

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.

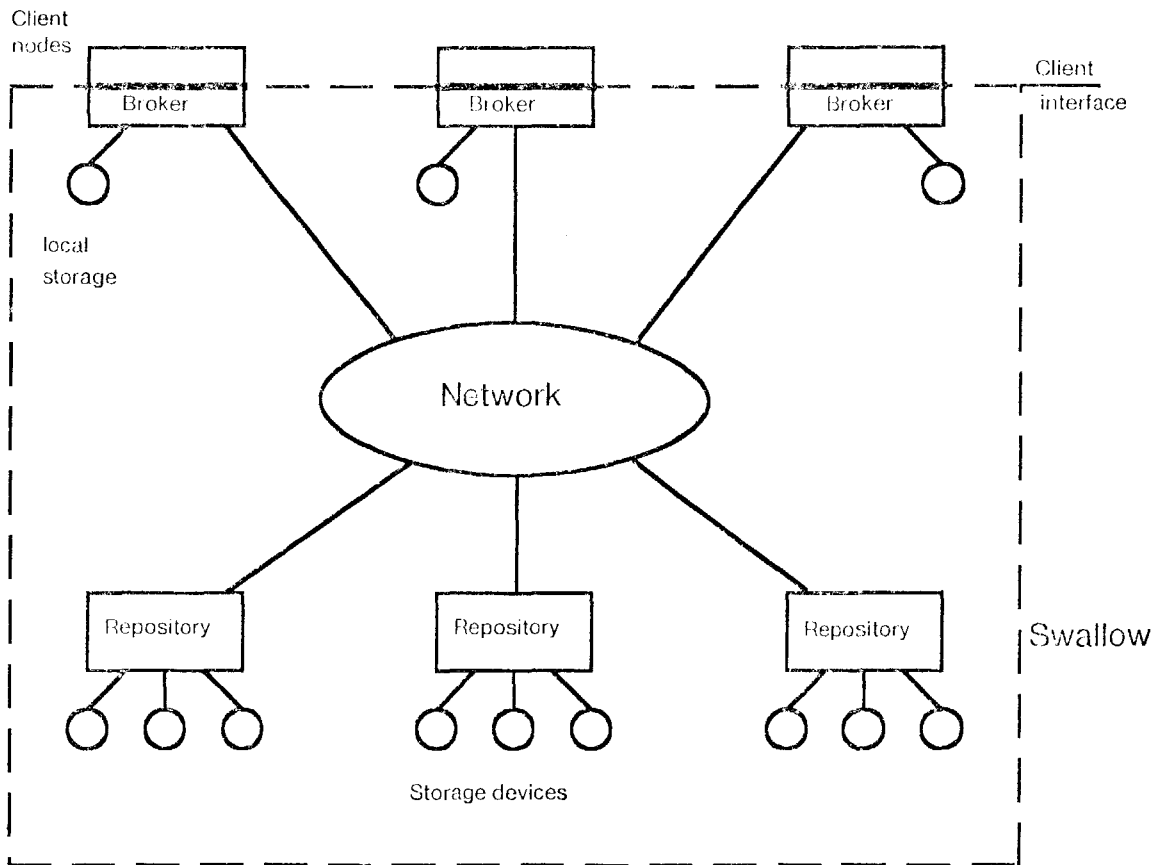


Figure 1: Structure of the SWALLOW system

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 Section 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 "education" have to be accommodated 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

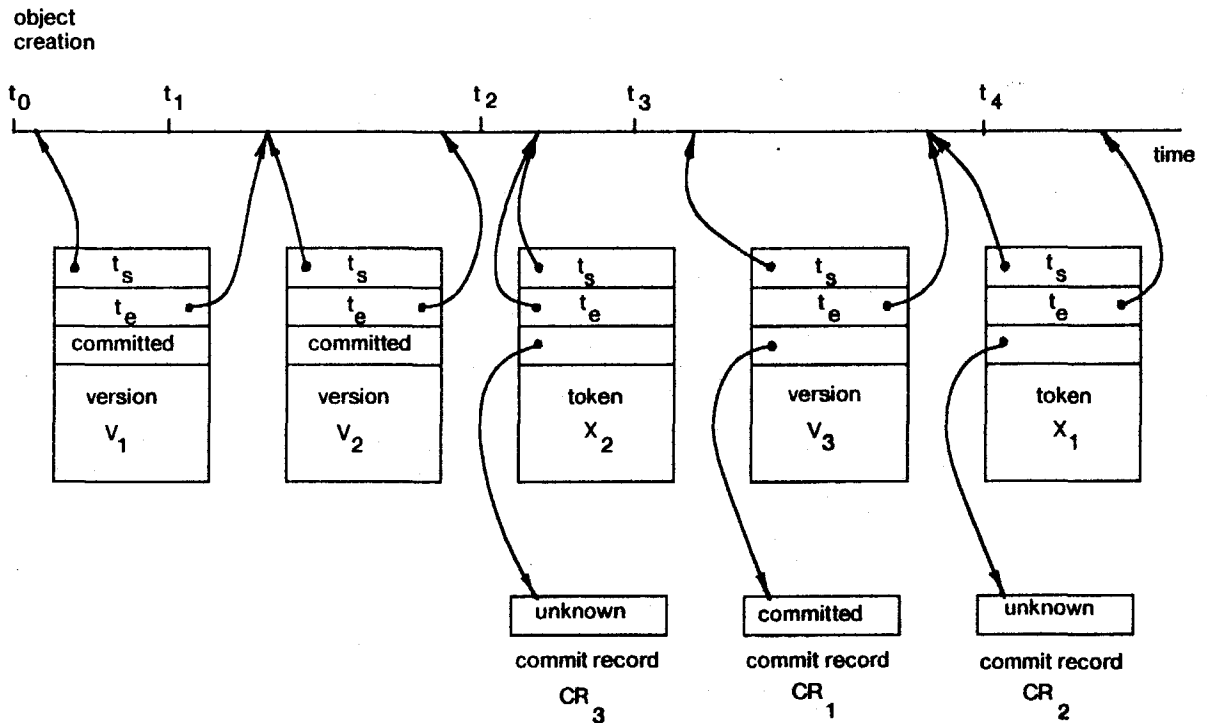
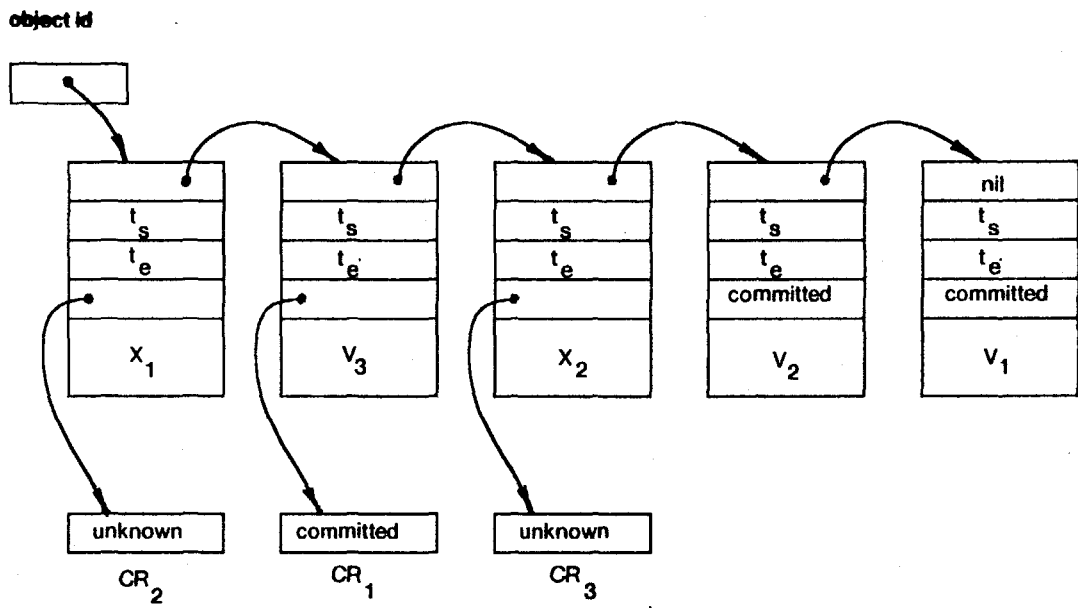
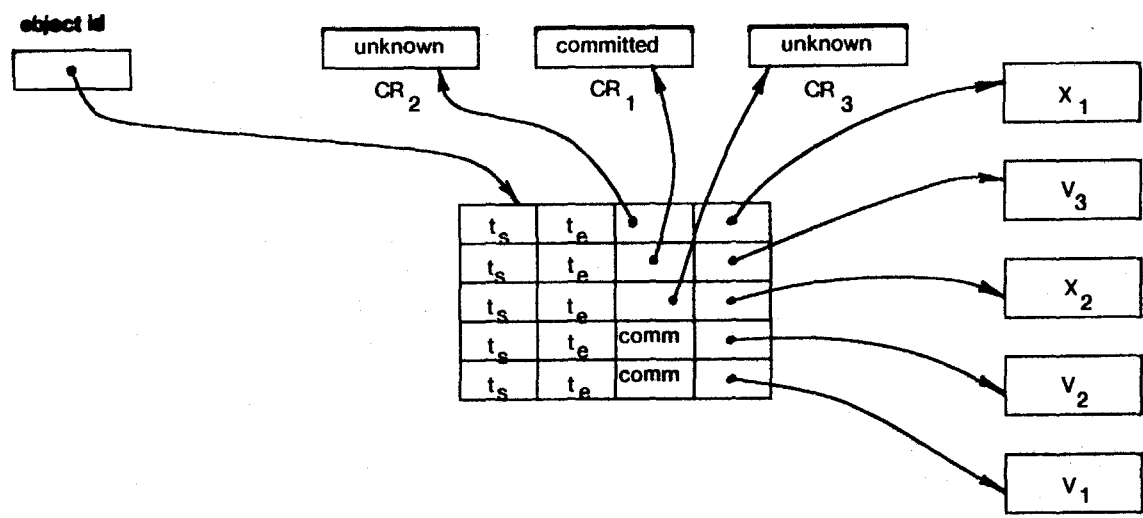


Figure 2: An example of an object history.

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 possibility 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

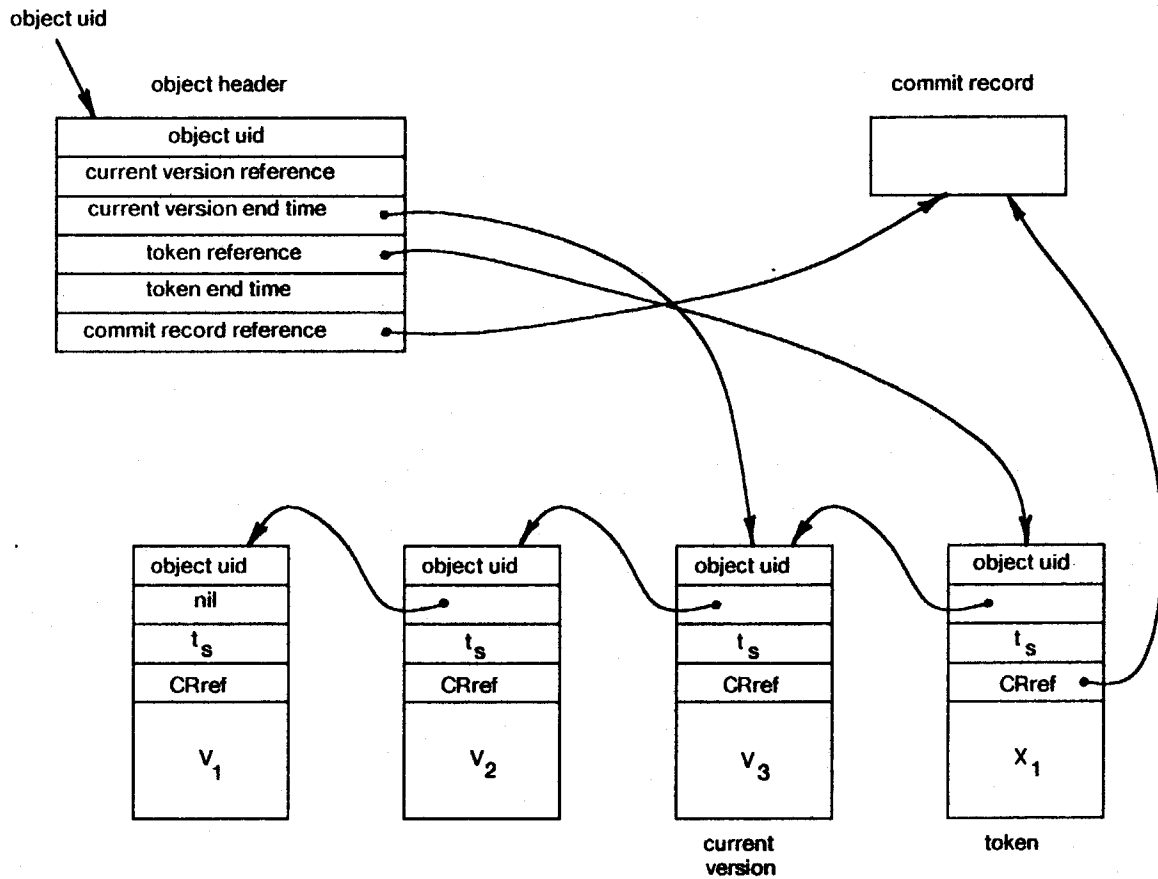


Figure 4: Representation of the object history for the modified object model; it is not possible to create token X₂ in this model.

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

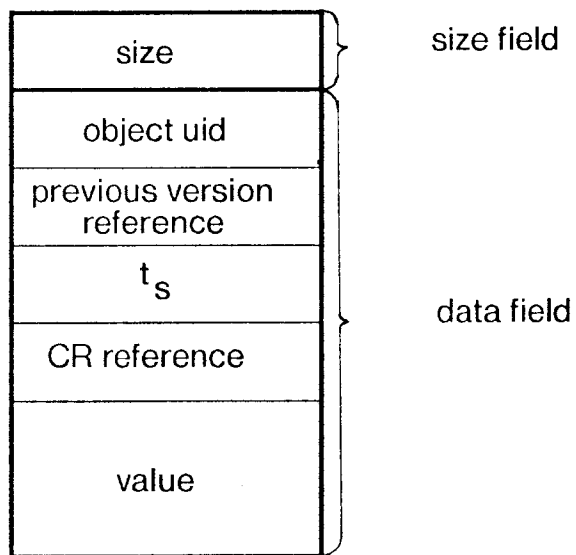


Figure 5: VS image representing a version image.

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 $(A_E - A_{vi}) \leq 2^n$, 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 $(t_c - T, t_c)$ 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.

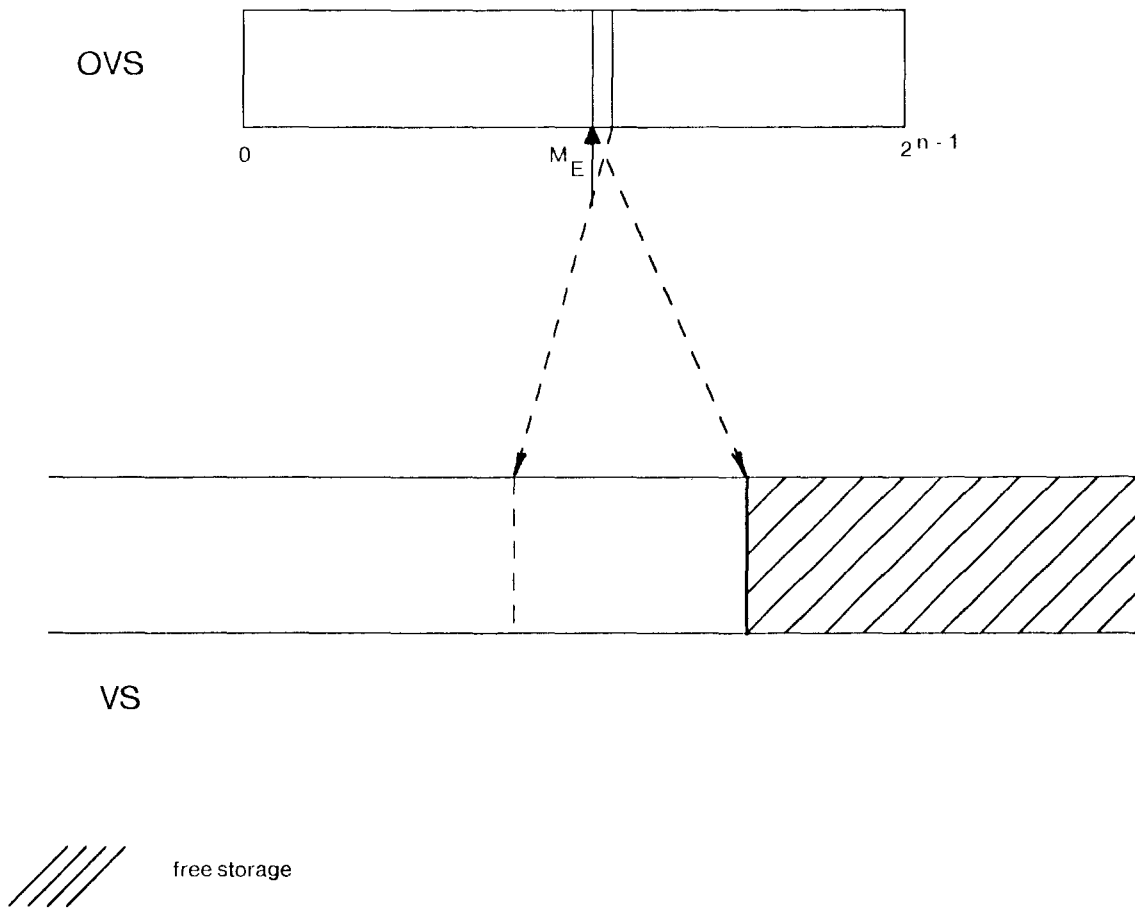


Figure 6: OVS as a circular buffer.
 New version images are appended at the mark M_E .

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

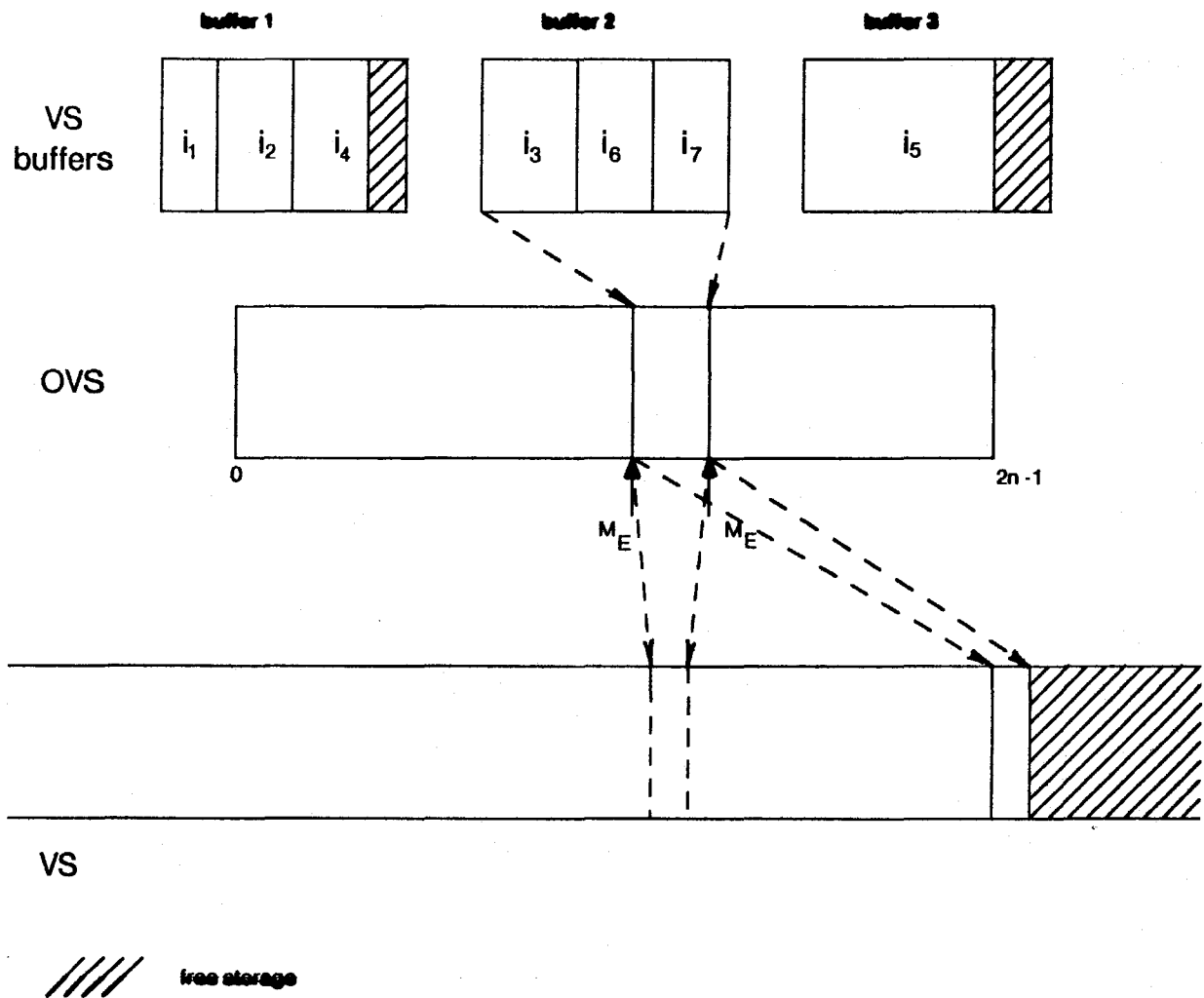


Figure 7: Writing VS image into VS.

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

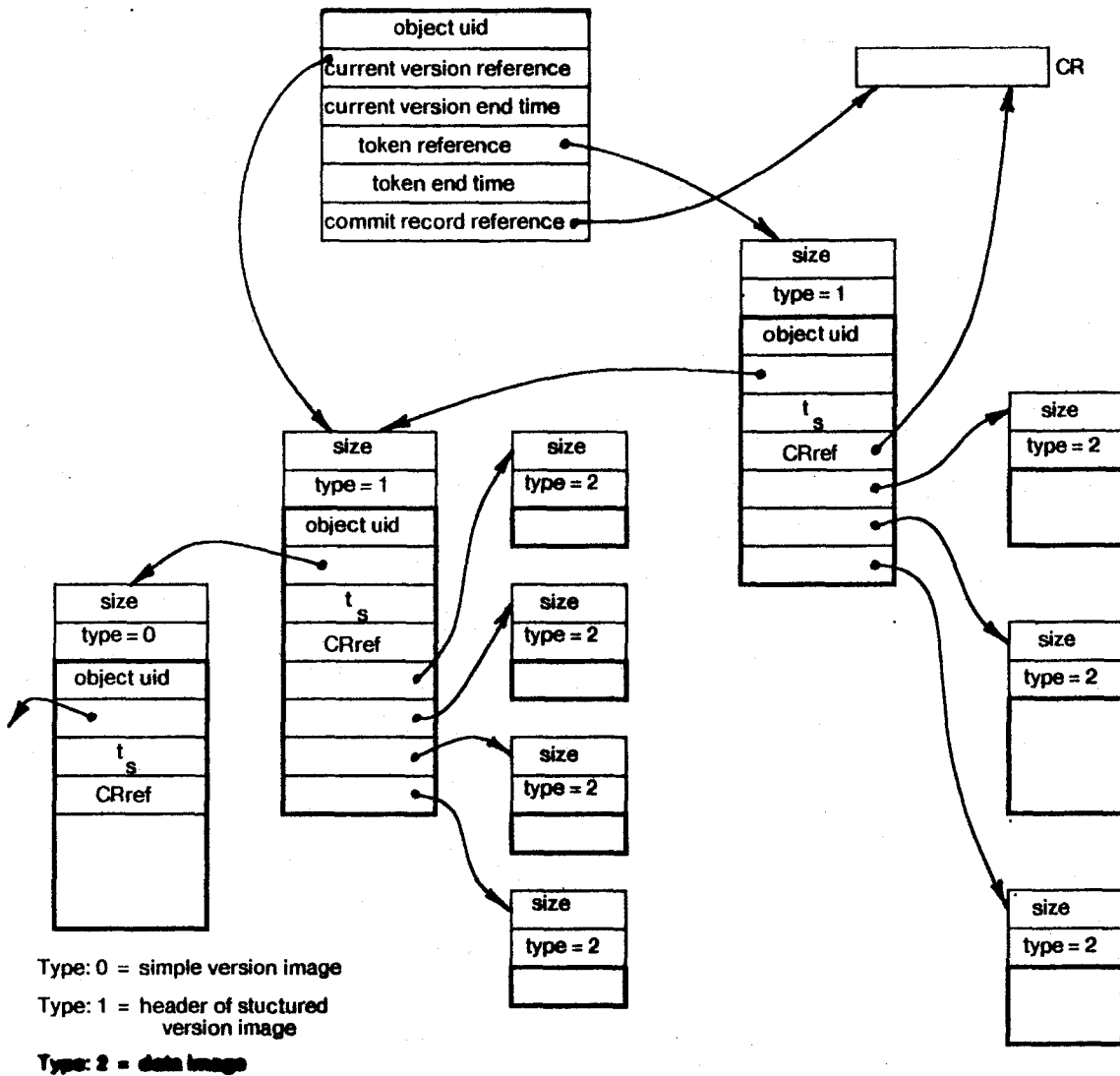


Figure 8: Representation of large object versions.

The current version and the token are represented 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 explicitly 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_F).

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_c , since that one may have been copied more recently. Let A_C be the VS address of M_C . Then $A_X \geq A_C$, 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.

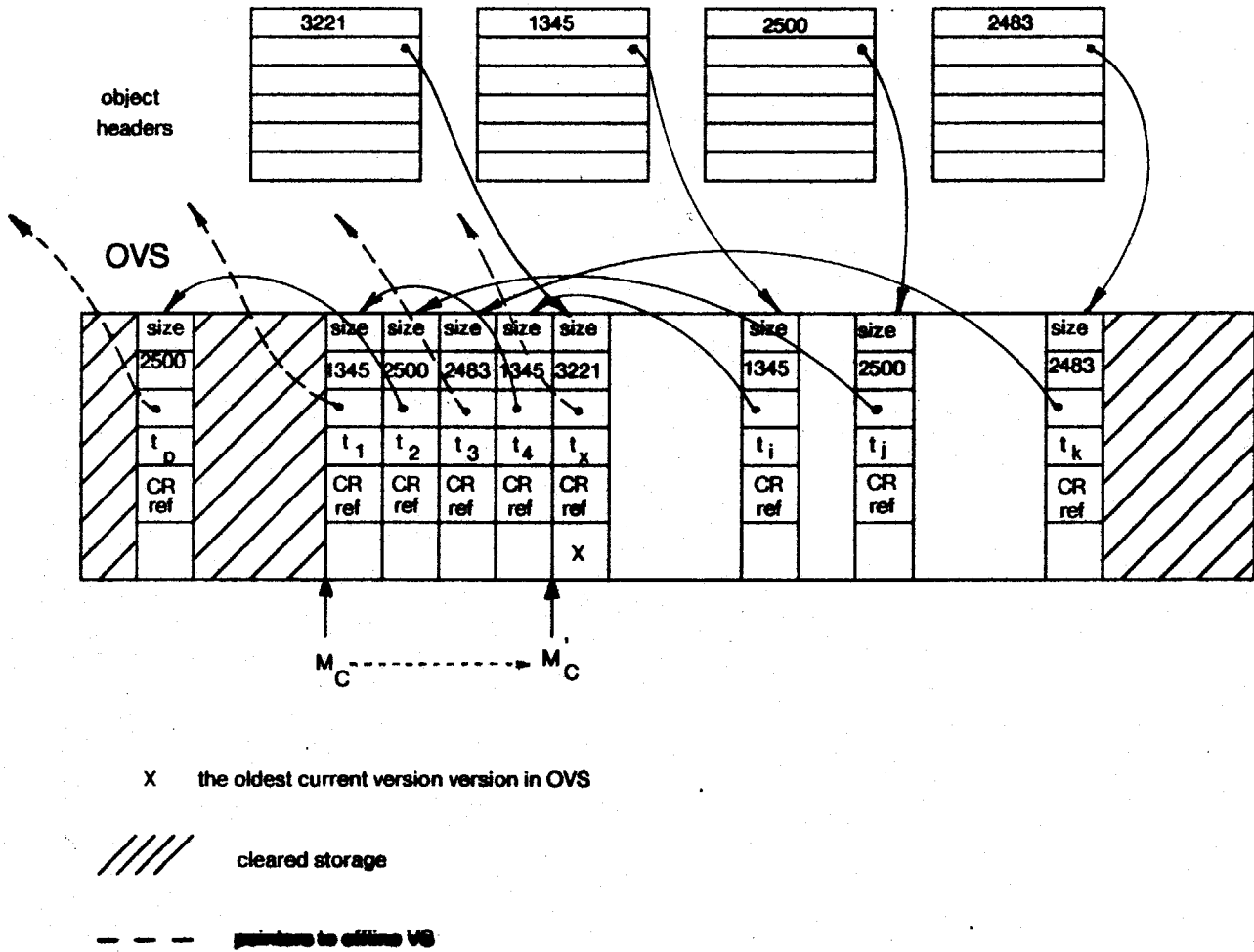


Figure 9: Release of OVS occupied by old versions.

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 | $A_t < A_{cv}$ | 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 preceding 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 $n \cdot T$ 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 version image to be copied is close to $M_E \pmod{2^n}$, then if one is willing to sacrifice the

intervening storage, such a version image does not have to be copied, since the storage between the version image and M_E is in a sense already "cleared."

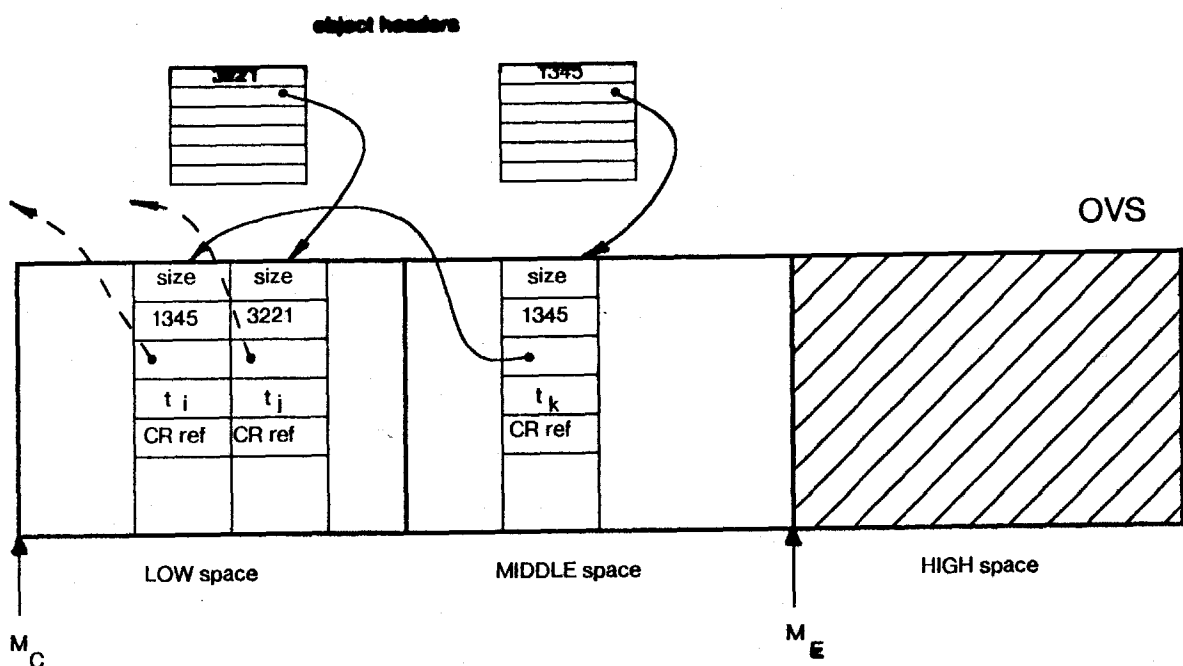
A version (token) reference is resolved as before: if $A_E - A_{vi} \leq 2^n$, the representing version image is in OVS. In addition, when a version image of a current version or a token is read, then if $A_{vi} < A_C$, a copy of this version image will be created in OVS. To improve the chances that the current version is in OVS, at the time a token is committed, that version image should also be copied, if its address is lower than A_C . If a current version is not represented in OVS, the appropriate version image is retrieved from the offline VS and written at the current end of OVS. Thus current versions of objects that have not been read for a long time can be reinstalled in OVS with this simple mechanism. Finally, it would be possible to provide a simple "refresh" process for those objects that should always stay online. This process would periodically read such objects to force their copying in OVS.

3.3 Adapting OVS management to an implementation with write-once devices.

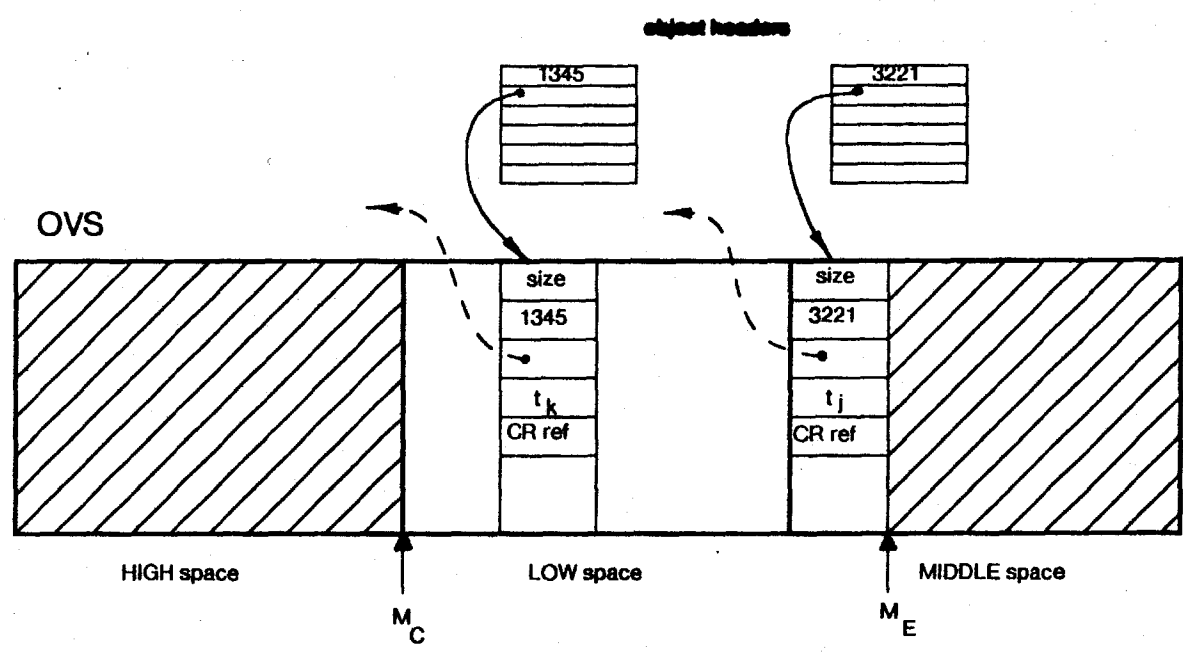
The two schemes presented in the preceding sections assumed that OVS is implemented with reusable physical storage, that is, that new and copied version images simply overwrite those with addresses lower than $A_E - 2^n$. This means, however, that the overwritten images must be preserved at some other device that is a part of the permanent VS. Alternatively, the storage devices used in OVS can be the actual VS. When a device is filled up, it is removed and stored offline, and a fresh device replaces it. Since the devices are written only once, VS can be implemented entirely with optical disks. Unfortunately, the fine tuning, which is the major attraction of the schemes presented so far cannot be achieved when OVS is implemented in this way since the OVS space can be "reused" only by replacing an entire device. Rather, OVS should be viewed as being divided into fixed-sized partitions, where each partition corresponds to one physical device.

To implement the same policy as the one used in Section 3.1, when the current versions of all objects are to be kept in OVS, it is necessary to have the minimum of three partitions. These partitions, called here LOW space, MIDDLE space, and HIGH space do not have to be of equal size, but for simplicity, let us assume that they are. Again, OVS will be managed as a circular buffer (Figure 10). When the MIDDLE space becomes full, all the version images in the LOW space will be purged and the spaces will be reassigned such that:

| | | |
|--------|------|--------|
| MIDDLE | ---- | LOW |
| HIGH | ---- | MIDDLE |
| LOW | ---- | HIGH |



a) The middle space is full; it is necessary to clear the LOW space



b) The current version of object 3221 was copied, and the spaces were reassigned


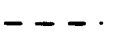
 cleared storage
 pointers to offline storage

Figure 10: Management of OVS: tripartite scheme.

M_E marks again the current end of VS in OVS; M_E falls into either the MIDDLE or HIGH space. M_C points always to the beginning of the LOW space; it moves only when the spaces are reassigned. To ensure that each existing object will retain an image of the current version in OVS, it is necessary to find all objects that have their current versions in the LOW space. Copies of these versions will be created in the NEW space, which is free at the beginning of the purge of the LOW space.

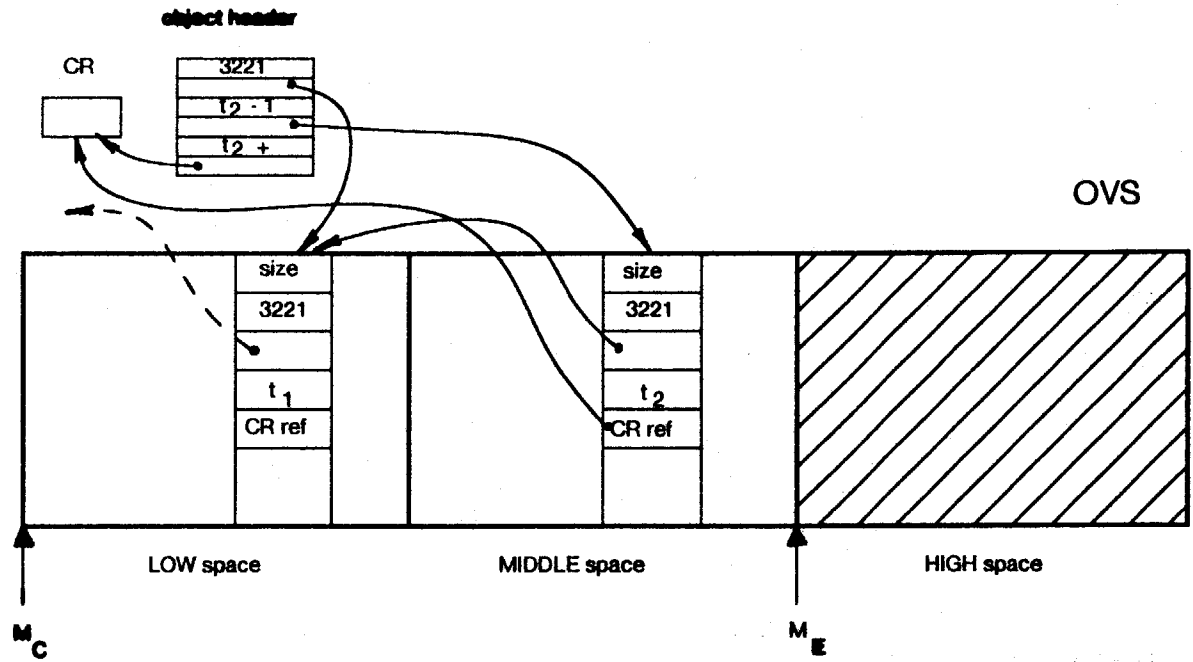
This scheme reduces the sorting problem into a tripartite sort. An object is logically mapped into the space which is the older of: the space that contains the last version image that represents the current version, and the space that contains the last version image that represents the token, if any. The conditions under which an object moves into a higher space are similar to those for the previous scheme. Let S_{cv} and S_t be the OVS spaces that correspond to the addresses A_{cv} and A_t at a given moment. An object is then mapped as specified by Table 2, where the ordering on the spaces is LOW<MIDDLE<HIGH. The possible changes in the logical mapping of an object into the three spaces are illustrated in Figure 11.

Table 2: Mapping to OVS spaces

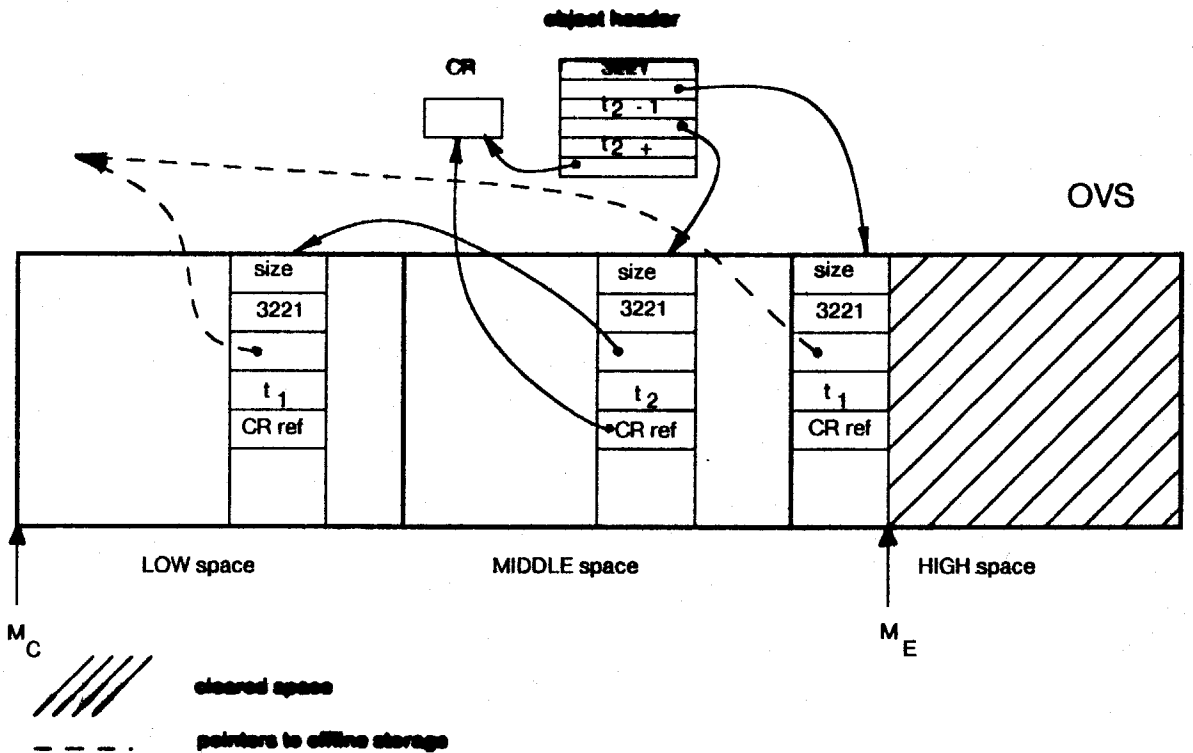
| event: object is eligible to move to a higher space | condition: object is moved to a higher space | result: mapping of the object to OVS spaces determined by |
|---|--|--|
| current version is copied | object has a token and $S_{cv} < S_t$ | S_t |
| token is committed | $S_{cv} < S_t$ | S_t |
| token is aborted | $S_t < S_{cv}$ | S_{cv} |
| token is copied | $S_t < S_{cv}$ | S_{cv} |

This OVS management scheme is not limited to an implementation with write-once devices. It is possible to take advantage of the simplified ordering on objects required by this scheme even if the physical OVS device is reuseable.

If OVS is implemented with write-once devices, then although the physical storage capacity of OVS is 2^n words, OVS does *not* contain the most recent 2^n words of VS as before. This is because when the LOW space is reassigned as the HIGH space, the physical device for this part of the OVS must be replaced with a fresh one and thus the corresponding OVS address space does not contain valid version images. In fact, on average, 50 percent of OVS will be empty. This has to be reflected in the resolution of version references. Let us use another mark, M_L , to identify the oldest valid VS image in OVS: M_L will point to the beginning of the LOW space. In this scheme, M_L is the same as

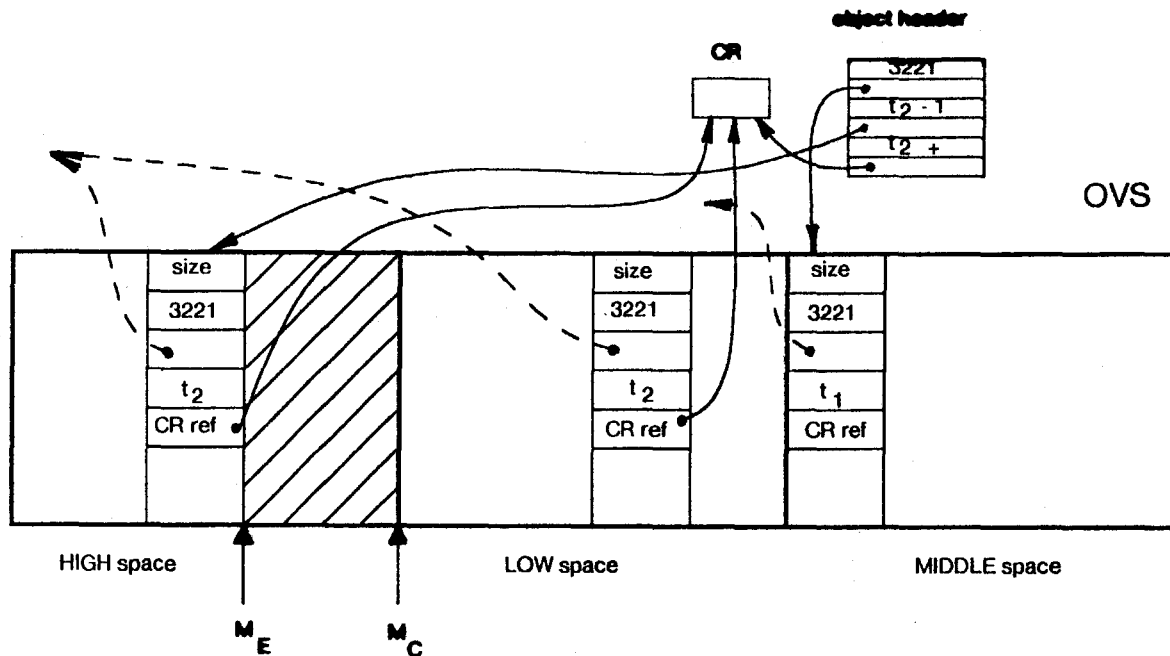


- a. Situation just prior to the beginning of a purge of the LOW space. Object 3221 has its current version in the LOW space and a token in the MIDDLE space.

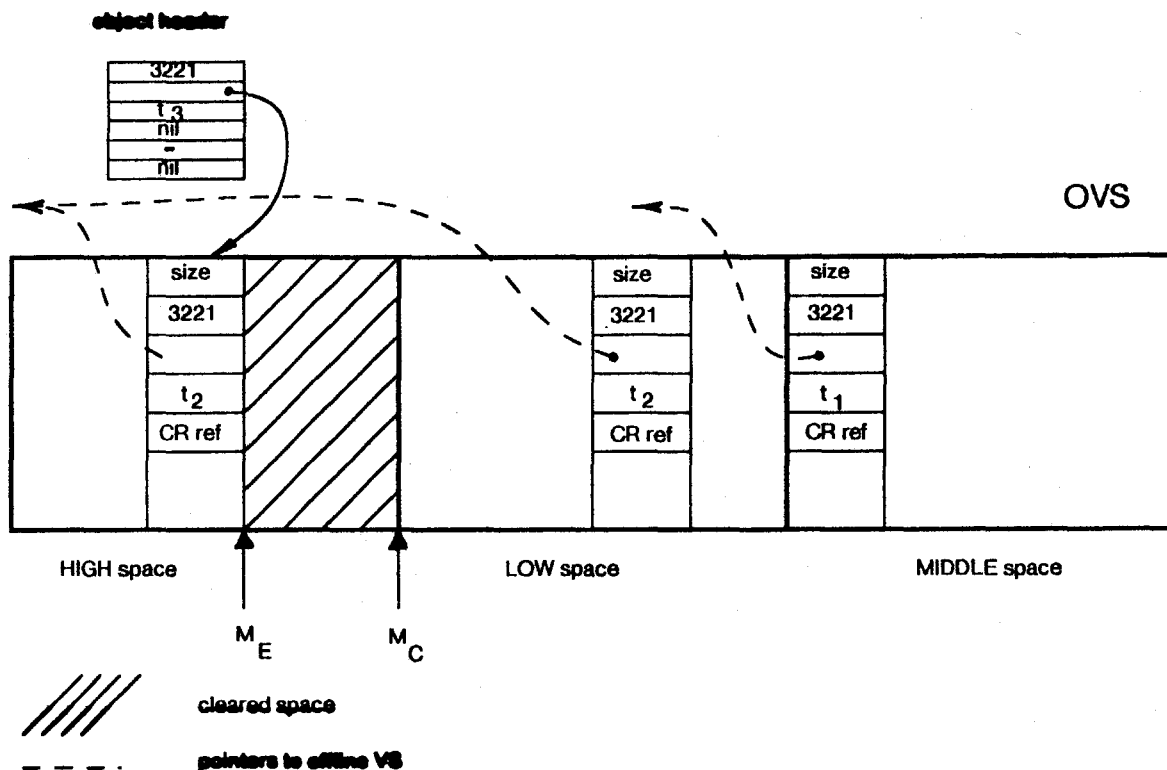


- b. Situation just after the current version has been copied into the HIGH space.

Figure 11: Resolving the token problem.

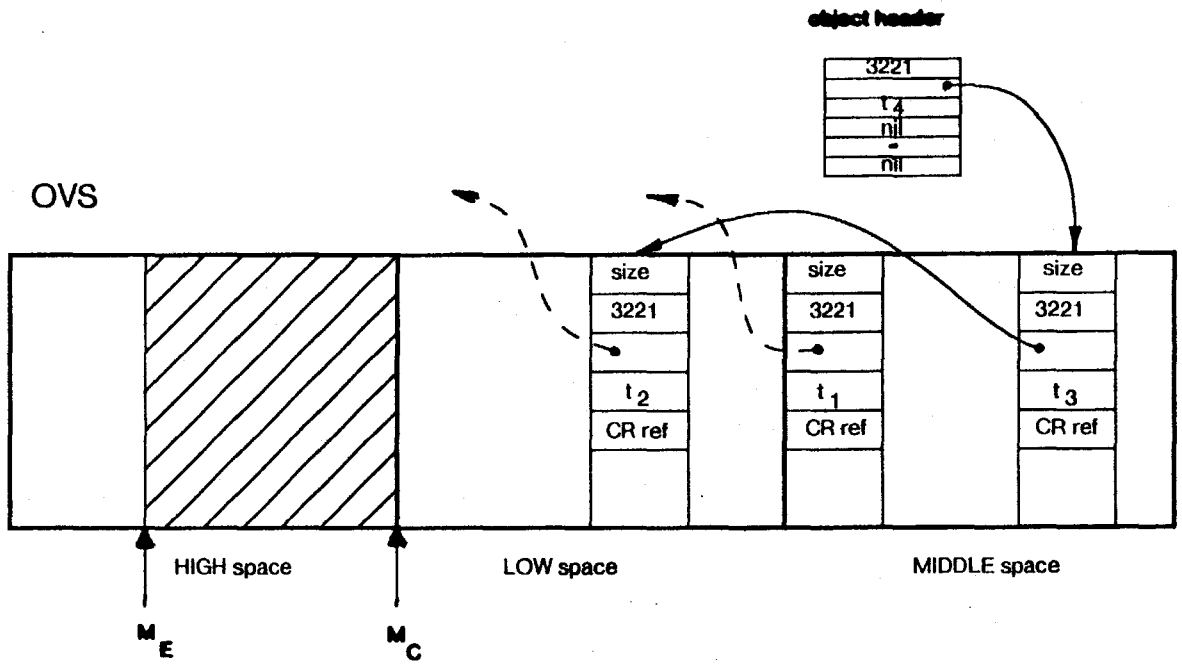


c. Situation during the next purge. The token of object 3221 has not yet been committed, so it has to be copied into the HIGH space.

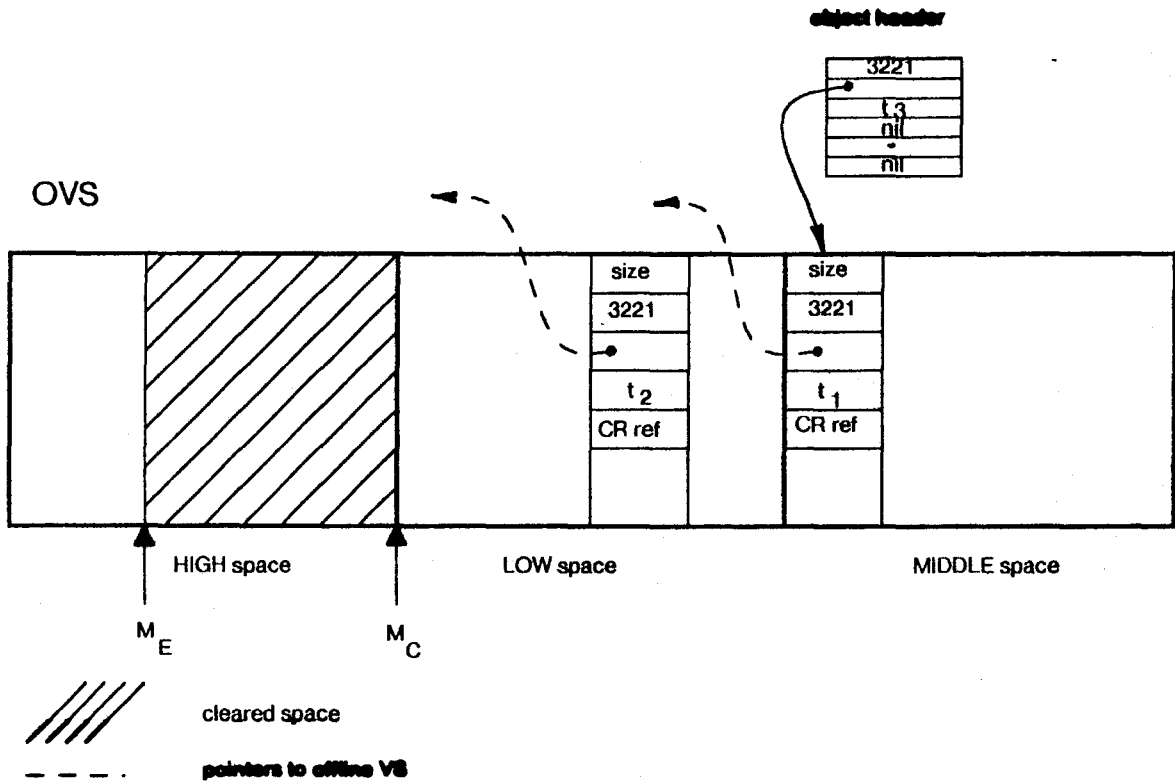


d. Situation during the next purge. The token was committed, the reference to the previously copied version is lost. The former token is the only version of object 3221 in OVS.

Figure 11: Resolving the token problem. (Cont.)



- e. Situation during the next purge. The former token was committed, but a new version was already created (and committed). Thus the former token does not have to be copied.



- f. Situation during the next purge. The token was aborted.

Figure 11: Resolving the token problem. (Cont.)

M_C , but it will be different for the other copy policy, as discussed below. A version image is in OVS only if $A_L \leq A_{vi} < A_E$.

If the management policy is to maintain in OVS only those current versions that have been actually used in the recent past, it is sufficient to divide OVS into two partitions, LOW space and HIGH space. When the current version of an object is read, the address of that version image, A_{CV} , is used to determine whether this image is in the LOW or the HIGH space. If it is in the LOW space, it is copied into the HIGH space. New versions (tokens) are created always in the HIGH space, that is, M_E maps always into the HIGH space. The copy mark M_C must point to the beginning of the HIGH space and the mark M_L to the beginning of the LOW space. Again, if $A_{vi} \geq A_L$, the version image is in OVS. If a version image represents a current version, then if $A_{vi} < A_C$, the version image will be copied.

These schemes resemble real-time copying garbage collection algorithms. However, in the context of garbage collection, objects that are not copied into the HIGH space are irretrievably lost. Thus, any object to which there exists a valid reference must be copied. This would mean copying the entire histories of all objects in the repository. Thus although the bipartite (and tripartite) OVS model and copying of version images was borrowed from the work on garbage collection, the implementation details are significantly different. A copying "garbage collector" for large paged virtual memory that works in a similar way as the schemes presented here was recently proposed for the LISP machine, but the details have not been worked out yet.

3.4 Online support for VS

As already discussed in the previous section, the physical support of OVS may be reusable storage devices that are maintained permanently online, or just "reusable" device drives, where the storage devices are replaced with fresh ones as they become full. The latter approach has the advantage that the entire VS can be implemented exclusively with optical disks. To implement the schemes presented in Section 3.3, one device drive is needed for each OVS space. When the LOW space is filled up, the device that contains the LOW space is replaced with a fresh device, and the replaced device becomes part of the offline version storage. In particular, if the policy that only those current versions and tokens actually accessed are to be maintained in OVS is adopted, two drives are needed; an implementation of OVS that uses this management scheme will be examined in more detail.

As said earlier, the entire VS should be duplicated for stability. However, since version images are created only in one space at any time, only one additional device drive is necessary, to duplicate this space. This duplicate is removed when that space is filled up, and replaced with a fresh device that

is assigned to the next space.

Finally, if it is necessary to read a version that is not available in OVS, the respective device has to be found and brought online. This requires yet another drive. Figure 12 illustrates the implementation with the minimum number of device drives.

To avoid long delays due to the manual replacement of the storage devices, it is necessary to add one more drive. Two drives are used for the LOW and HIGH spaces as before, and two drives are assigned to VS backup, but the actual assignment of the drives changes as illustrated in Figure 13. Each OVS space is divided into two equal parts, and each part is mapped into a *different* backup device. When the HIGH space is filled halfway, the backup device is full and the backup is redirected to the other backup device. The full backup device is replaced with a fresh device, and once the HIGH space is full, this device will become the new HIGH space; thus the drive is reassigned from the backup function to the "current VS" status. Basically, at any time, the assignment of the drives is:

| | | |
|-------------|-------------------------|---------------------|
| current VS: | LOW space | $D_i \bmod 4$ |
| | HIGH space | $D_{(i+1) \bmod 4}$ |
| backup: | low part of HIGH space | $D_{(i+2) \bmod 4}$ |
| | high part of HIGH space | $D_{(i+3) \bmod 4}$ |

when the HIGH space fills up, $i \leftarrow i+1$

The same scheme can be implemented with a reusable device such as a conventional magnetic disk in the following way. Both partitions, the LOW and the HIGH space, can be mapped to the same device. As the spaces are switched, the LOW space is simply overwritten. Of course, it is necessary to ensure that the version images that will be overwritten will not be lost from VS. If we assume that all images are written twice for stability, the second copy could be made in nonreusable storage, thus guaranteeing that when the OVS device is reused, there does exist another copy of each overwritten version image in VS. However, this does not ensure *future* stability, since once a version image is overwritten in OVS, only *one* copy will continue to exist. Thus if it is required that the copies of all images are maintained in VS, then either every image must be written *three* times when it is created, or, a copy of the LOW space must be made in nonreusable storage before the LOW space is reused. The latter looks like a better solution. In particular, as a fresh HIGH space begins to fill up, the LOW space can be copied onto another device (Figure 14).

The minimum number of device drives needed is the same as in the implementation that uses optical disks only. Although OVS can be put now on a single device, *two* devices are needed for

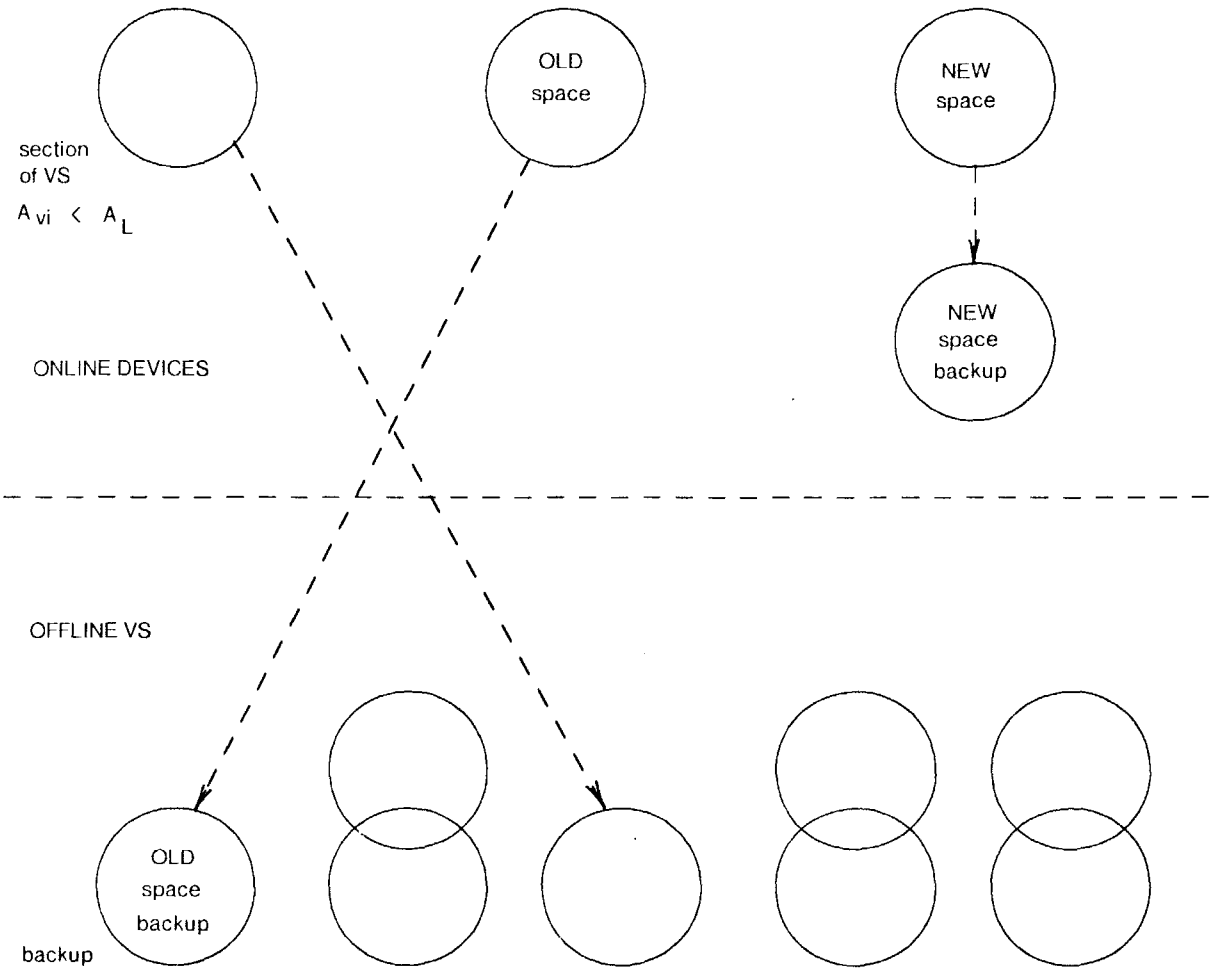
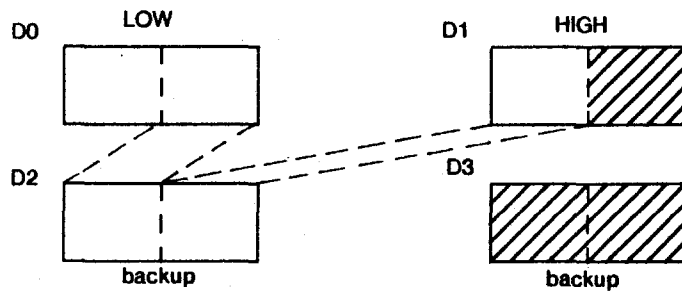
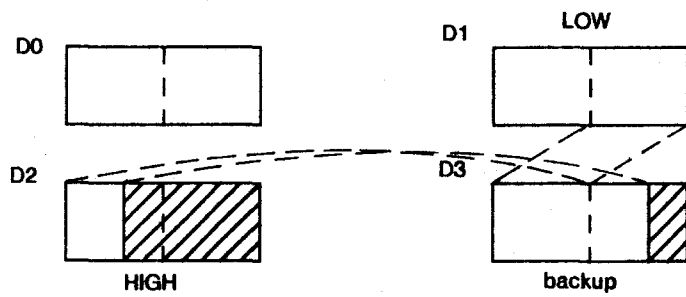


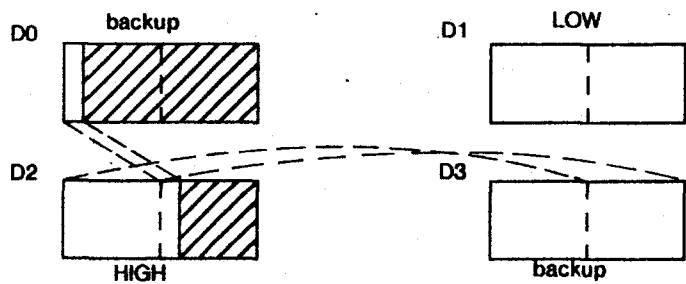
Figure 12: Implementing OVS/VS entirely with optical disks.



a) Device at the driver D2 is full and can be replaced;
backup switches to D3



b) Device at the driver D1 is full; D2 becomes the
HIGH space. D0 can be replaced.



c) D0 becomes the next backup

Figure 13: Implementation of OVS/VS with optical disks; management of device drives.

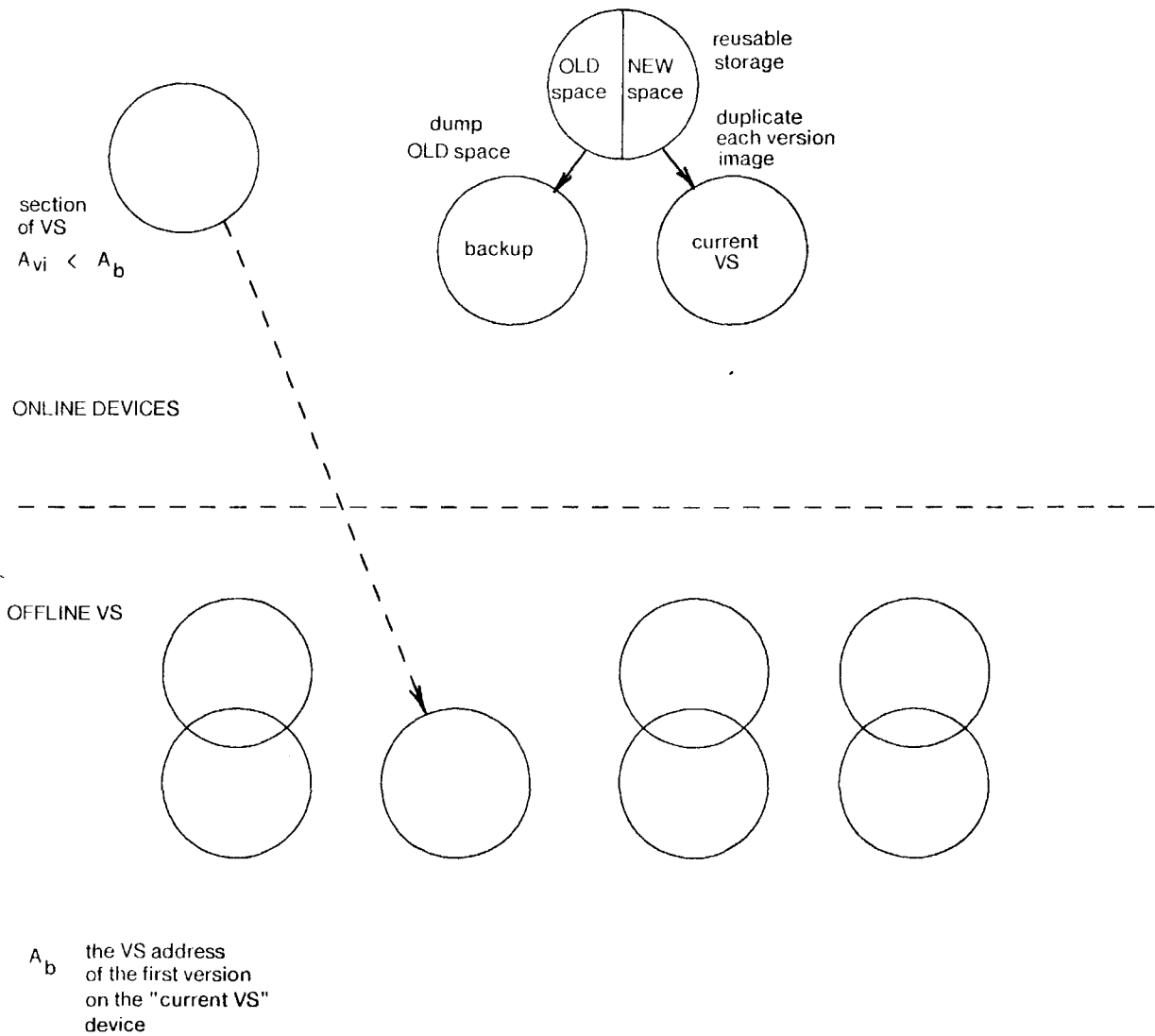


Figure 14: Implementation of OVS/VS with a reusable OVS device.

backup. Finally, as before, an additional drive is needed to bring selected pieces of VS online when a reference to an old version that is not in OVS is made.

The need to replace the backup device for the HIGH space creates again the problem of long delays. However, this problem can be resolved without an additional drive. If a "dump" of the LOW space to the backup device can be finished sufficiently fast, the backup device can be removed before the HIGH space fills up, and replaced with a fresh device which will become the next "current VS" device. When the "current VS" device is filled, the VS manager switches to the other drive which already has a fresh device mounted. Now a fresh backup device needs to be mounted on the other drive; it should be possible to perform this operation *and* dump the current LOW space before the HIGH space fills up again. Figure 15 illustrates the management of the device drives where the VS devices are twice the size of the reusable OVS device. To start this duplicated VS system, the first backup device will be partially empty, corresponding to the first dump of the LOW space, which is initially empty.

Although it is possible to save one device drive compared to the implementation that uses only nonreusable devices, the performance penalties for an interleaving of the normal operation of OVS with the dump of the LOW space could be severe. The only real advantage of using a reusable device for OVS is that it is possible to apply the more flexible moving window management scheme.

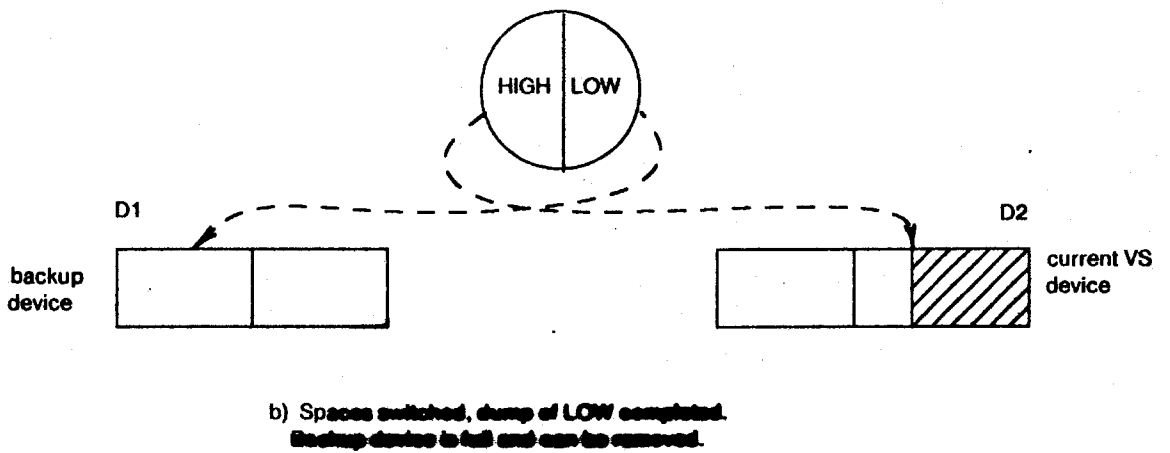
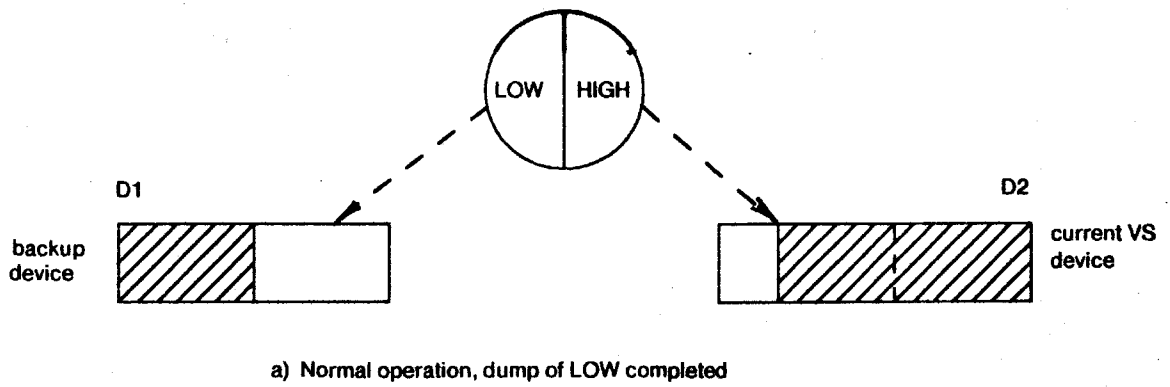
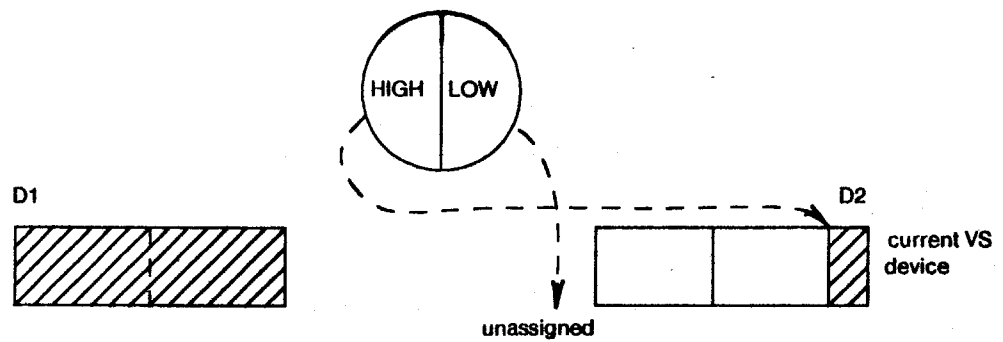
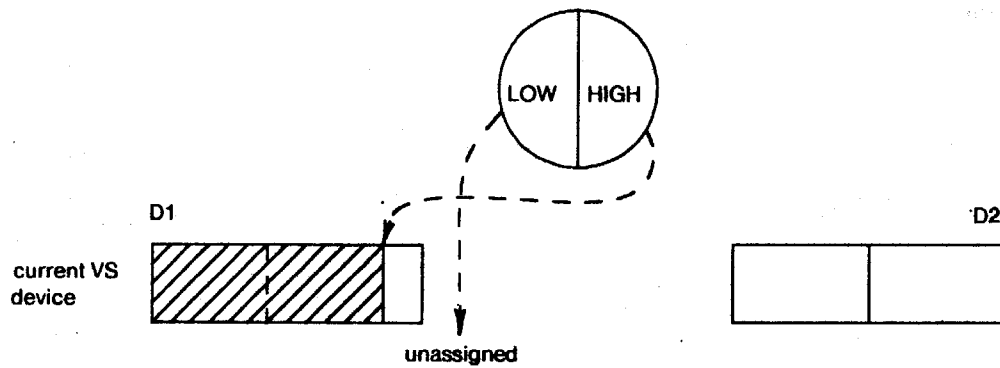


Figure 15: Implementation of OVS/VS with a reusable device; management of device drives.



c) Fresh device mounted on the driver D1;
momentarily, there is no backup device



d) Current VS device is full, the VS manager switched
to the other driver; no backup device yet

Figure 15: Implementation of OVS/VS with a reusable device;
management of device drives. (Cont.)

4. Management of objects

An *object* in the DDSS repository is an abstract type. The operations allowed on objects are:

- create** (pseudo-time, commit-record-id)
- read** (object-id, pseudo-time, commit-record-id)
- create-token** (object-id, pseudo-time, commit-record-id)
- commit-token** (object-id, commit-record-id)
- abort-token** (object-id, commit-record-id)
- delete** (object-id, pseudo-time, commit-record-id)

These operations are necessary to support the model described in Section 1. All of these operations are performed as part of some atomic action. A token can be read only by the atomic action that created it. Similarly, until the creation of an object is committed, only the atomic action that created the object should be allowed to create a token for that object. The commit record reference field in the object header can be used also for this purpose. When an object is created, this field will contain a reference to the commit record of the possibility for the creation; if a token is created later under the same possibility, the reference does not change. When the possibility is committed, this reference will be set to nil, regardless of whether the object has a token. Then a token can be created only if: the commit record reference in the object header is either nil or is the same as the commit record reference specified in the create-token request, *and* the object does not already have a version for the specified pseudo-time.

In addition to the *external* operations listed above, operations **copy-cv** (copy current version) and **copy-token** are needed for OVS management, but these are only *internal* operations, available solely to the object manager. Both the external and the internal operations must start at the object header.

Objects in the repository have identifiers that are unique both in space and time; all requests to perform operations on existing objects must include the uid of the desired object. The repository must map the object uid into a physical address of the object header. The most straightforward way is to have an object directory; this issue will be discussed in Section 4.3.

Since the object headers play such an important role, they should be stored in stable storage. However, the object header is updated twice for each update of the object (create-token and commit/abort token), and may be updated when the current version or token is read (extend the end time). Finally, the object header is updated when the version image of the current version is copied. The additional disk write for each update would represent a large overhead. Further, object headers should be updated in place, otherwise it would be also necessary to change the map that

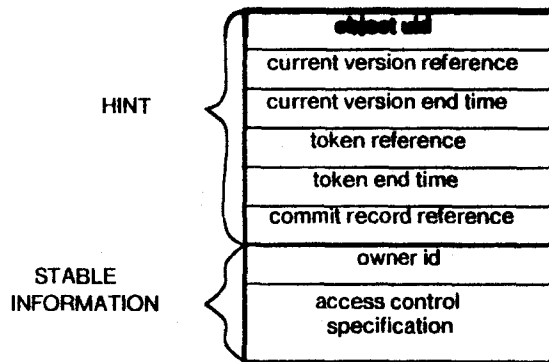
associates the object uid with the object header address. Thus read-write atomic stable storage would be needed, which is more difficult and expensive to implement than the append-only atomic stable storage used for VS. In particular, the two writes must be done sequentially. Thus the decision is *not* to reflect all changes in the object header in stable storage; Section 4.1 discusses how the object headers will be stored. Finally, Section 4.2 looks at the problem of synchronizing concurrent accesses to objects on the level of object representation.

4.1 Object headers

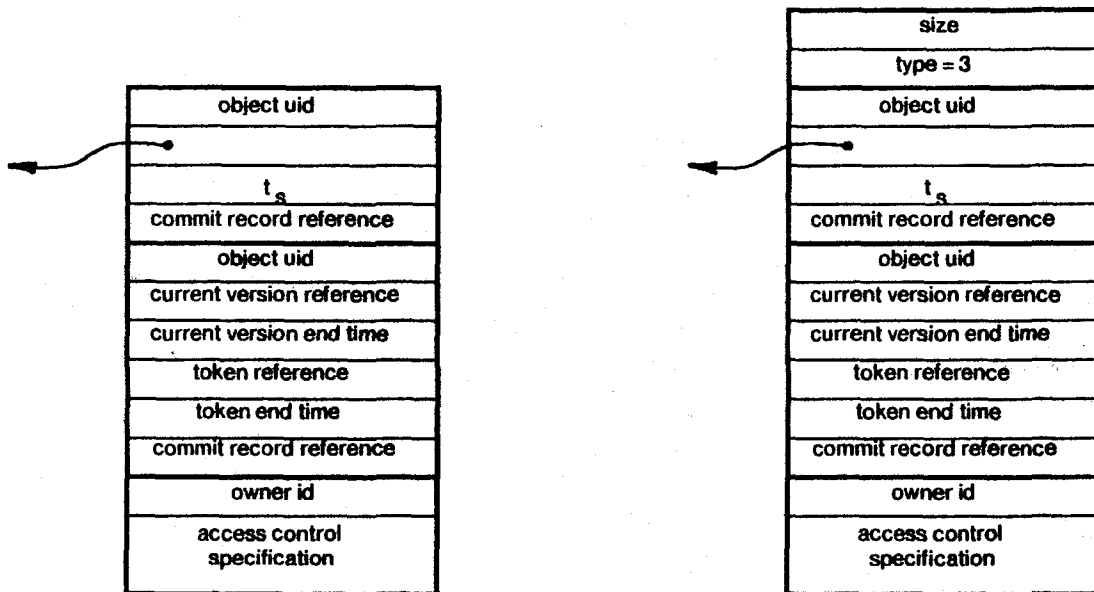
The object headers are stored on a nonvolatile storage device that allows unlimited writes (e.g., magnetic disk). This device provides Online Header Storage, or OHS. Object headers are brought into main memory as needed, and the changes made to an object header do not have to be propagated into the copy in OHS until the main memory used by the object header is to be reassigned. Since the current object headers might not be in stable storage at the time of a processor crash or a device crash, they must be reconstructable from the information that is in stable storage, in particular, the information contained in the version images. Consequently, the object headers themselves become *hints*: they are not necessary to guarantee correct operation, but of course are very important for good performance.

The object header as presented in Section 1.2 does not contain all control information that must be associated with an object. In particular, for accountability and protection, it is necessary to associate with each object the owner's id and access control specification. The access control information has to be checked for every remote request. It should be as easy to reach as the information contained in the object header; the simplest strategy is to include it in the object header. However, this additional information must be maintained in stable storage. The approach used so far, that is, inclusion of all such information in version images, is rejected for two reasons: first, it represents additional (and possibly substantial) storage overhead. Second, it is illogical to keep *write* permit information in *read-only* versions. To make it stable without having to maintain the entire object header in stable storage, the following strategy is proposed.

The object headers are maintained in OHS, but OHS is not stable (i.e., it is *not* duplicated). In the terminology of Lampson and Sturgis, OHS is *careful storage*. The object header consists of two parts, stable information and a hint, as shown in Figure 16a. When an object is created, and every time the stable information changes, the object header is created (updated) in OHS *after* a new *image* of the entire object header (that is, including the hint information) is written into VS. Finally, the object header should be written into VS when an object is deleted. The information that the object has been deleted has to be included in the object header; the access control specification field could be



a) Structure of an object header



b) Version image of an object header

c) VS image representing an object header

Figure 16: Object header and its image in VS.

used also for this purpose. Thus in addition to guaranteeing that this information will not be lost, the repository keeps a complete history of the changes of the access rights, which may be useful for auditing purposes.

To create an image of an object header, the object header is simply treated as data, and the same fields (envelope) are added as for version images (Figure 16b). The CR reference in an object header image refers to the commit record of the possibility under which the object was created, deleted, or the stable information changed. Thus treating object headers in this way solves not only the stability problem but extends the mechanism for committing tokens to the rest of the operations that modify the state of an object. In addition, the object may have a token, which has its own commit record.

The object header images in VS have to be distinguishable from the version images and data images: it must be possible to determine from the stored image itself that the *data field* represents an object header. Thus object header images represent yet another tagged *type* of entity that can be stored in VS, as shown in Figure 16c.

The hint information is guaranteed to be current *only* in the main memory. Once in a while, it is written into OHS, and it is also possible to create periodically new images of object headers in VS as checkpoints. Note that the images of the object headers will not be continuously copied in VS, since in the normal situation the object headers will be read from OHS; the VS images will be used only during recovery.

4.2 Synchronization

The repository must be able to handle several requests concurrently, since most requests will require one or more disk accesses. Also, the demon process of the OVS manager runs concurrently with the processes that execute the requests. In some cases, it is also possible to process concurrently several requests that pertain to the *same* object.

All accesses to individual object histories have to be negotiated at the object header; a single lock or monitor is needed per object. The most natural place for the lock is the object header. However, the locks must be "soft", that is, must be automatically released by a crash, otherwise if the operation that set the lock was aborted by a crash, the object could remain locked out. This can be achieved by allowing locks to be set only on the copies of the object headers in main memory, but this approach has a serious shortcoming. For easier memory management, objects should be packed in pages. Since it is not possible to "expand" the object header to add the lock when the object header is mapped into main memory, the object header must have a permanent "lock field". If

locked object headers are not allowed to appear in OHS, the pages of object headers in main memory have to be handled carefully: they must not be "write-through", and they cannot be automatically paged out by the virtual memory manager, at least not while some of the object headers on the page have their locks set. Further, it is not possible to *force* a modified object header into OHS while some other object header on the same page is locked. Alternatively, the locks for each page of OHS could be kept in a special data structure (a bit vector) in main memory. Since all object headers are of the same size, finding the appropriate lock given the OHS address of an object header is not difficult.

The "automatic release of locks" after a crash can be accomplished in yet another way: the recovery process can simply ignore the locks set on the surviving object headers in OHS, and clear the locks as part of reconstructing the object headers. This assumes that no normal processing is allowed on any object until the object header has been inspected by the recovery process; actually, as will be seen in Section 6, a read request that refers to a portion of the object history that is accessible from the surviving object header can be allowed to proceed, in spite of the object header being still locked from the epoch before the last crash.

The simplest locking policy is to lock the object header for the duration of each of the operations listed in the beginning of Section 4, but for maximum concurrency, object headers should be locked only for the shortest possible time. This corresponds to operations on the object *representation* that must be atomic. Locking guarantees only indivisibility in the absence of failures. Recoverability is provided by the underlying VS system. The individual operations on objects must lock the object header as follows:

- create:** locking is not necessary since the object does not become *known* until the create operation terminates (returns object-id)
- read:** find the appropriate version; if it is still a token, test if it can be read; change the end time if needed
- create-token:**
 - i. test if this token can be created; if yes, modify the object header to indicate that the object now has a token (note: the VS address of the token is not yet known)
 - ii. set the token reference after the version image of the token has been written into VS
- commit-token:**
 - i. change the current version reference and the current version end time
 - ii. clear the token reference and the related fields (the token end time and the commit record reference)
- abort-token:** clear the token reference and the related fields
- copy-vs:** change the current version reference

copy-token: change the token reference

The copying of version images, however, could cause problems when interleaved with execution of the external operations, in particular, commit-token or abort-token:

- i. If commit-token is executed while the current version is being copied, the OVS demon could change the current version reference *after* it has been changed to point to the new committed token.
- ii. If commit-token or abort-token is executed while the token is being copied, the OVS demon could change the token reference *after* it has been cleared to indicate that the object no longer has a token.

The latter is a lesser problem (on the first attempt to read such a copied token, it would be discovered that the token was committed and the object header would be properly reset), but it is still annoying. An additional problem arises if the "copy when read" policy is adopted (Section 3.2). When a version image of the current version or token is read and found to be past the copy mark, the OVS manager will initiate a copy operation. Now, if between the test for $A_{vi} < A_C$ and the completion of the copy operation the same image is read again, it would be copied again! In case of such a read and copy it is particularly undesirable to lock the object until the copy operation is completed, since the requested version image may have to be read from the offline VS. To solve this problem, two flags should be added to the object header: *cv.copy* and *token.copy*, to indicate that a copy operation on the current version or the token is in progress. Subsequent read requests can then proceed, but if the flag is set, the positive outcome of the $A_{vi} < A_C$ test will not start another copy operation.

The copy flags are also useful in commit-token and abort-token operations. Before changing the current version reference or the token reference field, these operations should check the appropriate copy flag. If the flag is set, the conflict can be resolved in two ways:

- i. Wait until the copy operation completes.
- ii. Abort the copy operation; that is, prevent the OVS demon from changing the current version or token reference. Note that the particular version image (copy) may have already been written into VS or will inevitably be written if it is already in some VS buffer when the copy operation is aborted. However, writing it into VS will not do any harm, not even with respect to object header reconstruction after a crash.

Note that the create-token operation does not have to be concerned with simultaneous copying. It

is impossible to copy the token before create-token terminates, and it does not matter whether the token refers to the old or the new version image of the current version.

4.3 Object directory

The object directory in a repository serves two purposes: it locates objects actually stored in the repository, and it serves as a forwarder if an object created in that repository is moved into another repository. For local objects, the directory contains the OHS address of the object header. For objects that were moved, it contains just the id of the new repository.

If the directory of some repository is lost or damaged, an exhaustive search of *all* repositories may have to be conducted to find an object known to have been created in that repository. Thus it is desirable to keep the directory in stable storage. The simplest way to accomplish this is to represent the directory as an object, with a reserved OHS address. Since the directory will be large, it will have to be represented as a structured object.

The OHS addresses do not have to change during the objects' lifetimes: thus the directory must be changed only when an object is created, moved to another repository, or deleted. Still, even with relatively infrequent changes, creating a new version of the entire directory would be very expensive. However, it should be possible to take advantage of the implementation of structured objects: for each change to the directory, it is only necessary to create a new data image of the affected piece and a new structured version image that differs from the previous one only in the reference to the modified piece. Since the size of individual pieces can change, the necessary modifications can be kept pretty localized, even if the directory is represented as a sorted list or a tree. If an entry is added and the size of the affected piece exceeds one page, it is simply split into two pieces.

Requests received by the repository must contain the uid of the desired object. The OHS address of the object header is obtained from the directory. To improve performance it is possible to return to the brokers also the OHS addresses. These addresses can then be included in future requests, in addition to the object uid. However, they are merely *hints*, that is, it is not guaranteed that the particular object can still be found at that address when the request is received. Prior to accessing an object, the object manager would have to check the validity of the hint by testing it against the object uid in the object header. If the hint as received is invalid, a new hint can be sent back as part of the response to that request. This kind of hint could be included also in the directory for those objects that were moved into another directory.

5. Management of commit records

Repositories must implement another abstract entity -- the commit record. A commit record includes the state of the possibility it represents, a timeout, and a list of tokens (references to tokens) created under that possibility. Commit records are mutable entities: both the possibility state and the list of tokens must be modifiable. While a commit record is still in an unknown state, tokens can be added to (and possibly deleted from) the list in the commit record. Once the possibility is completed, the state of a commit record is set to **committed** or **aborted** and tokens can only be removed from the list.

The list of tokens associated with each commit record is only an optimization; it is not needed to preserve consistency as required by the atomic action that created the possibility. Each token refers to its commit record; thus whether or not a token can be converted into a version can be determined by inquiring about the state of the commit record specified in this reference. This process can be sped up with the help of the token list: when the possibility is committed or aborted, all local tokens can be committed or aborted immediately. Another optimization is that it is possible to delete the commit record once all of the tokens on the list have been processed. If the token list cannot be guaranteed to include all tokens created under that possibility, then the commit record must never be deleted, because there is no other mechanism to insure that all tokens are informed about the final state of the possibility.

In Reed's original model [REED 78], the commit record of a committed possibility is assumed to be stored in atomic stable storage until all tokens on the list have been reliably changed to versions. Commit records of uncommitted possibilities (aborted, or possibilities the state of which is still unknown) do not have to be kept in stable storage: if the commit record cannot be found, the possibility can be assumed to have been aborted. Unfortunately, when the recovery of the repositories is considered, the list of tokens in a commit record is not sufficient to determine when a commit record can be deleted. In the present model, the conversion of tokens into versions is done merely by changing the references in the object header, and, as discussed in Section 4, the object headers are not stable. As it will be seen in Section 6, for recovery purposes, it is necessary to be able to determine the state of a possibility for a long time after all the tokens have been converted into versions. This means that *committed* commit records must never completely disappear from the repository; Section 5.1 presents a scheme that accomplishes this by representing commit records as *objects*. A consequence of the chosen representation is that the token lists need never be stored in stable storage. The fact that the token list does not have to be stable simplifies also the implementation of distributed possibilities as discussed in Section 5.2.

5.1 Representing commit records as objects

For stability, commit records can be mapped into VS. Since nothing ever disappears from VS, a commit record can be reconstructed even after it has been deleted at the level of abstraction implemented by the commit record manager. Commit records could be represented by yet another type of stable entity (similar to the object header image), or, they could be represented as objects. Implementing commit records as objects has the advantage that all externally accessible entities in the repository can be located and access to them controlled by the same mechanisms. On the other hand, the object abstraction needs to be extended to facilitate implementation of commit records, as will be seen later.

There are several possible ways to implement commit records as objects. The following approach was chosen because it utilizes best the mechanisms already present in the object model. When the repository receives a request to create a commit record, it creates an object. The objects and tokens created under this possibility will use, as their commit record reference, the uid of this object. Since creation of objects also must happen under some possibility, it is necessary to supply a commit record reference for the object that will represent a commit record. Recall that this commit record reference appears in both the OHS image and the VS image of the object header when an object is created. Creation of a commit record can be committed immediately. Thus a simple solution is to set the commit record reference for a commit record object to nil, to indicate that such an object is implicitly committed.

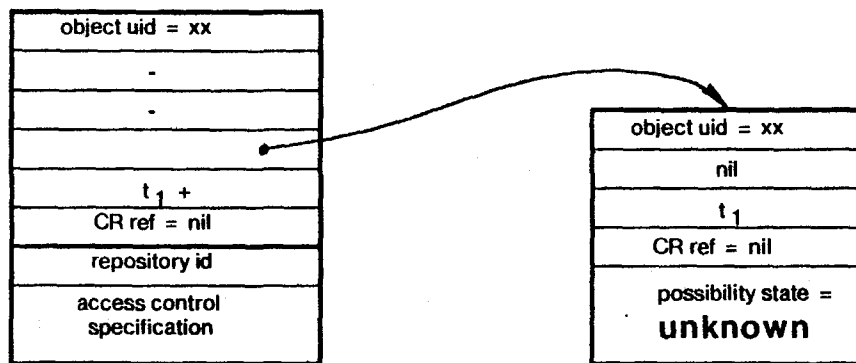
Each stable image of a commit record contains the state of the possibility. The commit record reference in the version image of an object representing a commit record is again nil. In this case, however, nil commit record reference does not mean that the version image is implicitly committed. Rather, such a version image refers indirectly to itself: the actual state of the possibility, and consequently of the representing version image, is embedded in the data field of that version image. It might be more suggestive to let the commit record reference in version images of a commit record refer to the commit record itself, but it is easier to test for a nil reference than to detect such a circular reference.

As will be seen in Section 6, the list of tokens associated with a commit record does not have to be stored in stable storage, since it is only a hint; it is not needed for recovery. If the repository crashes, all objects will be recovered individually by locating their latest version images in VS. In this process, the object manager will determine whether a version image represents a version or a token by inspecting the appropriate commit record; this must be done even for those version images that have earlier been determined to represent committed versions. Thus if the repository crashes after a possibility was committed but before all of the tokens have been converted into versions, it is

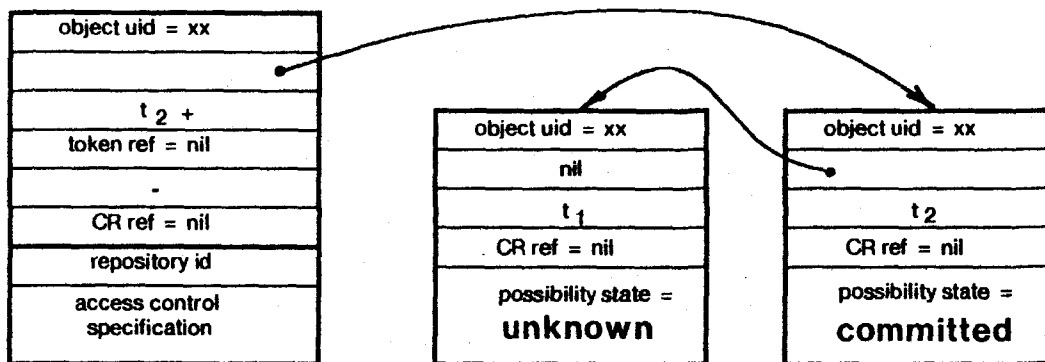
not necessary to resume or restart the conversion process since it will be finished automatically as part of recovery of the individual objects. The only reason for including the token list in a stable image of a commit record is to aid in error detection: prior to converting a token, the token list can be used to verify that this token is indeed part of that possibility.

The representation of a commit record is shown in Figure 17. In addition to creating an object as the commit record representation, the create-commit-record operation creates also a token for that object. Then the waiting for the outcome of a possibility can be accomplished through the already existing mechanism: a process attempting to read the commit record object will find a token, and consequently the read operation will be delayed until the token is either committed or aborted. To commit a possibility, the commit record manager creates the last version image for the commit record object that has the possibility state in the data field set to **committed**; this is a committed version which also commits all the preceding tokens. Now, if the possibility is aborted, it should be sufficient to abort the tokens of the commit record. For easier recovery from crashes, however, the commit record manager should, after aborting the existing tokens of the commit record, create a stable version with the possibility state set to **aborted** (Figure 17c). Finally, although deletion of an object is merely a deletion of the object header, it is still important to be able to delete commit records, since OHS is limited. With the chosen representation, commit records have to be explicitly deleted even if a possibility is aborted internally, by a repository crash or because of a timeout. The commit record manager should delete a commit record after it has processed the associated list of tokens. Such a deletion is again implicitly committed. Thus the VS image of the object header created by the delete operation will have the commit record reference set to nil. If the repository crashes before the commit record could be deleted, the commit record object will be recovered; it should be deleted as part of the recovery.

The present object model does not permit creation of another token and its commitment if the object already has a token. Since a token of a commit record cannot be turned into a version with the existing mechanisms, it is not possible to create the final version of a commit record as described above. It would be possible to add another operation, create-version, that would ignore the token, but a more general solution is to extend the object model such that it allows creation of more than one token for the same object within the same possibility. As presented in the beginning of Section 4, the object model already allows, within the same possibility, creation of a token for a newly created and uncommitted object; the extension needed to support multiple tokens is very simple. To create another token, a version image is created as for the first token, but the "previous version" field in this version image must refer now to the preceding token (Figure 18). The token reference in the object header is changed to point to the version image of the new token. The commit record reference is unchanged since the new token is created under the same possibility.

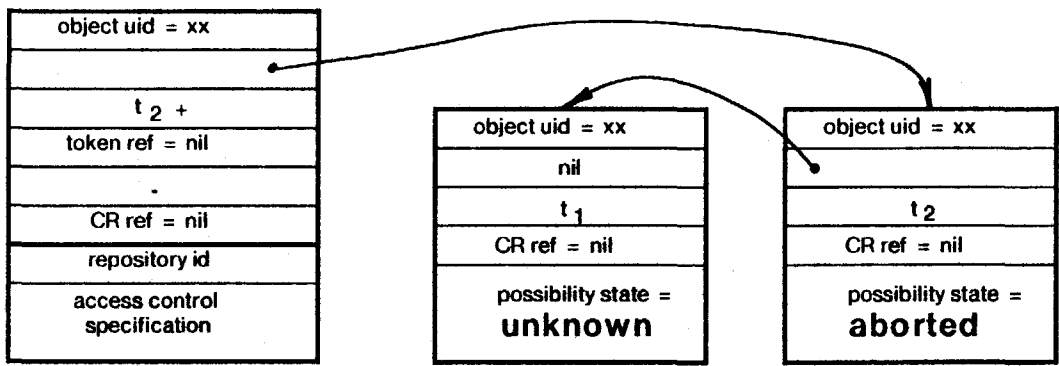


a) Creation of a commit record



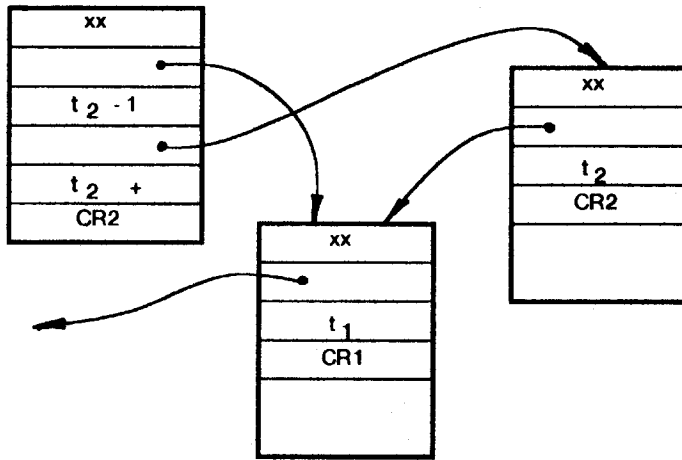
b) Commit record of a committed possibility

Figure 17: Representation of a commit record as an object.

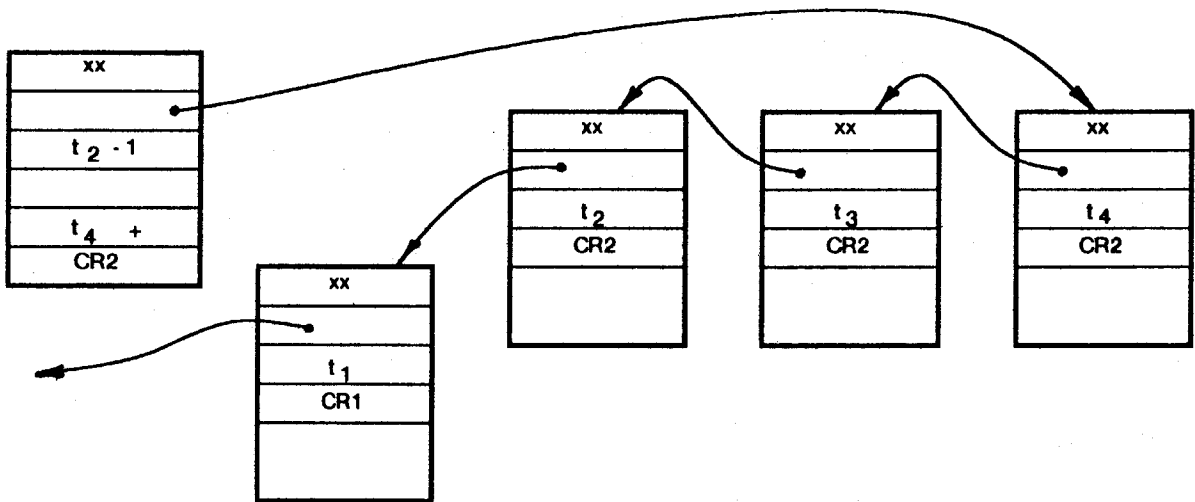


c) Commit record of an aborted possibility

Figure 17: Representation of a commit record as an object. (Cont.)

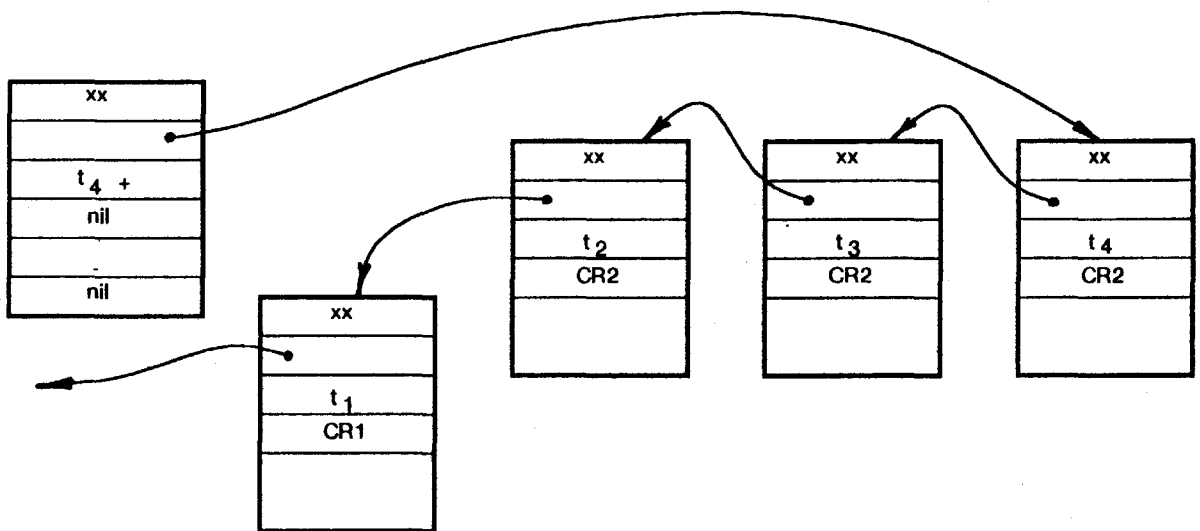


a) Creation of the first token



b) Multiple tokens

Figure 18: Creation of multiple tokens for an object within the same possibility.



c) Possibility committed

Figure 18: Creation of multiple tokens for an object within the same possibility.

(Cont.)

When the possibility is committed, this entire chain of tokens is committed at once. This does not require any changes: the current version reference becomes the reference to the last token, and the token reference is set to nil. Similarly, when the possibility is aborted, the entire chain of tokens is aborted. This extension to the object model facilitates checkpointing of commit records and data objects in general; as an extreme, commit records can be made stable throughout their lifetime. To achieve the latter, every time a token reference is added, a new version image (token) including the current list of tokens would have to be created for the commit record. However, special care must be taken when a token is copied by the OVS manager. First, only the latest token (which, if committed, will become the current version) should be subject to copying. Second, if a copy-token operation is in progress, it should be completed before an additional token can be created.

Commit records represent yet another problem. Once the possibility state is set to **committed** or **aborted**, it must not change in the future. If commit records are represented as objects, this means that it must not be possible to create another version of the representing object. This restriction must be enforced by the commit record manager, but it is aided on the object level by the access specification field, which can be set to restrict the right to update the representing object to the owner, that is, the repository.

5.2 Distributed possibilities

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 for this possibility (Figure 19). When a possibility is committed or aborted, this state is encached in the commit record representatives in all involved nodes, and the commitment or deletion of tokens is done locally.

The introduction of commit record representatives complicates the protocol for committing a possibility. To be able to rely on the token lists in deciding when to delete a commit record, all representatives with their lists of tokens must be first forced to stable storage before a decision can be made whether the possibility can be committed: a two-phase commit protocol is needed. An alternative solution is to treat the token lists in the representatives only as hints, and rely on the dual mechanism, that is, the commit record references embedded in the individual tokens. A protocol of this kind is outlined below.

Commit record representatives can be implemented in the following way. To create a commit record representative, the repository creates again an object with nil as the commit record reference (implicitly committed). In addition, it creates a token for this object, with the uid of the *primary* commit record as its commit record reference. All local tokens for this possibility will refer to the

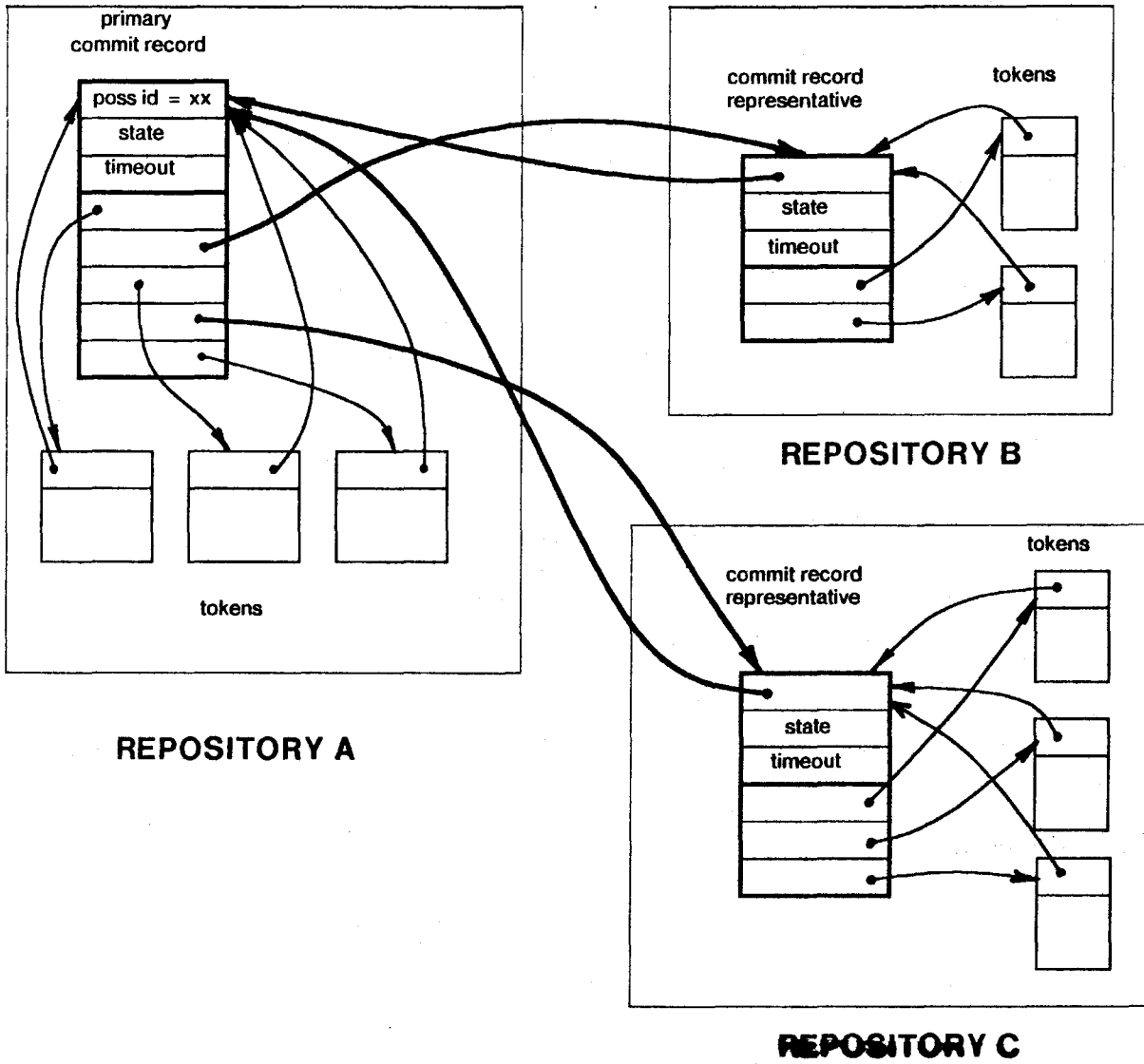


Figure 19: Implementation of a distributed possibility with commit record representatives.

object which is the local representative. When the final state of the possibility is known, the token of the commit record representative is either committed or aborted. If nothing else is done, then during crash recovery, it would be necessary to inquire again about the state of the primary commit record, and primary commit records would have to be maintained (be easily accessible) forever. Thus it is desirable to encache the state of the possibility locally in such a way that crash recovery can be confined to the failed repository. Again, it is only necessary to create a committed version of the commit record representative with the final state of the possibility (committed or aborted) embedded in it; the commit record reference in this version is now nil.

The actual protocol for distributed possibilities is summarized below:

Token accumulation phase: A repository receives a request to create a token for object x and examines the commit record id contained in the request; this is always the id of the *primary* commit record. If the respective object does not already have a committed version for the specified pseudo-time, or another token that was created under a different possibility, the repository proceeds to create the token. The create-token operation still can fail, if the repository finds out that the possibility specified in the request has already been committed or aborted. If this repository does not contain the primary commit record, it checks whether it already has a representative for this commit record. If not, it sends a request to the primary commit record for a permit to create a local representative. If approved, it creates the representative. Once the local commit record representative is located or created, the repository creates the token for object x and sets its commit record reference to the id of the object that represents the commit record.

When the request to create a commit record representative is approved by the primary commit record, a reference to that commit record representative, or, more precisely, a reference to the token of the representing object, is added to the list of tokens of the primary commit record. Note that obtaining an approval from the primary commit record is again only an optimization.

If a repository fails during the token accumulation phase, the list of tokens, if it existed only in the main memory, is lost. This does not mean, however, that the entire atomic action must be aborted, since the representing object is guaranteed to survive the crash. The only complication is that the tokens (including the tokens of the representatives in other repositories) will have to be converted individually, as other atomic actions attempt to access those objects.

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 version image of the primary commit record, with the possibility state being either committed or aborted. This version image may contain also the list of local tokens and the

references to the tokens 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 removed from the list and converted into versions or aborted. A message specifying the final state of the possibility is sent to each repository that contains a representative for this commit record. Each such repository, when it receives such a message, creates a version image of its local representative; the possibility state in this version image is set to the same value as the state in the version of the primary commit record. The repository then replies with a commit-ack message to the primary and starts converting the local tokens and removing them from the list of the local representative.

Deletion of commit record representative: When all local tokens in the list of a commit record representative are removed, the commit record is deleted, and consequently the representing object is deleted. This approach should be followed even if the possibility has been aborted.

Deletion of the primary commit record: When the primary record representative receives a commit-ack message from a representative, it removes the token reference for this representative from its list. The primary commit record can be deleted when its token list is empty.

Determining the state of a token during normal operation: To determine the real state of a token, the commit record reference in the token is used to find the local commit record representative. If the local object representing the commit record still has a token, then if the commit record reference in this token is nil, this object represents the primary commit record and the state of the possibility is still unknown. Otherwise, it is necessary to inquire at the primary commit record, which is specified by the commit record reference. If the commit record has a committed version, the state of the possibility is known locally, and is embedded in that version.

A repository should maintain a map from the primary commit record ids to the ids of the local commit record representatives. This map does not have to be stable. According to the protocol above, if a local commit record representative is not found through this map, the repository must send a request to the primary commit record to approve a creation of a representative. If the primary commit record contains a reference to a representative at that repository, its id (the uid of the representing object) will be returned. If the repository containing the primary commit record failed also and lost the token list but the atomic action continues, the requesting repository may receive an approval to create a new local record representative. This means that a repository may have more than one local representative for the same possibility, but the mechanisms of the object model and the particular implementation of the commit record representatives still guarantee consistency.

6. Recovery

At the core of the reliability measures adopted for the repository is the distinction between stable information and hints. A hint is information that is not essential for correct functioning of a system, but is important or even essential for good performance. In the SWALLOW repositories, all information in main memory and in OHS is considered to be hints reconstructable from the information in VS. The integrity of information stored in VS and OHS is assumed to be testable: this is accomplished by associating a checksum with each page.

Since during normal operation, the repository relies primarily on the hints, it is also important to be able to check the integrity of the hints. A checksum could be used also on each page in main memory, but since most hints (the object headers) change frequently, it is not feasible to recompute the checksum for each such change. In most cases, however, the validity of hints can be tested against the information in VS. For example, the current version and token reference fields of the object header must contain a VS address A_{vi} which is:

- i. valid in VS address space,
- ii. $A_{vi} < A_E$,
- iii. the object uid contained in the first word of the version image represented by this VS image matches the uid in the object header.

Only the last test is necessary to ensure that the accessed entity is indeed a version of the given object, but the first two tests can save time, since they can catch some errors without having to access VS.

The bulk of this section concentrates on the problem of recovering objects from system crashes and storage device decays. It is assumed that a system crash invalidates the entire content of the main memory. The major part of a crash recovery is reconstruction of object headers, since the current state of the recently active objects may have existed only in the main memory.

If the latest version image (the current version or a token) of an object is known, all older versions can be found by following the chain of references embedded in the individual version images. If this information is lost (when the current state of the object header is lost or damaged), it is necessary to find this version image by searching VS. This is why each version image must include the uid of the object of which it is a part. If each object is guaranteed to have at least the version images of the current version and the token in OVS, a backward search of OVS will find the beginning of all object histories. Otherwise the search must be extended to the offline portion of VS.

The recovery process must examine every VS image, starting from the end of VS. The issues of how to find the end of VS and how to isolate individual VS images on a VS page are discussed in Section 6.1. Section 6.2 presents an algorithm for reconstructing the object headers from the information in VS. Section 6.3 describes how recovery of individual objects can be distributed over time, triggered by an access to an object. Section 6.4 discusses the effect of a failure of a repository on the communication protocol between the repository and the brokers.

6.1 Retrieval of VS images

Before recovery of object headers can begin, it is necessary to find the current end of the version storage, that is, the address of the latest page written into OVS; the mark M_E can be viewed as pointing to the end of this page. This address could be found by searching from the *low* end of OVS or from the copy mark, in the direction of increasing VS addresses. In some of the OVS management schemes, these other marks are implicit, and thus no additional precautions must be taken. To remember the end of VS reliably, the mark M_E would have to be kept in stable storage. Otherwise, M_E can be found by searching for the first "free page." On an optical disk, this means the beginning of the area that has not yet been written. On a magnetic disk, each page, as released by the OVS demon, could be marked as "free." Such information provides a useful check in general: before the version buffer in main memory is written into VS, the specified OVS page should be checked if it is free.

The failure might have occurred between the two physical writes in the duplicated implementation of stable VS. If the latest page as written to one of the devices is found correct, the VS write can be completed, that is, that page is written also to the other device; otherwise that page should be marked as bad, and the end of VS set to the end of the preceding page (the latest page on the other device). No external request (that is, a request from a broker or another repository) for which some information has to be written into VS is acknowledged until *both* writes complete; thus if a (dual) VS page is declared bad because the second write did not complete correctly, no harm is done. However, if the write is completed during recovery, the create-token requests that caused creation of VS images on that page cannot be acknowledged, since the repository lost all information about these requests. The original requestors may retry their requests, in which case the recovered repository will send back an acknowledgement, as discussed in Section 6.4. Otherwise, the individual tokens on that page eventually will be aborted because of a timeout. Copies of version images made by the OVS manager will be found and incorporated into the chains representing the object histories by the main recovery process.

The next problem is to isolate the individual VS images. The scan of VS should proceed from its

high end towards the low end; for individual pages, this means from the end of a page towards its beginning. This means that the size field should be at higher-address end of a VS image. Since for normal use, the position of the size field must be computable from the VS address contained in a version or token reference, this implies that VS images should be stored so that their *first* word, that is, the word specified by those references, has the highest VS address. Finally, if a page is not completely filled when written into VS, a dummy data image should be created in the unused space. This dummy data image will be discovered only during recovery, but it will be automatically ignored since all data images are ignored during recovery: only the size field of the representing VS image is used to get to the beginning of the preceding VS image.

6.2 Reconstruction of object headers

Since most repository crashes will not damage OHS, the recovery process can use OHS image as the starting point. As stated earlier, it is assumed that a checksum is associated with each OHS page, and that it is sufficient to test the integrity of the object headers on the page. The value of the field that specifies the end validity time of the current version in the object header in OHS provides a logical delimitation for recovery: only if some version (token) was created *after* this time (this would mean that the OHS image was not updated), the hint in the object header must be updated (reconstructed) from the information in VS. Unfortunately, because of the copying of version images in VS, there is no simple unique mapping from time to a physical location in VS. Thus only the current version reference A_{CV} and the token reference, A_T , in the surviving object header are useful: VS must be searched only as far as the higher of these two addresses.

If the object header in OHS is damaged, VS must be searched until *all* of the following is found:

- 1) a version image of the current version
- 2) a version image of the token (if any)
- 3) an image of the object header.

If the OHS image is not damaged but is merely obsolete, it is only necessary to find the first two items. If the found image of the object header precedes the version images of both the current version and the token (i.e., it is the latest entity in VS pertaining to this object), the object header is recovered without any further search. If a version image of a token is found first, it is not necessary to search for a version image of the current version, since a reference to it is contained in the token. However, if a version image of the committed version is found first, it could be a copy, and thus it is still necessary to search for a token. Moreover, this version image does not necessarily represent the *current* version! This can happen if the current version had been copied while the object had had a token, such as in Figure 8b, after which the token was committed but not copied.

Fortunately, these two anomalies are mutually exclusive. Thus, if the first version image vi_L (*latest* in VS), found for a particular object represents a committed version, it is necessary to continue the search until a version image vi_X that represents a *different* version or a valid (not aborted) token is found. If an image of the object header is found next after vi_L , then vi_X is the version image pointed to by the token reference, if not nil, the current version reference otherwise. Now vi_L represents the current version if:

- 1) vi_X represents a token or
- 2) $t_s(vi_X) < t_s(vi_L)$ where t_s is the start validity time of that version.

If $t_s(vi_X) > t_s(vi_L)$, then vi_X is the current version and the object does not have a token.

A token representation is indistinguishable from a version representation. If there exists a reference to version image X in another version image, X must be a committed version. But if a version image is retrieved without such context, to distinguish between a committed version and a token, it is necessary to check the commit record, or, more specifically, the local commit record representative. This is why a version image of a token (and consequently, a version) must contain the uid of its commit record. Also, when an image of an object header is found, it may have been written into VS as part of an operation that has not yet been committed. Recall that a VS image of the object header is made when the object's status is changed: the object is created or deleted, or its access specification is changed. Again, it is necessary to use the commit record reference in the object header image to determine the state of the possibility under which the status of the object was to be changed. Thus an important part of reconstructing the object headers is finding the appropriate commit records.

Since commit records are represented by objects, they must first be recovered by the same mechanism as objects representing clients' data. However, at the time of a crash, a large portion of the commit records that will have to be inspected during recovery have been probably deleted. This means that their object headers were written into VS, marked as deleted. The repository does not have to recover deleted objects (given that the deletion was committed), but it must temporarily recover deleted commit records, so that other objects can be recovered. Since VS images of object headers are easily distinguishable (their commit record reference is nil), the handling of deleted commit records does not represent a major problem.

The copying of version images by the OVS manager complicates also the reconstruction of the relevant commit records. Without copying, the commit record of a possibility that reached the final state would be guaranteed to be recovered prior to all version images and object header images created under that possibility. When a committed version image is copied, it gets "ahead" of its

commit record, that is, the recovery process will find that version image before it recovers the commit record. This can happen even if the copied version image is still a token: if the copying of the token occurs just before the state of the possibility is finalized, the copy of the token and the version image of the commit record may end up in different VS buffers, and be written into VS in the reverse order. The images of object headers are always ordered correctly in VS, since they are read from VS only during recovery and therefore are not copied by the OVS manager.

The search process sketched in the beginning of this section must be expanded to take into account the problem of recovering the commit records. It is assumed that only the final state of a possibility is recorded in stable storage. Also, if the recovery process does not find a version representing the final state of a given possibility, it cannot abort the possibility, since the reconstructed local object might be just a representative of the commit record.

Again, the exact recovery of individual objects depends on in what order the various relevant entities are found:

- ▶ The first entity found is an image of the object header:

Since the VS images of object headers are not copied in OVS, then if the changes to the object status as reflected by this object header image were finalized (committed or aborted), the appropriate commit record version must have been already found by the recovery process. If it has not been found, the possibility is still in **unknown** state. In any case, the current version reference and token reference in this object header image can be used to rebuild the object header in OHS. If the found object header image is not committed, the version reference in this image can be used to find the preceding VS image of the object header which contains the correct stable information for this object.

- ▶ The first entity found is a version image; call it again vi_L :

1. The commit record for this version image has already been reconstructed. This can happen only if:
 - a. vi_L is a committed version that has *not* been copied; since this is the first image found, the object does not have a token.
 - b. vi_L is an aborted token. Embedded in this token is a reference to the current version; neither the current version nor the commit record for the current version have to be searched.
2. The commit record has not been reconstructed yet. This can happen if:

- a. The final version of the commit record has not been created yet, thus vi_L represents a token.
- b. vi_L represents a committed version that was copied by the OVS manager.
- c. vi_L represents an aborted token that got ahead of the final version of the commit record due to the nonsequential management of the VS buffers.

To resolve this uncertainty, it is necessary to continue the search of VS until one of the following is found:

- i. A version image of the commit record:
 - If the embedded possibility state is **unknown**, vi_L is a token, and it contains a reference to the current version.
 - If the embedded possibility state is **committed**, vi_L is a copy of the current version; it is still necessary to search for the possible token.
 - If the embedded possibility state is **aborted**, vi_L is a copy of an aborted token. vi_L contains a reference to the current version, and the object does not have a token.
- ii. Another version image, vi_X , created under a different possibility than vi_L (this restriction is sufficient to handle correctly situations where vi_X is just another copy of the same version image, and also the cases when multiple tokens were created under the same possibility):
 - If $t_s(vi_X) < t_s(vi_L)$ then vi_L must be a token. Embedded in vi_L is a reference to the current version; this is not necessarily vi_X , since vi_X could be an aborted token or a no longer accessible copy of an earlier version.
 - If $t_s(vi_X) > t_s(vi_L)$, then vi_L must be a copy. If vi_X is a token or an aborted token, then vi_L represents the current version, otherwise it is a copy of the preceding version. Thus it is necessary to continue the search of VS until the commit record for vi_X is reconstructed.

Finally, the object headers contain the end time of the current version and the token; this information also must be reconstructed somehow. If an object has a token, the end time of the current version must be one "tick" less than the creation time of the token. The end time of the token, and if an object does not have a token, then the end time of the current version, ought to be set to the current time, that is, the time when the object is recovered.

6.3 Real-time recovery

The actual 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 through special checkpoints. In addition, rather than recovering all objects in the repository before resuming normal processing, recovery can be distributed over time. In particular, individual objects can be

recovered as they are accessed.

For this, it is necessary to be able to distinguish the epochs between different recoveries. Thus the repository should maintain, as part of its state, the current recovery epoch number, REN. Every time the recovery process is started, the repository is assigned a new REN such that these numbers monotonically increase in time. REN must be included also in each object header. When an object is created, it is assigned the current REN. When an object is accessed through any of the operations listed in Section 4, then if its OHS image 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 the object header was written into VS during the recovery epoch as given by its REN. If the object is locked, the lock is simply broken; the locks must be honored only if the object REN and the current repository REN are the same. If an object is not used for a long time, several crashes (and recoveries) could have occurred since the object was created or recovered. However, since such an object has not been recovered earlier, it could not have been used (read or written) 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.

Thus, the recovery process should, at the commencement of a recovery, write a mark into VS that specifies the beginning of a new recovery epoch. For quick location of these marks, they should be chained together as are the histories of individual objects. Thus the recovery mark can be represented by an object: if the object header in OHS survives the crash, the last version is easy to find, and the new version of the mark can be created with the reference to the last one immediately. If the object header of the recovery mark is destroyed, it is necessary first to search VS for the last version version of the recovery mark. The object header of the recovery mark is modified *only* during recovery, and it should be forced immediately into OHS. This guarantees that the correct information is always in OHS and thus should survive most crashes.

When an object is recovered, its REN in the reconstructed object header is set to the current REN. Also, a VS image of the object header should be created: 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.

In the process of reconstructing an object, it is again necessary to reconstruct the appropriate commit record(s). Since atomic actions survive repository crashes, the fact that the final version of a commit record is not found in the same recovery epoch as the object in question does not mean that the state of the possibility has not been resolved. But since a commit record is also an object,

an attempt to access it will direct automatically the recovery process into the right recovery epoch.

6.4 Communication with brokers

A failure of a repository can also affect the brokers. It is the responsibility of the brokers to supervise that requested operations are indeed performed by the repositories. If a broker does not receive a reply from a repository, then unless the requested operation is not important for correct completion of the given atomic action, the broker has two options:

- i. abort the entire atomic action, or
- ii. repeat the request.

Now, of course it is possible that the first request was received and processed by the repository, but since all operations supported by the repositories are idempotent (if they carry the same pseudo-time), duplicate requests do not represent any problem. The only complication arises if a message from a broker containing data for a token is delivered in pieces. Unless the entire structured version image was already created, if the request is repeated, the previous incomplete message must be discarded, since the partitioning of the repeated message may be different from the previous one.

7. Summary

Figure 20 summarizes the structure of a SWALLOW repository as a lattice of abstractions. A more detailed description of the structure is given in the appendix. The entire design of the repository is centered around the Version Storage, which is the *only* stable storage in the repository. 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.

VS contains not only the versions of objects, but also the commit records and images of the object headers. However, the name Version Storage has been retained, since:

- i. commit records are represented by ordinary objects (and thus VS contains their *versions*), and
- ii. the object header images are in fact selected versions of the state of individual objects.

VS is append-only storage, in accordance with the basic object model. It provides a linear paged address space with a straightforward mapping from the VS address into a location on the physical device. VS is duplicated for stability, but since no update in place is possible, the two required writes can be concurrent.

Since VS may grow very large, it is impossible to maintain the entire VS online. Only the upper 2^n words of VS are kept in the Online Version Storage. OVS would thus contain the current versions and tokens of the recently updated objects. To make sure that the current versions of most objects are found in OVS, it is necessary to copy occasionally the images of current versions and tokens to the high end of VS. The most reasonable policy for managing OVS seems to be to copy a version image when the repository is processing a read request involving a current version or a token and the representing VS image is found to have a lower VS address than the copy mark. This policy preserves locality of reference, and automatically brings back online the current versions of the objects that have not been used for a long time.

OVS can be implemented with a reusable device, or with write-once devices. The latter form simplifies the transfer of version images from online to offline storage. The delays due to manual device replacement can be eliminated through a circular assignment of device drivers to different functions in the implementation of OVS.

The crash recovery of the repositories is based entirely on the information contained in VS. Current contents of object headers, although the object headers are the key elements in all

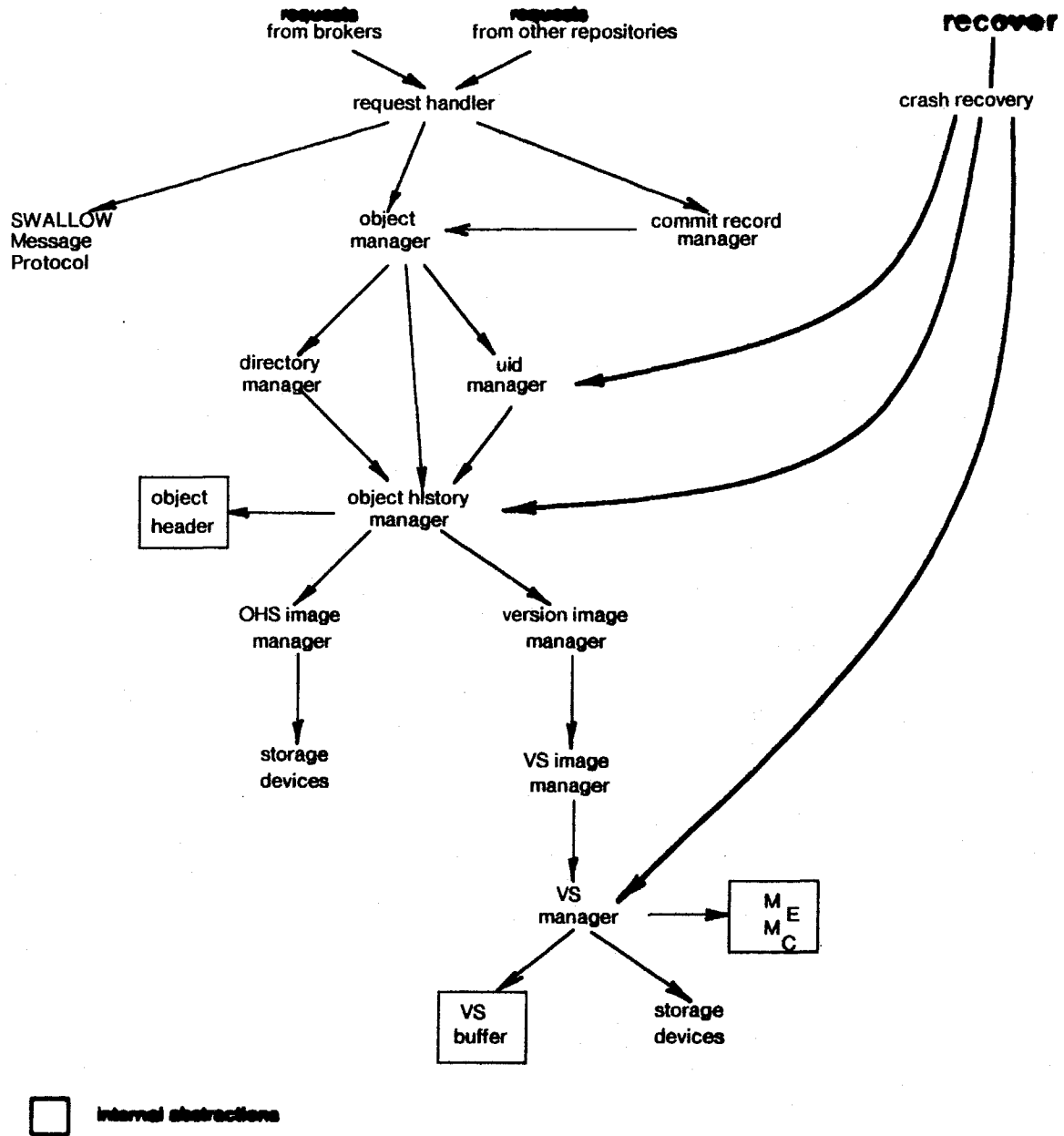


Figure 20: Structure of the repository.

operations on objects, are treated as hints that are fully reconstructable from the information found in VS. Since the commit records are implemented as objects, they are reconstructable by the same process. Finally, the object directory is an object itself and hence reconstructable from the information in VS.

This report presented only a skeleton for the design of the SWALLOW repositories. Many issues were touched on only very lightly, and some important issues have not been addressed at all. In particular, performance of OVS under the proposed copying policy needs to be evaluated and the sketched algorithm for reconstruction of the object headers ought to be analyzed more formally for possible inconsistencies. Some of the additional issues are:

- i. **Virtual memory.** It has been assumed that both VS and OHS are divided into pages, and that pages from both are brought into main memory on demand. So far, OHS and VS have been treated as distinct address spaces. This means that to implement virtual memory their pages would have to be mapped into main memory in different ways. Alternatively, OHS and VS can be made part of the same address space, e.g., OHS can be the lowest 2^k words of that space.
- ii. **Communication with brokers and other repositories.** Objects can be sent to repositories in pieces, subject to the constraints imposed by the communication substrate and communication buffer capacity of the receiver. Although the repository can deal with pieces of any size (if they are too big, they will be broken up further before being stored as data images), better performance can be achieved if the communication substrate already delivers pieces of the right size; the optimal size is the size of a page minus the amount of storage needed for the size field and the type tag which are added when a data image is created.
- iii. **Protection.** It is assumed that object versions in the repository will be stored in an encrypted form, where encryption provides the only kind of protection for read accesses [REED 80]. Some protection against modification is provided by the immutability of object versions, but it should be possible to control the ability to create and delete objects, create tokens and change the state (commit or abort) of commit records. Objects and commit records in the repository were designed to include an access control specification field which is stable; however, it is not clear what should be in this field and how the rights of the requestors should be checked. An interesting question is what the right to read means in the context of the given object model. In particular, does a revocation of such right apply only to the future versions of the object, or also to the current and the past versions?
- iv. **The repository provides mechanisms that facilitate building of atomic actions;** however, it is the responsibility of the users of SWALLOW to make sure that these

mechanisms are used properly. The division of responsibility for correct implementation of atomic actions should be studied in more depth. SWALLOW could assist in enforcing correct use by supervising that:

- a. a possibility cannot be committed until all outstanding requests to create a token have been received and processed
 - b. once a possibility is committed or aborted, no new tokens can be added.
- However, distributed possibilities make such checking difficult.

References

- COFF 73 Coffman, E.G., Jr., Denning, P.J., *Operating Systems Theory*, Prentice-Hall, Inc., Englewood Cliffs, N.J.
- GRAY 79 Gray, J., et. al., "The Recovery Manager of a Data Management System," IBM Research Laboratory Technical Report RJ2623, August 1979.
- 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," submitted to the International Workshop on Local Networks to be held in Zurich, Switzerland, August 1980.
- SWIN 79 Swinehard, 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.

Appendix

STRUCTURE OF THE REPOSITORY

This appendix describes in more detail the individual modules of the repository and their logical interconnection (the "uses" hierarchy presented in Figure 20). Note that some modules support more than one abstraction developed in this report. *External operations* are the operations provided at the module's interface, that is, operations that can be invoked from other modules. *Internal operations* are available only within the module. *Recovery operations* are special external operations that are invoked only by the recovery process.

Request handler

implements: *repository interface*

uses: *object*
commit record
SWALLOW Message Protocol

The request handler inspects messages delivered by the SWALLOW Message Protocol [REED 80] and invokes the appropriate manager to handle the request, and it constructs reply messages from the information returned by the manager.

Commit record manager

implements: *commit record*
commit record representative

uses: *object*

external operations:

create -->

add reference -->
commit -->

abort -->

use:

create object
create token
primitives of the implementation language
create token
commit token
delete reference
delete object
create token
commit token
abort token
delete reference
delete object

internal operations:

delete reference -->
delete -->

use:

primitives of the implementation language
delete object

recovery operations:

none (recovered only as *objects*)

Object manager

implements: *object*

uses: *directory*
object history
uid

external operations:

create -->

read -->

create token -->

commit token -->

abort token -->

set access control -->

delete -->

use:

get new uid
create object history
enter into directory
lookup directory
read object history
lookup directory
create token
lookup directory
commit token
lookup directory
abort token
lookup directory
set access control on object history
lookup directory
delete object history
delete from directory

recovery operations:

none

UID manager

implements: *uid*

uses: *object history*

external operations:

new -->

use:

may have to create new version

recovery operations:

reset uid -->

use:

reconstruct object history

Directory manager

implements: *directory*

uses: *object history*

external operations:

create -->
enter -->
lookup -->

use:

create object history
primitives of the implementation language
primitives of the implementation language

recovery operations:

recover -->

use:

reconstruct object history

Object history manager

implements: *object history*

uses: *version image*
OHS image

external operations:

create -->

use:

create object header
create version image (of object header)
create OHS image
read object header
read version image (returns also A_C)

read -->

copy current version
copy token

create token -->

read object header
create version image

commit token -->

read object header

abort token -->

read object header

set access control -->

read object header

delete -->

create version image (of object header)
read object header
create version image (of object header)
delete OHS image

internal operations:

| | | |
|----------------------|-----|---|
| create object header | --> | primitives of the implementation language |
| read object header | --> | read OHS image |
| write object header | --> | write OHS image |
| copy current version | --> | copy version image |
| copy token | --> | copy version image |

recovery operations:

| | | |
|-------------|-----|--|
| reconstruct | --> | read object header search version image create version image (of object header) write object header |
|-------------|-----|--|

Version image manager

implements: *simple version image*
structured version image
VS image of object header

uses: *VS image*

external operations:

| | | |
|----------------------|-----|---|
| create version image | --> | create VS image |
| read version image | --> | read VS image (returns also Λ_C) |
| copy version image | --> | create VS image |

recovery operations:

| | | |
|--------|-----|---------------|
| search | --> | next VS image |
|--------|-----|---------------|

VS image manager

implements: *VS image*

uses: *VS*

external operations:

| | | |
|--------|-----|--|
| read | --> | read VS page (returns also Λ_C) |
| create | --> | append VS |

recovery operations:

next (iterator) -->

use:

next VS page

VS manager

implements: *VS*

uses: *main memory page*
storage device

external operations:

append -->
read page -->

use:

append VS buffer
read storage device page (OVS or offline VS)
get A_C (returned together with the requested page)

internal operations:

append VS buffer -->
reset M_C -->
get A_C -->
assign device drivers -->

use:

allocate main memory page
write storage device page
primitives of the implementation language
primitives of the implementation language
primitives of the implementation language

recovery operations:

find end -->
next page (iterator) -->

use:

read storage device page (recover M_E)
read storage device page

OHS image manager

implements: *OHS image*

uses: *storage device*

external operations:

create -->
read -->
write -->
delete -->

use:

write storage device page
read storage device page
write storage device page
write storage device page

recovery operations:

none

use:

Crash recovery

uses:

uid
object history
VS

external operations:

start recovery

-->

use:

find end of VS
create recovery mark

internal operations:

create recovery mark

-->

use:

get new uid (new REN)
create token (for the recovery mark object)
commit token