Archival Storage for Digital Libraries

Arturo Crespo
Department of Computer Science
Stanford University
E-mail: crespo@cs.stanford.edu

Hector Garcia-Molina
Department of Computer Science
Stanford University
E-mail: hector@cs.stanford.edu

ABSTRACT
We propose an architecture for Digital Library Repositories that assures long-term archival storage of digital objects. The architecture is formed by a federation of independent but collaborating sites, each managing a collection of digital objects. The architecture is based on the following key components: use of signatures as object handles, no deletions of digital objects, functional layering of services, the presence of an awareness service in all layers, and use of disposable auxiliary structures. Long-term persistence of digital objects is achieved by creating replicas at several sites.

KEYWORDS: Digital library repository, archival storage, long-term preservation of data.

1 INTRODUCTION

A digital library *repository* (DLR) stores the digital objects that constitute the library. The two key requirements that distinguish DLRs from other information stores are archival storage and intellectual property management. The archival nature of a DLR means that the digital objects (e.g., documents, technical reports, movies) must be preserved indefinitely, as technologies and organizations evolve[11, 13]. Intellectual property management is required because digital objects will be served beyond the organization that runs the repository or that owns the information. In this paper we focus on the archival requirement.

There are two interrelated factors in the archiving of digital objects: data and meaning preservation. To illustrate, consider the Mayan inscriptions on their temples. For us to "read" them, first the carvings and paintings had to be preserved (data preservation) over the centuries. Second, the meaning of their hieroglyphs had to be decoded, say into English. Thus, to preserve the meaning there needs to be some translation machinery, which can be based on a lot of guesswork (as in the case of Mayan writings), or aids left behind (which are of course extremely hard to provide in advance). The translation could be done gradually and continuously, to avoid spanning large differences in representations (e.g., translating a document in MS Word 4 to Word 5 to Word 6).

In this paper we focus on data preservation only. This is admittedly the much simpler of the two problems, but clearly, without data preservation as a first step, meaning cannot be preserved. Thus, we view a *digital object* as a bag of bits (with some simple header information, to be discussed). We will not concern ourselves here on whether this object is a postscript file (or any other format), the document that explains how postscript is interpreted (an aid for preserving the meaning of the postscript file), or an object giving the metadata for the postscript file (e.g., author, title). However, we do wish to preserve *relationships* among objects. That is, we will develop an identification scheme so that one object can "point" or "reference" another one. This way, for instance, the metadata object we just discussed can identify the postscript file it is describing.

Given our problem definition, the reader may wonder if this is a solved problem. After all, a database system can very reliably store objects and their relationships. This may be true, as long as the same or compatible software is used to manage the objects, but is not true otherwise. For instance, suppose that the Stanford and MIT libraries wish to store backup copies of each other's technical reports, but they each use different database systems. It is not possible (at least with current systems) to tell the Stanford system that an object is managed jointly with MIT. Similarly, say that Stanford's database vendor goes out of business in 500 years, or Stanford decides to use another vendor. Then migrating the objects elsewhere can be problematic, since database systems typically represent reliable objects in ways that are intimately tied to their architecture and software.

The goal of this paper is to present an architecture for the archiving of digital objects. The objective is *not* to replace database systems, but rather to allow existing and future systems to work together in preserving an interrelated collection of digital objects (and their versions) in the simplest and the most reliable possible way. Also, keep in mind that what we are describing is the lowest layer(s) of a DLR; higher layers (not discussed here) would deal with intellectual property, metadata, security, and so on.

In Section 2 we present the key components of our architecture that make long term archiving feasible. Then in Sections 3 through 6 we describe the functional components of the architecture. In Section 7, we present a complete example that shows how those components work together. Finally, in Section 8 we discuss related work. Because of space limitations, we cannot provide full details of the architecture, nor explain how all failures and situations are handled. Instead, we focus on explaining the main features and on representative examples.

2 KEY COMPONENTS

Under our architecture, a Digital Library Repository (DLR) is formed by a collection of independent but collaborating *sites*. Each site manages a collection of digital objects and provides services (to be defined) to other sites. Each site uses one or more computers, and can run different software, as long as it follows certain simple conventions that we describe in this paper. Our architecture is based on following key components.

2.1 Signatures as Object Handles

Each object in a DLR has a *handle* used to identify and retrieve it. Handles are *internal* to the DLR and are not used by end users to identify documents. (Example: If a user is searching for report STAN-1998-347-B, a naming facility not discussed here will translate into the appropriate handle, or handles if the report has multiple components.)

Given an object, we define its handle to be a (large) signature computed exclusively from its contents, using a checksum or a Cyclic Redundancy Check (CRC). If the contents are smaller than the size of the signature, the object (at creation time) is "padded" with a random string to make its size larger than the size of a signature. This scheme has the following properties, which are important in an archival environment:

- Each site can generate objects and their handles without consulting other sites. This makes it possible for sites to operate independently. Furthermore, sites only need to agree on the signature function, not on software versions, character sets, timestamp services, and so on.
- The handle for an object can be reconstructed from the object itself. As we will see, this is an extremely useful property, since we do not need to reliably save any handle-to-object mappings.
- If copies of an object are made at different sites, all copies will have identical handles. This may seem disconcerting at first, but if the contents are identical, it makes management simpler to call "a spade a spade."
- Objects with different contents will, with *extremely* high probability, have different handles.

The last item requires some discussion, since it may be possible that two different objects share a handle, which would be disastrous. However, by making the signature large (e.g., 128 bits or more), the likelihood of this disaster happening is so extremely low that it is not rational to worry about it. To illustrate, in Appendix 1 we derive a bound for the probability p that there is no disaster in a DLR with n objects and signatures of size b bits. The bound is extremely conservative, but yet we see that, say, a 256 bit signature can make even a DLR with 10 billion objects incredibly safe.

Collection Size (n)	Probability of no collisions (p)	Signature Size (b)
10^7	$1-10^{-9}$	76 bits (10 bytes)
10^8	$1-10^{-9}$	83 bits (11 bytes)
10^9	$1-10^{-9}$	89 bits (12 bytes)
10^7	$1-10^{-24}$	128 bits (16 bytes)
10^7	$1-10^{-63}$	256 bits (32 bytes)
10^{10}	$1-10^{-18}$	128 bits (16 bytes)
10^{10}	$1-10^{-57}$	256 bits (32 bytes)

Figure 1: Number of bits required for typical n, p

If some applications (or paranoid users) need an absolute certainty that each signature is unique, then we offer the following enhanced identification scheme. Handles are extended to have two fields: a unique publisher field and the signature of the object. The publisher field is the unique code of the site that first publishes the object; this publisher code is assigned to the site by some authority. The publisher field of an object does not change when the object migrates to other repositories. The second field is the same as the signature described earlier. When a site creates a new object, it first stores its publisher field in the object header. Then it computes the signature of this extended object and checks if any other *local* object has the same signature. In the extremely rare case there is a conflict, we add a *discriminator*, a random string of bytes, at the end of the new object. The discriminator is included in the computation of the signature (and therefore will make the object map to a different signature), but it is filtered out when the object is returned to a user. From then on, the handle of an object is computed (at any site) by reading its publisher value and adding to it the object signature.

2.2 No Deletions

Because of our handle scheme, objects cannot be updated in place. That is, if the contents of an object are modified, it automatically becomes a new object, with a different handle. This is actually an important advantage, since it eliminates many sources of confusion. For instance, one cannot correct a typo in a report and make it pass as the same object. (We do provide a higher level mechanisms for tracking versions of an object; see Section 5.) Similarly, if a stored object is corrupted due to a disk error, the corrupted object will not be confused with the original.

Another fundamental rule in our architecture is that objects are never (voluntarily) deleted. Allowing deletions is dangerous when sites are managed independently; in particular, it makes it hard to distinguish between a deleted object and one that was corrupted ("morphed" into another) and needs to be restored. Ruling out deletions is natural in a digital library, where it is important to keep a historical record. Thus, books are not "burned" but "removed from circulation." We can provide an analogous high level mechanism for indicating that certain objects should not be provided to the public.

Having immutable objects presents some management challenges. For example, say we create a new version Y of some object (say a video clip) X. We cannot mark directly X to indicate there is a new version Y that should be accessed, because this would be an in-place update to X. In Section 5 we show how we can "indirectly" record such changes. Of course, having no deletions increases storage requirements. We do not believe this is an important issue because (1) storage costs are so low, and (2) we are only archiving in this fashion library objects, not all possible data.

2.3 Layered Architecture

Since each DLR site may be implemented differently, it is important to have well defined and as simple as possible site interfaces. Furthermore, it is also important to have clean interfaces for services *within* a site, so that different software systems could be used to implement individual components. We achieve this in our architecture by defining *service layers* at each site. The layers include:

1. *Object Store Layer:* The Object Store layer uses a *Data Store* (e.g., file system, database management system) to persistently save objects. This layer may use its own scheme to identify objects (e.g., file names, tuple-ids). We refer to these local identifiers as *disk-ids*.

2. *Identity Layer:* This layer has two main functions: (i) it provides access to objects via their handles (signatures); and (ii) it provides basic facilities for reporting changes to its objects to other interested parties.

3. *Complex objects layer:* Manages collections of related objects. Its services could be used to maintain the different versions (or representations) of a document.

4. *Reliability layer:* Coordinates replication of objects to multiple stores, for *long term* archiving. The assumption is that the Object Store layer makes a reasonable effort at reliable storage, but it cannot be counted on to keep objects forever

5. *Upper layers:* Provide mechanisms for protecting intellectual property, enforcing security, and charging customers under various revenue models. It can also provide associative search for objects, based on metadata or contents of objects, as well as user access.

In Figure 2 we illustrate the layers of a DLR. Each "column" in the figure represents a site, and each "row" a software layer. We call the implementation of a layer at a site a *cell*, and the complete repository a *cellular* DLR. Cells can collaborate with others to achieve their goals. For example, the reliability cell at Site 1 communicates with the reliability cell at Site 2 Cells below the reliability layer only deal with their local site. In this paper we only study the grayed-out cells.

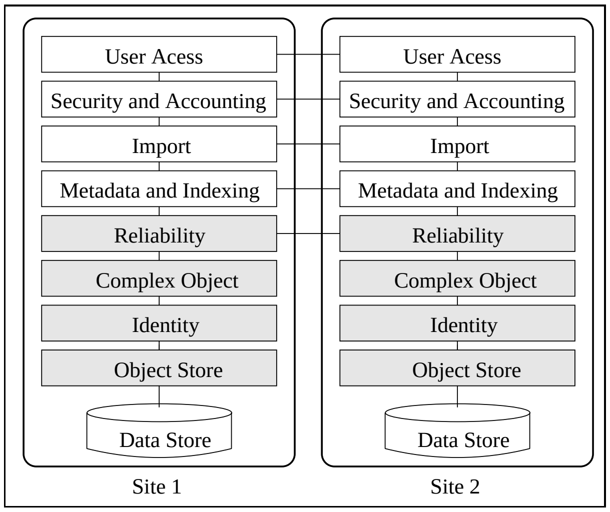

Figure 2: Layers of a Cellular Repository.

2.4 Awareness Everywhere

Awareness services (standing orders, subscriptions, alerts) are important in digital libraries. They are also important for our reliability and indexing layers: if one site is backing up another, it must be aware of new objects or corrupted objects to take appropriate action. Similarly, to maintain an index up-to-date, changes need to be propagated. In many systems, awareness services are added as an afterthought, once the base storage system is developed, and this makes it hard to detect *all* changes. In our architecture, awareness services are an integral part of every layer. This makes it possible to build very reliable awareness services, that can be used for replication and indexing.

2.5 Disposable Auxiliary Structures

Layers typically maintain auxiliary structures for improving performance. In our architecture these structures are designed to be *disposable*, so they can be reconstructed from the underlying digital objects. To illustrate, consider the Identity layer. For efficient lookup, it needs an index structure that maps a handle (signature) into the local disk-id (e.g., file name). One option would be to store this index as a digital object, making it part of the DLR. However, this opens the door for inconsistencies. For instance, the index may say that the object with handle H can be found at disk-id D, but the signature of the object at disk-id D is not H. Instead, we say that no auxiliary structures are part of the DLR. (The structures may be on secondary storage that in not part of the DLR.) If the structures become corrupted or inconsistent with the DLR, they should be deleted and reconstructed from scratch.

In addition to avoiding potential inconsistencies, this approach also makes it easy to migrate objects to a new store, when the old one becomes obsolete. Auxiliary structures, which are typically intricate, do not have to be migrated to the new system. The new system can simply obtains the digital objects, and builds its own structures, using whatever implementation it desires.

3 OBJECT STORE LAYER

The Object Storage Layer is the lowest DLR layer. This layer treats objects as sequence of bytes and uses a local disk-ids to identify objects. The disk-ids are meaningful only to a specific Data Store and their format varies from data store to data store. For example, if the Data Store is a standard file system and each object is saved in a different file, the disk-id could be the file name. On the other hand, if *all* objects are saved in a single sequential file, then the disk-id could be the name of that file, the offset into that file, and the length of the object.

3.1 Object Store Interface

The interface of the Object Storage Layer has the following functions:

- `OS_Get(disk_id)`:
Read an object given its disk-id.
- `OS_Put(bag_of_bits):disk_id`:
Insert a new object in the repository and return the disk-id associated with it.
- `OS_Awareness():list_of_disk_ids`:
List all disk-ids.

The last function, `OS_Awareness()`, lets a client perform a "scan" of the entire collection. This is the most primitive type of awareness service one can envision. Its simplicity makes it easier to implement an Object Store that is very robust. This awareness service is used by higher layers when they have lost their state, or when they wish to verify their state.

For building a reliable system, one must not only define the *desired events* (what we have done so far in this section), but also the *undesired expected events* [8]. The later are those events that may occur because of failures, but that recovery mechanisms (at higher layers) will handle. For this layer, the undesired expected events include: (i) `OS_Get()` returning a corrupted object; (ii) `OS_Put()` failing to insert an object (and returning an error); (iii) `OS_Awareness()` not returning the disk-ids of all objects ever inserted with `OS_Put()`).

3.2 Object Store Implementation

Having an extremely simple interface (e.g., no deletes, primitive awareness) reduces the number of undesired events that one needs to consider, and makes it possible to build a rock-solid store, with few "moving parts" and few things that can break. In addition, this simple interface allows us to us to use almost any secondary storage system as a Data Store, including legacy systems.

To illustrate a possible way to build a solid store that supports this interface, consider the following design. Objects can be placed sequentially on a disk (or tape), with a unique pattern separating them. The disk-id would be the disk address of the first byte. To list all handles, we just scan the disk sequentially looking for the special start-of-object pattern. Since there are no deletes or updates, any object found during the scan is an object to report. Since there are no auxiliary structures (e.g., no i-node tables, no free space tables), there are no structures that can be corrupted. To migrate this collection of objects to a different site, we simply must move this single stream of objects, and nothing else. We stress that this is not the only way to build a cell for this layer, but it is the way we expect it to be built in a good, reliable repository.

4 IDENTITY LAYER

The Object Identity Layer provides access to objects through their globally unique handles, provides an awareness service based on handles, and attempts to correct some of the failures of its underlying Object Store cell.

In our architecture, digital objects have two components: a header and a body. For example, from the point of view of the identity layer, the body of a digital object contains the bits given to an Identity cell for storage. In the header, the cell can store system data (e.g., size of object). The resulting object (header+body) can then be sent to the object store. Unknown to the identity layer, the body may contain headers added by higher layers (e.g., the `type` field discussed in Section 5). This analogous to how packets move between network layers, with lower layers adding their own headers. However, unlike network layers, our lower layers do *not* remove headers when returning an object to upper layers. The complete headers, as recorded in the Data Store, must be preserved, so that any layer can compute the signature and verify it is the correct object. Of course, each layer only interprets its own header, not those of lower or higher layers.

4.1 Identity Interface

The Object Identity Layer implements the following functions, analogous to the Object Store functions:

- `IL_Put(bag_of_bits):handle`:
Creates an object and returns the global handle associated with it.
- `IL_Get(handle):bag_of_bits`:
Gets an object given its handle.
- `IL_Awareness():list_of_handles`:
Returns all handles of objects in the repository.
- `IL_Latest(client):list_of_handles`:
Lists all handles created in the repository since the last time the `client` invoked this function.

The `IL_Put` function is used to create an object. The function receives the data and calls the Object Store layer `OS_Put()` function to save the data on secondary storage. The handle for the new object is computed and returned to the client. The `IL_Get()` function returns the object given its handle. We discuss below how this function can be implemented.

The `IL_Awareness()` function lists the handles of all objects in the local store. The `IL_Latest(client)` function is a specialized awareness service. We do not explain here in detail how it operates, but intuitively, it reports objects created since the last time the `client` invoked this function. It is provided to improve efficiency, since with it clients do not have to be informed of objects they have seen before. Since `IL_Latest()` must rely on auxiliary structures (somehow recording what new objects have not yet seen by clients) it is not as reliable as the `IL_Awareness()` function that simply scans the Object Store for all objects. Reference [3] discusses options for implementing such an awareness service.

Undesired expected behavior of this layer includes (i) losing some object; (ii) `IL_Put()` returning an error; (iii) the awareness functions not returning all of the handles. The Identity layer should attempt to make the probability of these and other undesired events as low as possible. One way to do this is to check for undesired events of the Object Store layer. Again, notice that our architecture significantly reduces the number of undesired events. In particular, the "wrong" object can never be returned by a `IL_Get` call because it can be trivially checked that the object matches the requested handle. Similarly, we never return a "deleted" object since there are no deleted objects!

4.2 Identity Implementation

There are two ways to implement the `IL_Get(handle)` function. The first is to obtain all disk-ids from the Object Layer, and then retrieve each object in turn and compute its signature, until we find an object whose signature matches the requested handle. The second way is by having the Identity layer keep an index mapping handles to disk-ids. The index can be initialized with a complete scan of the Object Store, and then can be incrementally maintained as new objects are created. The `IL_Get(handle)` function can then simply lookup the disk-id for the given `handle`, and fetch the object from the store.

Notice that indeed this index is disposable, as discussed in Section 2.5. As a matter of fact, in a good implementation, the index will be periodically discarded and rebuilt from scratch, to ensure that its structures have not been corrupted, i.e., to reduce the likelihood of undesired events at this layer.

Similarly, the `IL_Latest()` function uses auxiliary structures to track the objects not yet seen by a client. This structure should also be disposable. It should periodically be deleted, in order to force clients to use the more general `IL_Awareness`. This causes the client to check if it indeed has all the objects known to the Identity layer, and re-initialize the auxiliary structure used for future `IL_Latest` calls.

As discussed earlier, the Identity layer should try to handle as many undesired events of the lower cell. Specifically, suppose that the Identity layer is servicing a `IL_Get(handle)` call, and that through its structures has determined that the object is at `disk-id`. Since the call `OS_Get(disk_id)` may return a corrupted object, the Identity cell must check that the fetched object indeed has handle `handle`. If there is a discrepancy, the Identity Layer reports that the object is not found (and maybe attempt to reconstruct the mapping between handles and disk-ids). However, it cannot restore the object; this service will be provided by the Reliability Layer, discussed later on. (Actually, we cannot be sure the problem was caused by the Object Store; it could be the case that the auxiliary structure that told us that `disk-id` was the place to look for the object was incorrect.)

5 COMPLEX OBJECT LAYER

In a DLR, multiple digital objects may be interrelated. For example, a technical report may have several renditions (e.g., plain ASCII, postscript, Word97), where each of these is a simple object. Similarly, a report may consist of a sequence of versions, representing the state of the report over time. The Complex Object layer implements three useful constructs, tuples, versions, and sets (among others), that can be used for implementing higher level notions such as "technical report," and "access rights for a movie." In this paper we do not address the details of the high level concepts, which would be implemented by higher layers. References [5] and [12], among others, propose specific organizations for "documents" and other high level constructs.

Traditional methods for building complex structures do not work in our DLR environment because objects cannot be deleted or modified. For instance, we cannot implement a set as an object containing pointers to other member objects, since the membership could never be modified. (If the set represents the renditions of a report, it would mean that a new rendition could never be added, for example.) The schemes we propose in this section allow the structures to evolve.

A particular Complex Object cell interacts with a single Identity cell, so all the components of a complex object are assumed to reside in the same Identity cell. (A complex object may be replicated at another site as discussed in Section 6.)

The Complex Object layer adds a `type` field to all objects, as it hands them to the Identity Layer. The type field is used to record how the object is used by this layer. The Complex Object layer offers its clients an interface (not shown here) for accessing objects, analogous to that of the Identify Layer. For instance, the call `CO_Put(bag_of_bits)` is handled by adding the type `base` to the `bag_of_bits`, and calling `IL_Put(new_bag_of_bits)`. The `base` type indicates that this object is not one of the structural objects generated by the Complex Object layer.

5.1 Tuples

The basis for implementing any complex object is the *tuple* structure. A tuple is simply an object (of type `tuple`) containing an ordered list of object handles. The interface for tuples is:

- `CO_CreateTuple(list_of_handles):handle`:
Creates a tuple containing the handles passed as parameters; returns the handle of the new tuple object.
- `CO_GetTuple(handle):list_of_handles`:
Returns the list of handles in the given tuple.

Figure 3 illustrates two tuples. Tuple T_1 (created first) contains the handles of objects O_1 and O_2. We can represent this as $T_1 = \langle O_1, O_2 \rangle$. The second tuple T_2 is $\langle\langle O_1, O_2\rangle, O_3\rangle$. Notice one could also create the tuple $\langle O_1, O_2, O_3\rangle$, but it is different from T_2.

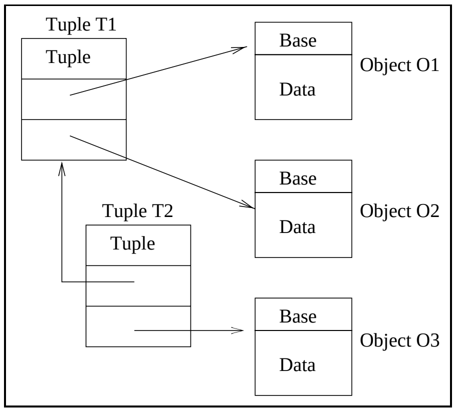

Figure 3: The tuple $<< O_1, O_2 >, O_3 >$

5.2 Versions

Versions are a way of implementing updateable objects in an environment where direct updates are not allowed. When using versions, we update an object, by creating a "new" version of it. Versions support these functions:

• `CO_CreateVersionObject(): handle :`
Creates a new version object and returns its handle.
• `CO_Update( handle, new_version ):`
Creates a new version of the object with the given handle.
• `CO_Read(handle):list_of_handles:`
Returns the list of handles that are the current versions of the object.
• `CO_Versions(handle):list_of_handles:`
Returns the list of all versions of the object.

Figure 4 illustrates how versions can be implemented using tuples. Object V_1 (type `version object`) is the "anchor" for the sequence of versions. Version 1 is recorded by the lower `tuple` object in the figure. Its list of handles contains (a) the handle of the anchor version object; (b) the handle of the object that constitutes this version; and (c) the handle for the previous version. (If this is the initial version, this last handle is null.) The upper `tuple` object records a second version. Notice that because objects cannot be updated, the version "chain" goes from more recent to earlier version. Also, the anchor version object, which identifies this chain, cannot contain a list of all versions. (We would need to update it as new versions are generated.) The structure of Figure 4 was created by the following sequence of calls:

• `CO_CreateVersionObject()`. This returns the anchor `V1`.
• `CO_Update( V1, O1 )`, where `O1` is the handle of the first version.
• `CO_Update( V1, O2 )`, where `O2` is the second version.

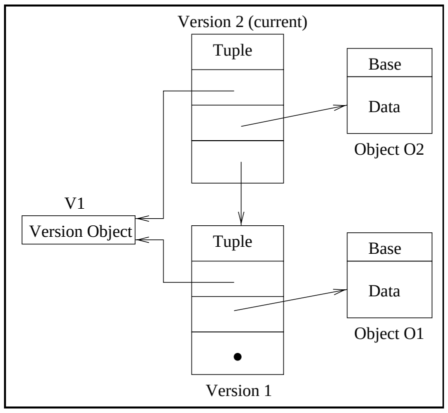

Figure 4: A document with versions v_1 and v_2

To read the latest version of `V1`, we use the call `CO_Read(V1)`, which returns a handle to `O2`. In our example there is only a single latest version, but as we discuss in Section 6, replicating a chain at several sites and independently updating it may lead to multiple latest versions.

The Update, Read, and Versions functions need to determine the latest version, given an anchor object `V`. This must be done indirectly. One way is to scan all `tuple` objects, looking for any that reference anchor `V`. The one(s) that are not referenced by other tuples are the latest versions. Another way is to build a disposable structure that maps anchors to their member objects. Such a structure can be built by scanning all `tuple` objects, and then incrementally maintained as new `CO_Update` calls are made. Our design ensures that this disposable structure is not essential for the long term survival of the DLR.

To record that a version chain has "ended" (e.g., it is inaccessible), we can generate a new version that points to distinguished `null` object. The `CO_Update` call will refuse to create new versions beyond this final one. (We could actually define several "ending" object to indicate different semantics, e.g., the version chain is frozen, it should not be accessed.)

In summary, version objects provide a mechanism "updating" and "deleting" DLR information. Since this mechanism builds upon our immutable objects, it still provides very reliable and long term storage.

5.3 Sets

Other structures can be implemented in a similar fashion. For example, Figure 5 illustrates how a set of objects can be implemented. Each member is a tuple that points to the set anchor (type `set`), and the actual member object. The interface for sets may include the functions:

• `CO_CreateSet():handle:`
Returns the handle of an empty set.
• `CO_InsertMember(set_handle, handle):`
Inserts a member into a set.
• `CO_Member(set_handle, obj_handle):boolean:`

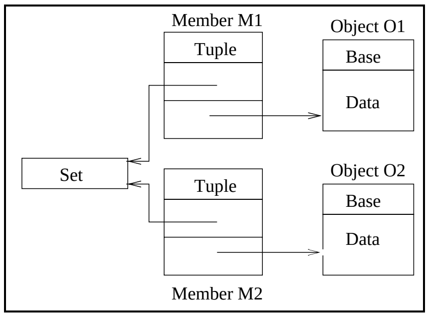

Figure 5: A set with two members

Returns TRUE if the object `obj_handle` is a member of set `set_handle`.

We can have additional functions for sets such as Union, Intersection, and Difference, but these are not discussed here. As with versions, set membership can only be determined by scanning all objects, and looking for those with a given set anchor. Disposable structures can be implemented to make this process efficient. As we discuss in the next section, when sets are replicated at different sites, there may be temporary inconsistencies regarding membership.

6 RELIABILITY LAYER

The Reliability Layer copies objects from one site to another to increase the probability that objects persist for extremely long times. This is achieved by establishing *replication agreements* between multiple sites to mutually maintain replicas of objects of a given *replication group*. For example, if the reliability layer at Site 1 establishes a replication agreement with Site 2 for objects of group G_1 (say a technical report series), then every time an object belonging to G_1 is created at one of the sites, a copy must be propagated to the other site. Note that agreements are multilateral: all members are responsible for backing up objects at the other members.

The Reliability layer adds two header fields to all objects, as it hands them to lower layers for storage. The `group` field records the replication group this object belongs to, i.e., it sets the desired level of replication. The group is selected by the client that creates the object in the first place. The second field, `agrmt`, is used to distinguish objects that represent agreements from those that do not.

Each replication agreement is recorded in a version complex object. The *agrmt* field in this object is set to True, and the *group* field is set to the identifier for this group. The content is a list identifying all the sites participating in the agreement. If the agreement changes, a new version is generated, with the new participants (and same `agrmt` and `group` fields). Note that *all* the objects that make up the version agreement for group G_1 are themselves in group G_1. Hence, they will also be backed up to participating sites. Also note that the replication functions we describe here can be used to migrate a collection from one site X to another site Y (by first adding Y to a replication group, and then dropping X).

6.1 Reliability Interface

The interface of the Reliability Layer includes the following functions:

- `RL_NewAgreement(): gr_hdl`
Creates a new replication agreement, identified by the returned `gr_hdl` handle. This handle is the group identifier, and should be given to all object in the group.
- `RL_Participants(site_list, gr_hdl)`:
Makes `site_list` the current set of participants in `gr_hdl`.
- The interface also includes the functions in the Complex Object interface. For the functions that create objects, an additional parameter `gr_hdl` is added, to indicate the replication group they belong to. Awareness functions are extended so that objects belonging to a given replication group can be requested.

6.2 Implementation

When `RL_NewAgreement()` call is received, the Reliability cell simply calls `CO_CreateVersionObject()`, receiving a handle `G` that will be used as the group identifier. Next, the function `CO_Update(G, O1)` is called to create the initial version of the agreement. Object `O1` has its `agrmt` field set to True, its `group` field set to `G`, and its contents to an empty set of sites. The result of the `RL_NewAgreement()` call is `G`, which can then be used by the client to create objects in this replication group.

A DLR administrator can then issue a `RL_Participants` call to record the participating sites. That call is issued at only one of the participating sites, since the site will immediately propagate the news to the other sites. The call generates a new version of the agreement (in the version chain anchored by `G`), containing the new list of participants.

Once an agreement is in place, the Reliability Layer can enforce it in a variety of ways. Here we illustrate one simple way, assuming Reliability cell A is the one actively ensuring Reliability cell B has copies for group `G`. (Cell B would perform a similar process concurrently.) Periodically, A requests from B its complete list of handles corresponding to object in group `G`. To comply, cell B uses its lower awareness services to get all object handles (in its storage partition), and forwards those in group `G` to A. Cell A performs a similar scan at its own site, and then compares the handles. If a handle is seen locally but not at B, that object must be copied to B. (Cell A asks cell B to create a new identical object. The object may have existed at B before, but it may have been corrupted.) Similarly, if an object is missing locally, it is requested from B and created at the local site.

Note that when asked to replicate objects of a complex type, the reliability layer creates shallow duplicates. For example, suppose that a version object `V1` is created, together with a first version, of say a postscript technical report. Assume that all these objects are defined to be in group `G1`. Next, a second `V1` version is created (e.g., an updated report), but for some reason its group is defined to be `G2`. A site that is only in `G1` will only receive the first version of the report, and not the second one. Thus, to ensure that a complex object is fully replicated, all of its components must be in the same group. Note that auxiliary tuple objects created by the Complex Object Layer do not have a replication group field, since are generated implicitly by the Complex Object layer. However, those objects still need to be replicated, as part of the complex structure they participate in. To achieve their replication, we implicitly assume that the replication group of a tuple object is the union of the replication groups of the base objects it points to.

The stored replication agreement is used by a Reliability cell to "remember" its agreements in case of problems. Let us consider a few sample problems to illustrate. (It is beyond the scope of this paper to do a detailed case-by-case failure analysis.) In our fist scenario, Reliability cell A fails while participating in group `G`, loses its state, but the latest agreement for `G` was not lost at the local site. Cell A restarts by scanning the local site for all objects[1] with their `agrmt` field set, eventually finding the latest version of agreement `G`. From that point on, it resumes its backup work with the other participants. Any `G` objects lost during the failure, will be reconstructed from the other participants.

In our second scenario, say that when cell A recovers, no record of agreement G is found locally. Hence, cell A does not know it is participating in `G`. However, other `G` sites are hopefully active, and they will realize that A has lost objects, and will restore them. Since the agreement for `G` is in the group, it will also be restored.[2] Eventually A realizes there is an agreement it is participating in, and resumes its activity. (Cell A needs to periodically scan its local object to ensure it has accurate information.)

In our third scenario, the latest version of agreement `G` is lost, but some older version survives. When A recovers, its starts its activity with an out of date list of participants. This may cause it to temporarily miss some of the sites that contain replicas, and may cause it to send object copies to sites that are no longer participants. However, the latest version of agreement `G` will eventually make it to A, and A will eventually operate correctly. We emphasize that the only "damage" done in this scenario is the creation of non-needed replicas at sites that had dropped out of the agreement. While un-needed copies may waste some space, they in no way compromise the objects that are already stored.

The reliability layer guarantees an "epidemic" [4] propagation of copies. If we look at a given object `X` in group `G`, with extremely high probability `X` will be at all `G` sites. There may be periods of time when `X` is missing at some sites (e.g., a copy was corrupted), but it would take an unlikely sequence of failures to make it disappear from all `G` sites. Note there is no notion of a distributed commit for `X`. Object `X` is committed when it is created at one site, and its probability of long term existence increases as copies are propagated. The fact that our objects are immutable, simplifies the protocol and increases the chances it works correctly. In particular, there is no danger that the distributed `X` copies become "inconsistent."

[1] This assumes that Cell A knows what its local site is. We can agree in advance on, say, fixed ports for the local layer interfaces.

[2] Object `G`, the anchor for the version chain, is not in group `G` since it was created before the group existed. However, the versions in `G` are in the group and are sufficient to reconstruct the latest version.

When a client creates an object `X`, it may wish to know when it has been replicated at all `G` sites, so it knows it has reached its "extreme safety" mode. For this, we can add a function to the Reliability layer that checks if an object is found at all participating `G` sites.

When complex objects are in the same group, they get replicated and their copies converge. Sites may temporarily have incomplete information, but we do not view this as a strict inconsistency. For example, site A may think that a technical report is available in ASCII and Postscript, while site B may think it is available in ASCII and Word97. If this information is encoded as a set, eventually both sites will know about all three formats.

7 A COMPLETE EXAMPLE

In this section we give an example of how the layers described in the previous sections work together. In this example, we will have two repositories containing technical reports, one at Stanford and another one at MIT. These two sites have a replication agreement for all objects belonging to the technical report group *TRG*.

Let us suppose that an upper cell at Stanford wants to publish a technical report. The publisher anticipates that several version of this document may be generated and decides to use a "Version" complex object. (For the sake of simplicity, we are assuming that each version of a technical report is just one object). First, the publisher asks the Reliability Cell at Stanford to create a new version object V belonging to the replication group *TRG*. Recall that the version object does not contain the data for the technical report (we will save this data as its first version). The Reliability Layer calls the Complex Object Layer function `CO_CreateVersionObject()`. In turn, the Complex Object cell generates the version object and saves it by calling the Identity cell, which calls the Object Store cell. As a result of these calls, the Reliability Layer obtains the handle of the version object V.

After creating the version object, the client is now ready to generate the first version of the technical report. First, the client creates the technical report object, TR_1, by calling the `Put()` function in the Reliability cell at Stanford. The reliability cell sets the group field to *TRG* and asks the lower layers to save the report. After creating TR_1, the client makes TR_1 a version of V by calling the `Update()` function in the

Reliability Layer. The Reliability Layer will pass the request on to the Complex Object Layer which will generate an object, V_1, containing a pointer to V, TR_1, and the previous version (which is a NULL pointer in this case as this is the first version). At the left of Figure 6 we show the state of the Stanford site (at this moment, the MIT repository would be empty).

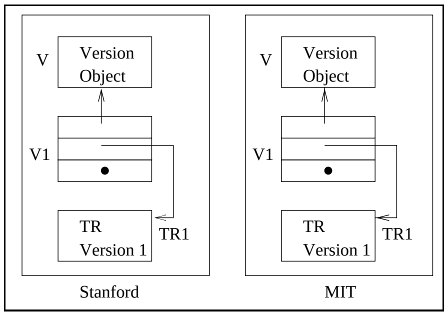

Figure 6: The repositories after replication.

As there is a replication agreement between MIT and Stanford for the objects in the Technical Report group, the MIT (or the Stanford) Reliability Cell will try sometime later to enforce the agreement by querying the other reliability cell and finding out that the newly created objects, TR_1, V, and V_1, are missing in the MIT site. As described in Section 6, the simple way of doing this query is to use the $IL\_Awareness()$ function to obtain all the handles in the other site and then compare those handles with the handles on our own site. A more efficient way of doing this query is to use the $IL\_Latest()$ function to find which handles have been added to the repository since the last time it was visited. There are more efficient awareness algorithms that are outside the scope of this paper. After finding the handles of the missing objects, the replication process will create replicas of those objects in the MIT site. At this moment, the content of the repository is shown in Figure 6. (We are not showing the Reliability Agreement Object that we are assuming was created earlier.)

Note that at this point we could have a synchronization problem if we concurrently add two new versions, one at MIT and the other at Stanford. Figure 7 illustrates this by showing the state after Stanford generated Version TR_2, and MIT independently created Version TR_3. When the replication process copies the new objects to the other sites, we end up with multiple latest versions, as shown in Figure 8. That is, the call `CO_Read(V)` will return both TR_2 and TR_3. We view this as an application "problem." Perhaps it was the intention to have multiple current versions for this report, i.e., the Stanford and MIT versions of a jointly authored paper. If this was not the intention, then the "report creation" layer should ensure that only one author at a time creates new versions of a report. This type of sequencing could be enforced by a synchronization service that is not discussed here.

Let us return to the state of the repository of Figure 6 and let us suppose that the Stanford Repository has a failure that

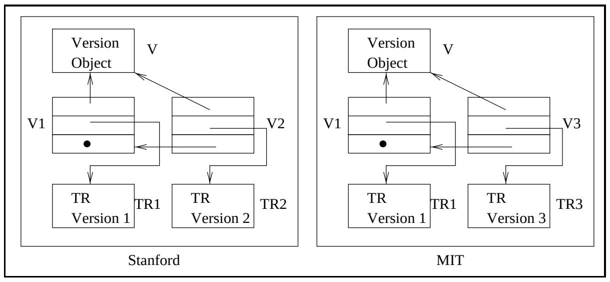

Figure 7: New versions at Stanford and MIT.

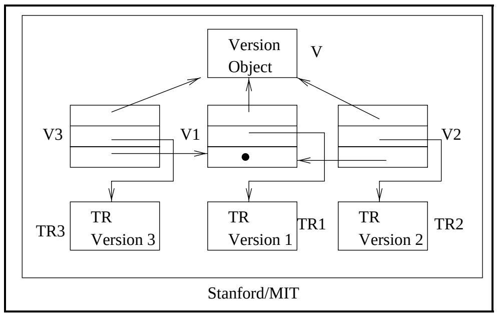

Figure 8: Inconsistent State.

completely destroys all its information. After this failure, the reliability process at Stanford cannot recover its data, since its Reliability Agreement Objects (that indicate where the replicas are) have been lost. However, some time later, the Reliability cell at MIT will visit Stanford and it will find out that some objects, including the Replication Agreement Objects have been lost at Stanford. The Reliability Cell at MIT will restore those objects (and potentially some others), allowing the Reliability cell at Stanford to also start recovering its destroyed digital objects.

8 RELATED WORK

Several architectures have been proposed and implemented for digital libraries [1, 7]. These architectures focus on interoperability and distribution, but are not directly concerned with the problem of long-term reliability.

The task force on preserving digital information [2] has investigated means of ensuring the long-term safekeeping of information in digital archives. As in our architecture, the task force regards migration as an essential tool in preserving digital archives. However, the task force, deliberately, avoids defining the implementation details for a digital archive.

At the secondary device level, the Petal [9, 10] and Frangipani [15] projects have designed highly-available, scalable block-level storage systems that are easy to manage. The availability of the system is achieved by using data striping and redundancy. Although these projects consider the problem of long-term data reliability, their aim is a "file system" replacement. They allow in-place updates and deletions, and use application generated filenames (handles).

In the business world, Computer Output to Laser Disk (*COLD*) systems have been very successful in solving the problem of long-term archiving of data that is not frequently accessed. COLD systems were originally designed to replace microfiche and paper archival applications with online computer systems. A typical COLD system captures the output of a computer program and stores it. Typically, the storage media are CD-ROMs but nowadays other types of storage media (magnetic disks, RAID, magnetic tape, and re-writable laser disks) are also used [6]. COLD system are monolithic with very few computers, all of them running exactly the same software. This is different from the heterogenous environment we consider. Storing data on a write-once COLD device forces the data to be immutable, as in our design. However, COLD systems always assume that some persistent storage is available on a write-many device, which can be used for some structures. We assume all DLR storage is immutable.

Systems based on layering have proven effective especially in the area of networking. Specifically, the Open System Interconnection model (OSI) provides a standard that divides a network in seven layers with clear responsibilities [14].

9 CONCLUSION

In this paper we have studied an architecture for long-term archival storage of digital objects. We have argued that we can build a simple, yet powerful, archival repository by using signatures as object handles, not allowing deletions, having awareness services in all layers, and using only disposable auxiliary structures. We believe this architecture is well suited for a heterogeneous and evolving environment because each site only needs to agree on some very simple interfaces, on a signature computation function, and on some simple object header structure (e.g., for type and group fields). Although sites may use auxiliary structures, they need not agree on their details and use. There are no i-node tables, out-of-synch clocks, inconsistent indexes that can cause us to lose or corrupt information. Since objects are never deleted or modified in-place, many sources of confusion are eliminated, yielding an extremely safe DLR. Migration of information from an obsolete site to a new one is simple, and can be performed by the replication services.

ACKNOWLEDGMENTS

Some of the early ideas in this papers were developed in talks with Jerry Saltzer. We would also want to thank Carl Lagoze for several useful suggestions.

REFERENCES

1. William Y. Arms. Key concepts in the architecture of the digital library. *D-Lib Magazine*, July 1995.
2. The Commission on Preservation and Access, and The Research Libraries Group. *Report of the Task Force on Archiving of Digital Information*, May 1996.
3. Arturo Crespo and Hector Garcia-Molina. Awareness services for digital libraries. *Lecture notes in computer science*, 1324:147–71, 1997.
4. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. *Operating Systems Review*, 22(1), January 1988.
5. Document Management Alliance. *DMA 1.0 Specification Draft*, January 1998.
6. P. Gawen. Computer output to optical disk and its application. In *Proceedings of the Seventh Annual Conference on Optical Information Systems*, pages 102–106, July 1990.
7. Robert Kahn and Robert Wilensky. A framework for distributed digital object services. Technical Report tn95-01, Corporation for National Research Initiatives (CNRI), May 1995.
8. W.H. Kohler. A survey of techniques for synchronization and recovery in decentralized computer systems. *Computing Surveys*, 13(2), June 1981.
9. Edward K. Lee. Highly-available, scalable network storage. *COMPCON*, 1995.
10. Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. *ASPLOS*, 1995.
11. Stephen Manes. Time and technology threaten digital archives. *The New York Times*, April 1997.
12. Jr. Ron Daniel and Carl Lagoze. Extending the warwick framework: From metadata containers to active digital objects. *D-Lib Magazine*, November 1997.
13. Jeff Rothenberg. Ensuring the longevity of digital information. *Scientific American*, 272(1):24–29, January 1995.
14. W. Richard Stevens. *UNIX Network Programming*. Prentice Hall, 1990.
15. Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. *SOSP*, 1997.

APPENDIX 1

The probability of not having a *signature collision*, p, depends on the size of the collection, n, and the number of bits, b, in the signature. When we insert the first object the probability of not having a collision is 1 (as there are no documents to collide with), for the second document the probability of not having a collision is $(2^b - 1)/2^b$ as there are 2^b possible signatures that can be generated and all but one of them will not create a collision. In general, when we have inserted k documents, the probability that the next document will not create a collision is $(2^b - k)/2^b$ if $k <= 2^b$, or 0 otherwise. In conclusion, if we assume that the signature function uniformly distributes documents in the signature space, and that the computation of each document signature is independent, then the probability that we will not have a collision in a collection of n documents is:

$$p = \prod_{k=0}^{n-1} \frac{2^b - k}{2^b} = \frac{2^b!}{(2^b - n)!2^{bn}} \tag{1}$$

Equation 1 is impractical to use when b and n are large numbers as the factorials will produce an overflow. We can derive an approximation by making $p = \prod_{k=0}^{n-1} 1 - \frac{k}{2^b}$, and using the Taylor expansion for the exponential function to obtain $p \approx \prod_{k=0}^{n-1} e^{-\frac{k}{2^b}}$. Solving the product, we obtain:

$$p \approx e^{-\frac{n(n-1)}{2^{b+1}}} \tag{2}$$