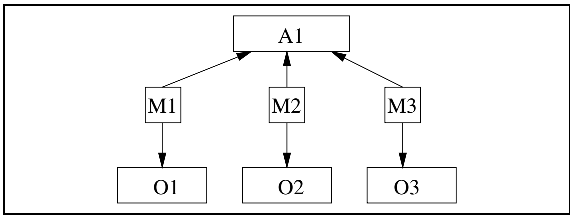


Figure 2: Structure of set $\{O_1, O_2, O_3\}$

The problem is that write-once structures are difficult to traverse. For instance, in order to find all of the members of A_1, it is necessary to scan all repository objects, looking for set member objects (e.g. M_1) that point to A_1. These set member objects would then point to the A_1 members. Clearly this traversal is very expensive, so we need auxiliary indexes to help us locate objects of interest. The most useful indexes for an archival repository are described in Subsection 3.1. Indexes need to be modified, so they cannot be stored as digital objects, and do not enjoy the high reliability of digital objects. Subsection 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 hash table, 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 hash table. 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. (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.

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 indeed its `HANDLE` is 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.

In our implemented SAV system, indexes are kept in main memory, so they need to be 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 managing and rebuilding indexes, we conducted experiments on our SAV prototype, running on an IBM Intellistation (450 MHz Pentium II, 256 MB RAM, 512 MB swap). The SAV itself is implemented in Java 2, and uses VisiBroker 3.4 CORBA to communicate between repository sites. Digital objects representing real documents from the Stanford Database Group's web site were stored in the archive. Six object sets of different sizes were tested in order to assess scalability. The smallest set contained over 300 objects and almost 5 MB of total data, while the largest contained over 30,000 objects and 600 MB of total data. In each set, the average object size was 18 KB. The results are shown in Figure 3. The solid line in the figure represents the total time required to import (in bulk), write and fully index all of the objects in the archive. The dotted line in Figure 3 indicates the time to rebuild all of the indexes for existing objects. Both tasks scale linearly with increasing number of digital objects. The object store requires an incremental writing and indexing time of 150 milliseconds per object or 8.5 seconds per megabyte. The index building operation requires an average of 19 milliseconds per object or 1.1 seconds per megabyte. It takes about 10.5 minutes to rebuild the indexes for the full 600 MB repository.

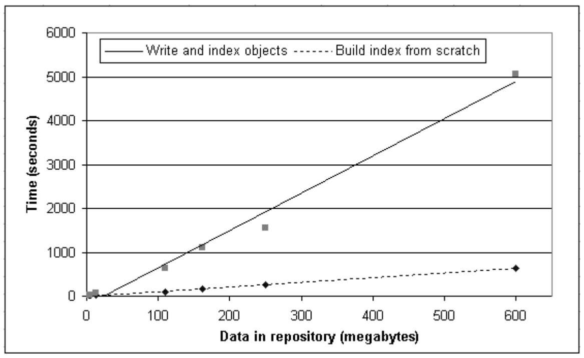


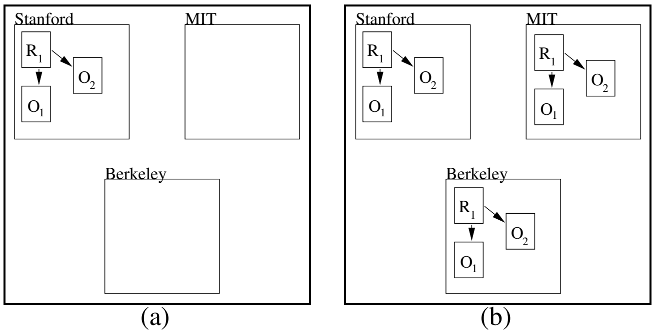
Figure 3: Performance of the object store.

Of course, it is very good that costs scale linearly, but they 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.

4 RELIABILITY LAYER

As described in Section 1, the replication layer backups 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.[1] Similarly, all three objects are copied to Berkeley (Figure 4b).



(a) (b)

Figure 4: A replication network

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, 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, as illustrated in Figure 5. 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, as shown in the figure, or another structure, such as a tree. 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 (two in the example) 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.

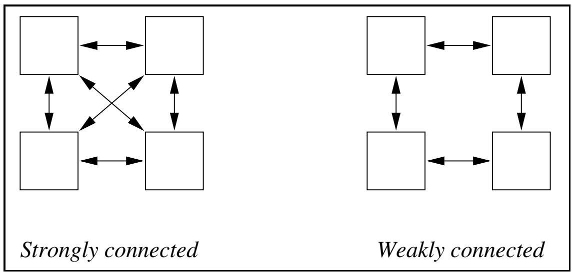


Figure 5: Different kinds of replication networks.

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

---

[1] 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.

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 6, 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.

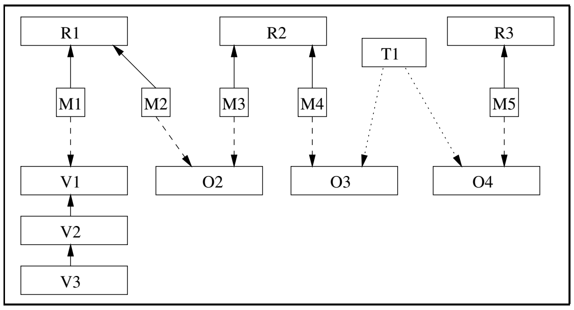


Figure 6: Example replication sets.

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 6. 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 6. 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 6, 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 6, 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 introduce 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 6 can serve as a template for determining how links should be marked. If it is desirable 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.

4.3 Detecting object corruption

Each site periodically constructs a *snapshot* of the replication set of each known agreement.[2] 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, our SAV writes objects to disk using Java 2's serialization operations, and when an object cannot be unserialized, 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] 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.

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 *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 is in line with the annotated 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.

4.4 Comparing replication set snapshots

The computed snapshot is compared to a remote replication set as follows:

1. The handle of every object discovered in the local traversal is stored in a hash table S_L by the local site.
2. The handle of every object discovered in the remote traversal is stored as an item in a linear list S_R by the remote site, and sent to the local site.
3. The local site creates a new, empty linear list L.
4. The local site traverses S_R, performing a lookup in the S_L hash table for each object.
   - If the object is found, it is removed from S_L.
   - If the object is not found, it is stored in L.
5. Every object remaining in S_L is "missing" at the remote site. Every object listed in L is "missing" at the local site.
   - If an object is missing at the remote site, the local site sends that object to the remote site. The remote site stores the object, overwriting any previously stored object with the same handle. (The previously stored object must be corrupted.)
   - If an object is missing at the local site, the local site requests the object from the remote site. The local site stores the object, overwriting any previously stored object with the same handle.

The snapshot construction and comparison process scales linearly. That is, if the number of objects currently in the local replication set is N, and the number of remote objects is M, then the total time is $O(N + M)$. The algorithm does require that an entire snapshot be sent from the remote site to the local site. This could be expensive over a low bandwidth line, even though the snapshot only contains handles, not the objects themselves. Some possible optimizations are discussed in the following subsection.

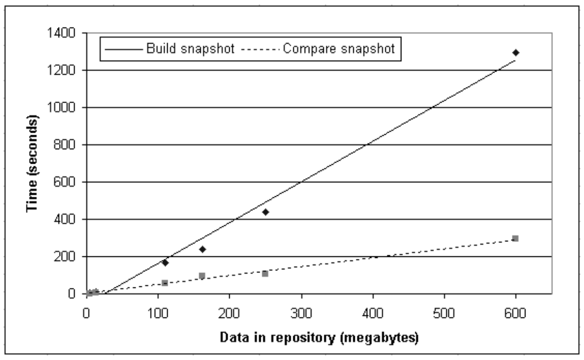


Figure 7: Performance of the reliability layer.

A useful variation to this algorithm is to have the remote site include the `CHECK` values for all objects in its snapshot. This would allow the local site to detect a scenario where an object was corrupted at a backup site before the object was created (and hence the backup site `CHECK` does not match the original `CHECK`).[3] In our SAV system, handles are identical to the `CHECK` values, so this extra check is implicitly performed for free.

4.5 Performance measurements

In order to evaluate the performance of the reliability layer, we conducted experiments on our SAV prototype. Two instances of SAV were started, one running at the IBM Intellistation described in Section 3.3, and another running on a Gateway E-4200 (450 MHz Pentium III, 256 MB RAM, 512 MB swap). The machines were connected by a 10 Mbit Ethernet LAN. The same data sets described in Section 3.3 were replicated between the two sites, and the resulting snapshot times are indicated in Figure 7. 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 can run concurrently. The snapshot construction time scales linearly with repository size, and represents an incremental duration of 39 milliseconds per object (2.2 seconds per megabyte). Moreover, the snapshot comparison time (dotted line in Figure 7) also scales linearly with increasing repository size, representing an incremental duration of 8.8 milliseconds per object (480 milliseconds/megabyte).

The amount of time to send a snapshot from one site to another was negligible in our experiments, due to the fast network. 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 compare both signatures. If the signatures match, then the snapshots are the same. If the signatures do not match, then

---

[3] This assumes both sites compute error codes in the same way.

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 [7].

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. Alternately, 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:

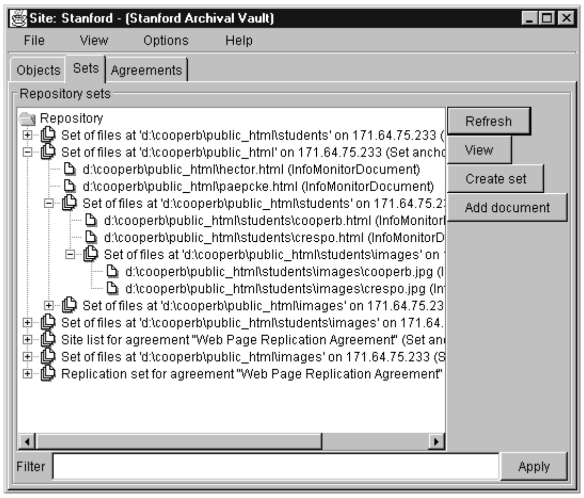
1. The user must be able to locate specific digital objects in the repository, even if the repository contains large numbers of objects.
2. The user must be able to easily perform structuring operations on objects, such as grouping related objects into sets, and to view the topology of object structures.
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 [4].

Figure 8 is a screen shot of our set interface (an example of a structure view). 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.



```
Repository
⊞ Set of files at 'd:\cooperb\public_html\students' on 171.64.75.233 (
⊟ Set of files at 'd:\cooperb\public_html' on 171.64.75.233 (Set ancho
    d:\cooperb\public_html\hector.html (InfoMonitorDocument)
    d:\cooperb\public_html\paepcke.html (InfoMonitorDocument)
  ⊟ Set of files at 'd:\cooperb\public_html\students' on 171.64.75.2
      d:\cooperb\public_html\students\cooperb.html (InfoMonitor
      d:\cooperb\public_html\students\crespo.html (InfoMonitorD
    ⊟ Set of files at 'd:\cooperb\public_html\students\images' on
        d:\cooperb\public_html\students\images\cooperb.jpg (I
        d:\cooperb\public_html\students\images\crespo.jpg (In
  ⊞ Set of files at 'd:\cooperb\public_html\images' on 171.64.75.23
⊞ Set of files at 'd:\cooperb\public_html\students\images' on 171.64.
⊞ Site list for agreement "Web Page Replication Agreement" (Set an
⊞ Set of files at 'd:\cooperb\public_html\images' on 171.64.75.233 (S
⊞ Replication set for agreement "Web Page Replication Agreement"
```

Figure 8: The *Sets* view

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 descriptive string.[4] The filter window (bottom of Figure 8) 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 8 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 9. 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

---

[4] 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.

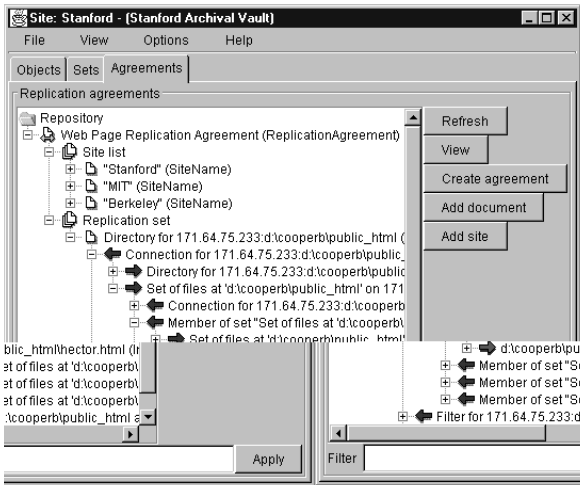


Figure 9: The *Agreements* view

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. So, the librarian running an archival repository 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 InfoMonitor serves as a "bridge" between a repository such as SAV and an existing environment where digital objects reside. We will use a web site as our running example for the existing environment, but the InfoMonitor can be used in other scenarios too. Users continue to create, edit and access web pages using standard tools (e.g., Netscape Composer, Explorer browser). 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 updates to the repository.

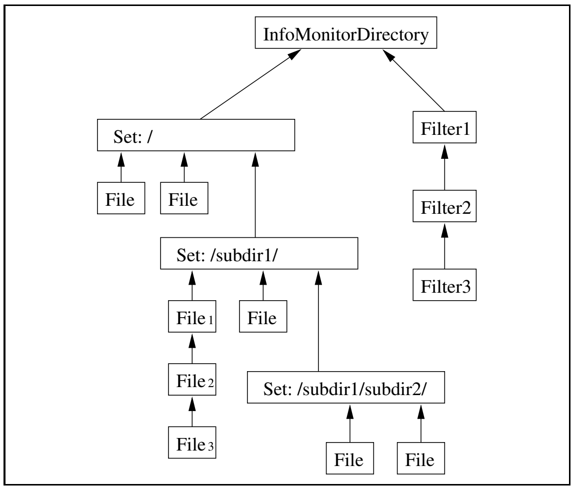


Figure 10: The data structure created in the Archival Repository by the InfoMonitor.

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

Figure 10 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 10 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 55,000 files (1.7 GB) of our group's web site. Additional details and performance numbers are available in [5].

7 RELATED WORK

The digital library community has begun to focus on the problem of designing and implementing long term archives. The Task Force on Archiving of Digital Information examined many of the aspects of the archiving problem in [10]. Several projects have focused on building archives, including the Computing Research Repository [12] and the Archival Intermemory Project [11, 2]. Both of these projects have investigated implementation issues, although they have focused on different archive architectures than the SAV design we discuss here.

The archiving problem is related to the problem of increasing the reliability of file systems. A few investigators have looked at ways to perform file backup [3, 13]. 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) [16], so that disk failures can be overcome. Others have suggested using logs to improve many aspects of the file system, including the reliability [18]. A third solution is to use hierarchical replication systems to reliably store digital objects, and several commercial products have incorporated this idea, including [15] and [6, 14]. 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 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 [1, 17]. 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.

8 CONCLUSIONS

In this paper we have discussed issues that arise when implementing a reliable archive storage system. Although we have discussed these issues from the perspective of our SAV design, these issues 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. 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.
2. 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.
3. 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.
4. Brian Cooper, Arturo Crespo, and Hector Garcia-Molina. Implementing a reliable digital object archive. http://www-db.stanford.edu/pub/papers/-arpaperext.ps, 1999. Extended version of paper.
5. Brian Cooper and Hector Garcia-Molina. Designing and implementing layered archival storage systems. http://www-db.stanford.edu/pub/papers/-fmpaper.ps, 1999. Submitted to ACM SIGMOD 2000.
6. IBM Corporation. Adstar distributed storage manager (ADSM) - distributed data recovery white paper. http://www.storage.ibm.com/storage/software/-adsm/adwhddr.htm, 1999.
7. Arturo Crespo and Hector Garcia-Molina. Awareness services for digital libraries. In *Lecture Notes in Computer Science*, volume 1324, 1997.
8. Arturo Crespo and Hector Garcia-Molina. Archival storage for digital libraries. In *Proceedings of the Third ACM DL Conference*, 1998.
9. Arturo Crespo and Hector Garcia-Molina. Modeling archival repositories for digital libraries. http://www-db.stanford.edu/pub/papers/archsim.ps, 1999. Submitted for publication to ACM DL 2000.

10. 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/.

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

12. 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.

13. 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.

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

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

16. 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.

17. 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.

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