MIT/LCS/TR-243

MANAGEMENT OF OBJECT HISTORIES IN THE SWALLOW REPOSITORY

Liba Svobodova

This blank page was inserted to preserve pagination.

MANAGEMENT OF OBJECT HISTORIES IN THE SWALLOW REPOSITORY

Liba Svobodova

July 1980

© Massachusetts Institute of Technology

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE
CAMBRIDGE, MASSACHUSETTS 02139

MANAGEMENT OF OBJECT HISTORIES IN THE SWALLOW REPOSITORY

ABSTRACT

SWALLOW is an experimental distributed data storage system that provides personal computers with a uniform interface to their local data and the data stored in shared remote servers called repositories. The SWALLOW repositories provide reliable, secure, and efficient long-term storage for both very small and very large objects and support updating of a group of objects at one or several repositories in a single atomic action. The repositories support, with some minor modifications, the object model developed by Reed [REED 78].

The core of the repository is stable *append-only* storage called the Version Storage (VS). VS is the only stable storage in the repository. It contains the histories of all objects in the repository and all the information needed for crash recovery. It is assumed that VS will be implemented with write-once storage devices such as optical disks. The upper 2^n words of VS are kept in the Online Version Storage (OVS). Techniques similar to real-time garbage collection are used to keep the current versions of frequently used objects in OVS. Two different policies for retaining current versions of objects in OVS are investigated; the actual implementation further depends on the type of storage devices used for OVS.

A critical concern addressed throughout the design of the repository is recovery from system crashes and storage device failures. The crash recovery of the repositories is based entirely on the information contained in VS; VS is scanned sequentially, starting from its current end, until all objects histories have been reconstructed. The recovery can be distributed over time, such that the recovery process is invoked for one object at a time, as individual objects are accessed. The same mechanism is used to recover *commit records*, which are data structures that record the state of atomic actions and group together the objects to be updated in a single atomic action. The implementation of commit records in the repository guarantees that all updates made by a specific atomic action are either all completed or all undone, regardless of failures. Further, interrupted atomic actions can be continued from the point of interruption, without any additional (backward) recovery.

Keywords: Distributed systems, atomic actions, storage management, reliability, recovery.

ACKNOWLEDGEMENT

This work builds directly on the PhD thesis of David Reed. The object model developed in that thesis forms the basis of the repository design; also, the idea of implementing a system of this kind with write-once storage devices is Reed's.

I am indebted to all those people who have participated in the discussion meetings of the SWALLOW project since its beginning in the fall 1979, in particular, to those people who have stayed with it: Gail Arens, Karen Sollins, and Dan Theriault, and of course, 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 report. Finally, I wish to thank Jerry Saltzer for his encouragement and critical comments on earlier drafts.

CONTENTS

1.	**Object model**		1
	1.1	Representation of object histories	4
	1.2	Modified object model	8
	1.3	Implementation issues	10
2.	**Version Storage**		12
	2.1	Online Version Storage	12
	2.2	Transfer of data between primary memory and VS	16
		2.2.1 Packing of version images in VS buffers	16
		2.2.2 Partitioning of large objects	17
	2.3	Mapping VS address space onto physical storage devices.	20
3.	**Management of OVS**		24
	3.1	Current versions of all objects maintained in OVS	24
	3.2	Most recently used current versions maintained in OVS	28
	3.3	Adapting OVS management to an implementation with write-once devices	29
	3.4	Online support for VS	35
4.	**Management of objects**		43
	4.1	Object headers	44
	4.2	Synchronization	46
	4.3	Object directory	49
5.	**Management of commit records**		50
	5.1	Representing commit records as objects	51
	5.2	Distributed possibilities	57
6.	**Recovery**		61
	6.1	Retrieval of VS images	62
	6.2	Reconstruction of object headers	63
	6.3	Real-time recovery	66
	6.4	Communication with brokers	68
7.	**Summary**		69
References			73
Appendix:		Structure of the repository	74

MANAGEMENT OF OBJECT HISTORIES IN THE SWALLOW REPOSITORY

SWALLOW is an experimental project that will test feasibility of several advanced ideas on design of object-oriented distributed systems. Its purpose is to provide reliable, secure and efficient storage in a distributed environment consisting of many personal machines and one or more shared repositories. The objectives and the overall structure of SWALLOW are presented in [REED 80]; the major components of the SWALLOW system are shown again in Figure 1.

Each personal machine runs a subsystem called a broker that interacts with the manager of the local storage device and the remote repositories; this broker implements a uniform interface to all objects accessible from the personal computer. The repositories provide stable, reliable, long-term storage for untyped objects. They must handle efficiently both very small and very large objects and provide mechanisms for updating of a group of objects at one or more physical nodes in a single atomic action.

This report discusses the organization and management of the repositories in the SWALLOW system. The repositories support, with some minor modifications, the object model developed by Reed [REED 78]. This model provides the basis for synchronization and recovery in the implementation of atomic actions. The main features of Reed's object model are outlined in Section 1; however, the material presented in this report assumes a much deeper knowledge of Reed's work.

1. Object model

An object can be viewed as a history of all the states assumed by the object since its creation. Each distinguishable (abstract) state of an object is represented by a special immutable entity called a *version*. In addition to having a value, a version has a time attribute that specifies its range of validity. The range of validity of a particular version is the time interval in the history of the object during which the object was known to be in the state represented by the version. Each version delimits the range of validity of the preceding version. All operations on objects include an implicit parameter, a pseudo-time, which specifies the exact point in the object's history to which this operation refers. A read operation selects a version that has the highest "start time" that is lower than the pseudo-time p specified in the read request. If the "end time" of that version is lower than p, it is extended to p. A write operation creates first a *token*, which has to be explicitly committed to become a version. The start time of that version is the pseudo-time specified in the write request. A token can be later discarded, thus returning the object history to the state that existed prior to the execution of the write operation.

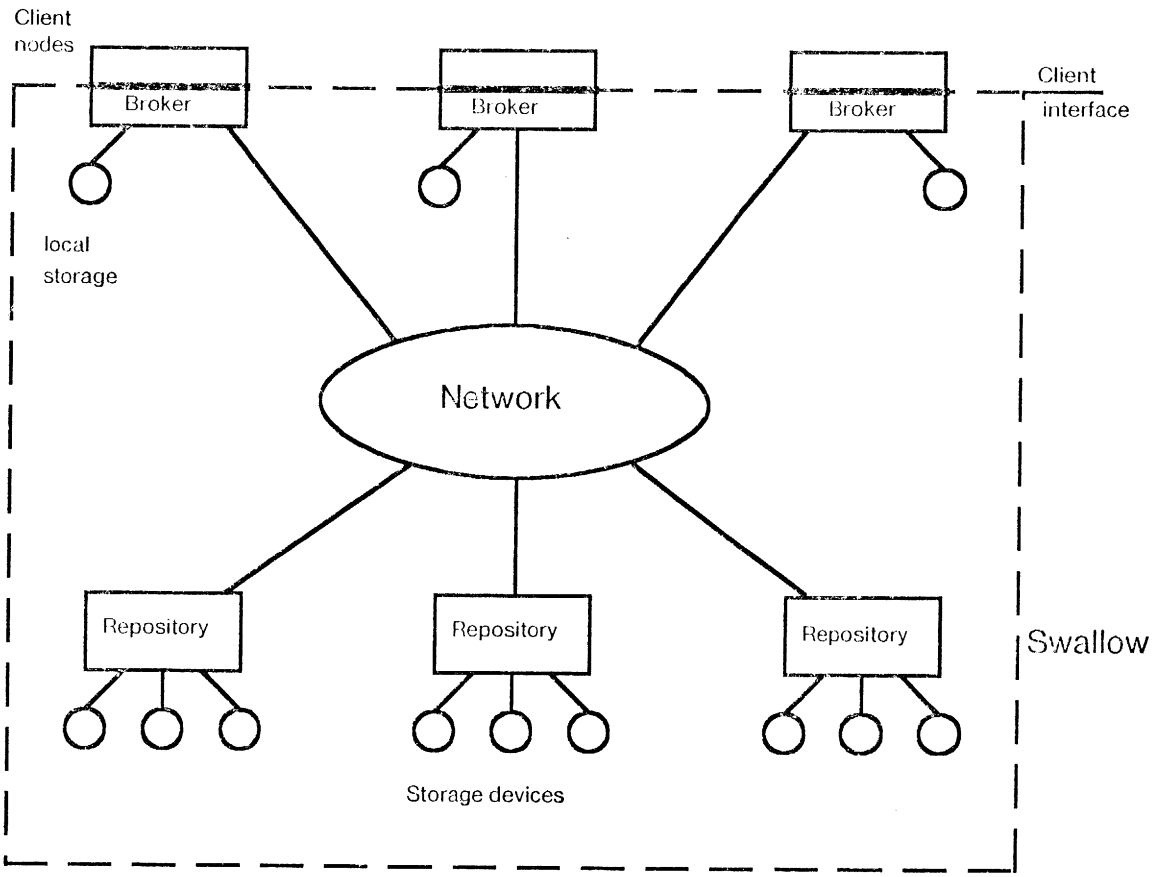


Figure 1: Structure of the SWALLOW system

The object model supports construction of *atomic actions.* An atomic action is a control abstraction that guarantees the following:

i. atomic actions are mutually exclusive, that is, operations performed as part of one atomic action cannot see or interfere with the tokens created within a different atomic action, and

ii. the tokens created as part of the same atomic action are either all committed (converted into versions) or all aborted (removed from the object histories).

Associated with an atomic action is a *pseudo-temporal environment* and a *possibility.* All operations performed within an atomic action are assigned pseudo-times from the same pseudo-temporal environment; the pseudo-temporal environment is a mechanism for making atomic actions mutually exclusive. A possibility is a group of tokens created by a specific atomic action. The possibility mechanism guarantees that only the atomic action that created the tokens can read them and that the tokens are either all committed or all aborted.

Possibilities are represented by *commit records.* A commit record is a data structure that records the state of a possibility and keeps track of what entities are dependent on the outcome of the possibility. A commit record is created with the possibility state set to **unknown.** When an atomic action completes successfully, the possibility that represents it is committed and the possibility state in the commit record is set to **committed.** If the atomic action is aborted, the possibility state in the commit record becomes **aborted.** The commit record includes a list of references to tokens created by the atomic action. Also, each token contains a reference to its commit record.

Construction of atomic actions is controlled by the brokers. This includes generation of the pseudo-temporal environment for atomic actions and creation and commitment or abortion of possibilities. The tokens in the same possibility can be created by different brokers; thus the commit records are shared data structures and must be in some repository. The repositories therefore must implement two abstractions: the object histories and the commit records. The following are the operations that can be requested by the brokers to be performed by the repositories. (Although the requests are shown in the form of procedure calls, this does not imply that a remote procedure call type of protocol will be used [LAMP 79]. Also, the lists of parameters as shown are not necessarily complete. Specifically, instead of a general acknowledgement, the repository will return 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.):

Requests that pertain to object histories:

create (pseudo-time, commit-record-id) **returns** (object-id)
read (object-id, pseudo-time, commit-record-id) **returns** (value)

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

Requests that pertain to commit records:

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

Additional operations on commit records must be supported in order to implement possibilities that involve objects in more than one repository (distributed possibilities); these operations, which can be requested only by a repository, will be discussed in Secion 5.

1.1 Representation of object histories

In Reed's original model, there may be time intervals in the object history that do not have corresponding versions (Figure 2). A new version can be created belatedly in any such time interval (by creating and committing a token), or the interval can be diminished when a request to read the value of the object at a time point that falls within this interval is executed. The latter action extends the validity range of the immediately preceding version, up to (including) the pseudo-time of the read request. Both of these forms of "eduction" have to be accomodated in the object history representation.

Figure 3a shows a linked list representation where the range of validity and the state of the version (token/committed) is physically a part of each version representation [REED 78, REED 79]. An alternative representation is to concentrate the various information about versions, including the pointers to the actual values, in a separate data structure which becomes a part of the object header (Figure 3b). The main problem with the first scheme is that the entities that represent versions are not immutable. The range of validity changes as versions are read. Also, if a new version is inserted into a gap, the "next version" link of the version that follows the new one in time must be changed. Similarly, if an action that produced a token is aborted, the token must be discarded, that is, the token must be removed from the history by destroying the pointer to the token. Another disadvantage is that if an operation refers to an older part of the history, it is necessary to inspect all newer versions to find the appropriate version (or gap). The other scheme (b) leads to more complicated storage management. The size of the object header varies from object to object and changes as new versions are created; also, since it must be possible to insert new entries anywhere in the version list, a simple array representation is not possible. Second, the number of versions in

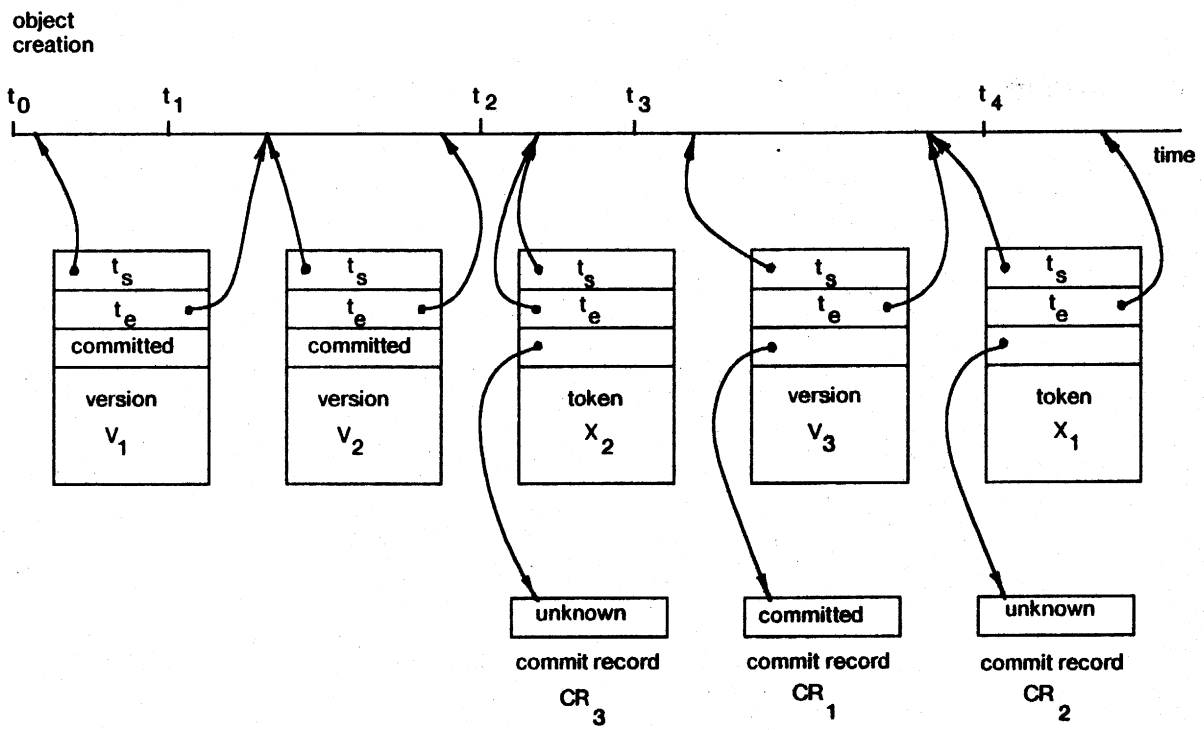
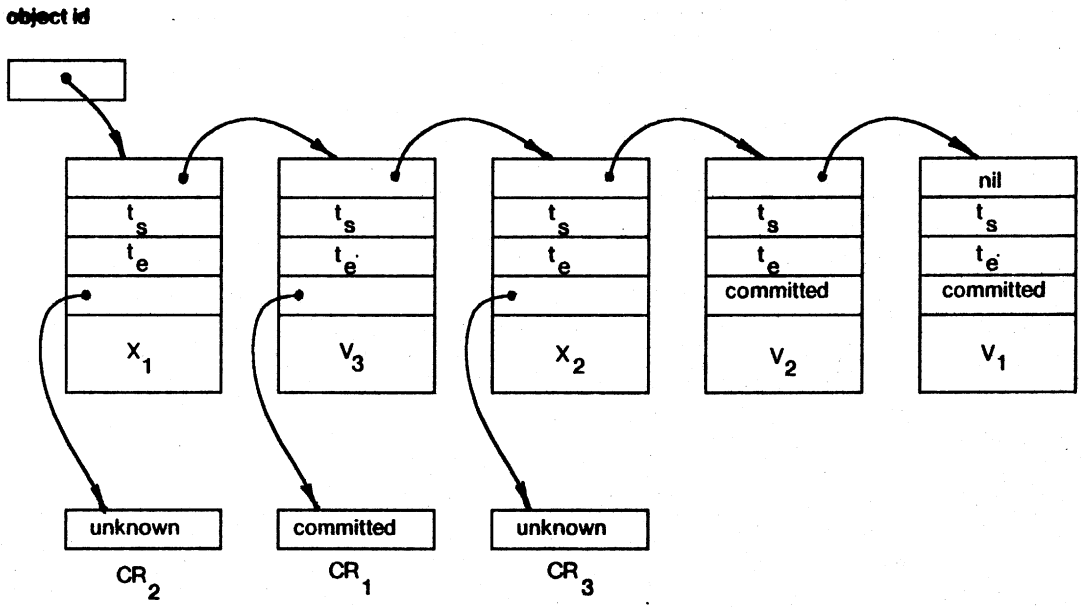
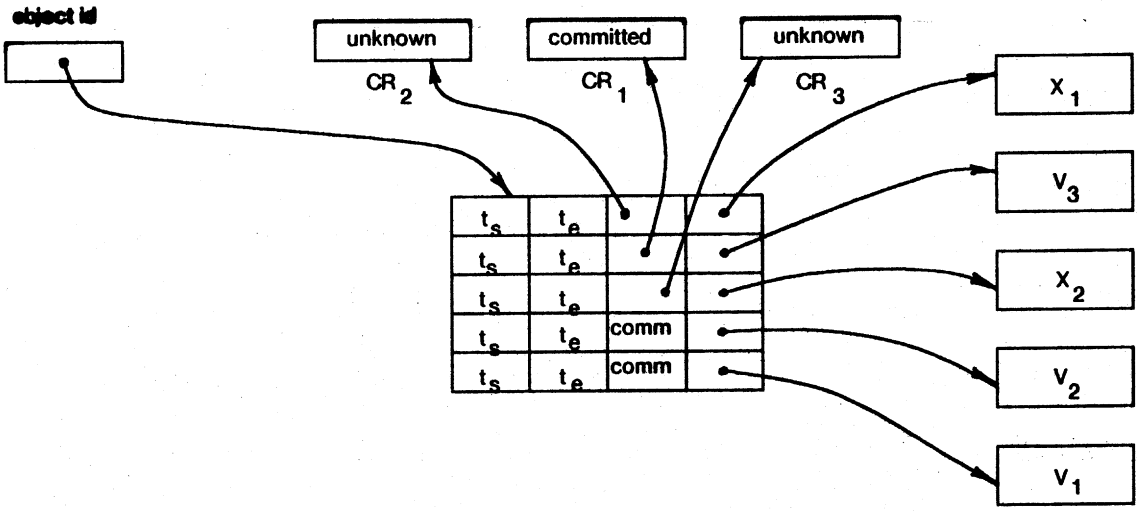


Figure 2: An example of an object history.

Token X_2 was created after version V_3 and token X_1. Version V_3 was committed recently, but has not had its state encached yet. Reading the object at time t_1 will return the value of version V_1. Reading the object at time t_2 will return the value of version V_2, after extending the validity of this version (end time t_e) to t_2. Attempts to read the object at time t_3 and t_4 will result in a wait, pending commitment or abortion of tokens X_2 and X_1 respectively, unless the read operation is requested from within the same possibility under which the token was created.



a. Version information stored with the version value



b. Version information concentrated in the object header

Figure 3: Possible representations of known object histories; shown for the example given in Figure 2.

an object history may grow very large, and old versions must be removed from online storage. If the stored versions physically contain the validity range and linking information, this information will be purged from online storage automatically with the old versions. If the list of version references is kept in the object header, it may have to be pruned separately.

It is highly desirable to represent versions by immutable storage entities. Perhaps the strongest reason for this restriction is that it is much simpler to design mechanisms to ensure integrity of stored versions.

One of the main functions of the repository is to provide very reliable storage. This means that the physical storage must be stable, that is, the information stored in it must not decay over time. In addition, it is necessary to ensure that information written to it is either written completely and correctly or not at all, that is, that the operations on stable storage are *atomic*. Since no physical device provides storage with these properties, the atomic stable storage must be implemented as an abstraction, using hardware components with less desirable properties. In particular, atomic stable storage must be designed to tolerate processor crashes during write operations and decays of the storage media. This is accomplished by writing the data twice, into decay-independent sets [LAMP 79].

An operation that is most difficult to perform atomically is an in-place update of stored information. An atomic update means that either the content of the updated entity is changed into the new value or, if the operation fails, the value of this entity is left unchanged. That is, atomicity guarantees that a stored entity is never left in an inconsistent state where the old value has been lost and the new value is incorrect. To perform an atomic update, the two copies of stored information in the decay-independent sets must be changed strictly sequentially, i.e. 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, the problem of atomicity is simplified. It is still necessary to have two copies for stability, and the ability to detect and correct bad writes, but the two writes into the two decay-independent sets can be done concurrently.

A second strong motivation for choosing an immutable representation for object versions and tokens is the possibility of using optical disks, which are write-once storage. The given object model will require a large amount of storage. Thus, it is important to utilize storage devices that are: 1) inexpensive, 2) easy to store offline. To provide fast access to old versions, a random access device is needed. Optical disks look promising in all these aspects.

To satisfy the immutability requirement with the present object model, it would be necessary to use the scheme of Figure 3b. However, it will be shown that with a minor modification to the

conceptual object model it is possible (and better) to include most information about versions in the version representation.

1.2 Modified object model

If we allow insertion of new versions in an arbitrary place in the list, the information about the ordering of the existing versions (the physical pointers to stored versions) must be kept in storage that allows multiple (unlimited) writes. In addition, the "end time" information for each version has to be kept in such storage, since it must be changed when a version is to be read at a pseudo-time greater than the current end time. Another possiblility would be to completely rewrite each version every time its end time must be extended and when a new version is inserted after it, but such a scheme does not seem practical.

Let us constrain the conceptual model such that when a new version is created, the end time of the previous version is extended to close the gap. This means that new versions can be inserted only at the "current" end of the list. Also, each object can have at most one token. Actually, an object could have multiple "dependent" tokens at the "current" end, as it is done in Takagi's scheme [TAKA 79]. This possibility will not be investigated in this report. However, with the exception of the current (latest) version and the token, the end time of a version can be derived from the start time of the next newer version and thus does not have to be included in the version representation. Consequently, an object history can be represented by a fixed-size object header and a growing list of immutable entities that represent the versions.

The data structures needed to represent an object history are shown in Figure 4. The object header contains a reference to the current version of the object and the end time of the current version. This time is updated every time the current version is read past its end time. The object header also includes a token reference that is either null if the object does not have a token or it contains the physical address of the current token. One reason for including both the current version reference and the token reference in the object header is that it is simpler to discard a token (remove it from the object history) when the atomic action that created it is aborted. However, having both of these references in the object header is crucial to the storage management, as will be seen later. Tokens can be read from within the atomic actions that created them; each such read extends the end time of this future version. Since the end time of the current version should not be automatically extended up to the start time of the token until that token is actually committed, it is necessary to keep track of the end time of the tokens as well as the end time of the current versions. It should be kept in mind that the current version end time and token end time in the object header are *pseudo-times* that do not necessarily correspond to real time. Finally, a reference to the commit record for the current token is

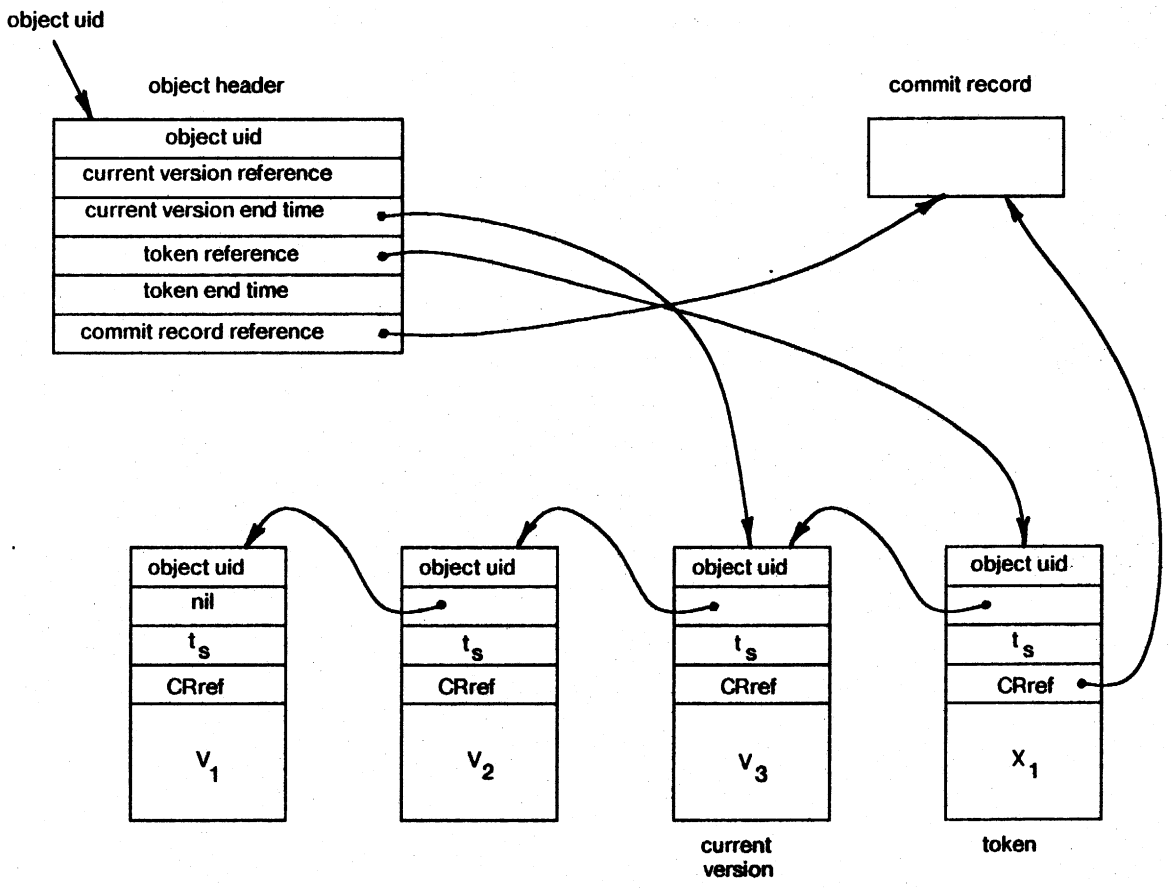


Figure 4: Representation of the object history for the modified object model; it is not possible to create token X_2 in this model.

contained in the object header, although this is only an optimization, since this information is present also in the token.

The data structures that represent the 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, the uid of the object it represents and a reference to the commit record for this version. The last two items are needed for recovery, as will be explained later. The time t_s specifies the beginning of the time interval in the object's history represented by that version. Again, t_s is not the real time when the version image was created, but the pseudo-time specified in the request to create a token.

It is important to make a distinction between versions and the representation of versions, that is, the version images. A version is a logical concept; it is the value of the object during a specific 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 copies of a version image may coexist in the repository. Since versions are immutable, this does not cause any synchronization problem. Also, a version image may remain in the repository although it no longer represents a valid version. Thus to discard a token when the action that created it is aborted, it is sufficient to set the token reference field in the object header to null.

In addition to eliminating the need to include mutable data structures in the version representation, the modified model also eliminates the need to perform a write operation when an older version is read. The lost ability to leave regions of the object's history undefined and create versions in such regions later does not reduce significantly the power of the object model. In most situations, an object that is to be updated is read first, and it is desirable to extend the end time of the read version up to the start time of the new version to ensure that the object has not been changed after it was read.

1.3 Implementation issues

A crucial problem is to find an efficient and reliable scheme for mapping object histories into physical storage. The two structures used to implement object histories, the object header and the list of version images, require different models of storage and different management policies. Object headers are mutable and therefore must be kept in storage that allows modifications of stored information. The version images are immutable and thus can be stored in write-once storage. In addition, the reliability requirements are different.

The main issue in the implementation of the lists of versions is storage allocation and management. Giving each object a section of consecutive physical storage locations for its entire history is clearly

infeasible. Rather, it seems natural to view the version storage as a *history* of creation and updates of *all* the objects in the repository. Section 2 develops a model of the version storage as an *infinite append-only file.* Since it is infeasible to keep the entire version storage online, the online portion of the version storage must be "reusable", that is, it must be possible to free it for newer version images. This problem is studied in more depth in Section 3. That section addresses also the problem of the assignment and management of the physical storage devices used to implement VS.

The role and management of object headers is discussed in Section 4. It is too expensive to immediately reflect all changes to an object header in stable storage. Therefore, the object headers are viewed only as hints that may be destroyed by a processor or storage device failure, but are reconstructable from the information contained in the version images. That section also addresses how objects are located and how concurrent requests for the same object are synchronized.

Section 5 discusses the implementation and management of commit records. Commit records are special data types provided by the repository, but are ultimately mapped into the same object model as other data. For possibilities that include objects in more than one repository, commit record representatives are added to the model.

Recovery issues are addressed throughout this report, but the major step, the reconstruction of object headers, is described in Section 6. Finally, Section 7 presents a summary, including a list of issues that must be studied in more depth.

2. Version Storage

The core of the repository is the Version Storage (VS). Abstractly, VS is an infinite append-only tape. VS stores information as stable immutable entities. These entities will be called *VS images*. A VS image consists of two fields: the data field, which at this level is simply an uninterpreted sequence of bits, and the size field. VS is the only stable storage in the repository. It will contain all versions of all objects in the repository. In addition, all the information needed for a crash recovery must be stored in VS, as immutable VS images.

Version images, as described in Section 1.2, are contained in the data field of VS images. That is, for storage in VS, an envelope that contains the size field is added (Figure 5). The version references in individual VS images as well as the current version reference and the token reference in the object header are directly the addresses of the representing VS images in VS, A_{vi}. The lists of versions representing histories of different objects are intertwined in VS; their ordering in VS is determined by the relative frequencies of updates of individual objects. And to some extent also by read activities, as will be seen later.

Since VS may 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 2.1. Section 2.2 is concerned with the transfer of data between the primary memory and VS. Small objects (version images of small objects) must be packed into buffers while large objects have to be partitioned into smaller pieces. Finally, Section 2.3 discusses some problems with the mapping of the VS address space into the physical address spaces of the used storage devices.

2.1 Online Version Storage

Only a fraction of the information contained in VS can be made available online. One approach is to add a special kind of cache for the current versions of all objects. The most straightforward policy for controlling the use of such a cache is to replace (overwrite) the version in the cache when a new version of that object is created. However, this new version may never be committed; when it is written into the cache, it is only a token. Alternatively, the cache could be assigned to contain the latest committed version of each object *and* the tokens. When a token is committed, the other, now old, version would be deleted and the freed space reused. Since version images can vary greatly in size, the cache storage would become fragmented and it would be necessary to do recompaction or garbage-collection. This problem arises even if tokens are allowed to overwrite the committed versions in the cache, since subsequent versions of an object can have greatly different sizes! Another unpleasant aspect of this form of caching is that there is no easy way to deduce the location of a version images in the cache from its address in VS and vice versa; thus two addresses

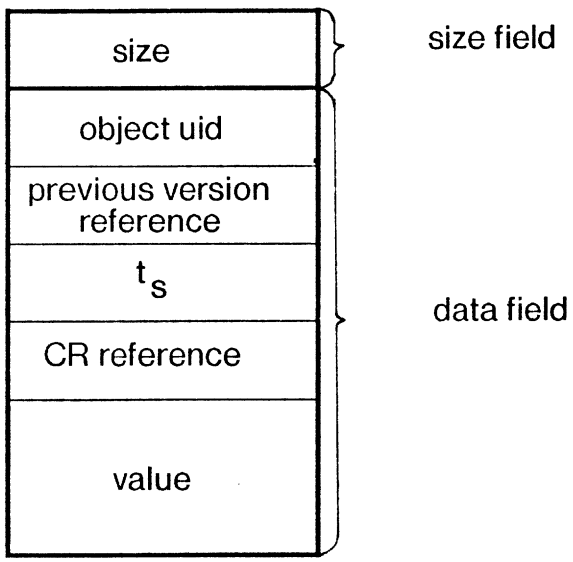


Figure 5: VS image representing a version image.

have to be remembered for each version image in the cache.

Instead of using a cache, the Online Version Storage (OVS), that is, the portion of VS currently available online, will be the most recent 2^n words of VS. OVS will be implemented as a circular buffer, as illustrated in Figure 6. Mark M_E will be used to specify the current end of VS on the device that serves as OVS. New version images are created always in OVS, but for read requests, it is necessary to determine if an image of the specified version exists in OVS. Such a check is very simple: if $(A_E - A_{vi}) <= 2^n$, where A_E is the VS address of M_E, then the version image is in OVS, and its address in OVS is $(A_{vi} \bmod 2^n)$.

OVS shall contain the version images created during the interval $(t_c\text{-}T, t_c>$ where t_c is the current time and T is determined by the speed with which the available online version storage fills up. Unfortunately, since versions of different objects are created at different rates even the current versions of some objects may disappear from OVS. To make sure that all or some objects (for example, those objects that are read frequently) retain their current versions in OVS, it is necessary to *copy* version images in OVS, and consequently in VS.

To preserve the current versions of objects in OVS, it is not sufficient to copy just the immediate current versions when the time comes to reuse the respective fragment of OVS space: the tokens have to be copied too. But, if an object has a token at the time the latest image of the current version is to disappear from OVS, it is still necessary to copy the current version, since the token later may be aborted.

When an image of a current version or a token is copied, the appropriate reference in the object header must be changed. But if an object has a token, a reference to the current version appears not only in the object header, but also in the token. Since the tokens are to be immutable, the reference to the current version embedded in the token cannot be changed; it will always refer to the version image that represented the current version at the time when the token was created. Fortunately, the fact that the reference in the token is not modified does not lead to an error. If the token becomes a version image, the reference to the copied version, which existed only in the object header, is replaced by the reference to the version image of the former token. The copied version image in OVS is effectively lost, but the object does have its current version in OVS. If the token is aborted, the current version is found in OVS as it should be.

To summarize, as a consequence of the copying, VS may contain many version images that represent the same version, but only *one* of these images is accessible by following the chain of pointers in the object history. The other images use up storage, but do not have an adverse impact on the implementation of the object histories.

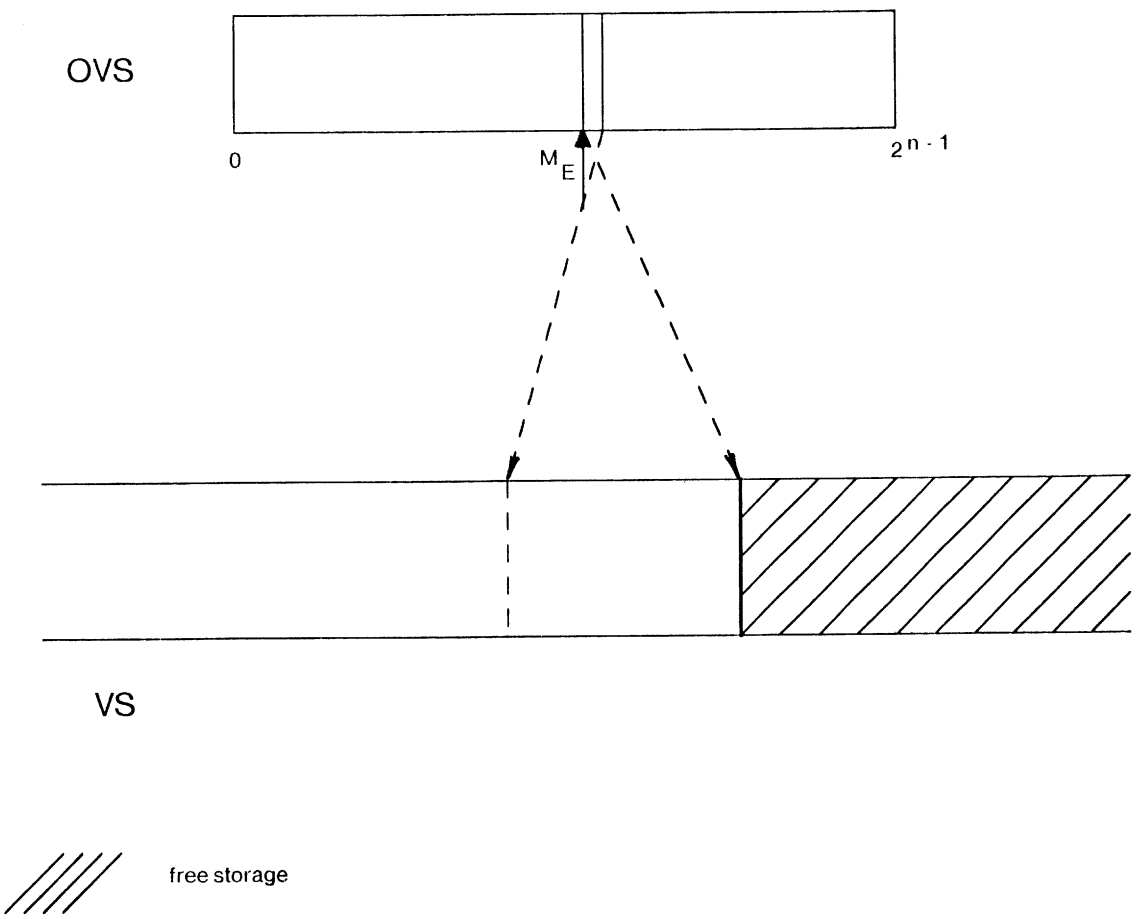


Figure 6: OVS as a circular buffer.

New version images are appended at the mark M_E.

A more detailed model of OVS will be presented in Section 3. Two different policies for retaining version images in OVS will be investigated: 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 in the recent past. The actual implementation of these policies depends further on the type of storage devices used.

2.2 Transfer of data between primary memory and stable VS

The repository has to handle efficiently objects of greatly varying size, from very small ones (< 100 bytes) to very large ones (>100 Kbytes). It would be very expensive to write small version images into VS individually. Because of the constraints of the communication network and protocols, very large objects will be sent to the repository in pieces; it would be very expensive if not impossible to buffer very large objects in primary memory.

Thus, prior to creating new versions of objects in VS, it is necessary to:

1. *pack* small version images (tokens) before writing them to VS
2. *fragment* large objects before writing them to VS.

For easier management of VS (mainly for faster VS address resolution and object location), it is desirable to allocate VS in fixed-sized blocks. These fixed-sized blocks, or *pages*, are the units of *atomic write* into VS. Both the packing and fragmentation must take this into consideration.

2.2.1 Packing of version images in VS buffers

Let us first look at the packing problem. Basically, as tokens for new versions are created, their version images are placed into a buffer in main memory. This buffer consists of one or more pages. When a buffer page is full, it is written atomically into VS. However, there are two problems with this scenario. First, creation of a token is a commitment that, regardless of processor, memory, or device failures, if and when the possibility under which the token was created is committed, the token is in the repository, undamaged. Thus a creation of a token cannot be acknowledged until the token has been written into stable VS. This action is delayed by the packing process; since new tokens will not be created at a constant rate, on an occasion, it may take a long time to fill up a page. Thus, a timeout should be associated with each buffer page; if a buffer page is not filled up before the timeout, it is written into stable storage partially empty. The filling of the buffer is sped up by the copying process which creates copies of old current versions and tokens at the "high" end of VS; these copies again are first written into the buffer.

The second problem is what to do if a version image just created or copied does not fit into the

space remaining in the buffer page. Or, restated, the question is whether a version image should be allowed to cross a page boundary. Although such a provision would lead to a better storage utilization and a possibility to deal more flexibly with large objects, there are strong reasons for not permitting it. Once split version images are permitted, almost every page will end with a split image, unless some restrictions are imposed in regards to how version images can be split. A read operation on a split image requires more than one VS access. Two VS accesses if the maximum permitted size of a version image is one page. Also, crash recovery would be slightly more complicated: since the repository may crash between the writes that involve a split image, the recovery algorithm would have to detect that the image is incomplete. The last consideration is that the buffer pages that contain parts of a split image have to be mapped sequentially into the VS address space. The alternative scheme described next will demonstrate the advantage of the lack of this restriction.

If split version images are not allowed, it does not mean that the buffer pages have to be written into VS half empty. As already indicated, the buffer in the main memory may consist of several pages, or, better, at any time, there may be *several one-page buffers* for VS in the main memory, as shown in Figure 7. The timeout for each buffer is set when the first version image is placed into that buffer. Now, new version images can be placed into any of the existing buffers, or, if no buffer offers enough space, a new buffer may 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. Since no ordering (precedence constraints) exist among the buffers, they can be written into VS in *any order.* 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. This means that the actual VS address of a version image is not known until the containing page is written into VS. The timeout associated with each buffer guarantees that no buffer will wait forever for a version image of the "right" size.

2.2.2 Partitioning of large objects

Large objects are partitioned invisibly to the brokers. However, 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 less than or equal to the page size in the repository, no further partitioning is needed; otherwise the data received in individual packets must be further divided. In either case, the fragments of an object (token) received in different packets can be processed and written into VS as they arrive; each fragment will become a separate version image. Since this partitioning is invisible to the brokers, a broker must always read or write the *whole* object, i.e., it is not possible to retrieve or to update only a small portion. This means that it should be sufficient to *chain* together the fragments of such an object and let the

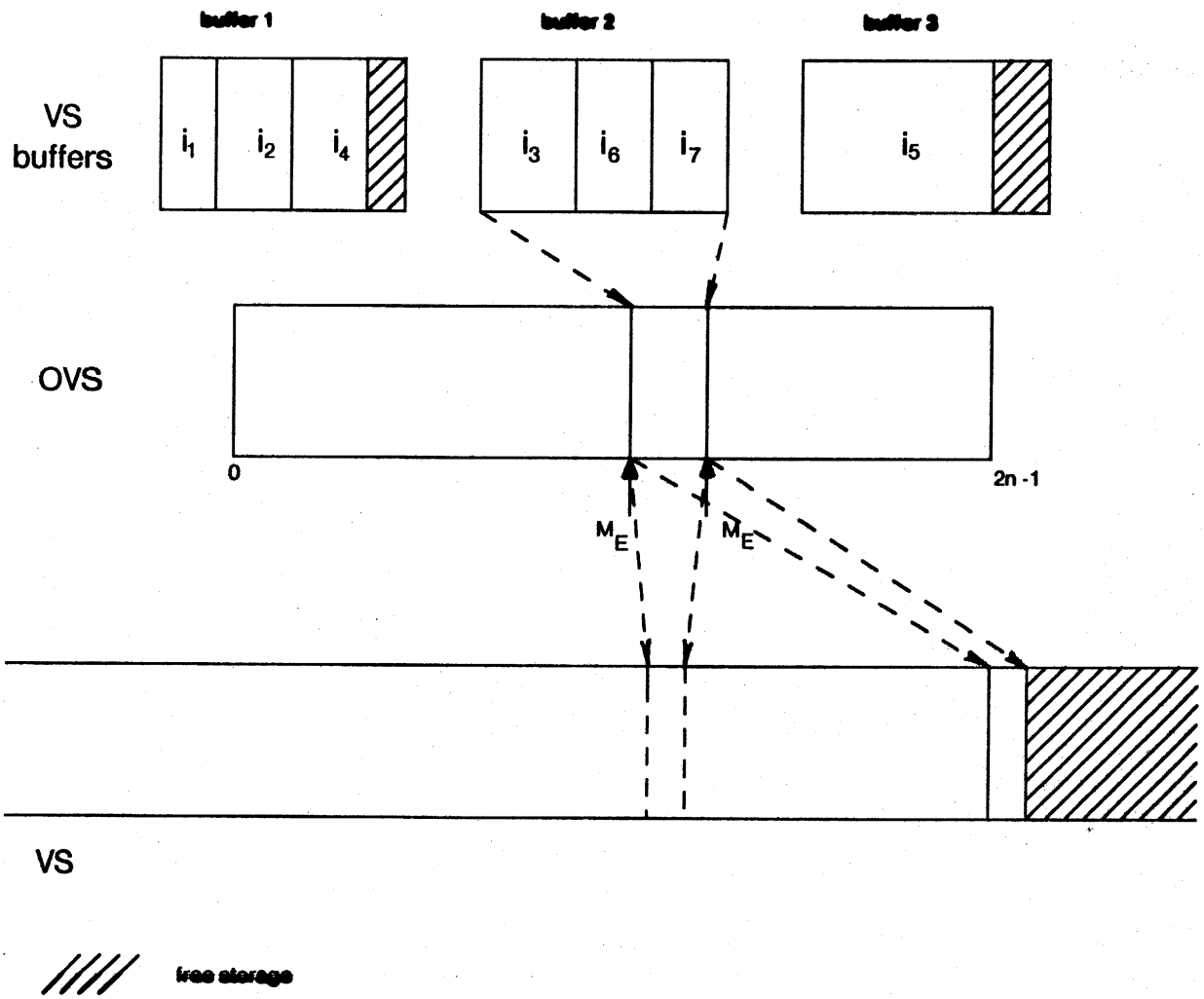


Figure 7: Writing VS image into VS.

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 (via OVS) before buffer 1.

object header point to the *last* fragment; it is not necessary to have *random* access to the individual fragments. Unfortunately, if a version image that represents such a fragment of an object is copied by the OVS manager, it would be necessary to modify a pointer in the version image that represents the next piece, but this is impossible since the version images are immutable. On the other hand, since the whole object (object version) will be read, all fragments should be copied, and the embedded pointers can be modified as each fragment is copied. However, although the object header must point to the last fragment, the copying must start with the first fragment, otherwise the new VS addresses of the individual fragments cannot be determined. Actually, this also impacts the initial creation of a version of a partitioned object. A version image of a piece k cannot be created until the VS address of the version image of the fragment k-1 is known; this again imposes precedence constraints on the set of buffers for VS.

To overcome these problems, it is necessary to have a special pointer array. There are several reasons for *not* including this pointer array in the object header: as will be seen in Section 4, the entire object header must be reconstructable from the information stored in VS and therefore the images of the individual fragments would have to include additional information; object headers would have different sizes, and the size of a particular object header could vary over its lifetime; but the most serious problem is that this would necessitate reconsideration of how to represent object histories. What would be the meaning of the "previous version" reference in each version image? Different versions of an object can be partitioned in different ways, so there is no meaningful mapping between fragment k of one version and fragment k of the preceding version.

Thus the pointer array will be stored in VS. In fact, it will look like a version image. This does not require any changes to the object header: the current version reference and the token reference simply point to images that contain the appropriate pointer arrays, as do the "previous version" pointers in each version image. A version image constructed in this way will be called a *structured* version image. The individual fragments referred to through this pointer array can be of different sizes. Both the VS image that contains the pointer array and the images of the individual fragments will be packed in VS buffers as before.

Both for normal operations on objects and for recovery, the information whether a version is simple (represented by a single version image) or structured must be included in the version images themselves. It does not make sense, though, to propagate this distinction into the definition of an object, since the representation may change during object's lifetime: as an object changes size, individual versions may be either simple or structured. This can also happen because of changes in the lower level communication protocols (flow control). Also, it is superfluous to include all the information so far associated with all version images in those images that represent the individual

fragments of a structured object version. In fact, none of these fields is needed! Thus for representation of object versions and tokens, the repository should provide three distinct types of stable entities:

simple version image:	self-identifying; data field contains the actual data
header of structured version image:	self-identifying; data field contains an array of pointers to data images
data image:	interpretable only in the context of the appropriate structured version image; not used during recovery.

Figure 8 shows a fraction of an object history that uses both simple and structured version images, and consequently all three types of stable entities just described. However, these distinct entities should be supported on a higher level of abstraction than VS; the stability is assured by mapping them into the same uninterpreted stable VS images.

Use of structured version images does not impose any precedence constraints on the transfer of main memory buffers to VS. Of course, the header of a structured version image cannot be *created* until all data images of that version have been written into VS, since the VS addresses are not known until then. If such a version image is copied by the OVS manager, it is necessary to *create* a new header after all data images have been copied. Structured version images are substantially more expensive than simple version images, thus fragmentation should be used only when necessary.

2.3 Mapping VS address space onto physical storage devices

To ensure that the version storage is stable, all VS images should be written twice, that is, the entire VS should be duplicated. It can be assumed that two separately controlled physical devices provide decay-independent sets from the point of view of physical failures of the driving hardware, e.g. head crashes. As discussed earlier, 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.

In addition to ensuring stability of *stored* information, it is necessary to ensure that version images are written correctly into VS. The usual approach is to follow each write by a read and a test

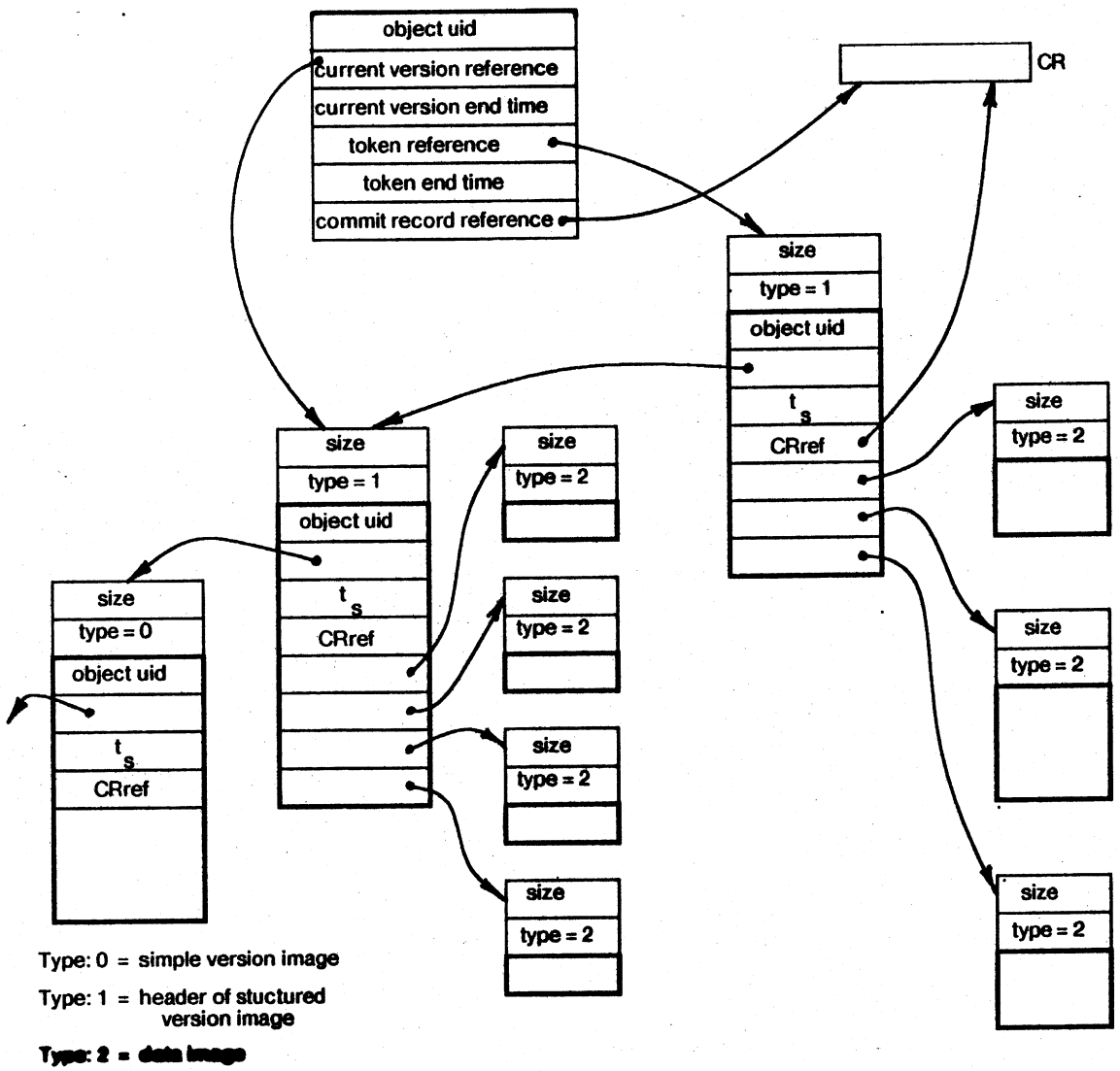


Figure 8: Representation of large object versions.

The current version and the token are reprresented as structured version images.

operation. 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 may happen even with devices that allow multiple writes to the same location, since some areas on a device may be faulty, and consequently a write operation to such a location can 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 the first case, the *correct* VS address cannot be determined until after the write to VS has succeeded. This means only that the token reference (or the current version reference, when a copy operation is performed by the OVS manager) in the object header cannot be set until the VS write terminates, but this order must be upheld anyway. However, the duplication of VS creates an additional problem. The address of each of the two copies of each version image must be easily computable from the VS address. Thus, for a duplicated write if *one* write operation does not succeed, the other one must be invalidated also. Thus, the same "hole" (bad data) has to be created on both devices. This scheme, however, cannot support recovery from later decays. When it is discovered that some *old* version was damaged on one device, than in order to restore the redundancy for the future, it would be necessary to copy the entire device, but in this process, different bad writes may occur, and the two copies of that part of VS would be out of sync! Note that it is *not* possible to copy just the respective version image (from the other device), since then the entire "newer history" of that object, that is, the portion of the object history between the current version and the version represented by the defective version image, would have to be recreated.

Thus, the chosen approach is to preserve the continuity of the VS address space. Each device must have a reserved area that provides substitute locations for data that could not be written into its correct address. There still may be "holes" on the device, but when such a hole is detected, the reserved area is searched for the missing data. Thus both write and read operations on VS may 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 does not guarantee that on a later read the same entity will be detected as bad. Thus, the device manager should explicitely mark (overwrite) the areas declared to be holes, in such a way that holes can be reliably detected in the future.

Finally, it is necessary to address the problem of VS performance. The provision for maintaining the current versions online is only the first step. The performance of the repository will depend strongly on the performance of OVS, that is, on the speed of reading from and writing to OVS.

Since write operations are multiplexed with random read accesses, the low overhead of the sequential write (append) operations on VS is lost. However, the repository is shared, and thus there may be many outstanding read requests to different locations of the OVS device. The performance of the device (throughput) can be improved significantly if these requests are processed in an order that minimizes the positioning overhead. The most effective disk scheduling algorithm is to scan the disk in alternating directions, servicing requests in the order of their physical addresses. Several variants of the basic SCAN algorithm were developed and analyzed [COFF 73]; however, since the address distribution of requests in OVS is not completely random, it may be possible to find a variant of SCAN that will perform better than these general algorithms. Also, a possible enhancement of the SCAN scheduling algorithm for the OVS device is to force a write of one of the VS buffers when the disk heads reach the current end of OVS (M_E).

In addition to finding a suitable algorithm for the OVS device management, performance of VS can also be influenced by:

i. assigning physical addresses to VS addresses
ii. mapping VS access requests to physical devices.

One possibility is to *interleave* VS, that is, assign consecutive VS blocks to different physical devices. This of course requires additional device drives. However, it is possible to take advantage of the *duplication* of VS. If both devices in this duplicated implementation provide fast random read access, a read request can be satisfied by either of the two devices and can be scheduled for that device which is more convenient (i.e., not currently busy, or needs less time to locate the requested version image).

3. Management of OVS

The Online Version Storage is very important to the performance of the repository. As presented in Section 2.1, OVS is an online address space managed as a circular buffer that contains the most recent 2^n words of VS. If no version images must be copied, removal of old version images is accomplished by simply overwriting them as M_E, the end of VS mark, reaches that part of OVS. However, if a version image must be copied to maintain the current version of the respective object in OVS, a rather unpleasant situation may arise: in order to write a version image for a new version, the OVS manager must copy one or more version images that lay ahead of M_E to make enough space for this new version image. However, in order to make space for the copied version images, more space has to be freed. Such a "chain reaction" can be prevented if the OVS manager looks ahead at which version images may have to be copied and performs the copying before that part of OVS space must be overwritten. On the other hand, if the copying is postponed, it may not be necessary to copy an old version image of a current version physically, since it is approximately in the right place with respect to M_E, but some storage may have to be wasted in return. M_C will be used to mark the *copy point* in OVS. M_C specifies how far the OVS manager has cleared OVS for an immediate reusal, that is, no version images need be copied before that part of OVS can be reused. $(M_E - M_C) \bmod 2^n$ is then the amount of the immediately reuseable space.

The main problem in managing OVS is how to determine when a version image must be copied. It is clearly wasteful to examine every single version image in OVS as the copy mark M_C moves; most version images should not have to be copied, since the respective objects already will have newer versions. If this assumption does not hold, then this whole approach is wrong. Since the information whether a version image represents the current version or the token of an object is embedded only in the object header, the decision process concerning what and when to copy should start at the object headers.

In order to maintain the current versions of all objects in OVS, the objects should be ordered according to the time when their current versions were last written into VS. This approach is investigated in Section 3.1. In Section 3.2 the requirement that each object must have at least one version in OVS is relaxed; this leads to a much simpler implementation. In Section 3.3, management of OVS is reexamined and adjusted to an implementation with write-once storage devices. Section 3.4 looks at the implementation of OVS from the point of view of the number of device drives needed.

3.1 Current versions of all objects maintained in OVS

The general moving window scheme outlined earlier can be restated as follows. When more OVS

has to be cleared for reuse, the OVS manager will search for the object that has not had a new version image *written into OVS* for the longest time. The current version of this object will be referred to as the *oldest current version in OVS*. Let us call it X. Note that this is not necessarily the oldest current version in the *repository*, that is, a current version with the lowest creation time t_s, since that one may have been copied more recently. Let A_C be the VS address of M_C. Then $A_X >= A_C$, where A_X is the VS address of x, since by definition the portion of OVS "older" than the position of M_C has already been cleared. All version images older than X, that is, with addresses $A_{vi} < A_X$, can be deleted; this means that M_C can be moved to A_X (Figure 9). However, if $A_C = A_X$, it is necessary to *copy* X to the "newer" portion of OVS.

The first problem is how to find X. First let us assume that all objects in the repository are ordered according to the time the last version image of their current version was written into OVS, that is, according to the VS address of the last image of their current versions. The OVS manager will maintain a sorted list of objects; let it be called COPYLIST. COPYLIST in fact would contain just pointers to the object headers. The object with the oldest current version in OVS is on the top of the list. When a new version image for some object is written into OVS, the object should move to the bottom of COPYLIST. Unfortunately, the new version image may, and in most cases will, represent a token. Since a token may be later aborted, it is not appropriate to move the object to the bottom of the COPYLIST at the time the version image for the token is created. Now, assume that an object has a token, and its current version will become subject to being overwritten if M_C is moved. The current version must be copied, again because the token may be later aborted. But what should be the relative position of the object in the COPYLIST after the current version has been copied? Since the version image of the token precedes the new version image of the current version, the position of the object in the COPYLIST is determined by the token. If the token is later committed, nothing need be done. If the token is aborted, the object must be moved to a position in COPYLIST that corresponds to the location of the current version in OVS. If the current version has not been copied since the creation of the token, no action is necessary. Finally, if the fate of the token is still undecided when M_C reaches the respective version image, the token must be copied, or, more precisely, the representing version image must be copied. That is, the OVS manager must also look for the *oldest token in OVS*, as it clears OVS.

To summarize, an object is eligible to move in the COPYLIST only when:

1. its current version is copied or
2. its token is committed or
3. its token is aborted or
4. its token is copied.

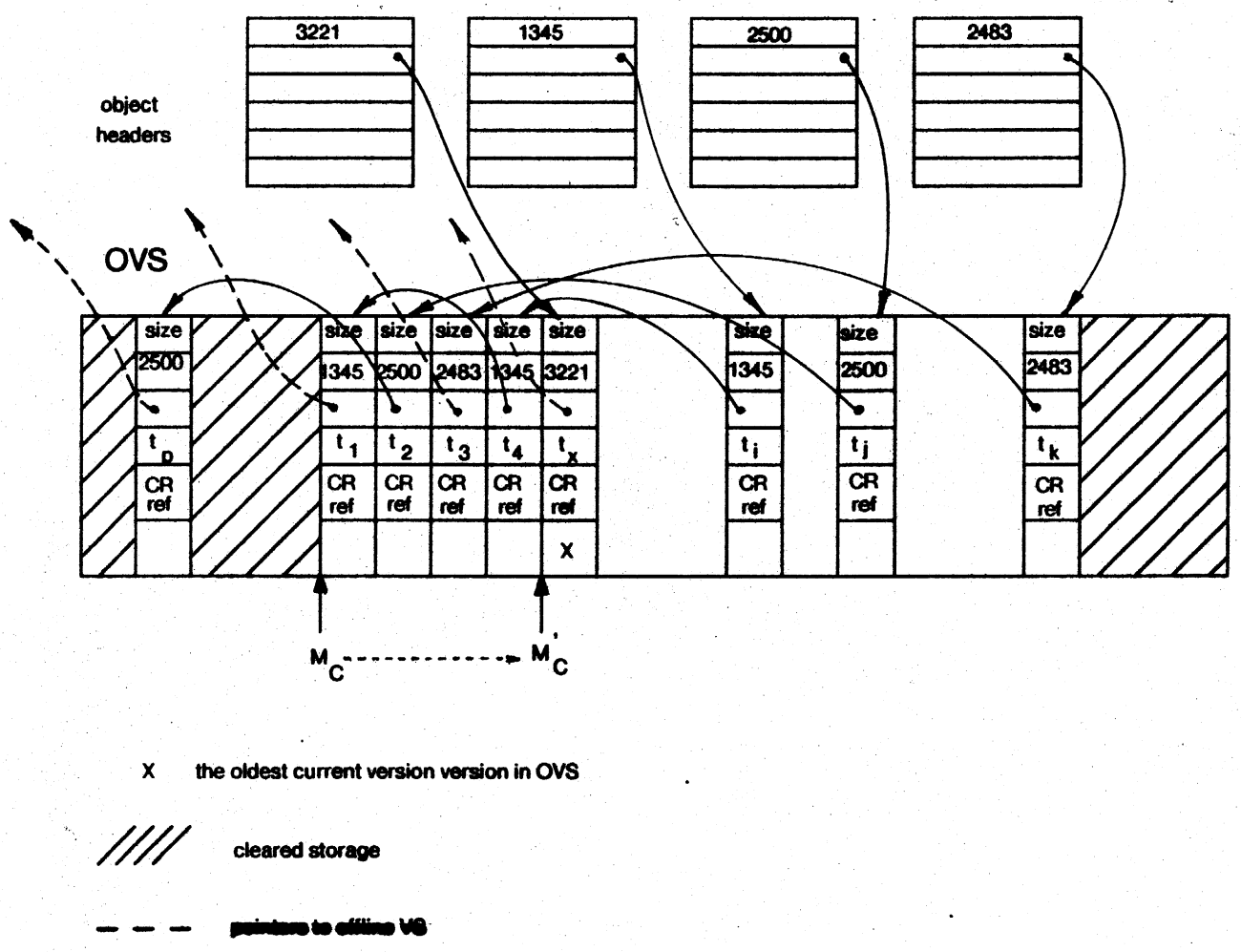


Figure 9: Release of OVS occupied by old versions.

Since objects 1345, 2500 and 2483 already have newer versions, the portion of OVS between M_C and M_C' can be released without having to copy any version image. Note that the version images in the cleared storage are still accessible.

Let A_{cv} and A_t be the current version and the token reference contained in the object header. Then Table 1 shows under what conditions the object does move in the COPYLIST. A graphical illustration for a simpler kind of COPYLIST will be found in Section 3.3, in Figure 11. If a nil reference (no token exists) is represented by a negative number, then to test for an existence of a token when the current version is copied, it is sufficient to test if $A_{cv} < A_t$. Thus, for any of the four kinds of events, the resulting position of the object in the COPYLIST is always determined by the greater of A_{cv} and A_t *prior* to that event.

Table 1: Management of COPYLIST

event: object is eligible to move in COPYLIST	condition: object is moved in COPYLIST	result: position of the object in COPYLIST determined by
current version is copied	object has a token	A_t
token is committed	$A_{cv} < A_t$	A_t
token is aborted	$A_t < A_{cv}$	A_{cv}
token is copied	$At < A_{cv}$	A_{cv}

The overhead of clearing OVS for reuse should be distributed over time. The OVS manager can be implemented as a demon process that runs concurrently with the processes that create and commit tokens. To maintain the amount of cleared OVS within specified limits, the demon is run when $\langle M_E, M_C \rangle$ drops below the lower limit, and it goes to sleep when it has cleared enough space as determined by the upper limit. A large amount of OVS may be cleared in just one step, by jumping to the oldest current version or token in OVS. Thus it is quite possible that the amount of cleared space far exceeds the upper limit; many new version images may be created before it is necessary to run the demon again. The demon should not copy the oldest current version or token unless more clear space is necessary. If the demon stops at such a version, it may be that the next time it is run, the respective object will by then have a newer version, and thus no copying is needed. On the other hand, the demon may run into a situation when it must copy almost every version; this, of course, will not free any space. If this is just a local phenomenon, that is, the images of the current versions of some objects became clustered, the demon will eventually release enough space (unless none of these objects is ever updated again). Otherwise, it might be an

indication that the system is saturated.

This scheme could be finely tuned to operate with a very small amount of cleared storage. This in turn means that multiple copies of a version or a token exist in OVS for only a very brief time interval; thus it is possible to achieve very good OVS utilization, in terms of the *useful* information stored. However, even if the entire COPYLIST could be kept in primary memory, the overhead of re-sorting the COPYLIST may be significant. This problem can be eliminated if a different policy for keeping current versions in OVS is adopted, as discussed in the following sections.

3.2 Most recently used current versions maintained in OVS

In the schemes described in the preceeding section, the OVS manager must maintain at least the current version of every object in OVS. This means that if T is the average time it takes to cycle through OVS, then the current version of an object that has not been updated for n·T will be copied n times. This represents a performance penalty that may be unnecessary, since some objects will not even be *read* for long periods of time, yet the OVS manager will keep copying them in OVS. To give a more specific example, in a reasonably busy repository, a 300 Mbyte disk used as OVS may fill up in less than a day. It is highly likely that many objects in the repository will be dormant for many days, weeks, or even months; copying them every day would be quite wasteful.

The OVS management policy will be relaxed such that only those objects that had their current version actually accessed (read, or had a new version created) since t_c - T will be kept in OVS, where T is again the time it takes to fill up OVS. With this relaxation, copying of dormant objects is avoided. In addition, the copying process can be simplified. In particular:

i. it is not necessary to sort objects to keep track of which objects must have their current versions or tokens copied as the OVS manager works on clearing OVS; the current versions and tokens can be copied as they are accessed,

ii. no special demon process is necessary to clear OVS; clearing of OVS is automatically distributed over time.

Let M_C specify again the copy point. If $M_C = M_E$, (i.e., $A_E - A_C = 2^n$), an object will maintain its current version in OVS only if a new version is created at least every T time units. If $A_E - A_C < 2^n$, the version images of the current version and tokens that are in this portion of OVS will be copied in OVS when read. The bigger the distance between M_C and M_E, the less frequently must the current versions be read to remain in OVS. An additional optimization is possible: if the