M.I.T. Laboratory for Computer Science
Computer Systems Groups

November 5, 1981
Request for Comments No. 210

A RELIABLE OBJECT-ORIENTED DATA REPOSITORY FOR A DISTRIBUTED COMPUTER SYSTEM

Liba Svobodova
INRIA[1]
Rocquencourt, France

The repository described in this paper is a component of a distributed data storage system for a network of many autonomous machines that might run diverse applications. The repository is a server machine that provides very large, very reliable long-term storage for both private and shared data objects. The repository can handle both very small and very large data objects, and it supports atomic update of groups of objects that might be distributed over several repositories. Each object is represented as a history of its states; in the actual implementation, an object is a list of immutable versions.

The core of the repository is stable append-only storage called Version Storage (VS). VS contains the histories of all data objects in the repository as well as all information needed for crash recovery. To maintain the current versions of objects online, a copying scheme was adopted that resembles techniques of real-time garbage collection. VS can be implemented with optical disks.

Keywords and phrases: distributed data storage system, server, atomic update, stable storage, optical disk, memory management, crash recovery

To appear in the Proceedings of the 8th ACM Symposium on Operating Systems Principles (December 14-16, 1981, Asilomar Conference Grounds, Pacific Grove, California)

1. Introduction

The data *repository* described in this paper is a remote server for a distributed system of highly autonomous machines. The repository itself is a component of a distributed data storage system called SWALLOW, described in [REED 80]. SWALLOW supports multiple repositories. Further, it includes another type of component, a software package called a *broker* that must be installed in each client machine. The brokers mediate all accesses to the data in the repositories.

SWALLOW implements atomic actions for its clients on the data stored in the repositories. An atomic action is a control abstraction that makes the program that forms its body appear indivisible with respect to other concurrent computations and with respect to failures.[2] It is up to the client to decide what constitutes an atomic action and to make sure that all requests that are part of it are properly acknowledged before the atomic action is committed, but the broker in the client machine will set up all the mechanisms needed to achieve atomicity.

Most of the client computers are assumed to be personal computers that do not have adequate facilities to store large amounts of data *reliably for long periods of time.* Further, it is assumed that the information stored in the individual client computers is not *directly* accessible from other client computers, or at least that it cannot be guaranteed that such information is always accessible; the requirement of autonomy is one of the main reasons behind this assumption. Thus the main functions of the repository are:

1. to provide large reliable long-term storage;
2. to support sharing of data among client computers.

Long-term reliable storage and sharing of data in a distributed system have been a subject of several research projects [SWIN 79, PAXT 79, LAMP 79, ISRA 78, STUR 80, DION 80].[3] The design presented in this paper differs in three important aspects:

1. the repository stores objects of arbitrary sizes; in particular, even small objects are handled as separate entities;
2. the repository supports a novel model of objects that incorporates synchronization and recovery mechanisms needed to update atomically single or multiple objects;
3. the repository is designed so that the principal long-term storage can be provided by optical disks.

The repository is an object-oriented system not only in the sense that it manages objects of arbitrary sizes but also because the synchronization and recovery mechanisms for an object are part of the object model rather than provided on top of it.

The object model that forms the basis of the design of the repository is described in the next section. The decision to design the repository so that the objects can be stored on optical disks has had important implications, as discussed in the successive sections. Section 3 presents an implementation of the object model that renders the essential part of the object representation immutable. Section 4 concentrates on the problem of management of the Version Storage, the append-only stable storage that contains the actual data as well as all the information needed for crash recovery. Section 5 discusses an implementation of the mechanisms that coordinate updates of multiple objects in one or more repositories. Section 6 describes the crash recovery procedures of the repository. Section 7 summarizes the major aspects of the design. An experimental implementation of this system is under way.

2. Object model

The repository supports, with some minor modifications, the object model developed by Reed [REED 78, REED 79]. Every time a client updates an object, the repository creates a new *version* of the object; the versions are linked together to form a *history* of the object. More precisely, a write operation first creates a tentative version called a *token*, which must be *committed* to make it a permanent version. A token also can be discarded, which returns the object history to the state that existed prior to the execution of the write operation.

In addition to having a value, a version has a time attribute that specifies its range of validity; all operations on objects include an explicit time parameter, which specifies the exact point in the object's history to which the operation refers.[4] The *start time* of a version is the time specified in the write request that created the token. The *end time* of a version is initially the same as its start time, but it is extended by both read and write operations to the time specified in the request; when a new version is created, the end time of the preceding version is frozen.[5]

If a valid version already exists for the time specified in a write request, the repository rejects that request. When an object has a token, the repository delays any write operation on the object until the token is either committed or discarded.[6] A read operation selects a version that has the highest start time that is lower than the time specified in the read request. If that version is not yet committed (it is only a token), the repository delays the read operation unless it is part of the same atomic action that created the token.

2.1. Implementation of atomic actions

Embedded in each token is a reference to a special object called a commit record. The commit record contains the state of the client's atomic action that created the token. A commit record is created with the state set to **unknown**. Eventually, the state of the commit record will be changed to **committed** or **aborted**. This change is irreversible, and it implicitly commits or discards any token that contains a reference to that commit record.

The set of tokens created as part of the same atomic action is called a *possibility*. Since all tokens in a possibility contain a reference to the same commit record, a possibility is either committed or discarded as a whole. This simple mechanism thus makes it possible to update several objects atomically. Moreover, these objects can be stored in different repositories.

To be able to execute clients' computations atomically in the presence of other concurrent computations, it is necessary to provide mutual exclusion mechanisms that give each atomic action a consistent view of the data on which it operates, and a simultaneous exclusive access to all the objects that it updates. The principal mechanism used here is the time parameters that appear in both the requests and the stored versions. Requests that belong to the same atomic action are assigned times from a time range which either completely precedes or completely follows the time ranges used by other atomic actions;[7] this is a simple extension of a transaction timestamp as used, for example, in [THOM 79, BERN 77, TAKA 79].

The repository implements two types of entities: the object histories and the commit records. The set of operations supported by the repository is summarized in Figure 1.[8] When a client starts an atomic action, its broker first assigns a time range to that atomic action and sends a request to some repository to create a commit record. The *commit record id* returned by the repository is not used directly by the client; the broker translates client's requests to create, delete, and access objects to those given in Figure 1, and inserts both the *commit record id* and the *time* parameter. Once the client indicates that the atomic action has completed successfully, the broker sends a commit request to the repository that contains the commit record. Since that atomic action might have been already aborted internally (for example, because of a timeout), the repository may reject the commit request. Of course, the client also can request that the atomic action be aborted.

3. Implementation of objects

From the point of view of implementing atomic updates, object histories provide a natural backward recovery mechanism: if a write operation on an object fails or if the computation which updated the object does not succeed, the previous version can resume its status as the most current version of the object. Clearly, for recovery from this type of failures it would be sufficient to maintain only the last committed version of an object, in addition to a possible token. However, there are many applications, particularly business applications, where it is actually desirable to store the entire object histories, either for auditing purposes or to aid in the recovery from certain types of errors at the application level.

Because of the large quantities of storage required, it is important to utilize storage devices that are: 1) inexpensive and 2) easy to store offline. To provide fast access to any of the many objects dispersed on the storage medium, a random access device is needed. Optical disks look promising in all these aspects, although their presently attained access speed is inferior to that of magnetic disks. But optical disks have a more severe limitation: they are "write-once" devices, that is, the information stored on them is not updateable. Consequently, it is not sufficient to make the object versions logically immutable (object versions appear to be immutable to the clients in the sense that a client cannot change the value of any version), but it is necessary to make immutable also their physical representation (the representation includes some control information).

There is another good reason to make the physical representation immutable. To ensure that stored data are not damaged while being modified, the update operation must be atomic. Since no physical storage device guarantees atomic write (an error might occur during a write, leaving the stored entity in an undefined state), it is necessary to add some redundancy for error detection and recovery. The technique described by Lampson and Sturgis [LAMP 79] is to maintain two copies of stored data, where the two copies must be changed strictly sequentially, that is, the first write must complete successfully (correct data written to correct address) before the second write is initiated. If the storage model does not have to support an update operation that involves overwriting the physical representation, the problem of achieving atomicity is simplified. It is still desirable to have two copies (or equivalent redundant information) in order to be able to cope with spontaneous device failures and decays of stored information, but the required writes can be done concurrently.

The data structures that represent an object history in the repository are shown in Figure 2. The information that controls access to the object history, which changes as the history evolves, is concentrated in the object header. In addition, some information already present in the object versions is duplicated in the object header for better efficiency. The entire object header is, however, only a *hint*: object headers are important for good performance, but they are not essential to guarantee correct operations on the stored objects. Designing the object header entirely as a hint means that it can be kept in fast but volatile storage without having to take special precautions to assure that it survives a crash.

3.1. Version images

The entities (data structures) that represent versions are called *version images*. A version image contains, in addition to the value field, the start time t_S, a reference to the immediately preceding version in the history, the unique indentifier (uid) of the object of which it is a part, and a reference to its own commit record. The last two items make version images fully self-identifying; this property is needed for crash recovery as will be explained.

It is important to distinguish between object versions and the representation of versions, that is, version images. A version is a logical concept; it is the value of the object during a specific time interval in the object's history. A version image represents either a version or a token; to determine which of these two it represents, it is necessary to inspect the object header or the commit record specified in the version image. Several images of a version may coexist in the repository, as a consequence of storage management (see Section 4). Also, a version image of an aborted token will remain in the repository; an aborted token is removed from the history simply by omitting it from the normal chain of references originating from the object header.

Version images are stored as stable immutable entities in the Version Storage described in Section 4; a version reference or a token reference is always the physical address of the representing version image in VS. By using a careful object update protocol and duplicate storage, the probability of losing a version after a write request has been acknowledged to the client can be made negligibly low. Note that tokens, although they represent tentative versions, must be stored in stable storage. In fact, a version image of a token must be written into stable storage before a create-token request is acknowledged.

3.2. Object headers

The object header contains a reference to the current version of the object and the end time of this version.[9] This time is updated every time the current version is read by an atomic action with a read time that exceeds the current end time of the object. The object header also includes a potential token reference which is null if the object does not have a token.[10] Finally, it contains a reference to the commit record for the current token. Besides this information, which is part of the object model, the object headers include additional fields (not shown in Figure 2) that are used for management and recovery of the object headers. Two of these fields will be discussed later: the hash table link and the Recovery Epoch Number.

The object headers play a very important role in controlling accesses to objects and locating proper versions. Nevertheless, the repository does not provide for them the same reliability as for the version images, because of the associated cost. The object header is updated twice for each update of the object (in create-token and commit/discard token), and may have to be updated when the current version is read (to extend the end time). An object header must be updated in place, otherwise it would be necessary to change also the table used to map object uids to the addresses of the object headers. Two disk writes for each update of an object header needed to make the object headers stable would increase significantly the internal overhead and the response time of the repository. Thus, as already explained, the object headers are designed to be only hints.

The object headers are stored on a nonvolatile storage device that provides updateable storage (e.g., magnetic disk). This device will be referred to as Object Header Storage, or OHS. Object headers are brought into main memory as needed, but the changes made to an object header are not propagated into its image in OHS until the object header is purged from the main memory. The current object header, that is, the instance of the object header that reflects correctly the current state of the object thus might get lost if the repository crashes. However, the object headers can be reconstructed entirely from the information contained in the version images. This means that the OHS images are not essential for correct operation, but the availability of even an obsolete object header in OHS reduces the amount of work it takes to put together a correct object header after a crash. The crash recovery will be discussed in more detail in Section 6.

OHS is divided into fixed-size pages where each page contains several object headers. The object headers in OHS are organized into a non-coalesced chain hash table. The hashing function maps the object id into the physical address in OHS. The headers of the objects that map into the same bucket are chained together using their hash table link field. When an object is created or deleted, it might be necessary to change the hash table link in several object headers. These changes are not performed atomically; instead, recovery procedures were designed that detect inconsistencies caused by crashes [AREN 81].

Since they contain mutable data, the object headers must be protected from incompatible simultaneous accesses. However, the repository should be able to handle concurrently several requests that pertain to the same object, since most requests will require one or more disk accesses. In particular, it should be possible to read an old version while the repository is in the process of creating a token for the same object. Thus each object is protected by a monitor that locks the object header only for the minimum time necessary; in most cases, it is only a fraction of the time needed to complete the entire operation on the object. The monitor state is kept only in the primary memory and thus all objects are automatically unlocked after a crash.[11]

4. Version Storage

The core of the repository is the stable storage called Version Storage (VS). Abstractly, VS is an infinite append-only address space. The append-only model of storage supports naturally the append-only model of object histories. In addition, this append-only model is well suited for an implementation on optical disks. In a sense, VS is similar to the transaction log of database management systems [GRAY 79]. However, there is an important difference: VS is used not just for recovery, but it is where the actual data are.

The VS address space is mapped in a straightforward fashion onto physical devices: a VS address is the number of the device and a sequential offset on the device. Histories of different objects are interleaved in VS. Versions of the same object appear in VS in the right sequential order, but versions of different objects are not necessarily ordered in VS by their start time. VS can be duplicated to prevent loss of information due to device failures, but since update does not involve overwriting old data, the two required writes can be concurrent.

Although the repository accepts objects of arbitrary sizes, the physical storage is allocated and accessed in fixed-size blocks. The issues of transfering data received from the brokers into VS are discussed in the following subsection. Since VS might grow arbitrarily large, it is infeasible to keep it online in its entirety. The issues of what information should be kept online and how the online storage is to be managed are discussed in Section 4.2. Since it is assumed that the physical storage (offline) is inexpensive, no attempt is made in this design to save on storage. Finally, Section 4.3

discusses the problem of errors in physical writes.

4.1. Transfer of data from brokers to VS

For easier management of VS, in particular, for faster VS address resolution and object location, it is desirable to allocate VS in fixed-size blocks. These fixed-size blocks, or *pages*, are the units of *atomic write* into VS. However, the repository has to handle objects of greatly varying size, from very small ones (< 100 bytes) to very large ones (>100 Kbytes). Thus it is necessary to:

1. *pack* small object versions before writing them to VS,
2. *fragment* large objects versions before writing them to VS.

Large object versions are partitioned invisibly to the brokers and stored as *structured version images* (Figure 3). This partitioning is not performed solely by the repository, but starts at the level of the communication protocols, since the amount of data that can be sent in a single packet is limited. If this amount is more than the page size in the repository, the data received in individual packets will be further divided. In either case, the fragments of a new object version (token) received in different packets can be processed and written into VS as they arrive, regardless of their actual order, and without first reassembling the entire version as sent by the broker. This is an example of a data transfer protocol that applies very strongly the end-to-end argument [SALT 81].

A structured version image consists of a header that contains a list of references to data images that represent the individual fragments. A type field was added to distinguish these two substructures and the simple version images. Finally, a size field had to be added. As an object changes size, its individual versions might be either simple or structured, as shown in Figure 3. The fragmentation can also change because of changes in the lower level protocols. The repository always creates new images for the entire new version; that is, no attempt is made to share data images between versions.

The create-token operation is finished when the header of the structured image is written into VS. At this point all respective data images must have been written into VS, since the VS address of a version image is not known until that time. If a transfer of the entire version does not complete successfully, the already processed fragments (data images) will remain in VS. Data images are ignored by the crash recovery procedures.

Let us look now at the packing problem. Basically, as tokens are created, their version images (or data images that represent fragments of large versions) are placed into a buffer in the main memory. This buffer consists of one or more pages, or rather, there are several one-page buffers as shown at the top of Figure 4. When a buffer page is full, it is written atomically into VS. However, creation of a token is a commitment that, regardless of processor, memory, or device failures, if and when the the atomic action that created the token is committed, that piece of data is in the repository, undamaged. Thus creation of a token cannot be acknowledged until the entire token has been written into stable VS. Acknowledgement is thus delayed by the packing process; since new tokens will not be created at a constant rate, on occasion, it might take a long time to fill up a page. Thus, a timeout is associated with each buffer. The timeout is set when the first version image is placed into the buffer. If the buffer is not filled up before the timeout, it is written into VS partially empty, and acknowledgements can be returned.

New version images can be placed into any of the existing buffers, or, if no buffer offers enough space, a new buffer might be created, subject to a limit on the number of buffers allowed. If no more buffers may be created, one must be written into VS before the new version image can be placed. No precedence constraints exist among the buffers, so they can be written into VS in *any* order.[12] Thus the VS manager may select the buffer which is most full, or the one which is closest to its timeout. That buffer is then assigned the next sequential VS page address.

4.2. Management of Online Version Storage

Only a fraction of the information contained in VS can be kept online. One possible approach is to use a two-level arrangement where the Online Version Storage (OVS) is a demand-driven cache. This type of organization always has the unpleasant side effect that it is necessary to do address translation. It is assumed that the online storage itself will be very large, and consequently the address mapping tables could grow very large; the management of these tables is a non-trivial problem in itself. Also, in this kind of environment, it is not possible to rely on locality of reference that normally makes such multi-level storage organizations effective.

An alternative approach that will be pursued here is to let OVS hold the "top of VS", that is, the most recent 2^k pages of VS. This approach has two pleasing properties. First, the location of version images in VS is very simple. Let A_O be the VS address of the beginning of the online portion of VS. Then given the VS address A_{vi} of a version image, this version image is in the online storage if $A_{vi} >= A_O$, and $A_{vi}-A_O$ specifies the offset in the online storage. Second, the entire storage for version images can be implemented with the same type of device, and particularly, with optical disks.

Of course, there are also some problems with this arrangement. At each moment, the most recent 2^k pages of VS will contain version images created during the last τ time units (of real time), where τ is a function of k, the version creation rate, and the size of version images. Unfortunately, since some objects may not be modified for a long time, their current versions might disappear from OVS even though they are frequently read. To make sure that objects retain their current versions online, it is necessary to *copy* version images to the top of VS. As a result of this copying process VS may contain several version images that represent the same version, but multiple images of a single version cause no confusion since only *one* of these images is ever accessible by following the chain of pointers starting from the object header.

Two different policies for retaining version images in OVS were investigated [SVOB 80]: one policy is to keep the current versions of *all* objects in OVS; the other is to keep in OVS only the current versions of those objects that have been used (read or modified) in the recent past. Only the second policy and its implementation on optical disks will be described here.

Let A_C be a VS address that specifies the *copy point* in VS (Figure 4). The copy point divides OVS into two partitions that will be called the LOW space and the HIGH space. The end of VS (A_E) is in the HIGH space, that is, tokens are created originally in the HIGH space. When a current version is read, and when a token is committed, the repository compares the address of the representing version image to A_C. If the address falls below the copy point, a copy of that version image is appended at the end of VS. Thus current versions are copied from the LOW to the HIGH space as they are accessed. Moreover, if the current version of an object is not represented in OVS, its version image, when retrieved from the offline VS, is copied to the HIGH space of OVS.[13] Thus current versions of objects that have not been read for a long time can be reinstalled in OVS with this simple mechanism. If some objects should always have their current versions online, a simple "refresh" process can be provided that will periodically read such objects to force their copying in OVS.

The two-partition scheme with copying resembles a copying garbage collector [BAKE 77], however, the copying in OVS is much more restrained and consequently much simpler. Since only the current versions of objects are ever copied, it is necessary only to change the appropriate reference in the object header; the rest of the stored history remains unchanged. A simple copy flag is sufficient to guarantee that the current version will not be recopied if it is read again before its new image is written into VS and the reference in the object header is changed to the new address.

If VS (and OVS) is implemented with write-once storage devices, OVS requires a minimum of two devices, one for each of the two spaces. The copy point is always at the starting address of the device representing the HIGH space. When the HIGH space fills up, the device representing the LOW space is removed from online and replaced with a fresh device which becomes the new HIGH space, while the old HIGH space becomes the LOW space. To avoid long delays during manual replacement of the storage devices, at least three devices should be available online. This completes the picture as given in Figure 4. If VS is duplicated, another scheme could be used that takes advantage of the fact that the duplicate of the LOW space does not have to be available online; then it is sufficient to have four devices online rather than six. The details can be found in [SVOB 80].

The outlined scheme for the management of OVS raises important performance questions. Copying of version images represents time overhead, but since this copying can be performed *after* the value of the current version has been sent to the requesting broker, it does not directly affect the response time for the read requests. Also, as said before, later read requests can be satisfied while the representing version image is being copied. The presence of multiple copies, however, reduces the effective storage capacity of VS. More seriously, it reduces the effective *online* space available, since two copies of the current version of an object might coexist in OVS for a long period of time, one in each OVS space. However, in the more conventional cache-like scheme, to make the online space management problem and particularly the address translation problem manageable, it would be necessary to make OVS paged. This means that when a particular version is read, the entire VS page must be brought into OVS. Since it is unlikely that the version creation pattern and the subsequent read demands will lead to good locality of reference within VS pages, the amount of online storage "wasted" in this fashion could be much higher than in the "top of VS" model. On the other hand, the copying process might in itself improve the locality of reference.

Other performance questions pertain to the physical accesses to the online devices. Unfortunately, since write operations are multiplexed with random read accesses, the low overhead of the sequential write (append) is lost. It might be possible again to take advantage of the duplication of VS. Unless an error is detected, it is sufficient to read only one copy of the requested version; thus the load of read requests can be divided between the two devices. However, the best solution would be to use a device with multiple heads, adapted to this mode of operation.

4.3. Problem of write errors

To ensure that the version storage is stable, the entire VS should be duplicated on two separately controlled physical devices. As discussed earlier, since old data is never overwritten, the two write operations to duplicate VS can be performed concurrently, thus the response time performance does not have to degrade significantly as a price for stability.

To be able to test integrity of information in VS, a checksum is associated with each VS page. Each write to the physical devices that implement VS is followed by a read and test operation (called *careful write* [LAMP 79]). If it is decided (after possibly several read and test attempts) that the write was incorrect, the write operation must be repeated. However, if the physical device is *write-once* only, the repeated write has to write the data to a new address! This might happen even with devices that allow multiple writes to the same location, since some areas on a device could be faulty, and consequently a write operation to such a location would never succeed.

This problem can be handled in two ways. One is to leave a "hole" in the VS address space. The other one is to mask the bad write on the device level by writing into an alternative address in an area specifically reserved for this purpose. In combination with the duplication of VS, the first strategy is awkward. To preserve the simple mapping from VS addresses to addresses on the physical storage devices, the offset of a VS page should be identical on both devices. Thus, if a write operation to one of the devices does not succeed, the other one would have to be invalidated too. In other words, the same "hole" (bad data) would have to be created on both devices. Recovery from later decays presents additional problems.

The strategy chosen for the repository is to reserve on each device an area that provides substitute locations for VS pages that could not be written into their actual home address. If the repository discovers, when it attempts to read a VS page, that the corresponding physical page is bad, the VS address is mapped into the reserved area, using a simple hashing function. Consequently, both write and read operations on VS might require several device accesses, but presumably the reserved area will be used only in rare cases, so the performance penalty should be low. However, the fact that the device manager decides that a write was unsuccessful after it read a just written page back is not a sufficient guarantee that a later read will detect that the same page is bad. Thus, the device manager should explicitly mark (overwrite) the area corresponding to the bad page so that this fact can be detected reliably in the future.[14]

5. Coordinating updates of groups of objects

The key mechanisms used to coordinate updates of groups of objects are the commit record and the commit record references embedded in the individual tokens. Whether or not a token can be converted into a version always can be determined by inquiring about the state of the commit record specified in its commit record reference. In addition, a commit record can include a list of tokens in the possibility, to make conversion of tokens more efficient.

In Reed's original model [REED 78], the token list was used also to determine when the commit record can be deleted.[15] In the present design, the question when a commit record can be deleted is more complicated. The conversion of tokens is done merely by changing the references in the object header, and, as discussed in the preceding section, object headers are not stable. If the repository crashes, objects will be recovered individually by locating their latest version images in VS. Once a version image is found, it is necessary to determine whether it represents a token or a committed version; the only way to distinguish between them is to inspect the appropriate commit record. Thus commit records have to be available for inspection for a possibly long time after all of the tokens have been processed as part of the normal operation of the repository. A commit record could be deleted once all of the objects that refer to it have new versions, but clearly this is difficult to know. Thus the simplest solution is to keep the commit records in the repository forever.

5.1. Implementation of commit records

A commit record contains the state of an atomic action, and therefore the state of the possibility created by that atomic action. In addition, it includes a timeout, and a list of tokens (references to tokens) in the possibility. However, the list of tokens is only a hint.

Commit records of atomic actions in progress are maintained in the main memory, in a hash table. To commit or abort an atomic action, the repository must write the final version of the commit record into VS. More precisely, it is sufficient to include in this stable version the uid of the commit record and the final state. A commit record can be deleted from the commit record table once all of the tokens on the list have been processed, since it can be reconstructed by the recovery procedure.

5.2. Commit record representatives

The mechanisms presented thus far are sufficient to implement even atomic actions that span several repositories, but treatment of non-local tokens would be rather costly. Thus for a distributed possibility, that is, a possibility that includes objects in more than one repository, a *primary commit record* is created in one repository, and *commit record representatives* are created in each other repository that contains a token that is part of this possibility (Figure 5). When a possibility is committed or aborted, this state is encached in the commit record representatives in all involved repositories, and the commitment or deletion of tokens is done locally. The repository that contains the primary commit record maintains also a list of the representatives, but again this list is viewed only as a hint.

A commit record representative carries the same id as the primary commit record. Thus the tokens in each repository refer both to the local representative and to the primary commit record. A commit record representative is kept in the local commit record table. Since the uid of a primary commit record (in fact, of any object) includes the uid of the repository that contains it, it is easy to tell whether or not a local version of a commit record is the primary commit record or only a representative (a representative carries a "foreign" uid). A commit record representative includes a list of local tokens of the possibility, but it does not inlude a timeout.

Commit record representatives can be viewed as a mechanism that groups together tokens in the same repository and thus reduces the number of messages that must be exchanged among the repositories in order to convert properly all of the tokens in a distributed possibility. Also, once the state is encached in the local representative, crash recovery is localized to the repository that failed. It is not necessary to use a two-phase commit protocol to coordinate the involved repositories as it is done, for example, in XDFS [ISRA 78, LAMP 79, STUR 80], since the essential information is written into VS when individual *tokens* are created.

5.3. Protocol for distributed possibilities

A protocol for distributed possibilities is outlined below:

Beginning of an atomic action:

The broker of the client creates the primary commit record in one of the repositories. The uid of this primary commit record is then included in every request that is part of this atomic action.

Token accumulation phase:

When a repository receives a request to create a token for object x, it examines the commit record id contained in the request; this is always the uid of the *primary* commit record. If the respective object does not already have a committed version for the specified time, or another token that is part of another possibility, the repository will proceed to create the token.[16] If this repository does not contain the primary commit record, it checks whether it already has a representative for this possibility. If not, it creates a local representative and sends a message about this fact to the repository that contains the primary commit record. The repository then adds the reference to the new token to the token list of the local representative.

When the repository that holds the primary commit record receives a message that a commit record representative was created in another repository, it adds the uid of that repository to the list of representatives kept with the primary commit record. Note that no provisions are made to guarantee that these messages are properly received and processed.

If a repository previously had a local representative for a particular atomic action but lost it in a crash, a request to create another token for the same atomic action will simply recreate the representative. The accompanying message to the repository that contains the primary commit record has no effect if the sender is already included in the list of representatives.

Commit point:

Requests to commit or abort a possibility must be sent to the primary commit record. When the repository that contains the primary commit record receives such a request, it creates a stable version of this commit record, with the possibility state being either **committed** or **aborted.** This version may contain also the list of the local tokens and the list of the representatives in other repositories.

Conversion of tokens:

After the commit point, the tokens at the same repository as the primary commit record are converted into versions or discarded and at the same time removed from the token list of the commit record. A message specifying the final state of the possibility is sent to each repository that according to the local list contains a representative for this possibility. Each repository, when it receives such a message, creates a stable version of its local representative that carries the same state, sends an acknowledgement to the primary and starts converting the local tokens and removing them from the token list of the local representative.

Determining the state of a token:

To determine the real state of a token when an object is accessed, the commit record reference in the token is used to find the local commit record representative. If the local representative is in the **unknown** state or if it cannot be located, it is necessary to inquire at the primary commit record.

The fact that all information necessary to finish an atomic action once the final state is set is embedded in the updated objects themselves makes the update protocol very simple and robust. A computation that a client wants to perform as an atomic action never needs to be aborted because of a failure of a repository, even if the failure occurs during the token accumulation phase. The commit record in the main memory is lost if the repository crashes, but if the final state of an atomic action has already been set, there is a stable version of the commit record in VS. Absence of such a stable version indicates that the state of the atomic action has not yet been finalized. Thus the state of a commit record always can be determined reliably, and the commit record can be reinstalled in the commit record table. If the list of tokens is lost or damaged, be it during the token accumulation phase or after the commit point, the tokens will be converted individually following the procedure outlined above as later computations attempt to access those objects.

In case of distributed possibilities, messages sent between repositories can get lost without affecting the correctness of the protocol, since in the worst case a repeated inquiry at the repository that contains the primary commit record will reveal the actual state of the possibility. However, if the primary commit record was already deleted or if it was lost in a crash, the recovery procedure has to be invoked to determine the state from the information in VS. The dependency on recovery procedures can be reduced in a variety of ways [AREN 81], but the simplest and possibly also the most effective enhancement is to postpone deletion of commit records from the commit record table.

If the repository that contains the primary commit record crashes before the final state is propagated to the representatives, objects in other repositories that have tokens created by that atomic action are essentially inaccessible to other computations. until that repository recovers. This problem exists also in the classical two-phase commit protocol, but the "critical window", that is, the interval during which the individual nodes involved in an atomic action (transaction) are dependent on the coordinator is shortened by a special "prepare" message [GRAY 78, LAMP 79]. In the scheme described here. the other repositories can never abort an atomic action on their own will; for a particular repository the critical window starts with the creation of the first local token. A possible solution is to replicate the commit record in different repositories. Since the commit record is not a general data object, but one that goes through a predictable and simple sequence of stable changes, it is possible to devise a fairly simple replication algorithm for it. A voting algorithm for setting the final state in a replicated commit record is presented in [REED 78].

6. Crash recovery

The major part of crash recovery is reconstruction of the object headers. At the time of a crash the correct object headers of the recently active objects might have existed only in the main memory. Since it is assumed that a repository crash invalidates the entire content of the main memory, all such object headers are lost. Further, the OHS images of object headers might get lost or damaged, either because the repository crashed during a write to OHS or because of a decay of a block of

OHS. In both cases the object headers can be reconstructed from the information in VS to reflect correctly the current state of the stored objects.

The only piece of an object header that cannot be reconstructed from the version images in the object history is the end time of the current version. It is acceptable, however, that this field be set to the time of recovery within the repository so long as that recovery time is equal to or greater than the highest time specified in any of the already processed requests. Thus a repository must keep track of this highest time in some form of stable storage or else be able to derive it externally, from the brokers or other repositories.

The recovery process should be as efficient as possible so that the delays experienced by the clients will not be noticeable. The repository can limit the extent of crash recovery by checkpointing the object headers in VS. In addition, rather than recovering all objects in the repository before resuming normal processing, the recovery can be distributed over time, since an object header does not have to be up to date until the time when the object is again accessed.

The history of a repository is divided into recovery epochs. Every time the recovery process is started, the repository is assigned a new Recovery Epoch Number (REN). It is sufficient that each REN be unique in the history of the repository, but normally these would be numbers that monotonically increase in time. To begin recovery, the repository writes a *recovery mark* into VS that specifies the beginning of a new recovery epoch; this mark contains the new REN. The recovery marks are chained together in a similar way as the histories of the individual objects, but they are not copied.

6.1. Reinitializing the repository

The minimum repository state that has to be reinitialized before a repository can resume processing of requests from brokers and other repositories consists of the following items:

the current REN,
the VS address of the last recovery mark,
the next available uid,
the current time,
the VS write pointer.

The VS write pointer is essentially the address A_E that marks the end of VS (Figure 4). The unique identifiers for new objects and for commit records are assumed to be generated as a monotonically increasing sequence of numbers; the *next available uid* is thus sufficient to reinitialize the uid generator.

Ideally, the hardware of the repository should contain a small amount of fast atomic stable storage to hold the state of the repository. If such special storage is not available, the first four items can be determined by examining VS from its end to the last recovery mark, but a more efficient alternative is to include them in each page written into VS. Thus it suffices to find the end of VS in order to be able to restore the rest of the repository state, and the only item that has to be recovered from outside of VS is the write pointer. An interesting possibility is to maintain the write pointer in the device controller. Otherwise the device will have to be scanned to find the last page written (the beginning of the free area).

6.2. Recovery of individual objects

To rebuilt an object header, it is sufficient to find a version image of the latest version of that object, be it a committed version or a token. This version image is found by a sequential backward scan of VS. When a version image of a particular object is retrieved from VS by this method, it must be decided if it is a committed version or a token: it is necessary to consult its commit record, which itself has to be recovered.

If the retrieved version image represents a committed version, there exists a stable version of its commit record in VS. However, the copying of version images used to keep the current versions online could place a version "ahead" of its commit record in VS. A simple solution that at the same time reduces the need to check commit records during recovery is to replace the commit record reference in a *copied* version image with the actual state, that is, **committed**. Thus if the state of an atomic action was finalized before the crash, a stable version of the commit record will be found before any version image with a reference to this commit record is retrieved.

To be able to reconstruct object headers individually, the REN must be included also in each object header. When an object is created, it is assigned the current REN. When an object header is accessed as part of any of the operations listed in Figure 1, if its image in OHS is not damaged, the REN in the object header is compared to the current REN of the repository. If they differ, the object header must be updated to reflect the changes since the time it was written into OHS during the recovery epoch as given by its REN. If an object has not been accessed for a long time, several crashes (and recoveries) could have occurred since the object was created or recovered. However, since the object was not recovered earlier, it could not have been accessed since the recovery epoch given by its REN, and thus to recover such an object, it is not necessary to search VS from its current end, but only from the point that corresponds to the end of that epoch.[17] As said earlier, the recovery marks that delimit individual recovery epochs are linked together, thus the recovery procedure can get to the right section of VS rather quickly.

When an object is recovered, its REN in the reconstructed object header is set to the current REN and the object header is written into OHS. Only after this point can the requested access be honored. In addition, the reconstructed object header should be checkpointed in VS: this will delimit the extent of the next recovery should the OHS image be damaged. In such a case, the recovery must start from the current end of VS since the REN of the object is unknown. Object headers also can be checkpointed in VS during normal operation, as a background process.

The reconstruction of object headers from version images is discussed in more detail in [SVOB 80]. A more recent report by Arens [AREN 81] covers recovery of the hashed object header table (breaking of merged and cyclic hash table chains) and the checkpointing of the object headers. Arens' report also includes a model of the cost of the recovery procedures.

7. Conclusion

The design of the repository explores several new techniques of data organization and storage. First, the repository implements the history model of objects with its associated mechanisms for detection of access conflicts and for coordination of updates of groups of objects. Second, it can use write-once storage devices as the principal stable storage medium. The actual representation of object histories had to be adapted to this type of storage medium.

The choice of the representation of object histories was influenced also by the desire to avoid implementation-based dependencies between the brokers and the repositories. The repositories handle objects of arbitrary sizes, although internally the object versions might have to be packed together or divided into smaller pieces. The broker/repository protocol for reading and updating objects is independent of the lower level data communication protocols, yet if a large object version is broken into pieces by the lower level protocols, these pieces can be processed immediately as they arrive. Finally, objects updated by a single atomic action can be distributed over several repositories; all the necessary coordination of the repositories is done invisibly to the brokers.

Throughout the design, strong emphasis was put on minimizing any reliance on the knowledge of the global state of the entities used to implement atomic update and even on the knowledge of the precise state of the individual objects. The design uses to advantage the notion of a hint, an information entity that does not have to be stored reliably either because it is recontructable from the basic data stored in stable storage or because there exists another mechanism that will eventually accomplish the same task as the mechanism that uses the hint.

A major challenge in the design of the repository is the management of the Version Storage. An append-only model of storage was chosen in accordance with the basic object model. However, to keep the current versions of objects online, it is necessary from time to time to copy their images to the top of VS. In spite of this complication, this scheme is attractive because of its simplicity. Performance of this scheme has not yet been evaluated; clearly, it is the next important step.

An implementation effort was undertaken at MIT to study the feasibility of the described solutions and of the underlying assumptions.[18] Notably, the design relies implicitly on the assumption that crashes will be so infrequent that it should be much less expensive to reconstruct the hints than to store that information reliably. A crucial assumption, of course, is that optical disks will be very inexpensive and sufficiently fast and reliable. Finally, it remains to be seen to what extent multi-object and in particular multi-repository atomic updates are desired by different applications.

Acknowledgement

I am indebted to all those people who participated in the the SWALLOW project since its beginning in the fall 1979, in particular to Gail Arens, Karen Sollins, Dan Theriault, and most of all to David Reed who started the project and is leading it. Their criticism and suggestions were very valuable in shaping and clarifying the ideas presented in this paper. David Reed originally suggested the use of optical disks to store object histories. I also want to express my deep gratitude to Phil Janson, Jerry Saltzer, and Karen Sollins for their constructive comments on earlier drafts that helped me to reorganize the material into the present form.

References

AREN 81	Arens, G.C., "Recovery of the SWALLOW Repository," MIT Laboratory for Computer Science Technical Report 252, January 1981.
BAKE 77	Baker, H.G., Jr., Hewitt, C., "The Incremental Garbage Collection of Processes," *Proc. of the ACM SIGART-SIGPLAN Symposium*, Rochester, New York, August 1977.
BERN 77	Bernstein, P.A., Shipman, D.W., Rothnie, J.B., Goodman, N., "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The General Case)," Computer Corporation of America Technical Report CCA-77-09, December 1977.
DION 80	Dion, J., "The Cambridge File Server," *ACM Operating Systems Review*, Vol.14, No.4, October 1980, pp. 26-35.
GRAY 78	Gray, J.N., "Notes on Data Base Operating Systems," *Lecture Notes in Computer Science*, Vol.60, Springer-Verlag, New York, 1978, pp. 393-481.
GRAY 79	Gray, J., et. al., "The Recovery Manager of a Data Management System," IBM Research Laboratory Technical Report RJ2623, August 1979.
ISRA 78	Israel, J.E., et al., "Separating Data from Function in a Distributed File System," *Proc. of 2nd International Symposium on Operating Systems*, IRIA, Rocquencourt, France, October 1978.
LAMP 79	Lampson, B.W., Sturgis, H.E., "Crash Recovery in a Distributed Data Storage System," Xerox Palo Alto Research Center, Palo Alto, California, April 1979, to be published in *Comm. of ACM*.
PAXT 79	Paxton, W.H., "A Client-Based Transaction System to Maintain Data Integrity," *Proc. of the ACM/SIGOPS Seventh Symposium on Operating Systems Principles*, Asilomar, California, December 1979, pp. 18-23.
REED 78	Reed, D.P., *Naming and Synchronization in a Decentralized Computer System*, MIT Laboratory for Computer Science Technical Report 205, September 1978.
REED 79	Reed, D.P., "Implementing Atomic Actions on Decentralized Data," presented at the ACM/SIGOPS Seventh Symposium on Operating Systems Principles, Asilomar, California, December 1979; submitted to *Comm. of ACM*.
REED 80	Reed, D.P., Svobodova, L., "SWALLOW: A Distributed Data Storage System for a Local Network," *Proc. of the International Workshop on Local Networks*, Zurich, Switzerland, August 1980.
SALT 81	Saltzer, J.H., Reed, D.P., Clark, D.D., "End-to-End Arguments in System Design," *Proc. of the 2nd International Conference on Distributed Systems*, Paris, France, April 1980, pp. 509-512.
STUR 80	Sturgis, H.E., et al., "Issues in the Design and Use of a Distributed File System," *ACM Operating Systems Review*, Vol.14, No.3, July 1980, pp. 55-69.

SVOB 80	Svobodova, L., "Management of Object Histories in the SWALLOW Repository," MIT Laboratory for Computer Science Technical Report 243, July 1980.
SWIN 79	Swinehart, D., McDaniel, G., Boggs, D., "WFS: A Simple Shared File System for a Distributed Environment," *Proc. of the ACM/SIGOPS Seventh Symposium on Operating Systems Principles*, Asilomar, California, December 1979, pp. 9-17.
TAKA 79	Takagi, A., "Concurrent and Reliable Updates of Distributed Databases," MIT Laboratory for Computer Science Technical Memo No. 144, Cambridge, Ma., November, 1979.
THOM 79	Thomas, R.H., "A Majority Consensus Approach to Concurrency Control of Multiple Copy Databases," *ACM Trans. on Database Systems*, Vol.4, No.2, June 1979, pp. 180-209.

Footnotes

[1] This paper was written while the author was a visiting scientist at the Institut National de Recherche en Informatique et en Automatique in Rocquencourt, France. The actual research was done at the Massachusetts Institute of Technology, where it was supported by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Reseach under Contract No. N00014-75-C-0661.

[2] Atomic transactions as defined in the context of database systems [GRAY 78] are an example of atomic actions. However, the concept of an atomic action is more general. For example, any operation on an abstract data type should be an atomic action.

[3] The remote shared component that provides these functions is called usually a *file server.* This term might be somewhat confusing, since most of these so called file servers include neither file directories nor specialized file access mechanisms: they simply store data in objects that have very primitive, machine readable names.

[4] In Reed's model, this time parameter is called pseudo-time. The pseudo-time plays two roles: it serves as ordering information for resolving access conflicts, and it delimits the states in the object history. To fulfill the first role, it would be sufficient to use a monotonically increasing counter. In its second role, however, pseudo-time should correspond to real time.

[5] As presented in [REED 78, REED 79], an object history can have gaps, that is, time intervals for which the object does not yet have a valid version. A new version can be created in any such gap by creating and committing a token. When a read request is executed with t_r that falls into a gap, the end time of the immediately preceding version is exended to t_r, as it is done for the last version. However, the need for this feature in typical clients' applications does not seem to be strong enough to justify the increase in the complexity of the implementation.

[6] It is easy to extend this model so that it allows multiple tokens to be created by the same atomic action.

[7] In Reed's work, these time ranges are called psedo-temporal environments.

[8] The parameter lists as shown omit certain non-essential details. Specifically, where the figure shows a general acknowledgement, the repository returns enough information about the request and its result to make the response self-identifying. If the requested operation cannot be performed, the repository returns an error message.

[9] Unless an object has a token, the end time of its current version is also the end time of the entire object.

[10] Having references to both the current version and the token in the object header makes it easier to discard a token (remove it from the object history).

[11] Note that these locks are entirely an *internal mechanism* of the repository; they mediate accesses to the *implementation* of objects. Accesses to objects as viewed by the clients are controlled, as discussed earlier, by the time parameters in the object model.

12 Potential precedence constraints are resolved automatically at the next higher level, due to the simple fact that the VS addresses of images in the buffers are not known until the buffers are written into VS.

13 The same would happen in case of committing a token, but it is very unlikely that a token would remain unresolved for such a long time as to lose its online version image.

14 Although optical disks are called write-once devices, it is possible to write into the same area more than once: a block of storage can be overwritten with additional 1's, but the results are unpredictable. Thus it would be difficult to leave certain fields on a page blank and overwrite a page so that only those fields are changed, but hopefully this form of overwrite is sufficient to ensure that a given page is always detectably bad.

15 Consequently, it was necessary to make the token list stable and to guarantee that it includes all of the tokens in the possibility before an atomic action could be committed, because once the commit record is deleted there is no other mechanism to insure that all tokens will be properly resolved. Such a requirement greatly complicates implementation of atomic updates that involve multiple repositories.

16 The create-token operation still might fail, if the repository finds out that the state of the commit record specified in the request has already been finalized. Normally this would mean that the transaction was aborted because of a timeout. However, if a client does not wait for acknowledgements to all of its create-token requests before committing a transaction, it is possible to get into a situation where a create-token request fails because that transaction has already been committed in the repository.

17 The search might have to be extended to more recent recovery epochs if the final version of the commit record specified in a retrieved version of the object being recovered was not found in the same recovery epoch.

18 Since the implementation is still in progress, it is inevitable that the system as implemented will differ somewhat from the design described in this paper, but the concepts that have been presented actually drive the implementation. VS is simulated currently on an ordinary magnetic disk.

Operations on object histories:

create (time, commit-record-id) **returns** (object-id)
read (object-id, time, commit-record-id) **returns** (value)
create-token (object-id, time, commit-record-id, value) **returns** (ack)
delete (object-id, time, commit-record-id) **returns** (ack)

Operations on commit records:

create (timeout) **returns** (commit-record-id)
test (commit-record-id) **returns** (commit-record-state)
commit (commit-record-id) **returns** (ack)
abort (commit-record-id) **returns** (ack)

Figure 1: Operations supported by the repository

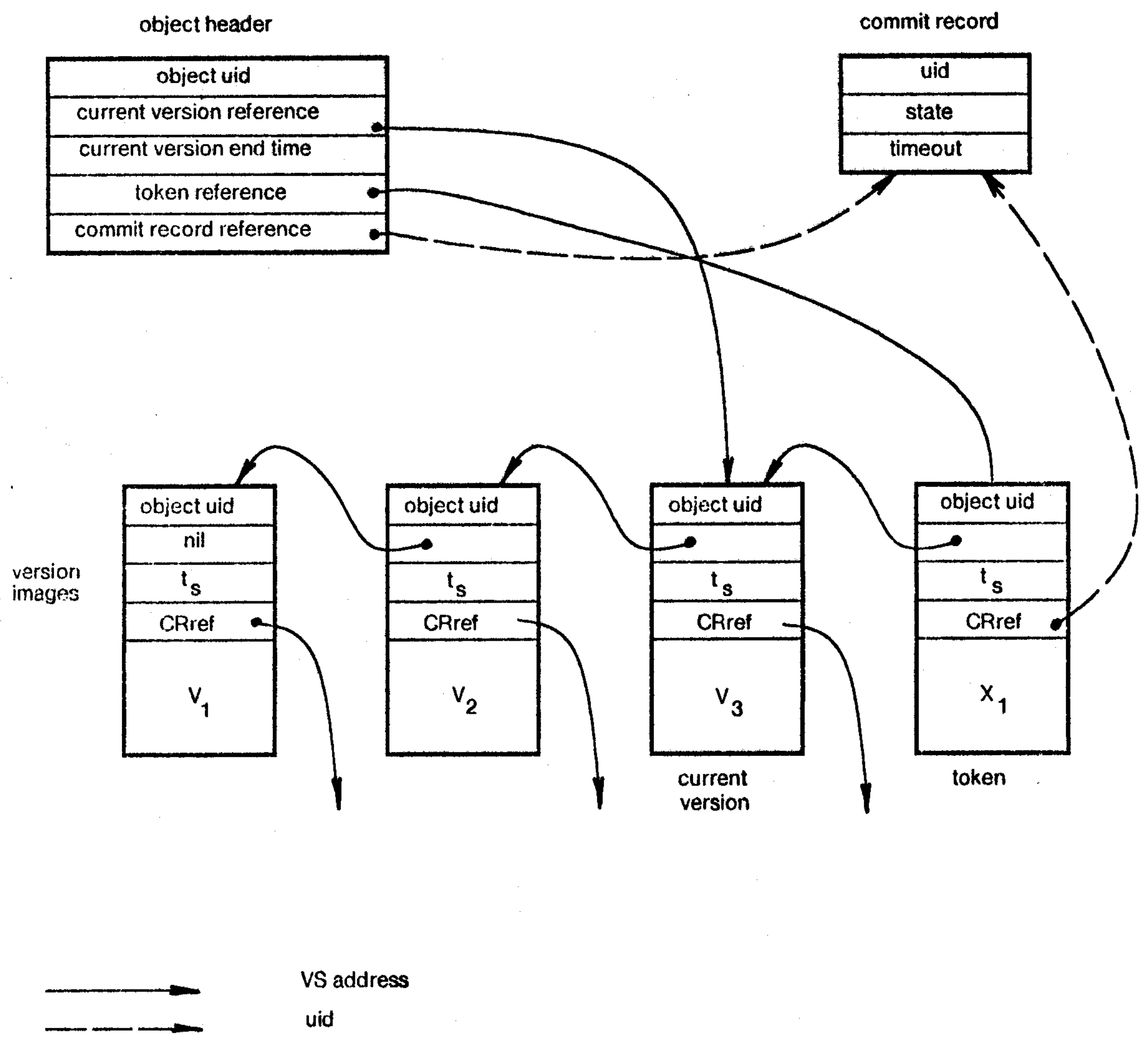

Figure 2: Representation of object histories.
Version images are immutable data structures stored in stable storage.

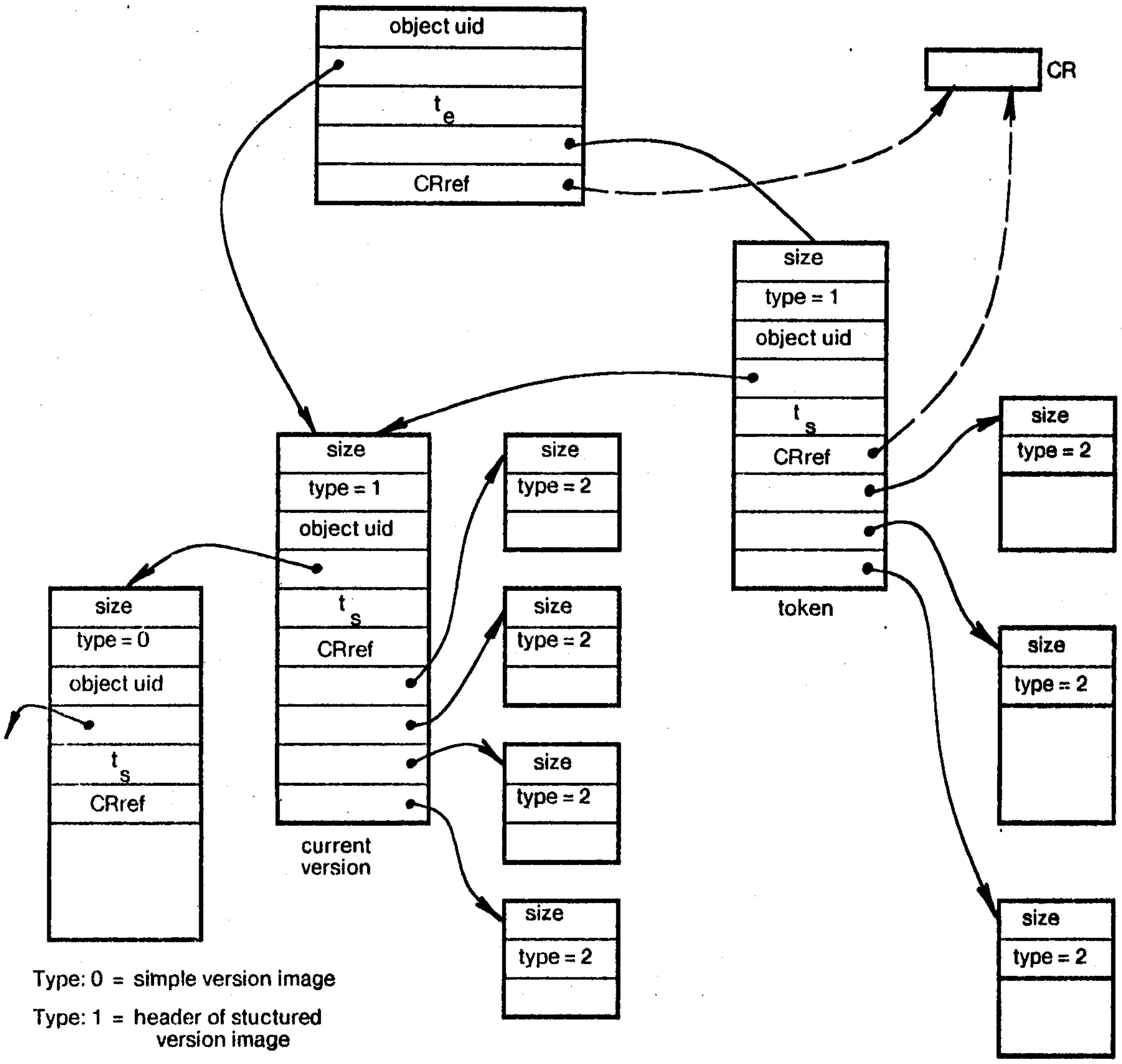

Figure 3: Representation of large object versions.
The current version and the token are stored as structured version images

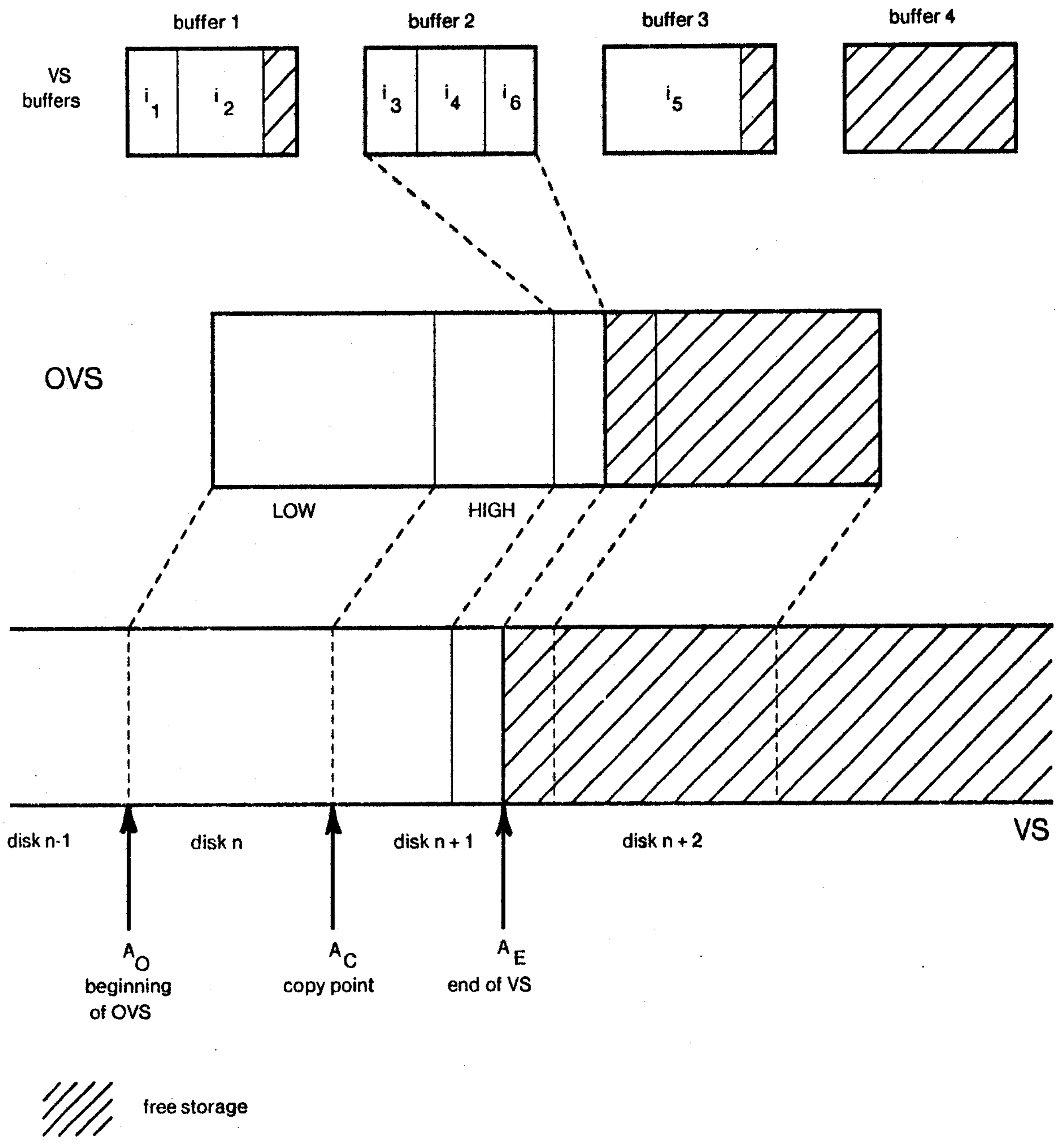

Figure 4: Organization of Version Storage.
Images i_k are packed in one-page buffers; k specifies the order in which they were created. Since buffer 2 is full, it is written into VS (OVS) before buffer 1.

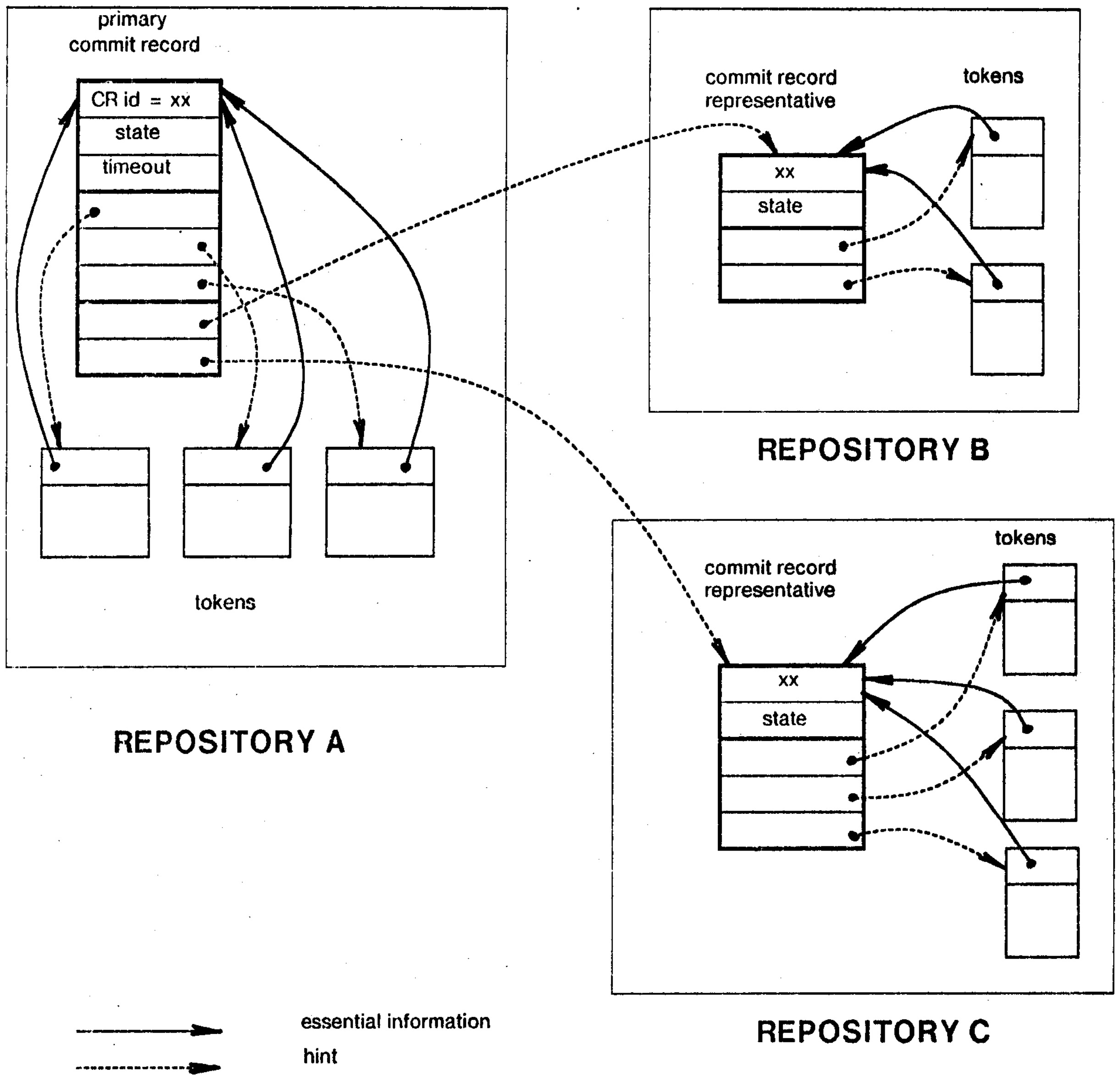

Figure 5: Implementation of a distributed possibility with commit record representatives.